Diplomarbeit

Adaptierung der Eclipse JDT Refactorings für die aspektorientierte Programmiersprache ObjectTeams/Java mit Hilfe von OT/Equinox

Johannes Gebauer
Mat.-Nr. 222068
Berlin, 9. Dezember 2009

Technische Universität Berlin
Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik
Prof. Dr. Ing. Stefan Jähnichen

Diplomarbeitsbetreuer: Dr. Ing. Stephan Herrmann
Erstgutachter: Prof. Dr. Ing. Stefan Jähnichen
Zweitgutachter: Dr. Ing. Steffen Helke

Eidesstattliche Erklärung

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt.
Berlin, den 9. Dezember 2009

. . . . . .
Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Die zunehmende Komplexität von Softwareprodukten stellt die Entwickler vor immer neue Herausforderungen. Ein bewährtes Mittel zur Reduktion der Komplexität ist die Modularisierung. Objektorientierte Programmiersprachen stellen eine Reihe von Abstraktionskonzepten zur Verfügung, mit denen ein Programm übersichtlich strukturiert werden kann. Allerdings lassen sich damit nicht alle Aspekte eines Programms in separate Module unterteilen. Querschneidende Anforderungen betreffen mehrere Module und verteilen sich über große Teile des Programms. Die aspektorientierte Softwareentwicklung erweitert die herkömmlichen objektorientierten Programmiersprachen um neue Konzepte zur Modularisierung dieser sogenannten *Crosscutting Concerns*.

Die Entwicklung eines Softwareprodukts endet in den wenigsten Fällen mit der Auslieferung. Die Software wird stetig verändert und an neue Anforderungen angepasst. Bei diesen Anpassungen muss die gute Struktur des Programms bewahrt werden und zusätzlich den neuen Anforderungen gerecht werden, damit die Software auch zukünftig evolutionsfähig bleibt. Eine gute Struktur zeichnet sich dadurch aus, dass sie leicht verständlich ist und Erweiterungen der Software mit möglichst geringem Aufwand durchgeführt werden können. Die Erweiterung von Programmen führt oft dazu, dass sich die Struktur und damit die Lesbarkeit verschlechtert. Es muss also zusätzliche Arbeit geleistet werden, um diese unerwünschten Nebeneffekte zu kompensieren. Refactoring ermöglicht es die Struktur an die neuen Bedürfnisse anzupassen ohne dabei das sichtbare Verhalten der Software zu verändern.

1.1. Zielsetzung

Sowohl Refactoring, als auch die aspektorientierte Softwareentwicklung sind Techniken um die Struktur von Software zu vereinfachen. Dabei behandeln beide Techniken verschiedene Probleme der Softwareentwicklung. In dieser Arbeit sollen beide Techniken miteinander verbunden werden, indem ein Werkzeug zur Unterstützung beim Refactoring von ObjectTeams/Java (OT/J) Programmen entwickelt wird. Als Grundlage für die Entwicklung wird das bereits existierende Refactoring Werkzeug des Eclipse Java Development Tooling (JDT) verwendet. Dieses Werkzeug

stellt bereits die volle Funktionalität für das Refactoring von Java Programmen zur Verfügung und soll mit Hilfe von OT/Equinox für die neuen OT/J Sprachkonzepte adaptiert werden. Zusätzlich soll an diesem Fallbeispiel untersucht werden, wie der geschickte Einsatz von Aspekten eine neue Form der Wiederverwendung realisieren kann. Abschließend wird das Werkzeug um spezielle OT/J Refactorings ergänzt, die exemplarisch zeigen sollen, welche neuen Möglichkeiten des Refactorings die aspektorientierte Softwareentwicklung bietet.

1.2. Aufbau der Arbeit

Die Arbeit ist wie folgt aufgebaut: Im folgenden Kapitel 2 wird die Verwendung von aspektorientierten Programmiersprachen motiviert und anschließend die Java Spracherweiterung OT/J vorgestellt. Dabei werden alle Sprachkonzepte vorgestellt, die einen Einfluss auf Refactoring haben können.

Kapitel 3 beschäftigt sich mit den theoretischen Grundlagen von Refactoring und beschreibt die grundlegenden Voraussetzungen für die Entwicklung eines Refactoringwerkzeugs.

Aus den in Kapitel 2 vorgestellten Sprachkonzepten werden die Einflüsse auf bestehende Java Refactorings abgeleitet. In Kapitel 4 wird ein detaillierter Regelsatz vorgestellt, der die wichtigsten Sprachregeln von OT/J zusammenfasst. Anschließend werden die wichtigsten Refactorings vorgestellt und die Einflüsse der Sprachregeln untersucht.

Bevor die praktische Arbeit beschrieben wird, wird in Kapitel 5 kurz die Plug-in Architektur der Eclipse Plattform erläutert, die die Grundlage für die praktische Arbeit bildet. Da die Adaption mit Hilfe von OT/Equinox umgesetzt werden soll, wird diese Erweiterung in einem Unterabschnitt vorgestellt.

Die praktische Umsetzung der verschiedenen Refactoring Adapter wird in Kapitel 6 beschrieben. Dazu wird als erstes das Refactoring Language Toolkit (LTK) vorgestellt, das ein Framework für die Entwicklung von Refactorings für Eclipse darstellt. Anschließend werden die entwickelten Adapter vorgestellt.

In Kapitel 7 werden zunächst die Möglichkeiten neuer OT/J Refactorings untersucht und ein paar neue Refactorings beschrieben. Im zweiten Teil des Kapitels wird die Implementierung von zwei neuen Refactorings im Detail vorgestellt.

In Kapitel 8 wird die Migration der Refactorings für die Eclipse Version 3.6 beschrieben.

Kapitel 9 enthält eine Zusammenfassung der Arbeit, eine Bewertung der Ergebnisse und einen Ausblick für mögliche Anknüpfungspunkte für zukünftige Arbeiten.

2. Einführung in ObjectTeams/Java

In diesem Kapitel wird die grundlegende Funktionsweise und Motivation von aspektorientierten Programmiersprachen vorgestellt. Anschließend wird die Betrachtung anhand von Object-Teams/Java vertieft und alle für die Arbeit relevanten Sprachkonzepte erläutert. Die neuen Sprachkonzepte bilden die Grundlage für die notwendigen Adaptierungen, die bei den herkömmlichen OO-Refactorings durchgeführt werden müssen. Außerdem wird OT/J selbst als Werkzeug für die Änderungen an den bestehenden Refactorings verwendet, um den ursprünglichen Code nicht zu verändern.

2.1. Motivation für aspektorientierte Programmiersprachen

Durch Modularisierung ist es möglich die Komplexität von Software zu verringern und auch bei großen Softwareprojekten eine übersichtliche Struktur zu schaffen. Modularisierung ermöglicht außerdem Arbeitsteilung, Wiederverwendung und die Zuordnung von Modulen zu bestimmten Anforderungen. Der von Dijkstra geprägte Begriff *Separation of Concerns* beschreibt das Prinzip, Lösungen für verschiedene Probleme voneinander zu trennen [Dij74]. Ein *Concern* beschreibt dabei einen Teil des Problems, dessen Lösung nach Möglichkeit in einem Modul implementiert werden sollte.

2.1.1. Crosscutting Concerns

Wie in der Einleitung erwähnt, lassen sich mit den herkömmlichen Konzepten der objektorientierten Programmierung nicht alle Anforderungen modularisieren. Querschneidende Anforderungen verteilen ihren Code über mehrere Module und lassen sich nicht in einem separaten Modul kapseln. Sie werden in der Literatur als *Crosscutting Concerns* oder auch *Aspekte*

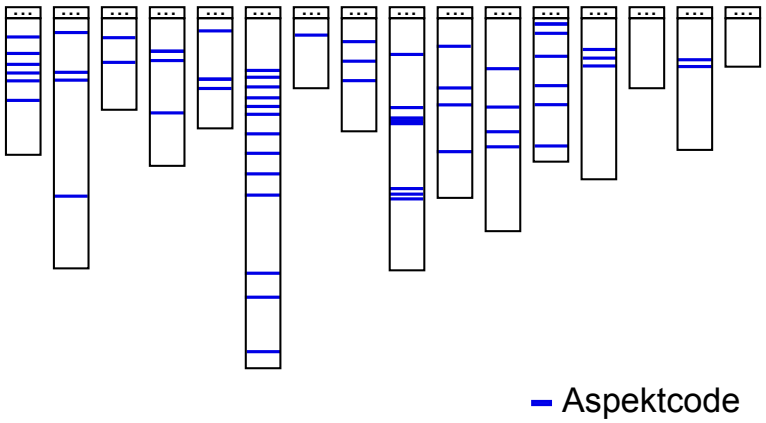

Abbildung 2.1.: Graphische Darstellung der Verteilung eines Aspekts über mehrere Klassen

bezeichnet. Häufig handelt es sich bei nichtfunktionalen Anforderungen wie z. B. Performanz, Fehlerbehandlungen oder Sicherheitsmechanismen[1] um Aspekte [KIL+97].

Ein beliebtes Beispiel zur Veranschaulichung eines Aspekts ist der Loggingmechanismus. Wenn beispielsweise Änderungen von Attributwerten protokolliert werden sollen, müssen in allen relevanten Set-Methoden Logginganweisungen untergebracht werden. Da sich Set-Methoden in vielen Klassen eines Programms befinden, könnte die Verteilung des Codes für einen solchen Loggingaspekt so aussehen wie die Darstellung in Abbildung 2.1. Die Rechtecke stellen dabei den Quelltext der verschiedenen Klassen dar, in denen die Anweisungen des Aspekts farblich hervorgehoben sind.

Wenn sich die Implementierung eines Concerns über mehrere Module verteilt spricht man von *Code-Scattering*. Code-Scattering erschwert die Wartung, da sich in den verschiedenen Modulen redundanter Code befindet, der bei Änderungen leicht zu Inkonsistenzen führen kann. Ein Aspekt besitzt nicht nur selbst eine schlechte Struktur, sondern beeinträchtigt auch den Code der ursprünglich gut modularisierten Concerns. Diese enthalten neben ihrer Kernfunktionalität zusätzlichen Aspektcode. Dies beeinträchtigt die Lesbarkeit, da das Modul nicht mehr nur einem bestimmten Zweck zugeordnet werden kann. Die Wiederverwendbarkeit eines Moduls wird ebenfalls eingeschränkt, da dieses durch den Fremdcode Abhängigkeiten zu verschiedenen Aspekten haben kann. Wenn sich der Code von verschiedenen Concerns in einem Modul vermischt, spricht man von *Code Tangling*.

[1] Im Sinne von Security, wobei Rollen und Rechte vor der Ausführung von Operationen geprüft werden müssen.

2.1.2. Modularisierung von Aspekten

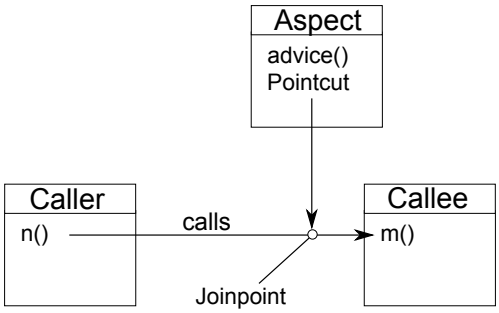

Abbildung 2.2.: Struktur eines aspektorientierten Programms

Der bekannteste Vertreter der aspektorientierten Programmierung im Java Umfeld ist die Sprache *AspectJ*. Im folgenden werden die neuen Modularisierungskonzepte mit Hilfe der in AspectJ gebräuchlichen Terminologie erläutert. Die grundsätzlichen Ideen lassen sich jedoch auch auf andere aspektorientierte Programmiersprachen übertragen. Im Abschnitt 2.2 wird dann detailliert erläutert, wie diese in der Sprache OT/J umgesetzt wurden.

Filman und Friedman nennen in ihrer Definition der aspektorientierten Programmierung zwei zentrale Eigenschaften:

> „Aspect-Oriented Programming is Quantification and Obliviousness.“[FF01]

Unter *Quantification* versteht man die Fähigkeit einer Sprache eine Menge von Punkten im Programm anzugeben, an denen ein Aspekt ausgeführt werden soll. Dies wird in AspectJ durch das Konzept der Pointcuts unterstützt. Die *Obliviousness*[2] beschreibt den Grad der Unabhängigkeit des Basiscodes gegenüber dem Aspektcode. Im Idealfall befinden sich im Basiscode keine Referenzen auf den zugehörigen Aspektcode. Die Untersuchung dieser Eigenschaft kann im späteren Verlauf der Arbeit Aufschluss darüber geben, inwiefern ein im Aspektcode durchgeführtes Refactoring den Basiscode beeinträchtigen kann.

Für die Modularisierung von Aspekten stellt AspectJ eine Reihe von Konzepten bereit. Der Code des Aspekts wird in einem separatem Modul implementiert und mit Hilfe von *Pointcuts* an den Basiscode[3] gebunden. Die Anweisungen des Aspekts werden dabei als *Advice* bezeichnet. Ein Pointcut beschreibt eine Menge von Punkten im Programm, an denen der Advice ausgeführt werden soll. Die Pointcuts werden dann beim *Weben* zu *Joinpoints* aufgelöst, die Stellen im Quelltext darstellen. In Abbildung 2.2 ist ein Aspekt dargestellt, der Methodenaufrufe der Methode `m` verändert.

[2] Aus dem Englischen: Vergesslichkeit

[3] Als Basiscode wird der Teil eines Programms bezeichnet, der mit herkömmlichen OO-Mechanismen implementiert wurde.

Beim Weben wird der zuvor modularisierte Aspektcode mit dem Basiscode zusammengeführt, damit zur Laufzeit die gesamte Funktionalität des Programms verfügbar ist. Dabei unterscheidet man im wesentlichen zwei Arten des Webens. Das statische Weben findet beim Übersetzen des Programms statt, wohingegen das dynamische Weben erst zur Laufzeit geschieht. Der wesentliche Vorteil des dynamischen Webens ist, dass auch zur Laufzeit neue Aspekte hinzufügt oder bestehende deaktiviert werden können. Das dynamische Weben führt dafür zu einer etwas schlechteren Performanz als ein statisch gewobenes Programm.

Zusätzlich zu der Verhaltensänderung mit Hilfe von Advices bietet die aspektorientierte Programmierung die Möglichkeit die Struktur eines Programms nachträglich zu verändern. In AspectJ spricht man dabei von *Inter-type Declarations*. Damit ist möglich neue Methoden, Felder oder Konstruktoren einzuführen. Auch ein Typ kann verändert werden, indem die Vererbungshierarchie angepasst wird. Dabei können neue Superklassen eingefügt werden oder ein zusätzliches Interface implementiert werden.

2.2. Sprachkonzepte von ObjectTeams/Java

Die Sprache ObjectTeams/Java wurde entwickelt, um die Möglichkeiten der Modularisierung von reinen OO-Sprachen zu verbessern. Jedoch ist der technische Ansatz ein anderer, als das reine separieren von Aspekten. Das Sprachdesign wurde von den Ideen des *Subject-Oriented Programming* beeinflusst. Beim Subject-Oriented Programming sieht man ein Objekt nicht als statisches Konstrukt, das Methoden für alle möglichen Klienten und deren Concerns bereitstellt. Da jeder Klient eine subjektive Sicht auf das Objekt besitzt, enthält dieses nur allgemeine Kernfunktionalitäten. Zusätzliches Verhalten wird in sogenannten *Rollen* implementiert, die zum benötigten Zeitpunkt das Verhalten eines Basisobjekts an die Anforderungen des jeweiligen Klienten anpassen können [HO93].

Durch die Verteilung des Verhaltens auf verschiedene Rollen ergeben sich neue Möglichkeiten der Modularisierung. Herkömmliche Sprachen bieten nur die Möglichkeit ein Programm hinsichtlich eines Merkmals zu strukturieren. So kann man beispielsweise einen Baum in einer hierarchischen Struktur darstellen, bei dem die einzelnen Knoten im Vordergrund stehen. Wenn Operationen auf dem Baum ausgeführt werden sollen, lassen sich diese mit dem *Visitor* Design Pattern implementieren [GHJV95]. Das Visitor Pattern schneidet jedoch die bestehende Hierarchie der Knoten quer und fügt allen Knoten zusätzliche Funktionalität hinzu. Ein Visitor lässt sich also nicht in die strenge Hierarchie eines Baums einfügen. Mit Rollen ist es jedoch möglich ein eigenes Modul für den Visitor zu bilden, der die Knoten für seine subjektive Sicht erweitert.

Die Strukturierung eines Programms hinsichtlich einer einzigen Dimension ist also nicht ausreichend. Nur mit einem Programmierparadigma, das eine mehrdimensionale Zerlegungen zulässt, können beliebige Concerns modularisiert werden. Beim Subject-Oriented Programming spricht man dabei von der Zerlegung des Programms in *Hyperslices* [TOHJ99]. Der Begriff wird hier sogar noch weiter gefasst und bezieht sich auf alle Artefakte, die während eines Softwareentwicklungsprozesses entstehen. Das Object Teams Programmiermodell beschränkt sich dabei auf die Zerlegung von Quellcode Artefakten. Ein Hyperslice bildet ein Modul, das einen bestimmten Concern implementiert. Rollen befinden sich innerhalb eines Moduls und ermöglichen nicht invasive Adaptionen von bestehenden Klassen und können dabei die Funktionalität der adaptierten Klassen wiederverwenden. Die verschiedenen Hyperslices werden dann zu einem kompletten Programm zusammengesetzt. Dies kann ähnlich wie das Weben in AspectJ beim Übersetzen oder erst zur Laufzeit geschehen.

2.2.1. Teams und Rollen

OT/J führt das neue Konzept des *Teams* ein. Ein Team wird durch das neue Schlüsselwort **`team`** gekennzeichnet und dient als Container für *Rollen*. Eine Rolle wird als *Inner Class* des Teams deklariert und kann wiederum selbst ein Team sein, das weitere Rollen beinhaltet. Diese Struktur lässt sich beliebig tief schachteln.

Rollen können an eine *Basis* gebunden werden. Als Basis kann jede herkömmliche Java-Klasse, eine Rolle oder ein Team verwendet werden. Die Bindung wird durch eine *playedBy-Relation* festgelegt, die hinter der Rollendeklaration mit dem Schlüsselwort **`playedBy`** und dem gewünschten Basistyp angegeben wird.

```
public class Role playedBy Base{}
```

Die playedBy-Relation stellt eine neue Typreferenz dar, die mit der Typreferenz in einer Extends-Relation in Java verglichen werden kann.

Die Rolle kann die Basisklasse um neue Attribute und Methoden erweitern. Außerdem kann sie die Methoden der Basis verändern, indem sie diese um zusätzliche Anweisungen erweitert oder die gesamte Methode ersetzt. Die genauen Mechanismen der Wiederverwendung und Adaption von Basismethoden werden in Abschnitt 2.2.3 und 2.2.4 genauer erläutert.

Role Directories

Rollen können nicht nur als Inner Class ihres umschließenden Teams deklariert werden. Alternativ können sie in einer eigenen Java-Datei implementiert werden, die sich in einem Ordner mit

dem Namen des Teams befindet. Um zu unterscheiden, dass es sich bei der Klasse um eine Rolle und keine gewöhnliche Java Klasse handelt, wird in der Java-Datei die Pfadangabe des Packages mit einem vorangestellten **`team`** Schlüsselwort gekennzeichnet. Das Konzept des Rollen Ordners stellt eine Abhängigkeit zur Namensgebung von Teams dar, die bei Umbenennungen berücksichtigt werden muss. Die Rolle in einer Rollendatei unterscheidet sich dabei semantisch nicht von Rollen, die als Inner Class deklariert wurden.

Externalized Roles

Normalerweise sind Rollen in einem Team gekapselt und außerhalb nicht sichtbar. Wenn Rollenreferenzen auch außerhalb verwendet werden sollen, müssen sie an einen sogenannten *Type Anchor* gebunden werden. Ein Type Anchor ist eine unveränderliche Referenz[4] auf eine Teaminstanz, an die die Rolle gebunden ist. Diese Zuordnung soll dafür sorgen, dass Rolleninstanzen nicht zwischen verschiedenen Teaminstanzen vertauscht werden können. Die Teaminstanz wird dabei als Parameter in spitzen Klammern angegeben.

```
final MyTeam myTeam = expression;
RoleClass<@myTeam> role = expression;
```

Das Konzept des Type Anchors erweitert Java um eine neue Referenz auf eine lokale Variable oder ein Feld.

Kapselung von Rollen

Für Rollen stehen einige Konzepte zur Verfügung um externe Zugriffe einzuschränken und damit die Integrität interner Rolleninstanzen zu wahren. Um *Aliasing* zu verhindern, kann eine Rolle von der in der Sprache vordefinierte Klasse `Confined` erben. Der Compiler stellt sicher, dass auf diese Rollen keine Referenzen außerhalb des umschließenden Teams existieren. Außerdem wird das Erben von Methoden verhindert, die dem Aufrufer eine **`this`** Referenz zur Verfügung stellen.

Um die Rollenreferenz auch außerhalb des Teams zu verwenden, aber trotzdem die Integrität zu wahren, kann das Interface `IConfined` implementiert werden. Es erlaubt externe Referenzen auf die Rolle, verbietet jedoch den Zugriff auf sämtliche Member der Rolle. Ein Verstoß gegen diese Regel führt zu einem Fehler beim Übersetzen des Programms.

Die Klasse `Confined` und das Interface `IConfined` stellen vordefinierte Typen dar, deren Namen nicht mehr für Rollen zur Verfügung stehen. Außerdem müssen Refactorings, die die

[4] Eine mit **`final`** deklarierte Referenz.

Programmstruktur derart verändern, dass neue This-Aufrufe entstehen, prüfen ob diese erlaubt sind.

2.2.2. Translation Polymorphism

Um eine Ersetzbarkeit zwischen Rollen und Basisklassen zu ermöglichen, führt Object Teams zwei neue Operationen ein. Zusammen bilden sie die Grundlage für den *Translation Polymorphism*, der Ähnlichkeiten mit dem Konzept des *Subtype Polymorphism* in der OO-Programmierung aufweist.

Lowering

Lowering ermöglicht die Navigation von einer Rolle zu ihrer Basis. Da eine Rolleninstanz immer an genau eine Basisinstanz gebunden ist, stellt Lowering keine besonderen Schwierigkeiten dar. Lowering ist eine sichere Operation, die mit einem *Down Cast* vergleichbar ist. Daher wird diese Operation bei Bedarf implizit ausgeführt und ist nicht im Quelltext sichtbar.

In bestimmten Situationen ist es jedoch sinnvoll ein explizites Lowering durchzuführen. Dazu implementiert die Rolle das Interface `ILowerable`. Daraufhin erzeugt der Compiler die Methode `lower()`, mit der ein explizites Lowering durchgeführt werden kann. Interfaces sind ein bereits bekanntes Java Konzept, daher stellt Lowering keine neuen Sprachfeatures bereit, die bei einem Refactoring berücksichtigt werden müssten. Es gilt zu beachten, dass der Name „ILowerable“ nicht für Rollen verwendet werden darf, da er innerhalb eines Teams auf das oben beschriebene Interface verweist.

Ob eine Rolle konform zu einer Basis ist hängt von der playedBy-Relation ab. Refactorings, die dazu führen, dass eine playedBy-Relation verallgemeinert wird, müssen prüfen ob vorhandenes Lowering weiterhin möglich sind.

Lifting

Lifting ist die Umkehroperation zum Lowering und soll zu einer Basisinstanz eine bestimmte Rolleninstanz finden. Da eine Basis mehrere Rollen gleichzeitig spielen kann, muss beim Lifting immer ein Kontext aus Teaminstanz, Basisinstanz und Rollenklasse angegeben werden. Implizites Lifting hat wie das implizite Lowering keinen Einfluss auf die Struktur des Programms. Explizites Lifting kann in Methodensignaturen von Rollen angegeben werden.

```
public void m(Base as Role param){}
```

Das Lifting für einen Parameter wird durch das neue Schlüsselwort **as** angefordert, auf das der gewünschte Rollentyp folgt. Dieses Konstrukt wird in der Sprachdefinition als *Declared Lifting* bezeichnet.
Der angeforderte Rollentyp stellt eine neue Referenz auf einen Typ dar, die bei einem Refactoring berücksichtigt werden muss.

2.2.3. Calloutbindungen

Mit *Callouts* kann eine Rolle Methoden und Attribute ihrer Basisklasse wiederverwenden. Es handelt sich dabei um eine Art selektiven Import, der in der Rolle deklariert wird. Dabei ist auch ein Zugriff auf private und protected Member möglich. OT/J beeinflusst dadurch die ursprünglichen Sichtbarkeitsregeln von Java, d. h. private Namensbezeichner können auch über die Klassengrenzen hinaus bekannt sein.

Callout Methodenbindung

Basismethoden können an abstrakte Methoden der Rolle gebunden werden.

```
abstract void roleMethod();
roleMethod -> baseMethod;
```

Die Basismethode wird in einer Bindung entweder durch ihren eindeutigen Namen oder durch ihre vollständig angegebene Signatur identifiziert. Wenn die Signatur in der Calloutbindung angegeben wird, kann auf die Deklaration einer abstrakten Rollenmethode verzichtet werden. Die Methode wird dann direkt in der Calloutbindung deklariert und der Compiler erzeugt eine Methode mit dem entsprechenden Namen und der angegebenen Signatur.

```
void roleMethod() -> void baseMethod();
```

Um Mehrdeutigkeiten auszuschließen, muss für überladene Methoden immer die vollständige Signatur angegeben werden. Refactorings, die zum Überladen von Methoden führen können, müssen diesen Punkt berücksichtigen. Außerdem stellt eine Calloutbindung eine neue Methodenreferenz dar, die mit einem Methodenaufruf in Java verglichen werden kann.

Callout to Field

Felder der Basisklasse können nach dem gleichen Prinzip wie Methoden gebunden werden. Der Zugriff erfolgt dabei nicht direkt, sondern über entsprechende Getter- und Setter-Methoden,

die mit den neuen Schlüsselwörtern **get** und **set** erzeugt werden. Listing 2.1 zeigt dazu ein Beispiel.

```
1  getValue -> get value;
2  setValue -> set value;
3  int getValue() -> get int value;
```

Listing 2.1: Beispiel für ein Callout to Field [HHM09]

Callout Parameter Mapping

Falls die Parameter- oder Rückgabetypen in einer Calloutbindung nicht konform sind, können hinter der Bindung Konvertierungen angegeben werden. Das Mapping wird mit dem neuen Schlüsselwort **with** eingeleitet und muss für alle Parameter, sowie einen möglichen Rückgabewert, eine Konvertierung angeben (siehe Listing 2.2). Refactorings, die Signaturen von Methoden verändern, müssen vorhandene Parameter Mappings beachten.

```
1  Integer absoluteValue(Integer integer) -> int abs(int i) with {
2    integer.intValue() -> i,
3    result <- new Integer(result)
4  }
```

Listing 2.2: Beispiel für ein Callout Parameter Mapping [HHM09]

Inferred Callouts

Es ist möglich den OT/J Compiler so zu konfigurieren, dass Callouts automatisch aufgelöst werden. Dann kann eine Rolle ohne Calloutbindungen direkt auf die Felder und Methoden ihrer Basis zugreifen. Dies erweitert Java um neue unqualifizierte Feldzugriffe und Methodenaufrufe, die mit einem Zugriff auf geerbte Felder und Methoden verglichen werden können.

2.2.4. Callinbindungen

Callins bieten in OT/J die Möglichkeit das Verhalten der Basis zu verändern. Eine Callinbindung fängt Aufrufe der Basismethode ab und führt in der Rolle definierte Instruktionen aus. Dabei stehen drei Modifikatoren zur Beschreibung der Bindung zur Verfügung. Mit den Schlüsselwörtern **before** und **after** wird der Code der Rollenmethode vor oder nach der Ausführung des Basiscodes ausgeführt. Ein Überschreiben der Methode kann mit dem Schlüsselwort **replace** erreicht werden. Innerhalb eines Replace Callins kann der Code der Basismethode mit Hilfe eines *Base Calls* wiederverwendet werden. Ein Base Call ist mit einem Superaufruf in Java vergleichbar. In Listing 2.3 ist die Syntax eines Replace Callins mit Base Call dargestellt.

```
1  public team class MyTeam{
2    protected class MyRole playedBy MyBase{
3      private void log(String log){ ... }
4      callin void mRole(){
5        log("begin m");
6        base.mRole();
7        log("m done");
8      }
9      void mRole() <- replace void m();
10   }
11 }
```

Listing 2.3: Beispiel für ein Callin mit Base Call

Auch bei Callins ist es möglich auf private Methoden zuzugreifen. Daher erweitern Callins die Sprache Java technisch in der gleichen Weise wie ein Callout. Der Sichtbarkeitsbereich von privaten Methoden wird vergrößert und Callinbindungen stellen eine neue Form der Methodenreferenz dar.

Callin Parameter Mapping

Auch Callins können ein Parameter Mapping angeben. Die Syntax ist ähnlich wie bei einem Callout Parameter Mapping. Es ergibt sich daraus eine weitere Abhängigkeit zur Signatur einer Methode, die bei Änderungen beachtet werden muss.

Reihenfolge bei mehreren Callins

Falls mehrere Callins mit dem gleichen Modifikator (**`before`**, **`after`** oder **`replace`**) an die gleiche Basismethode gebunden werden, ist es notwendig eine Ausführungsreihenfolge festzulegen. Dazu müssen die verschiedenen Callinbindungen mit einem Namen versehen werden und mit dem Schlüsselwort **`precedence`** eine Ausführungsreihenfolge der verschiedenen Callinbindungen angegeben werden. Es können auch Konflikte über mehrere Klassen entstehen, dazu ist es möglich innerhalb des umschließenden Teams eine Reihenfolge der Rollen anzugeben. Wenn nur der Rollenname angegeben wird, gilt die angegebene Reihenfolge für alle Callinbindungen dieser Rolle. Für eine feinere Festlegung der Callin Reihenfolge können die verschiedenen Callinbindungen auch über den umschließenden Rollennamen qualifiziert werden. Listing 2.4 zeigt ein komplexes Beispiel einer Precedence Deklaration. Refactorings müssen darauf achten die Semantik von Precedence Ausdrücken zu bewahren.

```
1  public team class MyTeam{
2
3    precedence MyRole.callin1, MyRole2, MyRole.callin2;
4
5    protected class MyRole playedBy MyBase{
6      callin void m(){}
7      callin void n(){}
8
9      callin1: void m() <- replace void m();
10     callin2: void n() <- replace void m();
11
12     precedence callin1, callin2;
13   }
14
15   protected class MyRole2 extends MyRole{
16     callin void k(){}
17
18     callin3: void k() <- replace void m();
19   }
20 }
```

Listing 2.4: Verschiedene Formen der Callin Precedence Deklaration

2.2.5. Teamaktivierung

Um die Verhaltensänderungen eines Teams besser kontrollieren zu können, kann es aktiviert und deaktiviert werden. Callinbindungen greifen nur in den Kontrollfluss des Programms ein, wenn ihr umschließendes Team aktiviert ist. Eine weitere Möglichkeit Callinbindungen zu unterbinden stellen *Guard Predicates* dar, die in Abschnitt 2.2.6 genauer beschrieben werden. Das Schlüsselwort **`within`** aktiviert ein Team für das folgende Statement oder den folgenden Block. Technisch stellt das within Konstrukt eine ähnliche Struktur wie *If-Statements* in Java dar, die beliebige Ausdrücke im Rumpf enthalten können.

within(myTeam){ stmts }

Die Teamaktivierung kann alternativ mit den Methoden `activate()` und `deactivate()` gesteuert werden. Optional kann bei der Aktivierung ein Thread übergeben werden, in dem das Team de-/aktiviert werden soll.

2.2.6. Guard Predicates

Guard Predicates erlauben eine feinere Kontrolle als die Teamaktivierung. Es können Prädikate für Callinbindungen, Rollenmethoden, Rollen und Teams angegeben werden. Wenn die Auswertung des Prädikats **`false`** ergibt, werden die betroffenen Callinbindungen deaktiviert[5]. Es wird

[5]Wenn das Prädikat an einer Rolle oder einem Team steht, sind alle enthaltenen Callinbindungen betroffen.

zwischen *regular Guards* und *Base Guards* unterschieden, wobei base Guards vor und regular Guards nach dem Lifting evaluiert werden.

```
when(predicateExpression)
```

Guard Predicates stellen technisch ein ähnliches Konstrukt wie die If-Statments in Java dar. Innerhalb des Ausdrucks können sich beliebige Feld- und Methodenreferenzen der Basis, Rolle und des Teams befinden, die beim Refactoring beachtet werden müssen. Dabei hängt der Sichtbarkeitsbereich davon ab, ob es sich um ein Base oder regular Guard handelt. Es können daher nicht Basis- und Rollenmethoden innerhalb des selben Guards referenziert werden.

2.2.7. Teamvererbung

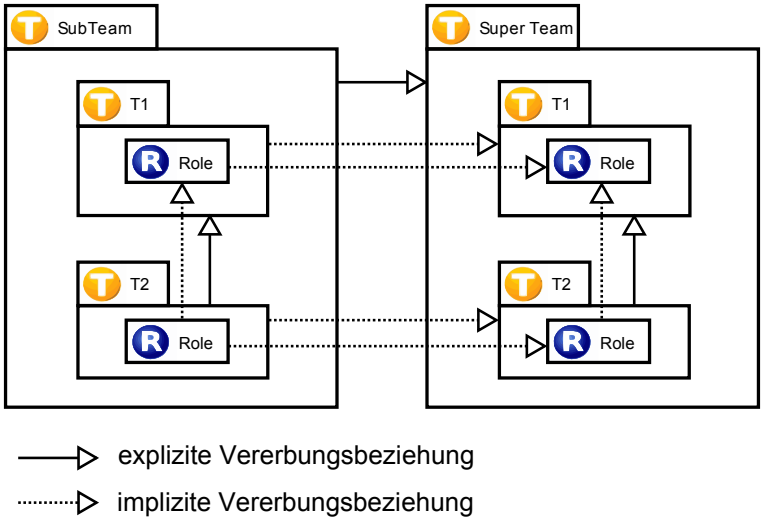

Abbildung 2.3.: Teamvererbung mit mehreren impliziten Superklassen

Mit der Teamvererbung erweitert OT/J die Konzepte der herkömmlichen Klassenvererbung. Ein Team erbt nicht nur die Methoden und Felder des Superteams, sondern auch dessen Rollen. Wenn ein Team eine Rolle mit dem gleichen Namen wie eine geerbte Rolle deklariert, findet ein Überschreiben der Rolle statt. Die überschreibende Rolle steht dabei in einer impliziten Vererbungsbeziehung mit der von ihr redefinierten Rolle.

Die Teamvererbung ermöglicht dabei drei wesentliche Konzepte:

- Eine Form der *Copy Inheritance*, bei der die vollständige Funktionalität der Rollen des Superteams in dem erbenden Team zur Verfügung steht.
- *Overriding* von Rollen, bei dem eine Rolle des Superteams durch Verwendung des gleichen Namens im erbenden Team redefiniert werden kann.

- *Implizite Vererbung*, bei der eine überschreibende Rolle die Features ihrer impliziten Superrolle erbt.

Abbildung 2.3 zeigt eine Teamvererbung, bei der durch Schachtelung mehrere implizite Superklassen für die Rolle `SubTeam.T2.Role` entstehen. Sie besitzt neben der expliziten Superklasse `Object` zusätzlich die impliziten Superklassen `SubTeam.T1.Role` und `SuperTeam.T2.Role`. In OT/J wird also ein zusätzliches Überschreiben von Typen eingeführt, was mit dem herkömmlichen Überschreiben von Methoden in Java verglichen werden kann. Ein Refactoring muss sich also der neuen impliziten Vererbungshierarchie bewusst sein, in der es nicht mehr nur eine direkte Superklasse geben muss.

Auch die Vererbung zwischen Rollen erweitert die herkömmlichen Klassenvererbung um einige Konzepte. Es werden auch private Member vererbt, was zur Folge hat, dass diese auch überschrieben werden können. Auch die neuen Sprachkonzepte, wie die playedBy-Relation, sowie Callout- und Callinbindungen werden vererbt. Ein Refactoring muss diese neue Vererbungsregeln beachten, um neue Redefinitionen zu erkennen und korrekt zu behandeln.

2.2.8. Phantomrollen

Durch Teamvererbung können Rollen entstehen, die im Quelltext nicht sichtbar sind. Wenn das Superteam eines Teams eine Rolle enthält, die nicht redefiniert wurde, ist sie trotzdem mit der vollen Funktionalität der im Superteam definierten Rolle vorhanden. In der internen Repräsentation des Object Teams Development Tooling (OTDT)s werden solche Rollen auch als *Phantomrollen* bezeichnet. Eine Phantomrolle ist semantisch äquivalent zu einer redefinierten Rolle, die einen leeren Rumpf besitzt.

Abbildung 2.4 zeigt ein Beispiel für eine Phantomrolle. Die Rolle `Chess.Move` wird nicht im Quellcode erwähnt, existiert jedoch durch die Vererbungshierarchie des umschließenden Teams `Chess` und stellt eine Kopie der Rolle `BoardGame.Move` dar.

Refactorings müssen daher berücksichtigen, dass Rollen nicht direkt im Quelltext vorhanden sein müssen um zu existieren. Erst eine Analyse der Vererbungshierarchie eines Teams kann alle Rollen des Teams bestimmen.

2.2.9. Implizite Vererbung von Referenzen

Durch die Teamvererbung ist es möglich, dass Referenzen implizit an Rollen vererbt werden. Solche Referenzen sind nicht unbedingt im Sourcecode sichtbar, sondern müssen aus der Deklaration in einem Superteam abgeleitet werden.

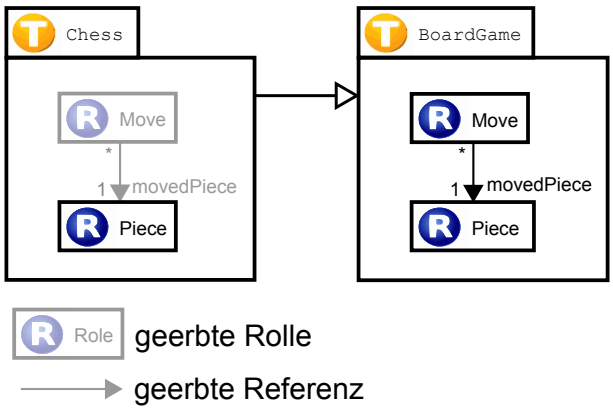

Abbildung 2.4.: Die Rolle `Chess.Move` erbt die Referenz `movedPiece`

Abbildung 2.4 zeigt ein Beispiel für die implizite Vererbung einer Referenz.
Die Rolle `BoardGame.Move` ist nicht in `Chess` redefiniert worden und wird daher durch *Copy Inheritance* in `Chess` reproduziert. Die Referenz wird vererbt und zeigt durch die Eigenschaften des *Family Polymorphism* jetzt auf die Rolle `Chess.Piece`.

Refactorings müssen daher bei Referenzanalysen für Rollen auch die Typhierarchie betrachten.

3. Refactoring

In Kapitel 2 wurde die aspektorientierte Softwareentwicklung als eine Technik zur besseren Strukturierung von objektorientierten Programmen vorgestellt. In diesem Kapitel soll Refactoring als weiteres Hilfsmittel zur besseren Strukturierung von Software vorgestellt werden. Zunächst werden die theoretischen Grundlagen von Refactoring erläutert. Darauf aufbauend wird beschrieben, wie Werkzeuge dem Entwickler bei der Durchführung von Refactorings unterstützen und zur Fehlervermeidung beitragen können.

3.1. Theoretische Grundlagen

Die Softwareentwicklung unterliegt einem ständigem Evolutionsprozess. Dies ist darauf zurückzuführen, dass fortlaufend neue Funktionalität hinzugefügt wird oder auf Grund von fehlenden oder unklaren Anforderungen Änderungen an bestehendem Code durchgeführt werden müssen. Während diesem Prozess verschlechtert sich die Struktur der Software stetig und man spricht von *Software Decay*. Um diesem Verfall entgegen zu wirken, ist es notwendig die Struktur an die neuen Anforderungen anzupassen. Eine solche Anpassung kann das einfache Umbenennen einer Methode sein oder das Ersetzen von doppelten Quelltext-Passagen durch einen einheitlichen Methodenaufruf. In dem Standardwerk „Refactoring: Improving the Design of Existing Code“ stellt Fowler einen Katalog von Refactoring-Mustern bereit und definiert *Refactoring* wie folgt:

> „Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.“
>
> „Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behavior.“[Fow99]

Die Definitionen zeigen, dass ein Refactoring nicht das sichtbare Verhalten eines Programms verändern darf. Der Vorgang des Refactorings muss daher separat von der Implementierung von Features durchgeführt werden, um sicherzustellen, dass keine unerwünschten Seiteneffekte auftreten.

Dabei gibt es zwei Bedingungen, die nach der Durchführung eines Refactorings gelten müssen:

1. Das Programm ist nach dem Refactoring syntaktisch korrekt und lässt sich fehlerfrei kompilieren.
2. Das Programm liefert nach dem Refactoring für gleiche Eingabewerte die gleichen Ergebnisse. Es bleibt semantisch äquivalent.

Mit Hilfe eines Parsers lässt sich leicht die syntaktische Korrektheit eines Programms überprüfen. Der Compiler gibt Auskunft über die Korrektheit der statisch analysierbaren Semantik, wie der Existenz von Namen und Typkorrektheit von statisch typisierten Programmen. Die Überprüfung der semantischen Äquivalenz von dynamischen Eigenschaften stellt hingegen eine deutlich größere Herausforderung dar. Diese kann z. B. mit einer *Testsuite* überprüft werden, wobei die Aussage jedoch direkt von der Qualität der eingesetzten Testsuite abhängt.

Im Rahmen dieser Arbeit wird der Begriff der Verhaltenserhaltung auf die Funktionalität eines Programms beschränkt. In speziellen Softwareprojekten könnten auch andere Aspekte des beobachtbaren Verhaltens von Interesse sein, wie z. B. Performanz oder Security Eigenschaften.

Das Verhalten von *reflexiven* Programmen kann ebenfalls durch Refactoring beeinflusst werden, da es sich bei referenzierten Namen nur um Strings und nicht um echte Referenzen handelt, die von einem Compiler aufgelöst werden könnten. Dadurch kann nach einer Verschiebung oder Umbenennung eines Features eine Verhaltensänderung auftreten.

Auch einige Joinpoint Sprachen aspektorientierter Programmiersprachen arbeiten mit Reflexion. Weder die Java Reflection-API, noch die Joinpoint Sprache von OT/J werden in dieser Arbeit bei der Verhaltenserhaltung berücksichtigt[1].

3.1.1. Abgrenzung

Es ist schwer eine genaue Grenze zu ziehen, bei welchen Transformationen es sich um ein Refactoring und bei welchen nicht. Die zwei folgenden Beispiele beschreiben Grenzfälle, die die Problematik verdeutlichen.

[1] Laut § 8.1. in [HHM09] wird die Joinpoint Sprache in der Version 1.2 noch nicht unterstützt.

Quelltext Generatoren

Auf der einen Seite gibt es in modernen Entwicklungsumgebungen eine Reihe nützlicher Quelltext Generatoren, die das beobachtbare Verhalten des Programms nicht verändern. Dies kann z. B. das Erzeugen von Getter- und Setter-Methoden für ein Feld oder die Erzeugung von *Methoden-Stubs* für neu implementierte Interfaces sein. Diese Operationen sind so trivial, dass eine komplexe Analyse des Programms nicht nötig ist.

Verhaltensändernde Refactorings

Auf der anderen Seite gibt es Refactorings, die Verhaltensänderungen erzeugen können. Das Refactoring *Inline Local Variable* ersetzt alle Referenzen einer lokalen Variable durch ihre Initialisierung. Falls die Initialisierung Seiteneffekte hat, können diese nach dem Refactoring mehrmals ausgeführt werden [Wlo06]. Eine solche Verhaltensänderung wird in derzeitigen Werkzeugen nicht erkannt, da diese nur statische Analysen durchführen und Seiteneffekte nur mit einer Flussanalyse erkannt werden könnten[2].

3.2. Werkzeuge zur Unterstützung beim Refactoring

Der Mensch neigt dazu Fehler zu machen und Codestellen bei einem Refactoring zu vergessen. Um die Sicherheit beim Refactoring zu erhöhen, empfiehlt es sich daher ein Werkzeug zu verwenden, das die Refactorings automatisch[3] durchführt. Ein Refactoringwerkzeug bietet die Möglichkeit die Durchführbarkeit eines Refactorings zu prüfen und kann bei möglichen semantischen Veränderungen eine Warnung ausgeben.

3.2.1. Bedingungen zur Verhaltenserhaltung

William Opdyke formulierte in seiner Dissertation für 23 gängige Refactorings Vorbedingungen, die notwendig für eine verhaltenserhaltende Transformation sind [Opd92]. Da die Analysen der

[2] Eine Ausnahme bildet das *Extract Method* Refactoring, bei dem die Flussanalyse zur Bestimmung der benötigten Parameter durchgeführt wird.

[3] Da Refactorings oft ein Verständnis des Programms voraussetzen, ist die Eingabe weiterer Parameter und ein Review der resultierenden Änderungen durch den Benutzer nötig. Aus diesem Grund können viele Refactorings nur semi-automatisch durchgeführt werden.

von Opdyke formulierten Vorbedingungen sehr aufwändig sein können, erweiterte Donald Roberts die von Opdyke vorgestellten Refactorings um Nachbedingungen, die einen Zustand nach dem Refactoring definieren und sich leichter in einem Werkzeug umsetzen lassen [Rob99].

Diese und andere Arbeiten bilden die Grundlage für die Refactoringwerkzeuge in aktuellen Entwicklungsumgebungen wie Eclipse. Die Entwicklung von Werkzeugen für reine OO-Refactorings ist mittlerweile so weit fortgeschritten, dass diese in der Praxis eingesetzt werden können. Allerdings kommen auch hier neue Konzepte wie z. B. Generics in Java 1.5 hinzu, die von den bestehenden Refactorings berücksichtigt werden müssen. Die aktuelle Forschung deckt immer noch Schwächen bei den bestehenden Ansätzen auf und zeigt, dass auch hier noch Verbesserungen möglich sind. Aktuelle Forschungsthemen sind z. B. Probleme durch Sichtbarkeitsänderungen, die das Programmverhalten beeinflussen können oder neue Ansätze zur Analyse der Verhaltenserhaltung (siehe [ST09] und [SVEdM09] für weitere Informationen).

3.2.2. Analyse der Programmstruktur

Eine Grundvoraussetzung für die Entwicklung eines Refactoringwerkzeugs ist die statische Analysierbarkeit der Programmstruktur. Ein Metamodell, das Informationen über den Zusammenhang verschiedener Codeelemente bereitstellt, erleichtert die statische Analyse. Aus diesem Grund verwendet man für die Transformationen und Analysen eines Refactorings den aus dem Compilerbau bekannten *abstrakten Syntax Baum* [Rob99]. Eine abstrakte Repräsentation des Programms ermöglicht eine komplexere Verarbeitung des Quelltextes als einfache String Manipulationen. Durch weitere Analysen wird der Baum in einen Graphen umgewandelt, indem Referenzen aufgelöst und mit ihrer Deklaration verbunden werden. Der dadurch erzeugte Graph stellt zusätzlich semantische Informationen über das Programm bereit.

Auch die Typisierung einer Sprache spielt eine Rolle bei der statischen Analysierbarkeit. Java ist eine statisch typisierte Sprache, daher können die meisten Typfehler schon beim Übersetzen festgestellt werden. Dies kann auch von Refactoringwerkzeugen genutzt werden, die bei Veränderungen in der Vererbungshierarchie überprüfen können, ob das Programm nach der Änderung typkorrekt bleibt. Frank Tip hat in einem Artikel beschrieben, wie die Typisierungsregeln von Java bei der Durchführung von Refactorings berücksichtigt werden können [TKB03].

3.2.3. Verwandte Ansätze und Grenzen von Refactoringwerkzeugen

Die Durchführung eines Refactorings wird durch den Entwickler angestoßen. Wenn eine *strukturelle Schwäche* feststellt wird, wählt der Entwickler ein passendes Refactoring aus, um diese

zu beseitigen. Es gibt andere Ansätze die weiter gehen und sogenannte *Bad Smells* erkennen und Vorschläge für Refactorings zu deren Beseitigung machen. In der Diplomarbeit von Thomas Dudziak und Jan Wloka wurde ein solches Werkzeug vorgestellt [DW02].

Ein weiteres Problem stellt die Wahrung von Schnittstellen dar. Bei einem Refactoring können Schnittstellen oder öffentliche Datenstrukturen verändert werden und dadurch die Kompatibilität beeinträchtigen. Einfache Refactoringwerkzeuge können solche Probleme nicht erkennen. Die Wahrung von Schnittstellen kann durch *API Tools* unterstützt werden. Ein API Tool berücksichtigt Metadaten über die öffentlichen Schnittstellen und prüft die Kompatibilität des aktuellen Programms gegenüber der zuletzt veröffentlichten Version. Das Plug-in Development Environment (PDE) von Eclipse stellt unter anderem ein solches API Tool zur Verfügung.

Refactoringwerkzeuge können nicht alle Teile eines Programms mit absoluter Sicherheit analysieren. In Fließtexten, reflexiven Programmstrukturen oder anderen Abschnitten, die nicht derart formalisiert sind, dass die Semantik durch eine Maschine interpretiert werden kann, können nur einfache Textsuchen verwendet werden. Die Ergebnisse müssen dann vom Benutzer auf Korrektheit und Relevanz überprüft werden.

Für weitere formalisierte Daten, wie z. B. Metadaten in XML-Dateien oder Javadoc Kommentaren können zusätzliche Compiler in das Refactoringwerkzeug integriert werden. Refactoringwerkzeuge sind daher nicht nur auf die Umstrukturierung des Quelltextes beschränkt.

3.2.4. Zusammenfassung

Refactoringwerkzeuge haben die Aufgabe sicherzustellen, dass das beobachtbare Verhalten des Programms nicht verändert wird. Dazu prüfen sie eine Reihe von Vorbedingungen und suchen alle relevanten Stellen im Code die verändert werden müssen. In Kapitel 4 wird der Einfluss der Spracherweiterung OT/J auf bestehende Vor- und Nachbedingungen untersucht, sowie konzeptionelle Anpassungen formuliert, die notwendig sind, um eine Verhaltenserhaltung für das Refactoring von OT/J Programmen zu garantieren.

4. Einfluss von ObjectTeams/Java auf bestehende Java Refactorings

Nachdem die aspektorientierte Sprache OT/J und Refactoring als Technik zur strukturellen Verbesserung von OO-Programmen vorgestellt wurden, können die Einflüsse der neuen Sprachkonzepte auf bestehende Refactorings untersucht werden. Dabei muss vor allem sichergestellt werden, dass die Eigenschaft der Verhaltenserhaltung eines Refactorings gewahrt wird.

4.1. Dimensionen von aspektorientiertem Refactoring

Das Refactoring von aspektorientierten Programmen lässt sich in vier Szenarien unterteilen, die Jan Wloka in einem Bericht als die vier Dimensionen des aspektorientierten Refactorings bezeichnet [Wlo06].

1. Refactoring im Basiscode:
   - Ein Refactoring am Basiscode muss die Aspektbindungen berücksichtigen und ebenfalls anpassen. Man spricht hierbei von *aspect-aware Refactoring*.
2. Refactoring im Aspektcode:
   - Das Refactoring muss dabei die neue Syntax und Semantik der Aspektsprache beachten.
3. Verschieben von Basiscode in einen Aspekt:
   - Darunter fällt das Identifizieren von querschneidendem Code und die anschließende Modularisierung in einem Aspekt. Die Suche nach solchem Code wird auch als *Aspect Mining* bezeichnet und ist die Voraussetzung für solche Refactorings.

4. Verschieben von Aspektcode in die Basisklasse:

    - Wenn ein Aspekt nur noch wenige Codestellen betrifft oder so wichtig geworden ist, dass er einen Teil der Kernfunktionalität darstellt, soll das Refactoring den Aspektcode in den Basiscode integrieren und anschließend den Aspekt entfernen.

4.2. Frühere Diplomarbeit über das Refactoring von ObjectTeams/Java Programmen

Gregor Brcan untersuchte in seiner Diplomarbeit [Brc05] den Einfluss von OT/J auf bestehende Refactorings. Hierzu traf er die Annahme, dass die Ziele eines Refactorings Teil des Basiscodes sein müssen und beschränkte damit die Untersuchungen auf die erste Dimension von aspektorientiertem Refactoring.

Die Arbeit enthält einen Satz von Regeln zur Bewahrung der syntaktischen Korrektheit, Regeln zur Erhaltung der impliziten Vererbungshierarchie, sowie eine Regel zur Erhaltung der semantischen Äquivalenz von OT/J Programmen. Im Rahmen der Arbeit wurden 18 atomare und 9 zusammengesetzte Refactorings untersucht und Bedingungen zur Verhaltenserhaltung von den Regeln abgeleitet.

Als Grundlage für die Regeln zur Verhaltenserhaltung dienten die von William Opdyke aufgestellten Regeln für die Sprache C++ [Opd92], sowie ein davon abgeleiteter Regelsatz für die Sprache Java, die in der Diplomarbeit von Shimon Rura erarbeitet wurden [Rur03]. Diese grundlegenden Betrachtung zur Verhaltenserhaltung bei reinen OO-Refactorings werden in dieser Arbeit vorausgesetzt. Es gelten daher bei jedem Refactoring die bereits bekannten Regeln, es sei denn eine Einschränkung ist ausdrücklich formuliert. Die bereits bestehenden JDT Refactorings setzen die bekannten Regeln bereits zufriedenstellend um (siehe Abs. 6.3) und werden als Basis für die praktische Arbeit verwendet und erweitert.

Die von Gregor Brcan aufgestellten Regeln und abgeleiteten Bedingungen werden in dieser Arbeit wiederverwendet und für die zweite Dimension erweitert. Die Regeln wurden auf Basis der Version 0.8 der ObjectTeams/Java Language Definition (OTJLD) erstellt und müssen daher außerdem für die aktuelle Version 1.2 überarbeitet werden. Der komplett überarbeitete Regelsatz ist im Anhang (siehe A.2) zu finden.

4.2.1. Übersicht der Regeln für ObjectTeams/Java

Die folgenden Überschriften geben einen Überblick über die Regeln aus der Diplomarbeit von Gregor Brcan [Brc05].

Sprachanforderungen von ObjectTeams/Java

- OTL1. Regeln für Teamklassen
- OTL2. Regeln für Rollenklassen
- OTL3. Regeln für Team- und Rollenverschachtelung
- OTL4. Regeln für externe Rollen
- OTL5. Regeln für Rollendateien
- OTL6. Regeln für Methoden-Bindungen
- OTL7. Regeln für Parameter Mappings
- OTL7.1. Implizite Parameter Mappings
- OTL8. Regeln für Declared Lifting
- OTL9. Regeln für Namen entlang impliziter Vererbung
- OTL10. Regeln für playedBy
- OTL11. Regeln für within

Vererbungsbeziehungen (Sub-Typ-Beziehungen und Overriding) in ObjectTeams/Java

- OTI1. Overriding von Methoden in der impliziten Vererbungshierarchie
- OTI2. Spezialisierung der Extends-Relation in impliziten Subrollen
- OTI3. Spezialisierung der playedBy-Relation
- OTI4. Unveränderliche playedBy-Relationen
- OTI5. Mehrdeutige Methodenbindungen

- OTS1. Erstellen/Entfernen von unreferenzierten Feldern, Methoden und Typen

4.3. Überarbeitung der bestehenden Regeln

In diesem Abschnitt werden Änderungen und Ergänzungen an den von Gregor Brcan aufgestellten Regeln beschrieben. Die Angabe der Paragraphen bezieht sich dabei auf die OTJLD Version 1.2 [HHM09].

4.3.1. Änderungen an bestehenden Regeln

In Regel OTL2. (d) muss ergänzt werden, dass Rollennamen keine sichtbaren Typen des umschließenden Teams verdecken dürfen.

OTL2. (d): Eine Rollenklasse darf nicht denselben Namen haben, wie eine Methode oder ein Feld ihres umschließenden Teams. Außerdem darf der Name einer Rolle keine sichtbaren Typen des umschließenden Teams verdecken (*Shadowing*) (§1.4(a)).

Der Regelsatz OTL2. muss um eine Regel ergänzt werden, die die Namen der vordefinierten Typen IConfined, Confined und ILowerable für Rollen verbietet.

OTL2. (f): Rollenklassen dürfen nicht die Namen „IConfined“, „Confined“ oder „ILowerable“ haben (§7.).

In den Regeln für Methoden-Bindungen (OTL6) fehlt eine Beschreibung von Callinmethoden.

OTL6. (g): Der Modifikator **`callin`** ist nur für Methoden von Rollen erlaubt. Eine Callinmethode darf keine Sichtbarkeit definieren (§4.2(d)).

Regel OTL10. (b) muss um eine Ausnahme bei *Base Imports* ergänzt werden.

OTL10. (b): Eine Basisklasse, die hinter dem Schlüsselwort **`playedBy`** angegeben ist, darf von keiner Rollenklasse des umschließenden Teams verdeckt werden. Das heißt, Rollenklasse und Basisklasse müssen verschiedene Namen haben (§2.1.2.(a)). Eine Ausnahme bilden Basisklassen, die mit einem Base Import importiert wurden (§2.1.2.(d)).

Die Regel OTS1. bezieht sich jetzt nicht nur auf Basis-Code sondern das ganze Programm.

Regel OTS1. muss daher berücksichtigen, dass Rollen nicht unbedingt referenziert werden müssen, um das beobachtbare Verhalten eines OT/J Programms zu beeinflussen. Durch die implizite Vererbungshierarchie kann sie das Verhalten von impliziten Subrollen verändern, ohne direkt referenziert zu werden.

Es kann zunächst so scheinen als ob eine ungebundene Rolle unreferenziert sei und dadurch das beobachtbare Verhalten des Programms nicht beeinflussen würde. Erst durch eine Analyse der Superteam-Hierarchie des umschließenden Teams kann sichergestellt werden, dass keine Referenzen auf eine Rolle existieren. Daher dürfen Rollen auch nicht in geerbten Referenzen referenziert werden (siehe Abs. 2.2.9).

Beim Erstellen von Typen und Methoden müssen außerdem Mehrdeutigkeiten durch *Shadowing* (OTL2. (d)) oder *Overloading* (OTI5.) ausgeschlossen werden.

Neu hinzugefügte Methoden dürfen in der impliziten Vererbungshierarchie weder überschrieben werden, noch andere Methoden überschreiben.

OTS1. Unreferenzierte Variablen (Felder), Methoden und Typen können im OO-Code (Basis-Code) hinzugefügt oder entfernt werden. Beim Entfernen dürfen keine Referenzen auf diese Elemente in Teamklassen, in Rollenklassen oder implizit geerbte Referenzen existieren. Rollen dürfen außerdem keine impliziten Subrollen haben, es sei denn sie haben einen leeren Rumpf. Beim Hinzufügen von Methoden und Typen dürfen außerdem keine Mehrdeutigkeiten durch *Shadowing* (OTL2. (d)) oder *Overloading* (OTI5.) produziert werden. Neu hinzugefügte Methoden dürfen keine in der impliziten Vererbungshierarchie existierende Methoden überschreiben oder von einer solchen überschrieben werden.

4.3.2. Neue Regeln

OTL12. Regeln für Guard Predicates:

(a) Der Ausdruck in einem Guard muss vom Typ `boolean` sein (§A.7.(a)).
(b) Die Bezeichner in dem Ausdruck eines Guards müssen innerhalb des *Scopes* des deklarierenden Teams oder der deklarierenden Rolle sichtbar sein (§5.4.1.(a)–(d)).
(c) Die Bezeichner in dem Ausdruck eines Base Guards dürfen Features der Basis referenzieren, jedoch keine Features der Rolle (§5.4.2.(d)).

OTL13. Regeln für Precedence:

(a) Der Ausdruck in einer Precedence Deklaration besteht aus einer Liste von Namen, die auf eine Callinbindung verweisen. Eine Precedence Deklaration befindet sich in einem Team oder einer Rolle (§4.8.(a)).
(b) In einem Team muss der qualifizierte Name einer Callinbindung angegeben werden. Innerhalb einer Rolle können Callinbindung mit ihrem unqualifiziertem Namen referenziert werden (§4.8.(b)).
(c) In einem Team können alle Callinbindungen einer Rolle mit dem Namen der Rolle referenziert werden (§4.8.(c)).
(d) Falls innerhalb eines Teams mehrere Callinbindungen eine Basismethode referenzieren und dabei den gleichen Modifier (**`before`**, **`after`** oder **`replace`**) verwenden, muss eine Precedence Ausdruck angegeben werden.

OTL14. Regeln für Rollenkapselung:

(a) Auf Features von Rollen, die das Interface `IConfined` implementieren, darf außerhalb des umschließenden Teams nicht zugegriffen werden (§7.1.).
(b) Auf Rollen, die von `Team.Confined` erben, darf es außerhalb des umschließenden Teams keine Referenzen geben (§7.2.).

Den Regeln zur Erhaltung von Vererbungsbeziehungen muss eine Regel hinzugefügt werden, die die implizite Vererbungsbeziehung deutlich beschreibt.

OTI6. Rollen können zusätzlich zu den expliziten Super- und Subklassen implizite Super- und Subrollen haben. Eine Rolle erbt von einer Rolle eines Superteams ihres umschließenden Teams, wenn sie den gleichen Namen hat. Um die Vererbungsbeziehungen zu erhalten, müssen die Namen der Super- und Subrollen konsistent gehalten werden (§1.3.1.(c)).

4.4. Neue Referenzen in ObjectTeams/Java

Um die Semantik eines Programms bei einem Refactoring zu bewahren muss die Semantik und Integrität von Referenzen erhalten werden. Eine Referenz muss nach einem Refactoring ein eindeutig identifizierbares Element referenzieren, das das gleiche Verhalten zeigt, wie vor dem Refactoring. Eine wichtige Voraussetzung für Referenzen ist die Sichtbarkeit. Vorhandene Featurezugriffe müssen nach dem Refactoring immer noch möglich sein. Da OT/J auch Zugriffe auf private Member zulässt (siehe 2.2.3 und 2.2.4) und dadurch deren Sichtbarkeitsbereich vergrößert, müssen bei vorhandenen Refactorings umfangreichere Analysen für private Methoden durchgeführt werden.

Beim Refactoring sind nicht nur Referenzen auf Member, Variablen oder Parameter relevant. Es müssen auch Typreferenzen innerhalb der Sprache berücksichtigt werden. In einem OT/J Programm befinden sich z. B. Typreferenzen in playedBy-Relationen oder an Parametern mit einem Declared Lifting (siehe 2.2.1 und 2.2.2). Auch Konstruktoren haben eine Abhängigkeit zu ihrem Typ, da sie den gleichen Namen wie ihre Klasse haben müssen. Daher werden z. B. bei dem Refactoring *Rename Type* auch die Konstruktoren und dessen Aufrufe mit umbenannt.

Als Grundlage für Referenzanalysen wird eine Liste mit durch die Sprache OT/J neu eingeführten Referenzen erstellt. Dies schafft eine bessere Trennung zwischen den von Gregor Brcan aufgestellten Bedingungen zur Verhaltenserhaltung und zusätzlichen Änderungen und Analysen von Referenzen, die von einem OT/J Refactoring durchgeführt werden müssen.

OTR1. Typreferenzen:

(a) In einer playedBy-Relation kann ein Team, eine Rolle oder eine reguläre Klasse referenziert werden (§2.1.(a)).
(b) In einem *Declared Lifting* wird nach dem Schlüsselwort **`as`** eine Rolle referenziert (§2.3.2.(a)).
(c) Eine Rolle kann innerhalb eines Precedence Ausdrucks referenziert werden (§4.8.(b)–(c)).

OTR2. Referenzen auf Felder und lokale Variablen:

(a) In der Deklaration eines *Anchored Type* verweist der Type Anchor auf einen Methoden-Parameter, eine lokale Variable oder ein sichtbares Feld (§1.2.2.(b)).
(b) Felder einer Klasse, die in einer playedBy-Relation zu einer Rolle stehen, können in einer Callout to Field Bindung referenziert werden. Dies gilt auch für Felder, die keine *public* Sichtbarkeit haben (§3.5.).
(c) Felder können in *Inferred Callouts* referenziert werden (§3.5.(h)).

OTR3. Methodenreferenzen:

(a) Methoden können in Callout- und Callinbindungen referenziert werden. Auf der linken Seite können Methoden der deklarierenden Rolle referenziert werden. Auf der rechten Seite der Bindung können Methoden der gebundenen Basisklasse referenziert werden, wobei auch Referenzen auf Methoden, die keine *public* Sichtbarkeit haben, möglich sind (§3.1.(b) & §4.1.(c)).
(b) Methoden können in *Inferred Callouts* referenziert werden (§3.1.(j)).

OTR4. Referenzen in Ausdrücken:

(a) Parameter Mappings enthalten Ausdrücke zur Konvertierung von Parametern oder Rückgabewerten. Dabei können in der Rolle sichtbare Features und die Parameter der Methode[1] referenziert werden (§3.2.(d)).
(b) Das within Konstrukt enthält einen Ausdruck, der eine Teaminstanz bereitstellt und einen Rumpf, in dem beliebige Anweisungen auftreten können (§5.2.(a)).
(c) Die Bezeichner in dem Ausdruck eines Methodenbindung-Guards können Parameter eines vorhandenen Parameter Mappings referenzieren (§5.4.1.(a)).
(d) Die Bezeichner in dem Ausdruck eines Methoden-Guards können Parameter der zugehörigen Methode referenzieren (§5.4.1.(b)).
(e) Die Bezeichner in dem Ausdruck eines regular Guards können Features der Rolle, Features des umschließenden Teams oder sichtbare statische Features eines in der Rolle sichtbaren Typs referenzieren (§5.4.1.(a)–(d)).
(f) Die Bezeichner in dem Ausdruck eines Base Guards müssen Features der Basis, Features des umschließenden Teams oder sichtbare statische Features eines in der Rolle sichtbaren Typs referenzieren (§5.4.2.(d)).

[1] Bei den Parametern handelt es sich um die Parameter, die in der Methodenbindung spezifiziert wurden. Eine Verbindung zu den eigentlichen Parametern der Rollenmethode oder Basismethode besteht nicht direkt über den Namen, sondern nur über den Typ und die Position des Parameters.

4.4.1. Die Rolle als Klient der Basis

Eine Rolle ist im Prinzip eine Art *Klient* der Basisklasse, der auf deren Features zugreift und diese dadurch referenziert.
Durch die Eigenschaft der Obliviousness (siehe Abs. 2.1.2) enthält der reine Basiscode[2] keine Referenzen auf OT Elemente. Die Liste der neu eingeführten Referenzen enthält ausschließlich Konstrukte für Rollen und Teams. Aufgrund der Abhängigkeit zwischen Rolle und Basis, müssen bei Refactorings der ersten Dimension Basisreferenzen im Aspektcode angepasst werden. Nur so kann gewährleistet werden, dass die Semantik von Joinpoints und die syntaktische Korrektheit des Aspektcodes bewahrt wird.

Im Umkehrschluss müssen bei Refactorings der zweiten Dimension keine Referenzen in der Basis angepasst werden, da eine Basis keine Abhängigkeiten zu ihren Rollen besitzt.

4.5. Untersuchung von atomaren Refactorings

Mit den überarbeiteten Regeln lässt sich der Einfluss auf bestehende Refactorings untersuchen. Hierzu werden 18 atomare Refactorings aus William Opdyke's Dissertation [Opd92] und dem Refactoring Katalog von Martin Fowler [Fow99] untersucht. Mit Hilfe dieser Refactorings lassen sich viele weitere zusammengesetzte Refactorings konstruieren, wobei dann auf die hier definierten Vorbedingungen zurückgegriffen werden kann. Die Tabellen 4.1 und 4.2 geben einen Überblick über den Einfluss der verschiedenen Regeln zur Verhaltenserhaltung. Ein Haken in der Tabelle signalisiert einen möglichen Regelverstoß durch das Refactoring. Für einen möglichen Regelverstoß ist es notwendig eine oder mehrere Vorbedingungen zu formulieren, die sicherstellen, dass das Refactoring durchgeführt werden kann, ohne dabei die betreffende Regel zu verletzten.

Im Rahmen dieser Diplomarbeit werden nicht alle der untersuchten Refactorings in aller Ausführlichkeit beschrieben. Exemplarisch werden die in der praktischen Arbeit adaptierten Refactorings im Detail beschrieben. Für die nicht adaptierten Refactorings werden Gemeinsamkeiten in den zu berücksichtigenden Regelverstößen untersucht und generelle Aussagen über nicht beeinträchtigende Regeln formuliert.

[2] Im Fall von *Stacking* oder *Layering* stellt der Basiscode gleichzeitig Aspektcode dar und kann nicht als reiner Basiscode bezeichnet werden ([HHM09], §2.7.(c)).

4.5.1. Regeln ohne Einfluss

Bei der Untersuchung der atomaren Refactorings hat sich herausgestellt, dass es Regeln gibt, die keinen Einfluss auf die untersuchten Refactorings haben. Es kann daraus nicht geschlossen werden, dass diese Regeln generell keinen Einfluss auf Refactorings haben, da es viele Refactorings gibt, die in dieser Arbeit nicht untersucht werden. Außerdem können beliebig viele neue Refactorings entwickelt werden, weshalb sich diese Arbeit auch nur auf die gängigsten atomaren Refactorings beschränkt.

Außerdem bleibt zu untersuchen, ob die nicht beeinflussten Regeln einen Einfluss auf die in Kapitel 7 untersuchten Refactorings der 3. und 4. Dimension haben.

Um die nicht beeinflussten Regeln nicht für jedes im Detail untersuchte Refactoring zu wiederholen, werden die nicht beeinflussten Regeln an dieser Stelle zusammengefasst.

- Die untersuchten Refactorings können Methodenbindungen nicht direkt verändern. Die Regeln OTL6.(a),(e)-(f) haben daher keinen Einfluss auf die untersuchten Refactorings.
- Keines der untersuchten Refactorings kann den Rückgabetyp einer Methode verändern oder neue Methodenbindungen erzeugen. Daher kann die Regel OTL7.1.(b) von keinem der untersuchten Refactorings verletzt werden und hat dadurch keinen Einfluss.
- Der Typ eines Ausdrucks kann durch die untersuchten Refactorings nicht verändert werden, daher haben die Regeln OTL11.(a) und OTL12.(a) keinen Einfluss auf die untersuchten Refactorings.

4.5.2. Regeln mit geringem Einfluss

Unter den Regeln befinden sich Regeln, die sehr speziell sind und damit nur wenige Refactorings betreffen. Um diese nicht für jedes nicht beeinflusste Refactoring zu wiederholen, werden diese im Folgenden zusammengefasst.

- Regel OTL1.(a) beeinträchtigt nur Create Type, falls eine Teamklasse erzeugt werden soll und dabei eine Superklasse angegeben wird. Die anderen Refactorings erzeugen keine neuen Vererbungsbeziehungen oder verändern den **`team`** Modifikator einer Klasse. Daher hat die Regel OTL1.(a) nur einen Einfluss auf Create Type.
- Regel OTL2.(e) kann nur beim Verschieben bzw. Erzeugen eines Typs, dem Erzeugen einer playedBy-Relation oder dem Hinzufügen eins **`static`** Modifikators verletzt werden. OTL2.(e) hat daher bis auf Move Class und Create Type keinen Einfluss auf die untersuchten Refactorings.

- Regel OTL2.(f) verbietet ungültige Namen für Rollen und kann nur von einem Rename Type oder Create Type Refactoring verletzt werden, da diese die einzigen der untersuchten Refactorings sind, die den Namen einer Rolle verändern bzw. bestimmen können. Auf die anderen untersuchten Refactorings hat diese Regel keinen Einfluss.
- Die Regeln OTL3.(a)-(b) zur Verschachtelung von Rollen und Teams können nur beim Erzeugen oder Verschieben von Typen verletzt werden. Eine andere Möglichkeit wäre die Veränderung eines **`team`** Modifikators, ein solcher kann von keinem der untersuchten Refactorings verändert werden[3]. Daher haben die Regeln OTL3.(a)-(b) nur einen Einfluss auf Create Type und Move Class.
- Regel OTL3.(c) kann nur durch die Veränderung oder Erzeugung einer Vererbungsbeziehung oder einer Umstrukturierung der Verschachtelung von Teams und Rollen verletzt werden. Die einzigen Refactorings, die dies bewirken können, sind Create Type und Move Class. Auf die anderen Refactorings hat die Regel OTL3.(c) daher keinen Einfluss.
- Precedence Ausdrücke werden von den untersuchten Refactorings nicht beeinflusst, da sie keine Methodenbindungen verändern. Eine Ausnahme bildet das Move Class Refactoring, dass beim Verschieben einer Rolle in ein anderes Team eine Anpassung oder Erzeugung eines Precedence Ausdrucks erfordern kann. Auf die anderen untersuchten Refactorings haben die Regeln für Precedence (OTL13.(a)-(d)) keinen Einfluss.
- Die untersuchten Refactorings implementieren weder neue Interfaces, noch erzeugen sie Extends-Relationen. Bis auf Move Class, Move Field und Move Method werden auch keine neuen Referenzen oder Featurezugriffe durch die untersuchten Refactorings erzeugt. Daher haben die Regeln für Rollenkapselung (OTL14.(a)-(b)) auf die übrigen Refactorings keinen Einfluss. Für die erwähnten Move Refactorings bleibt zu untersuchen, ob sich Beispiele konstruieren lassen, in denen ein nicht erlaubter Featurezugriff erzeugt werden könnte.

4.5.3. Rename Type

Das Rename Type Refactoring benennt einen Typ um. In OT/J kann es sich dabei auch um Teams oder Rollen handeln, wobei zusätzliche Regeln beachtet werden müssen.

[3] Bei einem Move Class Refactoring bleibt zu diskutieren ob dieses beim Verschieben einer Rolle in eine Rolle, die noch keinen **`team`** Modifikator besitzt, einen solchen erzeugen sollte. Die Alternative ist, es das Refactoring in einem solchen Fall zu verbieten und zu fordern, dass die Rolle in einem separaten Schritt mit einem **`team`** Modifikator ergänzt werden müsste.

Parameter

- Der umzubenennenden Typ T.
- Der neue unqualifizierte Name N.

Vorbedingungen

1. Der neue Name N darf in keinem Team, in dem T sichtbar ist, von einer Rolle überschattet werden (OTL2.(d)).
2. Wenn T eine Rolle ist, darf der neue Name N keinen sichtbaren Typ des umschließenden Teams überschatten (OTL2.(d)).
3. Wenn T ein Team ist, das einen Rollenordner besitzt, darf es kein Package in dem Package des Teams geben, das den Namen N trägt (OTL5.(b)).
4. Wenn T als Basis verwendet wird und nicht mit einem Base Import importiert wurde, darf der neue Name N nicht von einer Rolle des umschließenden Teams der an T gebundenen Rolle verdeckt werden (OTL10.(b)).
5. Wenn T eine Rolle ist, darf N nicht den Namen eines im umschließenden Team als Basis verwendeten Typs verdecken, der nicht mit einem Base Import importiert wurde (OTL10.(b)).
6. Wenn N der neue Name für eine Rolle ist, darf dieser nicht „IConfined“, „Confined“ oder „ILowerable“ lauten (OTL2.(f)).
7. Damit kein Overriding entsteht, darf es beim Umbenennen einer Rolle keine Rolle mit dem Namen N in einem Super- oder Subteam des umschließenden Teams geben (OTI6.).
8. Um die Vererbungsbeziehung zu erhalten müssen auch implizit verwandte Rollen umbenannt werden. Als implizit verwandte Rollen werden alle von T überschriebenen Rollen und Rollen die T überschreiben bezeichnet. Daher müssen die Vorbedingungen auch für alle implizit verwandten Rollen gelten. Dies schließt auch die hier nicht erwähnten Vorbedingungen für reine Java Programme mit ein (OTI6.).

Änderungen im Programm

1. Beim Umbenennen einer Rolle oder eines Teams müssen auch vorhandene Referenzen in der Deklarationen eines Anchored Types umbenannt werden (OTL4.(b)).
2. Wenn ein Team umbenannt werden soll, das einen Rollenordner besitzt, muss dieser den neuen Namen N bekommen (OTL5.(a)).
3. Wenn eine umzubenennende Rolle in einer Rollendatei gespeichert ist, muss der Name der Datei angepasst werden (OTL5.(b)).
4. Wenn ein umzubenennendes Team Rollen in Rollendateien besitzt, muss der Name des Teams in vorhandenen Teampackage-Deklarationen umbenannt werden (OTL5.(c)).
5. Es müssen alle von OT/J neu eingeführten Typreferenzen umbenannt werden (OTR1.(a)-(c)), sowie vorhandene Referenzen auf den Konstruktor in OT/J Ausdrücken (OTR4.(a)-(b),(e) und (f)).
6. Um die impliziten Vererbungsbeziehungen beim Umbenennen einer Rolle zu erhalten, müssen alle implizit verwandten Rollen den neuen Namen N bekommen (OTI6.).
7. Die zuvor aufgezählten Änderungen im Programm müssen für alle implizit verwandten Rollen durchgeführt werden, dies beinhaltet auch Referenzen auf implizit verwandte Phantomrollen (OTI6.).

Regeln ohne Einfluss

- Da ein Rename Type Refactoring die Vererbungshierarchie nicht verändert sind folgende Regeln nicht zu beachten: OTL2.(b)-(c), OTL3.(a),(c), OTL9., OTI1., OTI2., OTI3. und OTI4.
- Das Rename Type Refactoring hat keinen Einfluss auf Methodenbindungen oder Methoden, daher müssen die Regeln OTL6.(a)-(g), OTI1., OTI5. und OTS1. nicht beachtet werden.
- Parameter Mappings werden nicht von Rename Type beeinträchtigt. Daher haben die Regeln OTI7.(a)-(b) und OTI7.1.(a)-(b) keinen Einfluss.
- Rename Type hat keinen Einfluss auf die Sichtbarkeit von Typen oder Features, daher können die Regeln OTL2.(a),(b), OTL4.(a), OTI7.(c) und OTL12.(b) nicht verletzt werden.

- Ein Rename Type Refactoring verändert oder erzeugt keine playedBy-Relationen, dadurch haben die Regeln OTL10.(a) und (c) keinen Einfluss.

4.5.4. Pull Up Method

Das Pull Up Method Refactoring verschiebt eine Methode in eine Superklasse der deklarierenden Klasse. In OT/J kann als Ziel auch eine implizite Superklasse angegeben werden. Durch ein Pull Up Method wird in der Regel der Sichtbarkeitsbereich einer Methode vergrößert[4], da sie nach dem Refactoring auch in der Zielklasse vorhanden ist und möglicherweise an weitere Subklassen der Zielklasse vererbt wird.

Parameter

- Eine Methode M.
- Der Zieltyp T.

Vorbedingungen

1. Wenn M durch das Refactoring an Rollen vererbt wird, in denen M noch nicht bekannt war, darf M nicht den gleichen Namen wie eine Methode oder ein Feld des umschließenden Teams haben. Dies muss in allen Subklassen von T überprüft werden (OTL2.(d)).
2. Falls der Typ, der M deklariert als Basis verwendet wird und M eine private Methode ist, die auf der rechten Seite einer Callin- oder Calloutbindung referenziert wird, kann das Refactoring nicht durchgeführt werden, es sei denn die Sichtbarkeit wird auf **`protected`**[5] erweitert (OTL6.(b)-(c)).
3. Falls M eine private Rollenmethode ist und T eine explizite Superklasse der Rolle ist, darf M nicht verschoben werden, wenn sie auf der linken Seite einer Callout- oder Callinbindung referenziert wird, es sei denn die Sichtbarkeit wird auf **`protected`** erweitert (OTL6.(b)-(c)).

[4] Eine Ausnahme bilden private Methoden, die innerhalb der expliziten Vererbungshierarchie nicht in Subklassen sichtbar sind oder Methoden, die eine *protected* oder *default* Sichtbarkeit besitzen und durch das Pull Up in ein anderes Package verschoben werden.

[5] Wenn sich der Zieltyp T im gleichen Package wie der deklarierende Typ befindet, würde eine Ausweitung der Sichtbarkeit auf *default* ausreichen. Mit einer Anpassung auf die Sichtbarkeit *protected* können zusätzliche Überprüfung eingespart werden.

4. Falls M eine Callinmethode ist, d. h. den Modifikator **`callin`** besitzt, muss das Ziel eine Rolle sein (OTL6.(g)).

5. Wenn ein Declared Lifting für einen Parameter von M angegeben wurde, muss die angegebene Rolle in T enthalten sein und bereits an die angegebene Basisklasse gebunden sein (OTL8.).

6. M darf in impliziten Subklassen von T keine zuvor sichtbaren Namen verdecken oder durch andere Namen verdeckt werden (OTL9.).

7. M darf im Ziel keine mehrdeutige Methodenbindung erzeugen. Mehrdeutigkeiten können entstehen, wenn das Ziel eine Basisklasse, eine Superklasse einer Basisklasse oder eine Rolle ist. Da die Methode eventuell an weitere Klassen vererbt wird, muss auch in Subklassen von T geprüft werden ob Mehrdeutigkeiten entstehen können (OTI5.).

8. Wenn M nach dem Refactoring von Methoden der impliziten Subklassen von T überschrieben wird, darf das Verhalten des Programms nicht dadurch verändert werden, dass diese in einem Tsuper-Aufruf die Methode M referenzieren. Auch einfache Methodenreferenzen, die sich vor dem Refactoring auf eine von M überschriebene Methode bezogen haben dürfen nach dem Refactoring nicht M referenzieren (OTI6.). Abbildung 4.1 zeigt, wie durch Overriding das Verhalten eines Programms verändert werden kann. Nach einem Pull Up Method von `T4.R.m` nach `T1.R` würde sowohl der Tsuper-Aufruf in `T2.R.m`, als auch der Aufruf von `m` in `T3.R.n` auf die verschobene Methode `T4.R.m` zeigen. Wenn `T4.R.m` nicht das gleiche Verhalten wie die zuvor referenzierte Methode `T0.R.m` implementiert, kann es zu Verhaltensänderungen kommen. Die Veränderung des Tsuper-Aufrufs zeigt, dass auch bei einer Redefinition der verschobenen Methode Verhaltensänderungen auftreten können.

9. T muss eine explizite oder implizite Superklasse des Typs sein, in dem M deklariert ist (OTI6.).

10. M muss nach dem Refactoring weiterhin sichtbar in Typen sein, die M referenzieren. Eine Verringerung der Sichtbarkeit kann bei einem Pull Up einer privaten Methode auftreten und durch eine Erhöhung der Sichtbarkeit auf **`protected`** gelöst werden. Dabei müssen auch die von OT/J neu eingeführten Referenzen untersucht werden (OTL7.(c), OTL12.(b)-(c), OTR3.(a)-(b) und OTR4.(a)-(f)).

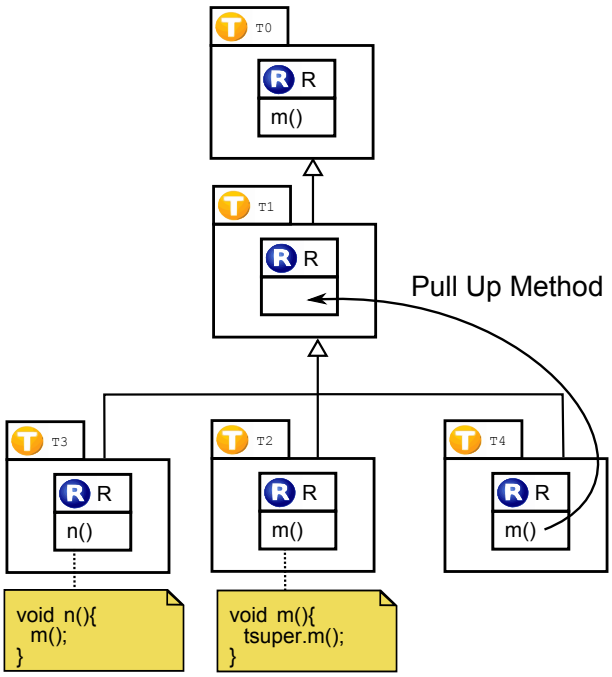

Abbildung 4.1.: Verhaltensänderung durch Pull Up Method

Änderungen im Programm

- Um das Refactoring durchzuführen sind in bestimmten Fällen Sichtbarkeitsänderungen bei M nötig, die bereits in den Vorbedingungen erwähnt wurden.

Regeln ohne Einfluss

- Ein Pull Up Method Refactoring kann keine Vererbungsbeziehungen verändern, daher haben die Regeln OTL2.(b)-(c), OTI1., OTI2., OTI3. und OTI4 keinen Einfluss.
- Parameter Mappings werden nicht von Pull Up Method Refactorings beeinträchtigt, daher haben die Regeln OTL7.(a)-(b) und OTL7.1.(a)-(b) keinen Einfluss.
- Pull Up Method hat keinen Einfluss auf die Sichtbarkeit von Typen, daher können die Regeln OTL2.(a),(b) und OTL4.(a) nicht verletzt werden.
- Pull Up Method verändert keine Rollen/Team Referenzen oder Deklarationen. Aus diesem Grund können Deklarationen mit einem Anchored Type nicht durch ein Pull Up Method Refactoring beeinflusst werden (OTL4.(b)).
- Pull Up Method verändert nicht die äußere Struktur von Typen (weder Rollen noch Teams) und muss dadurch keine Regeln für Rollendateien beachten (OTL5.(a)-(c)).

- Eine Callout to Field-Bindung kann durch ein Pull Up Method Refactoring nicht beeinträchtigt werden, da selbst beim Verschieben der gebundenen Methode die Sichtbarkeit der Methode durch Vererbung erhalten bleibt. Die Regel für Callout to Field-Bindungen (OTL6.(d)) hat daher keinen Einfluss.
- Durch ein Pull Up Method Refactoring werden keine Typen (daher auch keine Rollen oder Basisklassen) verändert oder verschoben. Auch playedBy-Relationen können durch ein Pull Up Method Refactoring nicht erzeugt oder verändert werden. Deshalb haben die Regeln für **`playedBy`** (OTL10.(a)-(c)) keinen Einfluss.

4.5.5. Push Down Method

Das Push Down Method Refactoring verschiebt eine in Methode in eine oder mehrere Subklassen der deklarierenden Klasse. Durch die neuen Vererbungsbeziehungen durch OT/J können als Ziel auch implizite Subklassen gewählt werden. Durch ein Push Down Method Refactoring wird der Sichtbarkeitsbereich der verschobenen Methode verkleinert und es muss daher vor allem darauf geachtet werden, dass alle Referenzen auf die verschobene Methode nach dem Refactoring weiterhin gültig sind.

Parameter

- Eine Methode M.
- Der Zieltyp T.

Vorbedingungen

1. Da die Methode M nach einem Push Down Refactoring in der deklarierenden Klasse nicht mehr sichtbar ist, darf M nicht auf der linken Seite einer Methodenbindung referenziert werden. Dies beinhaltet sowohl Callout- und Callinbindungen, als auch Callout to Field-Bindungen (OTL6.(b)-(d)).
2. Falls M auf der rechten Seite einer Methodenbindung auftaucht, muss die Basisklasse der Rolle, die die Methodenbindung beinhaltet, der Zieltyp T oder eine explizite oder implizite Subklasse von T sein (OTL6.(b)-(c)). Nur dadurch ist gewährleistet, dass M nach dem Refactoring weiterhin als Basismethode referenziert werden kann.

3. Wenn M in dem Ausdruck eines Parameter Mappings referenziert wird, muss M nach dem Refactoring weiterhin im Sichtbarkeitsbereich der Rolle sichtbar sein, die das Parameter Mapping beinhaltet (OTL7.(c)).

4. Wenn der Zieltyp T eine Rolle ist und implizite Super- oder Subklassen hat, darf M keine sichtbaren Namen der impliziten Super- oder Subklassen verstecken bzw. verdecken (OTL9.). Das gleiche gilt für Methoden, Felder oder sichtbare Typen des umschließenden Teams oder dessen Super- und Subteams (OTL2.(d)). Dies kann dadurch verursacht werden, dass eine private Methode entlang der expliziten Vererbungshierarchie verschoben wird, die vor dem Refactoring in T nicht sichtbar war.

5. Wenn M in einem regulären Guard referenziert wird, muss M weiterhin im Sichtbarkeitsbereich der deklarierenden Rolle oder des deklarierenden Teams sichtbar sein (OTL12.(b)).

6. Wenn M in einem Base Guard referenziert wird, muss die betreffende Basis entweder T oder eine explizite bzw. implizite Subklasse von T sein (OTL12.(c)).

7. In der Regel ist M bereits in den Subklassen sichtbar und kann nach einem Push Down Method Refactoring kein neues Overloading verursachen. Wenn M jedoch privat ist und entlang einer expliziten Vererbungsbeziehung verschoben wird, muss sichergestellt werden, das kein Overloading entsteht, das zu mehrdeutige Methodenbindungen führt (OTI5.).

8. T muss eine explizite oder implizite Superklasse des Typs sein, in dem M deklariert ist (OTI6.).

9. Da OT/J Mehrfacherben ermöglicht kann der Zieltyp T mehrere direkte Superklassen haben (OTI6.). Da implizit geerbte Methoden Vorrang vor explizit geerbten Methoden haben, kann es durch ein Push Down Method dazu kommen, dass eine implizit geerbte Methode von M überschrieben wird und dadurch das Verhalten des Programms verändern kann. Listing 4.1 zeigt ein Beispiel für ein verhaltensänderndes Overriding. In Zeile 13 wird die Methode `T2.R.m` referenziert, da die implizite Vererbung stärker bindet als die explizite Vererbung ([HHM09], §1.5.(e)). Nach einem Push Down Method von `A.m` nach `T1.R` würde in Zeile 13 die verschobene Methode `A.m` referenziert werden, was zu einer Verhaltensänderung führt, wenn `A.m` und `T2.R.m` kein identisches Verhalten implementieren. Ein Push Down Method von `T2.R.m` nach `T1.R` würde hingegen in keinem Fall das Verhalten des Programms verändern.

10. Der Zieltyp T darf keine Phantomrolle (siehe Abs. 2.2.8) sein. Das heißt, T muss im Quellcode existieren.

Eine Lösungsmöglichkeit wäre es die Phantomrolle während des Refactorings zu materialisieren und damit die beschriebene Situation nicht zu verbieten. Da eine leere Rolle semantisch die gleiche Bedeutung wie eine Phantomrolle hat, wäre eine solche Anpassung verhaltenserhaltend. Es bleibt zu diskutieren, ob ein solcher Schritt von einem Refactoring durchgeführt werden sollte oder in Form eines *Code Completion* Mechanismus realisiert werden könnte, der ähnlich wie die Unterstützung für das Überschreiben von Methoden funktioniert.

```
1  public class A {
2    public void m(){}
3  }
4
5  public team class T2 {
6    protected class R {
7      public void m(){}
8  }}
9
10 public team class T1 extends T2 {
11   protected class R extends A {
12     public void n(){
13       m();
14 }}}
```

Listing 4.1: Overriding durch Push Down Method

Änderungen im Programm

Bei einem Push Down Method Refactoring sind keine zusätzlichen Änderungen nötig.

Regeln ohne Einfluss

- Ein Push Down Method Refactoring kann keine Vererbungsbeziehungen verändern, daher haben die Regeln OTL2.(b),(c), OTI1., OTI2., OTI3. und OTI4 keinen Einfluss.
- Parameter Mappings werden nicht von Push Down Method Refactorings beeinträchtigt, da die Signatur einer Methode nicht verändert werden kann. Die Regeln OTL7.(a)-(b) und OTL7.1.(a)-(b) keinen Einfluss.
- Ein Declared Lifting wird durch ein Push Down Method Refactoring nicht beeinflusst, da die referenzierten Rollen und Basisklassen in Subklassen weiterhin sichtbar bleiben (OTL8.).
- Push Down Method hat keinen Einfluss auf die Sichtbarkeit von Typen, daher können die Regeln OTL2.(a),(b) und OTL4.(a) nicht verletzt werden.

- Push Down Method verändert keine Rollen/Team Referenzen oder Deklarationen. Aus diesem Grund können Deklarationen mit einem Anchored Type nicht durch ein Push Down Method Refactoring beeinflusst werden (OTL4.(b)).
- Push Down Method verändert nicht die äußere Struktur von Typen (weder Rollen noch Teams) und muss dadurch keine Regeln für Rollendateien beachten (OTL5.(a)-(c)).
- Durch ein Push Down Refactoring werden keine Typen (daher auch keine Rollen oder Basisklassen) verändert oder verschoben. Auch playedBy-Relationen können durch ein Push Down Method Refactoring nicht erzeugt oder verändert werden. Deshalb haben die Regeln für **`playedBy`** (OTL10.(a)-(c)) keinen Einfluss.

4.5.6. Pull Up/Push Down Field

Die Untersuchung der Pull Up Field und Push Down Field Refactorings wird nicht so ausführlich beschrieben wie für die zuvor vorgestellten Refactorings. Der Grund dafür ist, dass die grundlegenden Mechanismen dieser Refactorings sehr ähnlich zu den analogen Varianten für Methoden sind (siehe Abs. 4.5.4 und 4.5.5).

Analogien

- Für Felder gelten ähnliche Regeln für Namenskonflikte wie für Methoden (OTL2.(d) und OTL9.).
- Auch Felder können in Ausdrücken referenziert werden und müssen nach dem Refactoring sichtbar sein, wenn sie referenziert werden (OTL7(c), OTL12.(b)-(c), OTR2.(a)-(c), OTR4.(a)-(f)).
- Beim Verschieben von Feldern steht in OT/J auch die implizite Vererbungshierarchie zur Verfügung (OTI6.).

Unterschiede

- Im Gegensatz zu Methoden können Felder in der Deklaration mit einem Anchored Type verwendet werden und es muss daher sichergestellt werden, dass nach dem Refactoring der Type Anchor weiterhin in der deklarierenden Klasse verfügbar ist (OTL4.(b)).

- Felder spielen in reinen Methodenbindungen keine Rolle (OTL6.(b)-(c)). Dafür werden sie jedoch in Callout to Field-Bindungen referenziert und müssen nach dem Refactoring in der entsprechenden Basisklasse zur Verfügung stehen (OTL6.(d)).
- Durch das Verschieben von Feldern kann kein Overloading oder Overriding entstehen und dadurch auch keine mehrdeutigen Methodenbindungen (OTI5.).

4.6. Überblick der Einflüsse auf die restlichen Refactorings

Es wurden bereits einige Aussagen über Regeln ohne Einfluss und Regeln mit geringem Einfluss (siehe Abs. 4.5.1 und 4.5.2) getroffen. In diesem Abschnitt wird der Einfluss der Regeln auf die nicht im Detail untersuchten Refactorings abgeschätzt um eine Grundlage für weitere Untersuchungen zu schaffen.

4.6.1. Delete Refactorings

Die Refactorings Delete Type, Field und Method setzen voraus, dass das zu löschende Element unreferenziert ist. Daher ist der Einfluss der neuen Regeln eher gering. Hauptsächlich müssen bei der Referenzanalyse die neuen Referenzen in OT/J beachtet werden (OTR1. - OTR4.).

Bei einem Delete Type Refactoring muss zusätzlich die impliziten Vererbungsbeziehungen von Rollen berücksichtigt werden (OTI6.). Da für implizite Vererbungsbeziehungen keine expliziten Referenzen existieren wie bei herkömmlichen Extends-Beziehungen, empfiehlt es sich für die Analyse nicht den reinen Quelltext zu verwenden, sondern eine angereicherte Repräsentation, wie das *Java Model*, das nach dem Auflösen der Bindungen auch Informationen über alle impliziten Super- und Subklassen bereit stellt.

4.6.2. Create Refactorings

Die Refactorings Create Type, Field und Method müssen hauptsächlich Regeln bei der Erzeugung von OT-Elementen beachten (Regeln für Rollen, Teams, Callinmethoden, Team/Rollen-Verschachtelung).
Damit keine Verhaltensänderung durch Overriding auftritt, muss die neue implizite Vererbungshierarchie berücksichtigt werden. Dies gilt besonders bei der Erzeugung von Rollen, die allein

durch ihren Namen eine implizite Vererbungsbeziehung erzeugen und dadurch das Verhalten eines OT/J Programms beeinflussen können (OTI6.).

Die Erzeugung von neuen Elementen hat außerdem einen Einfluss auf Regeln, die das *Naming* betreffen. Die neuen Elemente werden in einem bestimmten Bereich des Programms sichtbar und dürfen die Regeln bezüglich *Shadowing* und *Hiding* (OTL2.(d) und OTL9.) nicht verletzten. Shadowing kann nicht nur bei der Erzeugung eines Teams oder einer Rolle auftreten, sondern betrifft auch herkömmliche Java Typen, da diese potentiell in einem Team überschattet werden können.

Bei der Erzeugung von Methoden können durch die neue Sichtbarkeit des Methodennamens Mehrdeutigkeiten in Methodenbindungen auftreten (OTI5.). Dies gilt sowohl für Rollenmethoden, als auch für reguläre Java Methoden.

4.6.3. Signaturverändernde Refactorings

Die Refactorings Add/Remove Parameter führen Veränderungen in der Signatur einer Methode durch. Die Veränderung einer Methodensignatur kann zu Overriding führen. Um Verhaltensänderungen durch Overriding auszuschließen muss auch die implizite Vererbungshierarchie untersucht werden und die darin enthaltenen verwandten Methoden[6] (OTI1. und OTI6.).

Rollenmethoden, die mit einer Methodenbindung an die zu verändernde Methode gebunden sind, müssen bei diesen Refactorings beachtet werden. So kann es sein, dass ein Parameter, der in einer Basismethode nicht genutzt wird, in einer Rollenmethode verwendet und dadurch nicht entfernt werden kann. Es gilt auch hier die durch OT/J neu eingeführten Referenzen nach Parameterreferenzen zu durchsuchen (OTR2. und OTR4.)

Die wichtigsten Regeln für Signaturänderungen sind die Regeln für Parameter Mappings (OTL7. und OTL7.1.). Beim Entfernen von nicht verwendeten Parametern muss daher bei einem vorhandenen Parameter Mapping untersucht werden, ob der angegebene Ausdruck seiteneffektfrei ist und entfernt werden kann ohne eine Verhaltensänderung nach sich zu ziehen.
Beim Hinzufügen eines Parameters müssen in vorhandenen Parameter Mappings default Mappings erzeugt werden, um die Vollständigkeit eines Mappings zu bewahren.

[6] Verwandte Methoden bezeichnen Methoden, die eine Methode überschreiben oder von ihr überschrieben werden. Verwandte Methoden schließen auch Methodendeklarationen der betreffenden Methode in Interfaces oder Abstrakten Klassen mit ein.

```
1  public class Base {
2    public void m(int i, int j){}
3  }
4
5  public team class Team {
6    protected class Role playedBy Base{
7      callin void f(int x){}
8      f <- replace n;
9    }
10 }
```

Listing 4.2: Beispiel für eine unvollständige Signatur einer Rollenmethode

Da eine Rollenmethode nicht die komplette Signatur der Basismethode besitzen muss, muss bei Veränderungen der Signatur genau auf die Reihenfolge der Parameter geachtet werden. Listing 4.2 zeigt ein Beispiel für eine Rollenmethode, die nur ein Teil der Signatur der Basismethode besitzt. Wenn der Parameter `i` der Methode `Base.m` entfernt werden sollte, müsste der Parameter `x` der Methode `Team.Role.f` ebenfalls entfernt werden. Das Entfernen des Parameters `j` hingegen dürfte keine Veränderung in der Signatur von `Team.Role.f` nach sich ziehen.
Die Reihenfolge der Parameter muss auch beim Hinzufügen von Parametern zwischen Rollen- und Basismethoden synchron gehalten werden.

4.6.4. Move Refactorings

Die Refactorings Move Class, Move Field und Move Method verschieben Codeelemente innerhalb des Programms.

Für Move Field und Move Method wurden bereits viele Einflüsse untersucht, da Pull Up/Push Down Field/Method Spezialfälle der Move Refactorings darstellen. Das heißt, dass immer wenn eine Regel durch einen Spezialfall verletzt werden kann, kann diese auch durch das allgemeine Move Refactoring verletzt werden. Diese Implikation lässt sich in den Tabellen 4.1 und 4.2 ablesen.

Ein weiterer Teil der Bedingungen eines Move Refactorings lässt sich aus den Create Refactorings ablesen. Es können zwar nicht alle Regeln verletzt werden, da das zu bewegende Element bereits syntaktisch korrekt ist, die Probleme bezüglich Naming, Shadowing, Hiding oder Mehrdeutigkeiten sind jedoch die gleichen (OTL2.(d), OTL9. und OTI5.).

Um die Vererbungsbeziehungen des Programms zu erhalten muss außerdem die implizite Vererbungshierarchie beachtet werden um Beispielsweise Overriding zu verhindern (OTI6.). Ein weiterer Punkt, der bei einem Move Refactoring beachtet werden muss ist die Korrektheit vorhandener Referenzen. Um die Semantik des Programms nicht zu verändern, müssen nach dem Refactoring alle Referenzen auf das verschobene Element und dessen Features möglich

sein. Hierbei müssen auch die von OT/J neu eingeführten Referenzen untersucht werden (OTR1.-OTR4.).

Die Move Refactorings sind die einzigen hier untersuchten Refactorings, die möglicherweise gegen Regeln der Rollenkapselung verstoßen könnten. Im Rahmen dieser Arbeit konnten jedoch keine Beispiele gefunden werden in denen dies nachzuweisen wäre. OT/J stellt im Prinzip die zwei folgenden Konzepte für Rollenkapselung zur Verfügung.

1. Die zu schützende Rolle erbt von der vorgegebenen Klasse `Confined` und wird als **`protected`** deklariert. Dadurch ist sichergestellt, dass es außerhalb des Teams keine Referenzen auf die Rolle gibt.

2. Die zu schützende Rolle implementiert das vorgegebene Interface `IConfined` und kann außerhalb ihres Teams über diesen Typ referenziert werden, Featurezugriffe sind jedoch nicht möglich.

Um ein referenziertes Feature in eine andere Klasse zu verschieben, muss der deklarierende Typ eine Referenz auf eine Instanz des Ziels besitzen. Nur dann ist es möglich vorhandene Referenzen anzupassen. Für das 1. Konzept der Rollenkapselung stehen keine Referenzen außerhalb des Teams zur Verfügung, die als Ziel dienen könnten. Das 2. Konzept der Rollenkapselung lässt zwar Referenzen vom Typ `IConfined` zu, allerdings ist eine solche Referenz ein ungültiges Ziel, da durch die Sprache vorgegebene Interfaces nicht verändert werden können.

Das Move Class Refactoring kann beim Verschieben einer Rolle verschiedene Regeln verletzten, die das Verhalten des Aspektcodes erheblich beeinflussen können. Wenn eine Rolle in ein anderes Team verschoben wird, kann es zu Konflikten zwischen Methodenbindungen kommen, die eine Precedence Deklaration erfordern (OTL12.(d)). Auch die Teamaktivierung durch **`within`**-Statements oder Guards kann beim Verschieben einer Rolle beeinträchtigt werden (OTL11. und OTL12.). Hier wären genauere Untersuchungen nötig um herauszufinden in wie weit bei einem solchen Refactoring eine Verhaltenserhaltung gewährleistet werden kann.

Es ist zu vermuten, dass dies kaum möglich ist. Der initiale Zustand der Teamaktivierung könnte leicht aus der plugin.xml Datei gelesen werden. Eine Flussanalyse, die bestimmt ob die Teamaktivierung der betreffenden Teams äquivalent ist, wäre hingegen sehr aufwändig und müsste große Teile des Programms analysieren. Eine weitere Form der Teamaktivierung stellen Guard Predicates dar, deren Wahrheitswert statisch schwer zu analysieren ist.

4.6.5. Rename Method und Rename Field

Rename Method wurde bereits umfassend adaptiert und ausführlich in der Diplomarbeit von Gregor Brcan [Brc05] besprochen.

Rename Field erfordert wenig neue Vorbedingungen. Es müssen die Regeln für Naming eingehalten werden (OTL2.(d) und OTL9.). Außerdem müssen alle durch OT/J neu eingeführten Referenzen auf Felder mit umbenannt werden (OTR2. und OTR4.).

	Create Type	Create Field	Create Method	Delete Type	Delete Field	Delete Method	Rename Type	Rename Field	Rename Method	Add Parameter	Remove Parameter	Move Class	Move Field	Pull Up Field	Push Down Field	Move Method	Pull Up Method	Push Down Method
OTL1																		
(a)	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OTL2																		
(a)	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(b)	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(c)	✓	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
(d)	✓	✓	✓	-	-	-	✓	✓	✓	-	-	✓	✓	✓	-	✓	✓	✓
(e)	✓	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
(f)	✓	-	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-
OTL3																		
(a)	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(b)	✓	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
(c)	✓	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
OTL4																		
(a)	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
(b)	-	-	-	-	-	-	✓	-	-	-	-	✓	✓	✓	✓	-	-	-
OTL5																		
(a)	✓	-	-	-	-	-	✓	-	-	-	-	✓	-	-	-	-	-	-
(b)	✓	-	-	-	-	-	✓	-	-	-	-	-	-	-	-	-	-	-
(c)	✓	-	-	-	-	-	✓	-	-	-	-	✓	-	-	-	-	-	-
OTL6																		
(a)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-	✓	✓	✓
(c)	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-	✓	✓	✓
(d)	-	-	-	-	-	-	-	✓	✓	-	-	-	✓	-	✓	✓	-	✓
(e)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(f)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(g)	-	-	✓	-	-	-	-	-	-	-	-	✓	-	-	-	✓	✓	-
OTL7																		
(a)	-	-	-	-	-	-	-	-	-	✓	✓	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-	✓	✓	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-	-	✓	-	✓	-	-	✓	✓	✓	✓

Tabelle 4.1.: Übersicht der Regeleinflüsse auf atomare Refactorings (1/2)

	Create Type	Create Field	Create Method	Delete Type	Delete Field	Delete Method	Rename Type	Rename Field	Rename Method	Add Parameter	Remove Parameter	Move Class	Move Field	Pull Up Field	Push Down Field	Move Method	Pull Up Method	Push Down Method
OTL7.1																		
(a)	-	-	-	-	-	-	-	-	-	✓	✓	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OTL8	-	-	-	-	-	-	-	-	-	✓	-	✓	-	-	-	✓	✓	-
OTL9	-	✓	✓	-	-	-	-	✓	✓	-	-	-	✓	✓	✓	✓	✓	✓
OTL10																		
(a)	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
(b)	✓	-	-	-	-	-	✓	-	-	-	-	✓	-	-	-	-	-	-
(c)	✓	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
OTL11																		
(a)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OTL12																		
(a)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-	-	-	✓	✓	-	✓	✓	✓	✓
(c)	-	-	-	-	-	-	-	-	-	-	-	✓	✓	-	✓	✓	✓	✓
OTL13																		
(a)	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
(d)	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	-	-	-
OTL14																		
(a)	-	-	-	-	-	-	-	-	-	-	-	✓	✓	-	-	✓	-	-
(b)	-	-	-	-	-	-	-	-	-	-	-	✓	-	-	-	✓	-	-
OTI1	-	-	-	-	-	-	-	✓	✓	✓	✓	-	-	-	-	-	-	-
OTI2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OTI3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OTI4	✓	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
OTI5	-	-	✓	-	-	-	-	-	✓	-	-	-	-	-	-	✓	✓	✓
OTI6	-	-	-	✓	-	-	✓	-	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
OTS1	✓	✓	✓	✓	✓	✓	-	-	-	-	-	-	-	-	-	-	-	-

Tabelle 4.2.: Übersicht der Regeleinflüsse auf atomare Refactorings (2/2)

4.7. Weitere Refactorings

Die in Abschnitt 4.5 vorgestellten atomaren Refactorings bilden bereits eine gute Grundlage für die Erstellung komplexerer Refactorings und geben einen ersten Überblick über die Einflüsse der OTJLD.

Es gibt Refactorings, die sich nicht allein aus den zuvor vorgestellten Refactorings zusammenstellen lassen. Dabei handelt es sich unter anderem um das Extract Method und Inline Method Refactoring, die direkter in den Ablauf des Programms eingreifen und dadurch eine Flussanalyse durchführen müssen. Diese Refactorings sind für diese Arbeit besonders interessant, da sie die Grundlage für einen Teil der Refactorings der 3. und 4. Dimension bilden. Aus diesem Grund werden diese beiden Refactorings kurz erläutert und der Einfluss der Regeln zur Verhaltenserhaltung abgeschätzt.

Eine weitere Gruppe von Refactorings, die sich nicht direkt aus den atomaren Refactorings ableiten lassen, sind erste OT/J spezifische Refactorings, die neue Möglichkeiten der Strukturierung eines OT/J Programms nutzen. Hierzu werden in diesem Abschnitt die Refactorings Pull Up Role/Methodbinding, Push Down Role/Methodbinding und Inline/Extract Rolefile vorgestellt.

4.7.1. Beispiel für den Einsatz von OT/J Refactorings

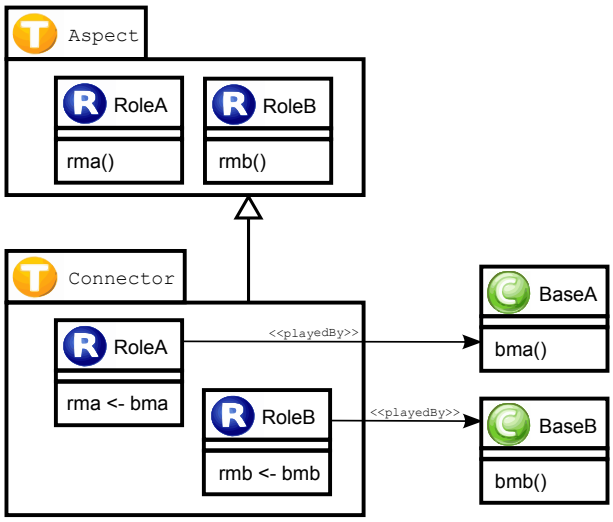

Abbildung 4.2.: Connector Pattern

Gute Software zeichnet sich dadurch aus, dass für wiederkehrende Probleme Design Patterns verwendet werden, die die Lesbarkeit und Struktur des Programms verbessern. Für die Wiederverwendung von mehrmals benötigtem Aspektcode hat sich in OT/J das sogenannte *Connector*

Pattern etabliert (für weitere OT-Patterns siehe [SMHS06] und [obj]). Abbildung 4.2 zeigt die Struktur des Connector Patterns. Der Aspekt wird in den Rollenmethoden des `Aspect` Teams implementiert und erst im `Connector` Team an die gewünschten Basismethoden gebunden. Wenn sich der Bedarf für ein Connector Pattern erst während des Entwicklungsprozesses herausstellt, kann Refactoring den Softwareentwickler dabei unterstützen dieses Pattern aus einem einfachem Team zu extrahieren. Ein Push Down Methodbinding Refactoring kann dem Entwickler beim Trennen von Aspektcode und Aspektbindung behilflich sein.

4.7.2. Extract Method

Extract Method ersetzt eine bestimmte Folge von Anweisungen durch einen Methodenaufruf. Dazu wird eine neue Methode erzeugt, die die zu ersetzenden Anweisungen enthält. Um die Parameter der neu erzeugten Methode zu bestimmen, muss eine Flussanalyse durchgeführt werden, die bestimmt welche lokalen Variablen nicht im Sichtbarkeitsbereich der neu erzeugten Methode sichtbar sind und somit als Parameter übergeben werden müssen. Falls der zu ersetzende Anweisungsblock einen Wert oder eine Instanz erzeugt, kann daraus ggf. ein Rückgabetyp für die erzeugte Methode ermittelt werden.

Der Einfluss durch die Regeln zur Verhaltenserhaltung von OT/J Programmen ist eher gering. Im wesentlichen können Konflikte bei der Erzeugung der neuen Methode entstehen und sind daher die selben, wie bei einem Create Method Refactoring. Der zu extrahierende Code darf außerdem keinen Tsuper oder Base Call beinhalten. Da Aspekte in OT/J nicht direkt an Anweisungen, sondern nur an Methoden gebunden sind, muss der Einfluss von Rollen auf den extrahierten Code nicht analysiert werden.

4.7.3. Inline Method

Inline Method ist die Umkehroperation zu Extract Method. Dieses Refactoring ersetzt den Aufruf einer Methode durch die im Rumpf der Methode enthaltenen Anweisungen. Dabei werden alle verwendeten Parameter durch die an die Methode übergebenen Werte ersetzt. Das Refactoring ist nur möglich, wenn die im Methodenrumpf verwendeten Bezeichner auch an den Stellen im Programm, an denen sie aufgerufen wird sichtbar sind.

Der Einfluss durch OT/J bezieht sich hier hauptsächlich auf die Aspektbindungen. Normale Methodenaufrufe lassen sich durch die Anweisungen der Methode ersetzen. Da Rollen ihre Callin- und Calloutbindungen jedoch direkt an Methoden binden müssen, können diese nicht

auf einen Block aus Anweisungen übertragen werden. Es dürfen also keine Methodenbindungen in einer Rolle existieren, die die Zielmethode referenzieren[7] (OTR6.(a)-(b)).

4.7.4. Extract Rolefile

In OT/J Programmen können Rollen innerhalb einer Teamklasse oder in einer separaten Rollendatei deklariert werden (siehe Abs. 2.2.1). Das Extract Rolefile Refactoring soll eine als innere Klasse deklarierte Rolle in eine Rollendatei umwandeln. Um diese Änderung durchzuführen müssen die Regeln für Rollendateien beachtet werden (OTR5.(a)-(b)).
Dieses Refactoring greift kaum in die Struktur des Programms ein und ist daher fast immer möglich[8]. Falls noch kein Rollenordner existiert muss ein neuer erzeugt werden. Dann kann dort eine .java Datei mit dem Rollennamen erzeugt werden und der Code der Rolle in die Datei verschoben werden. Abschließend muss noch die Teampackage-Deklaration erzeugt werden und das Refactoring ist abgeschlossen.

4.7.5. Inline Rolefile

Das Inline Rolefile Refactoring bildet die Umkehroperation zu Extract Rolefile. Die Rolle wird in die zugehörige Teamklasse verschoben und die Rollendatei gelöscht. Falls es sich um die letzte Rollendatei handelte, kann der Rollenordner ebenfalls gelöscht werden. Auch dieses Refactoring kann immer durchgeführt werden und muss nur die syntaktischen Regeln für Rollendateien beachten (OTR5.(a)-(b)).

4.7.6. Pull Up Methodbinding

Eine Methodenbindung kann neben Feldern und Methoden als ein durch OT/J neu eingeführtes Feature betrachtet werden. Beim Verschieben einer Methode kann es sinnvoll sein die zugehörige Methodenbindung mit zu verschieben.

Pull Up Methodbinding verschiebt eine Methodenbindung (oder Callout to Field Bindung) in eine explizite oder implizite Superrolle der deklarierenden Rolle. Es muss daher sowohl in der Rolle, als auch in der Basis geprüft werden, ob die referenzierten Felder und Methoden sichtbar

[7] Bei gewöhnlichen Referenzen in Methodenaufrufen wird der Aufruf einfach durch den Rumpf der Methode ersetzt. Diese Strategie kann bei Methodenbindungen nicht angewandt werden, da diese nur Methodenreferenzen und keine Statements enthalten dürfen.
[8] Bei fehlenden Schreibrechten bzw. einer binären Teamklasse könnte dieses Refactoring nicht durchgeführt werden

sind (OTL6.(b)-(d)). Auch muss die Zielrolle bereits gebunden sein, um Bindungen enthalten zu können (OTL6.(a)).

Eine Methodenbindung kann Ausdrücke in regulären Guards, Base Guards oder Parameter Mappings besitzen. Da die Zielrolle und möglicherweise auch die gebundene Basisklasse genereller werden, muss geprüft werden ob alle dort referenzierten Bezeichner noch sichtbar sind (OTL12.(b)-(c), OTL7.(c)).

Der Gültigkeitsbereich der verschobenen Methodenbindung wird vergrößert und betrifft auch die impliziten Subrollen der Zielklasse (OTI5.) und es kann dadurch zu mehrdeutigen Methodenbindungen kommen (OTI5.)

Bei einem Pull Up kann es außerdem passieren, dass konkurrierende Methodenbindungen entstehen, die eine Precedence Deklaration erfordern OTL13.(d). Da dieser Konflikt jedoch auch schon in der deklarierenden Rolle aufgetreten sein muss, ist es möglich, aus den dort angegebenen Informationen einen Precedence Ausdruck in der Zielrolle zu erzeugen. Falls jedoch Konflikte in weiteren Subrollen der Zielrolle auftreten, kann das Refactoring nicht mehr voll automatisch durchgeführt werden, da hier Precedence Angaben durch den Entwickler nötig wären.

Listing 4.3 zeigt ein Beispiel in dem eine Precedence Deklaration auf Rollenebene durch eine Precedence Deklaration auf Callinebene ersetzt werden müsste. Wenn die Callinbindung in Zeile 8 in die Rolle `Role1` verschoben werden sollte, müssten Labels für die Callinbindungen in Zeile 8 und 12 erzeugt werden und in Zeile 3 statt der Rollennamen referenziert werden. Das Beispiel zeigt, dass es im Umgang mit Precedence Deklarationen Randfälle gibt, die bei einer technischen Umsetzung beachtet werden müssen.

```
1  public team class Team{
2
3    precedence Role1, Role2;
4
5    protected class Role1 playedBy Base{
6      protected void m(){...}
7      protected void n(){...}
8      void m() <- after void m();
9    }
10
11   protected class Role2 extends Role1{
12     void n() <- after void m();
13 }}
```

Listing 4.3: Codebeispiel für Precedence Anpassung

Da Methodenaufrufe in Guards dynamisch gebunden werden, sollte die Semantik in der deklarierenden Klasse erhalten bleiben. Ein weitaus größeres Problem stellt die semantische Erhaltung durch die Teamaktivierung dar, bei der durch den vergrößerten Sichtbarkeitsbereich der Methodenbindung eventuell das Verhalten vorher nicht betroffener Methoden verändert werden kann.

4.7.7. Push Down Methodbinding

Ein Push Down Methodbinding Refactoring verschiebt eine Methodenbindung in eine explizite oder implizite Subrolle (OTL6.). Bei diesem Refactoring kann es sogar gewünscht sein, dass das Verhalten in gewisser Weise verändert wird, da sich der Gültigkeitsbereich der Methodenbindung verkleinert. Daher sollte zur Sicherheit überprüft werden, welche Methoden nach dem Refactoring nicht mehr von der Methodenbindung erfasst werden, um dem Benutzer dies in einer Warnung mitzuteilen.

Im Prinzip sind bei einem Push Down Methodbinding Refactoring ähnliche Regeln wie bei dem zuvor erläutertem Pull Up Methodbinding zu beachten. Auch bei einem Push Down kann die Sichtbarkeit referenzierter Felder oder Methoden beeinträchtigt werden, wenn diese in der Super-Basisklasse privat waren (OTL12.(b)-(c)).

Wenn die verschobene Methodenbindung in einer Precedence Deklaration erwähnt wird, muss sie aus dieser entfernt werden und in der Zielrolle die Precedence Deklaration ergänzt bzw. erzeugt werden (OTL13.(a)).

4.7.8. Pull Up Role

Da Rollen entlang der Teamvererbungshierarchie vererbt werden, kann das Verschieben von Rollen ein nützliches Refactoring sein. Dieses Refactoring kann nur durchgeführt werden, wenn die Rolle nicht im Zielteam definiert wurde. Es darf also höchstens eine Phantomrolle (siehe Abs. 2.2.8) vorhanden sein, die durch die zu verschiebende Rolle materialisiert wird (OTI6.).

Die verschobene Rolle darf in keinem Subteam des Zielteams überschrieben werden (OTI6.). Falls die zu verschiebende Rolle selbst explizit von einer Rolle erbt, muss diese bereits im Zielteam existieren bzw. sichtbar sein (OTL2.(c)). Da das Verschieben einer Rolle eine Reihe von Typen, Feldern und Methoden im Zieltyp sichtbar macht, müssen mögliche Probleme mit Naming ausgeschlossen werden (OTL2.(d)).

Alle referenzierten Features des umschließenden Herkunftsteams müssen auch im Zielteam existieren. Dies betrifft Referenzen in Parameter Mappings, regulären Guards und Methodenrümpfen der zu verschiebenden Rolle (OTL7.(c) und OTL12.(b)).

Ähnlich wie bei einem Pull Up Methodbinding kann die Erhaltung von Precedence Ausdrücken recht kompliziert oder sogar unmöglich sein (in diesem Fall wären weitere Angaben durch den Entwickler erforderlich) (OTL13.(d)).

4.7.9. Push Down Role

Push Down Role ist die Gegenoperation zu Pull Up Role und verschiebt eine Rolle in ein explizites oder implizites Subteam des umschließenden Teams. Dieses Refactoring ist ebenfalls nur möglich, wenn die zu verschiebende Rolle im Zielteam nicht redefiniert wird, es muss sich also auch hier um eine Phantomrolle handeln (OTI6.).

Falls eine andere Rolle innerhalb des umschließenden Teams von der zu verschiebenden Rolle erbt ist das Refactoring nicht möglich (OTL2.(c)). Die Rolle darf generell nicht in dem umschließenden Team referenziert werden (OTL4.(b), OTL8. und OTR1.(a)-(c)).

Ähnlich wie das Push Down Methodbinding Refactoring ist dieses Refactoring nicht unbedingt semantikerhaltend. Das Verhalten der betreffenden Rolle wird nicht mehr aktiviert, wenn das zuvor deklarierende Team aktiviert wird. Daher sollte bei einer technischen Umsetzung eine Warnung ausgegeben werden, wenn Callinbindungen in der Rolle enthalten sind, die möglicherweise nicht mehr aktiviert werden könnten.

	Extract Method	Inline Method	Extract Rolefile	Inline Rolefile	Pull Up Methodbinding	Push Down Methodbinding	Pull Up Role	Push Down Role
OTL1								
(a)	-	-	-	-	-	-	-	-
OTL2								
(a)	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	✓	✓
(d)	✓	-	-	-	-	-	✓	-
(e)	-	-	-	-	-	-	-	-
(f)	-	-	-	-	-	-	-	-
OTL3								
(a)	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-
OTL4								
(a)	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	✓
OTL5								
(a)	-	-	✓	✓	-	-	-	-
(b)	-	-	✓	✓	-	-	-	-
(c)	-	-	✓	✓	-	-	-	-
OTL6								
(a)	-	-	-	-	✓	-	-	-
(b)	-	✓	-	-	✓	✓	-	-
(c)	-	✓	-	-	✓	✓	-	-
(d)	-	-	-	-	✓	✓	-	-
(e)	-	-	-	-	-	-	-	-
(f)	-	-	-	-	-	-	-	-
(g)	✓	-	-	-	-	-	-	-
OTL7								
(a)	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-
(c)	-	-	-	-	✓	-	✓	-

	Extract Method	Inline Method	Extract Rolefile	Inline Rolefile	Pull Up Methodbinding	Push Down Methodbinding	Pull Up Role	Push Down Role
OTL7.1								
(a)	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-
OTL8	-	-	-	-	-	-	-	✓
OTL9	✓	-	-	-	-	-	-	-
OTL10								
(a)	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-
OTL11								
(a)	-	-	-	-	-	-	-	-
OTL12								
(a)	-	-	-	-	-	-	-	-
(b)	-	-	-	-	✓	-	✓	-
(c)	-	-	-	-	✓	-	-	-
OTL13								
(a)	-	-	-	-	-	✓	-	✓
(b)	-	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-
(d)	-	-	-	-	✓	-	✓	-
OTL14								
(a)	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-
OTI1	-	-	-	-	-	-	-	-
OTI2	-	-	-	-	-	-	-	-
OTI3	-	-	-	-	-	-	-	-
OTI4	-	-	-	-	-	-	-	-
OTI5	✓	-	-	-	✓	-	-	-
OTI6	-	-	-	-	✓	✓	✓	✓
OTS1	✓	✓	-	-	-	-	-	-

Tabelle 4.3.: Weitere Refactorings

4.8. Zusammenfassung

Die Untersuchung der verschiedenen Refactorings bildet eine gute Grundlage für die praktische Umsetzung in Kapitel 6. Insbesondere für die Refactorings Rename Type, Pull Up Method und Push Down Method (siehe Abs. 4.5.3, 4.5.4 und 4.5.5) wurde versucht eine möglichst vollständige[9] Liste von Vorbedingungen zu erstellen.

Während den Untersuchung haben sich einige Regeln herausgestellt, die von mehreren Refactorings verletzt werden können. Häufig können Probleme mit Overriding von Methoden und Rollen innerhalb der impliziten Vererbungshierarchie auftreten. Ein weiteres Problem stellt das Naming dar, wobei Shadowing und Hiding entlang der impliziten Vererbungshierarchie oder in Team-/Rollenverschachtelungen verhindert werden muss. Auch mehrdeutige Methodenbindungen können durch einige Refactorings verursacht werden.

Es bleibt daher zu untersuchen in wie weit eine wiederverwendbare Lösung für die Prüfung dieser Vorbedingung realisiert werden kann, um bei der Adaption weiterer Refactorings von Regelüberschneidungen zu profitieren. Im Idealfall lassen sich OT spezifische Vorbedingungen auf bereits vorhandene Java Vorbedingungen zurückführen und so weit vereinfachen, dass die Überprüfungsmechanismen des JDTs wiederverwendet werden können.

Für die von OT/J neu eingeführten Referenzen sollte ebenfalls eine für alle Refactorings anwendbare Lösung gefunden werden. Im Idealfall lassen sich auch hier die Suchroutinen der JDT Refactorings wiederverwenden.

[9] Eine Garantie für die Vollständigkeit der Vorbedingungen kann nur schwer gegeben werden. Die Praxis zeigt, dass selbst in professionellen Refactoringwerkzeugen, wie die in Eclipse enthaltenen Implementierungen, immer noch Randfälle entdeckt werden, die beim initialen Entwurf nicht bedacht wurden (siehe Abs. 6.3).

5. Die Eclipse Plug-in Architektur

Als Vorbereitung auf Kapitel 6, das die praktische Arbeit beschreibt, werden in diesem Kapitel die wichtigsten Grundlagen der Eclipse Plug-in Architektur beschrieben. Darauf aufbauend werden die Mechanismen von OT/Equinox vorgestellt, die für die Adaptierung des JDT Plug-ins verwendet werden.

5.1. Eclipse Plattform

Eclipse ist eine quelloffene integrierte Entwicklungsumgebung, die ursprünglich für die Entwicklung von Java Programmen entwickelt wurde. Aufgrund der guten Erweiterbarkeit der Eclipse Plattform wird sie mittlerweile für viele verschiedene Sprachen und Entwicklungsaufgaben eingesetzt. Aus diesem Grund wurde Eclipse auch als Basis für das OTDT ausgewählt, das die in Eclipse existierenden Werkzeuge zur Repräsentation und Manipulation von Java Programmen für die durch OT/J eingeführten Sprachfeatures erweitert.

5.1.1. Plug-ins

Das Geheimnis der guten Erweiterbarkeit von Eclipse liegt in der Plug-in Architektur. Eclipse verwendet dafür eine erweiterte Version des OSGi-Frameworks[1], das in Eclipse unter dem Namen *Equinox* bekannt ist. Eclipse besteht aus sogenannten *Plug-ins*, deren Abhängigkeiten untereinander durch das Equinox Framework verwaltet werden. Um vertraglich gesicherte Schnittstellen zu garantieren, wird in den Metadaten[2] eines Plug-ins angegeben, welche Packages für andere Plug-ins sichtbar sind und welche nur intern verwendet werden. Die durch diesen Mechanismus exportierten Packages können von anderen Plug-ins wiederverwendet werden.

[1] OSGi – Open Services Gateway initiative, ist ein Komponentenmodell für Java. Es beschreibt eine Infrastruktur, mit der eine Software aus vielen einzelnen *Bundles* zusammengesetzt werden kann. Für weitere Informationen siehe [OA09].

[2] Die Metadaten werden in einer Datei Namens MANIFEST.MF gespeichert, die ein Bestandteil jedes Plug-ins ist.

Dazu werden in den Metadaten die Namen und Versionen der exportierenden Plug-ins spezifiziert. Für eine eindeutige Identifikation gibt jedes Plug-in eine eindeutige ID und eine Version an. Die Abhängigkeit zwischen zwei Plug-ins wird durch eine *required*-Relation beschrieben, die im Prinzip einen Plug-in übergreifenden Import der exportierten Packages bewirkt.

Die Plug-in Architektur ermöglicht dadurch eine schwache Kopplung zwischen den einzelnen Plug-ins. Um dies zu erreichen ist es notwendig, dass Plug-ins alle Abhängigkeiten explizit angeben. Dadurch können Plug-ins verschiedener Hersteller miteinander kombiniert und ein individuelles Softwareprodukt zusammengestellt werden.

Um die Erweiterung und Wiederverwendung bestehender Plug-ins zu verbessern bietet das Equinox Framework außerdem die Möglichkeit, Plug-ins durch andere Plug-ins zu erweitern.

5.1.2. Extension Points

Ein Plug-in kann einen oder mehrere *Extension Points* anbieten, die von einem oder mehreren Plug-ins erweitert werden können. Das Plug-in, das erweitert werden soll nimmt in einer solchen Beziehung die Rolle des *Host Plug-ins* ein und kann durch ein oder mehrere *Extender Plug-ins* erweitert werden. Bei einer Erweiterung kann die Kommunikation über einfache *Listener*[3] funktionieren, um auf Aktionen des Host Plug-ins reagieren zu können. Es sind aber auch Verhaltensänderungen durch das Extender Plug-in möglich, indem es ein Callback Objekt bereitstellt, dessen Verhalten von dem Host Plug-in ausgeführt wird.

Die Klassen und Interfaces, die für die Kommunikation zwischen Host- und Extender Plug-in benötigt werden, stellt das Host Plug-in bereit und spezifiziert diese in der Deklaration des Extension Points. Extension Points oder die Erweiterung eines solchen durch eine Extension, werden in der plugin.xml Datei deklariert.

5.2. OT/Equinox

Das Equinox Framework bietet bereits gute Möglichkeiten für die Wiederverwendung und Erweiterung von Plug-ins. Das Problem dieser Lösung ist, dass schon bei der Implementierung eines Plug-ins Extension Points bedacht werden müssen. Oft stellt sich erst zu einem späterem Zeitpunkt heraus, welche Anpassungsmöglichkeiten in einem Plug-in benötigt werden.

[3] Im Java Umfeld werden Klassen, die in einem *Observer Pattern*[GHJV95] als *Observer* fungieren, als Listener bezeichnet.

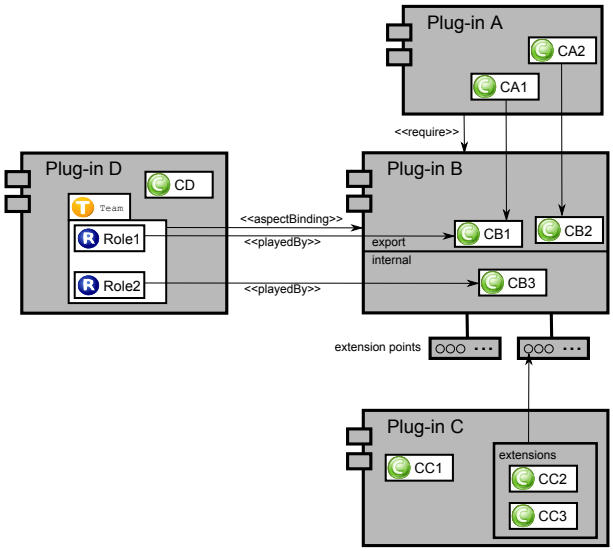

Abbildung 5.1.: Plug-in Beziehungen in OT/Equinox

Außerdem verursacht jeder Punkt, an dem ein Plug-in variabel sein soll, zusätzlichen Code und verschlechtert dadurch die Lesbarkeit des Programms.

5.2.1. Aspektbindungen

OT/Equinox erweitert Equinox um eine *aspectBinding* Beziehung. Eine solche Aspektbindung verbindet ein Team mit einem Plug-in, das in der Beziehung die Rolle des Basis Plug-ins einnimmt. Aspektbindungen werden in der plugin.xml Datei des Plug-ins, das das Team bereitstellt deklariert. Das Team darf Rollen für die Klassen der exportierten Packages des Basis Plug-ins deklarieren und somit dessen Verhalten verändern. Um auch Klassen aus nicht exportierten Packages verändern zu können, bietet OT/Equinox die Möglichkeit sogenannte *Forced Exports* zu deklarieren. Dadurch können auch interne Klassen als Basis in einer playedBy-Relation verwendet werden. Ein solcher Forced Export gilt nur für das Plug-in, das ihn deklariert. Eine ausführliche Beschreibung der OT/Equinox Konzepte können in [HM07] nachgelesen werden.

Abbildung 5.1 zeigt die verschiedenen Relationen, die in einer OT/Equinox Architektur möglich sind. Die einfache required-Relation zwischen `Plug-in A` und `Plug-in B` stammt aus der Spezifikation des OSGi-Frameworks. Dabei darf `Plug-in A` auf die von `Plug-in B` exportierten Klassen zugreifen und diese wiederverwenden. `Plug-in B` definiert 2 Extension Points, wobei einer von `Plug-in C` erweitert wird. Diese Form der Erweiterung stammt aus der Equinox Implementierung. Die aspectBinding-Relation zwischen `Plug-in D` und `Plug-in B` wurde durch OT/Equinox eingeführt und ermöglicht die Erzeugung von playedBy-Relationen zu Klassen des Basis Plug-ins.

6. Adaptierung von Eclipse JDT Refactorings

In diesem Kapitel wird das Vorgehen bei der Adaptierung der bestehenden JDT Refactorings beschrieben. Die Grundlage für diese Adaptierungen bilden die Untersuchungen aus Kapitel 4. Zunächst wird das LTK vorgestellt, ein Framework für die Entwicklung von Refactorings für Eclipse. Anschließend wird das allgemeine Vorgehen bei der Adaption von bestehenden JDT Refactorings beschrieben und einige Mängel beschrieben, die bei der Arbeit in bestehenden Implementierungen gefunden wurden. Abschließend wird die Adaptierungen von drei Refactorings im Detail beschrieben.

6.1. Das Refactoring Language Toolkit

Das LTK ist ein Framework zum Erstellen von Refactorings für Entwicklungsumgebungen, die auf der Eclipse Plattform basieren. Die Refactorings des JDTs sind selbst mit Hilfe des LTKs realisiert worden. Das Framework bietet dem Entwickler eine Reihe nützlicher Features für die Umsetzung eines Refactorings, die im folgenden Abschnitt an Hand der Bestandteile eines Refactorings erläutert werden.

6.1.1. Bestandteile eines Refactorings

Für die Implementierung eines neuen Refactorings müssen die folgenden Bestandteile implementiert werden. Einige von ihnen sind optional und für die einfach Durchführung eines Refactorings nicht zwingend notwendig.

Refactoring Implementierung

Die Refactoring Implementierung befindet sich in einer Klasse, die von der Klasse `Refactoring` erbt und die für den in Abschnitt 6.1.2 beschriebenen Lebenszyklus notwendigen Methoden `checkInitialConditions()`, `checkFinalConditions()` und `createChange()` implementiert. Das Framework kümmert sich dabei um die richtige Ausführungsreihenfolge der implementierten Methoden.

Refactoring Wizard

Der Refactoring Wizard bildet die graphische Benutzerschnittstelle für die Ausführung des Refactorings. Hier können notwendige Parameter eingegeben werden und einfache Fehleingaben abgefangen werden. Das Framework bietet darüber hinaus eine Fehleransicht für fehlgeschlagene Vorbedingungen oder Warnungen bei der Ausführung des Refactorings. Vor der endgültigen Durchführung des Refactorings erzeugt das Framework eine Vorschau aller Änderungen im Quelltext, die vom Benutzer begutachtet und bestätigt werden können.

Refactoring Action

Die Refactoring Action wird verwendet um das Refactoring zu starten. Es bestimmt aus der aktuellen Selektion die Parameter des Refactorings oder deaktiviert den Menüpunkt, falls das Refactoring für die selektierten Elemente nicht durchgeführt werden kann.

Refactoring Descriptor (optional)

Optional kann ein Refactoring Descriptor bereitgestellt werden, der alle Eingabeparameter des durchgeführten Refactorings speichert und somit dem Framework ermöglicht eine neue Instanz des Refactorings zu erstellen. Ein Refactoring Descriptor wird für die Möglichkeit Refactoring Skripte zu schreiben oder die Refactoring History Funktion verwendet, die anzeigt, wann welche Refactorings durchgeführt wurden.

Refactoring Contribution (optional)

Mit der Implementierung einer Refactoring Contribution Klasse wird das Refactoring beim Eclipse Kern registriert. Diese Klasse ist ebenfalls notwendig für die Unterstützung von Refactoring Skripten und der Refactoring History.

6.1.2. Der Lebenszyklus eines Refactorings

Das LTK Framework legt eine feste Ausführungsreihenfolge des Refactorings fest. Dabei werden die folgenden Schritte immer in der gleichen Reihenfolge ausgeführt, wobei einige Schritte auch mehrmals ausgeführt werden können, falls sich beispielsweise die Eingabeparameter ändern.

1. **Initale Vorbedingungen prüfen:**
   Hierzu wird die Methode `checkInitialConditions()` aufgerufen, die prüft ob es sich um ein gültiges Ziel handelt und die Datei nicht binär ist oder keine Schreibrechte vorhanden sind. Die Analysen in dieser Phase sollten sehr kurz gehalten werden um keine zu großen Verzögerung in der Bedienung zu verursachen.

2. **Zusätzliche Benutzereingaben (optional):**
   Falls weitere Angaben notwendig sind, werden weitere Parameter gesetzt. Bei Refactoring Skripten werden alle Parameter bei der Erzeugung des Refactorings angegeben.

3. **Finale Bedingungen prüfen:**
   Wenn alle Parameter des Refactorings feststehen, können die finalen Bedingungen durch einen Aufruf der Methode `checkFinalConditions()` geprüft werden. Dies beinhaltet die vollständige Analyse der Vorbedingungen zur syntaktischen Korrektheit und semantischen Verhaltenserhaltung des Refactorings.

4. **Änderungen erzeugen:**
   Wenn alle Vorbedingungen erfüllt sind, können die Änderungen durch die Methode `createChange()` erzeugt werden.

5. **Änderungen anzeigen (optional):**
   Der Benutzer hat die Möglichkeit die Änderungen zu begutachten und ggf. das Refactoring abzubrechen oder abzuschließen.

Inhaltlich sind die verschiedenen Phasen in der Praxis nicht unbedingt voneinander getrennt. Durch die feste Ausführungsreihenfolge können Quellcodeänderungen schon während der Prüfung der Vorbedingungen erzeugt werden, um die Performanz zu verbessern. Dies kann beispielsweise sinnvoll sein, wenn bei einem Rename Refactoring Vorbedingungen für Referenzen geprüft werden müssen, die später umbenannt werden sollen.

6.1.3. Refactoringstatus

Das Framework stellt die Klasse `RefactoringStatus` zur Verarbeitung der verschiedenen Analyseergebnisse bereit. Die Lebenszyklus Methoden `checkInitialConditions()` und

`checkFinalConditions()` liefern daher jeweils ein `RefactoringStatus` Objekt zurück. Ein `RefactoringStatus` Objekt enthält eine Nachricht, die den Fehler beschreibt, sowie einen Schweregrad des Fehlers. Die verschiedenen Schweregrade eines Fehlers sind wie folgt zu interpretieren.

Warning

Bei einer Warnung kann das Refactoring durchgeführt werden, es bestehen jedoch Probleme, die vom Benutzer begutachtet werden sollten. Eine Verhaltensänderung ist möglich, das resultierende Programm ist jedoch fehlerfrei. Typischerweise werden Warnungen verwendet, um dem Benutzer zusätzliche Änderungen wie eine Sichtbarkeitsanpassung mitzuteilen.

Error

Ein *Error* Status signalisiert eine Beeinträchtigung der Syntax oder Semantik des Programms. Im Unterschied zu einer Warnung ist die Verhaltensänderung bereits durch das Refactoring erkannt worden und muss nicht durch den Benutzer überprüft werden. Das Refactoring kann zwar trotzdem durchgeführt werden, jedoch kann keine semantische Äquivalenz oder ein fehlerfreies Ergebnis garantiert werden.

Fatal Error

Bei einem *Fatal Error* Status kann das Refactoring technisch nicht durchgeführt werden und muss abgebrochen werden.

6.1.4. Refactoring Participants

Das LTK bietet eine Infrastruktur für sogenannte *Refactoring Participants* an. Dabei handelt es sich um Extension Points eines Refactorings, die von einem Plug-in selektiv erweitert werden können. Das erweiternde Plug-in stellt dann einen Participant dar und wird über durchgeführte Refactorings informiert um zusätzliche Vorbedingungen zu prüfen und eigene Quellcodeänderungen zu Erzeugen. Um Konflikte auszuschließen, darf der Participant nicht die gleichen Dateien wie das ursprüngliche Refactoring verändern.
Participants werden z. B. innerhalb des Eclipse PDEs verwendet um bei einem Rename Type Refactoring auch Referenzen innerhalb der Plugin.xml Datei umzubenennen.

6.2. Allgemeines Vorgehen

Für die praktische Umsetzung der in Kapitel 4 beschriebenen Refactorings kommen verschiedene Lösungsansätze in Frage. Im Folgenden werden die verschiedenen Ansätze kurz beschrieben und die Vor- und Nachteile diskutiert.

1. **Neuentwicklung**
   Das Refactoring könnte als neues Plug-in mit Hilfe des LTKs entwickelt werden. Um die Verhaltenserhaltung der reinen OO-Refactorings zu gewährleisten, wurden die dazu in früheren Arbeiten formulierten Vorbedingungen vorausgesetzt (siehe Abs. 4.2). Um den dazu existierenden Code der JDT Refactorings wiederzuverwenden könnte das neu entwickelte Refactoring von dem JDT Pendant erben. Da viele Anpassungen jedoch auch private Methoden betreffen würden, wäre Vererbung allein nicht ausreichend und Code müsste kopiert werden. Ein weiteres Problem stellt die Initialisierung der neuen Klassen dar. Dazu müssten alle Stellen im JDT Code angepasst werden, die das Refactoring erzeugen. Da auch diese veränderten Klassen erzeugt werden müssten, setzt sich das Problem fort und eine Vielzahl von Klassen müsste neu geschrieben werden, um ein einziges Refactoring zu adaptieren. Erfahrungen bei der Adaption anderer JDT Klassen haben gezeigt, dass eine solche Lösung nicht evolutionsfähig ist [HM07]. Aus diesem Grund wird dieser Lösungsweg ausgeschlossen.

2. **Entwicklung eines Participants**
   Das in Abschnitt 6.1.4 vorgestellte Participant Konzept könnte für die Überprüfung zusätzlicher OT/J Vorbedingungen eingesetzt werden. Die Probleme einer solchen Lösung wären ähnlich wie bei einer Neuentwicklung. Die Evolutionsfähigkeit wäre zwar besser, da ein Participant über eine wohldefinierte Schnittstelle mit dem Host Plug-in kommuniziert. Die Probleme der schlechten Wiederverwendbarkeit der JDT Methoden würde jedoch an anderer Stelle auftreten.
   Die Vorbedingungen für die Verhaltenserhaltung von Java Programmen würden zwar vom Host Plug-in überprüft werden, aber viele OT/J spezifische Vorbedingungen ließen sich idealerweise zur gleichen Zeit prüfen. Dadurch würde redundanter Code im Participant entstehen, der die Performanz des Refactorings beeinträchtigt, da Vorbedingungen doppelt geprüft werden müssten. Darüber hinaus stellt redundanter Code immer ein Problem in der Wartbarkeit und Evolutionsfähigkeit einer Software dar.
   Ein weiteres Problem stellen die Quelltextänderungen der JDT Refactorings dar, die nicht die selben Dokumente verändern dürfen, wie die des Participants. Eine Erweiterung durch Participants eignet sich daher nur für zusätzliche Quellen, die nicht so stark mit dem reinen OO-Code verwoben sind wie eine in die Sprache integrierte Erweiterung.

3. **Adaptierung mit OT/Equinox**
   Ein OT/Equinox Plug-in, dass mit einer Aspektbindung an die zu adaptierenden Refactoring Klassen gebunden wird, erlaubt einen vollen Zugriff auf die benötigte Infrastruktur. Der selektive Import durch Calloutbindungen und die Adaption durch Callinbindungen bieten die nötige Granularität, um möglichst viel JDT Funktionalität wiederzuverwenden und den Umfang der Änderungen so gering wie möglich zu halten. Wegen dieser Flexibilität und der dadurch am besten gewährleisteten Evolutionsfähigkeit wird im Rahmen dieser Arbeit die Adaption mit Hilfe von OT/Equinox durchgeführt.

6.2.1. Allgemeiner Aufbau des Refactoring Adapters

Da die Adapter für die verschiedenen Refactorings einige Gemeinsamkeiten aufweisen, wird ein Überblick über die allgemeine Struktur des Adapter Plug-ins und die darin enthaltenen Adapter Teams gegeben.

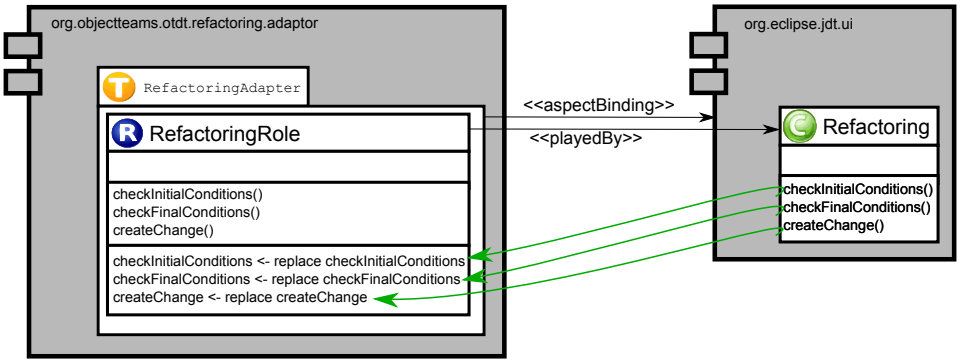

Abbildung 6.1.: Aufbau des Refactoring Adapters

Der Adapter ist als OT/Equinox Plug-in realisiert und enthält für jedes zu adaptierende Refactoring ein Team, das der Klasse, die das Refactoring implementiert (siehe Abs. 6.1.1), eine Rolle hinzufügt. Um die eigentliche Adaption durchzuführen, werden die Methoden des Refactoring Lebenszyklus durch Replace-Callinbindungen überschrieben[1]. Durch Base Calls können dann die Vorbedingung und Quelltextänderungen des JDT Refactorings geprüft und erzeugt werden. Die Klasse `RefactoringStatus` bietet eine `merge()` Methode an, mit der die Ergebnisse der Überprüfung der Vorbedingungen von Basis und Rolle leicht vereinigt werden können.

[1] In einigen Fällen kann auch eine Before- oder After-Callinbindung verwendet werden.

6.2.2. Compiler versus Refactoring

Bei genauer Betrachtung der verschiedenen Regeln zur Verhaltenserhaltung (siehe Abs. A.2) fällt auf, dass diese viele Ähnlichkeiten mit Regeln haben, die ein Compiler beim Übersetzen des Programms prüfen muss. Daher stellt sich die Frage, ob die Algorithmen des Compilers für die Überprüfung der Vorbedingungen eines Refactorings wiederverwendet werden können. Bei genauerer Betrachtung des Problems fallen jedoch einige grundlegende Unterschiede beider Techniken auf.

1. Ein Compiler betrachtet das Programm in seinem aktuellen Zustand und produziert statische Fehlermeldungen. Ein Refactoring hingegen untersucht eine Programmtransformation, die von verschiedenen Eingabeparametern abhängt.

2. Ein Refactoring kann spezifischere Fehlermeldungen produzieren als ein Compiler, da Kontextinformationen über das durchzuführende Refactoring vorhanden sind und nicht nur der statische Quelltext.

3. Ein Refactoring kann den Einfluss der Regeln abschätzen und auf die Regeln beschränken, die durch das aktuelle Refactoring verletzt werden. Ein Compiler muss alle denkbaren Sprachregeln untersuchen, was die Performanz eines Refactorings deutlich verschlechtern würde.

Auf Grund der genannten Unterschiede ist es sinnvoll eigene Strategien für die Untersuchung der Vorbedingungen eines Refactorings zu entwickeln.

6.2.3. Anmerkung zu Regel OTL2.(d)

Während der praktischen Arbeit ist aufgefallen, dass ein Teil der Regel OTL2. (d) (OTJLD §1.4 (a) [HHM09]) nicht vom Compiler überprüft wird. Das dort erwähnte *Shadowing* wird nur für Typen überprüft, daher dürfen Rollen den gleichen Namen wie ein Feld oder eine Methode ihres umschließenden Teams haben. Um in den Vorbedingungen eines Refactorings keine vom Compiler akzeptierten Programme zu verbieten, wird dieser Teil der Regel in der praktischen Arbeit nicht umgesetzt.

6.2.4. Wiederverwendung für wiederkehrende Vorbedingungen

In den Tabellen zur Übersicht über Regelverstöße durch Refactorings ist zu sehen, dass verschiedene Refactorings die gleichen Regeln brechen können. Um redundanten Code zu vermeiden,

besitzt das Plug-in die Klasse `RefactoringUtil`, in der Methoden zur Überprüfung wiederkehrender Vorbedingungen gesammelt werden können.

Ein Problem was bei einer solchen zentralen Verwaltung von Regelverstößen entsteht ist, dass Kontextinformationen des aktuellen Refactorings verloren gehen, worunter die Aussagekraft der Fehlermeldungen leidet. Um dieses Problem zu beheben werden in solchen Fällen Callbackobjekte verwendet, die von dem jeweiligen Refactoring implementiert werden und aussagekräftige Fehlermeldungen erzeugen.

6.2.5. Berücksichtigung neuer OT Referenzen

Refactorings müssen häufig Referenzsuchen durchführen, die z. B. beim Löschen eines Elements sicherstellen, dass dieses unreferenziert ist. Das JDT stellt für solche Zwecke eine Java Suche bereit, die nicht nur eine reine Textsuche durchführt, sondern auch die Java Sprachspezifikation berücksichtigt. Dadurch ist es möglich nach einem bestimmten Java Element wie einem Typ oder einer Methode zu suchen.

Die Java Suche wurde im Zuge der Entwicklung des OTDTs bereits adaptiert und findet daher bereits die neu eingeführten Referenzen.

6.2.6. Testgetriebene Entwicklung

Für die praktische Umsetzung wurde eine Testgetriebene Entwicklung umgesetzt. Dies ermöglichte eine Anforderungsdefinition der adaptierten Refactorings über Testfälle, bevor mit der eigentlichen Entwicklung begonnen wurde. Dazu wurden aus den in Kapitel 4 untersuchten Regelverstößen Quelltextbeispiele konstruiert, die einen entsprechenden Regelverstoß produzieren. Dabei ließ sich für jeden möglichen Regelverstoß mindestens ein Testfall ableiten, wobei einige auch durch mehrere Beispiele verursacht werden können.

6.3. Festgestellte Mängel in JDT Refactorings

Während der Analyse der zu adaptierenden Refactorings sind einige Mängel festgestellt worden. Dabei handelt es sich in allen Fällen um fehlende Vorbedingungen. Dieses Fehlverhalten erschwert die Adaption, da fehlende Vorbedingungen teilweise OT/J spezifische Vorbedingungen

mit abdecken[2]. In anderen Fällen würde eine JDT Implementierung eine Infrastruktur liefern, die von einer adaptierenden Rolle wiederverwendet und erweitert werden könnte. Beide Fälle führen zu redundantem Code, der zu einem späterem Zeitpunkt gelöscht oder integriert werden müsste. Da diese Diplomarbeit keinen direkten Einfluss auf den Basiscode (das JDT) hat, wurden die Mängel in Form von Bugreports festgehalten, um die Adaptierung in einer späteren Version des JDT/OTDT zu ermöglichen.

Im Folgenden sind die Mängel kurz beschrieben. Eine detaillierte Beschreibung lässt sich in den entsprechenden Bugreports nachlesen.

- Ein Move Method Refactoring kann Overriding erzeugen, ohne dies in einer Fehlermeldung zu signalisieren.
- Bei einem Pull Up Method Refactoring kann das Verhalten durch Overriding verändert werden. Der Fall wurde bereits für die implizite Vererbungshierarchie in den Vorbedingungen von Pull Up Method (Abs. 4.5.4 Vorb. 8) diskutiert und lässt sich analog auf die explizite Vererbungshierarchie übertragen.
- Bei einem Inline Method kann der Methodenrumpf nach dem Refactoring ungültige Referenzen auf private Methoden und Felder haben.
- Wenn bei einem Move Type Refactoring der Typ in ein anderes Package verschoben wird, können ungültige Methoden- und Feldzugriffe entstehen. Dies betrifft Methoden und Felder, die eine protected oder default Sichtbarkeit besitzen.
- Ein Move Method Refactoring kann doppelte Methoden mit gleichem Namen und gleicher Signatur erzeugen, die zu Fehlern beim Übersetzen des Programms führen.
- Ein Push Down Refactoring sucht nur innerhalb des deklarierenden Typs nach Referenzen. Dadurch können nach dem Refactoring ungültige Referenzen entstehen, die zu Fehlern beim Übersetzen führen.

Die detaillierten Bugreports können in [ecl09a], [ecl09c], [ecl09d], [ecl09b] und [ecl09e] nachgelesen werden.

6.4. Adaptierung von Rename Type

Auf Grundlage der in Abschnitt 4.5.3 formulierten Vorbedingungen wurde das Rename Type Refactoring adaptiert. Bevor mit der eigentlichen Adaption begonnen wurde, wurde analysiert,

[2]Beispielsweise überprüfen Referenzanalysen automatisch neue OT Referenzen, da die Java Suche im OTDT bereits entsprechend adaptiert wurde.

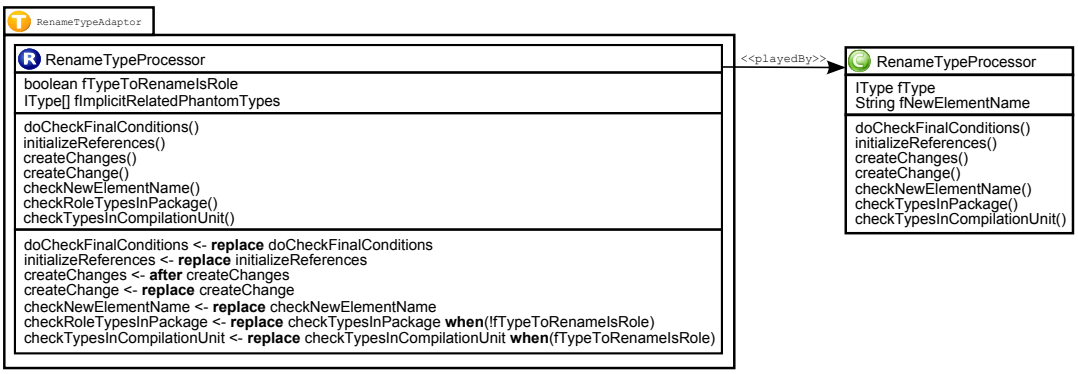

Abbildung 6.2.: Klassendiagramm des Rename Type Adapters

welche Vorbedingungen und Änderungen bereits durch die JDT Implementierung abgedeckt werden.

6.4.1. Bereits behandelte Vorbedingungen und Änderungen

- Die Vorbedingung 4 wird bereits von der Methode `checkConflictingTypes()` überprüft. Hier führt das JDT Refactoring eine Referenzsuche durch und prüft ob der neue Name bereits im Namensraum der Referenz bekannt ist.
- Die zusätzlichen Programmänderungen 1, 4 und 5 werden bereits vom JDT Refactoring durchgeführt, da dieses die adaptierte Java Suche verwendet.

6.4.2. Durchgeführte Adaptionen

Abbildung 6.2 zeigt ein Klassendiagramm des entwickelten Adapters[3]. Im Folgenden werden die Adaptionen der verschiedenen Methoden beschrieben und der Bezug zu den behandelten Vorbedingungen hergestellt.

doCheckFinalConditions()

- Beim Aufruf der Methode `doCheckFinalConditions()` stehen alle Informationen des Refactorings zur Verfügung (Zieltyp und neuer Name). Daher werden an dieser Stelle die in Vorbedingung 8 erwähnten implizit verwandten Rollen gesucht und gespeichert.

[3]Für eine bessere Übersicht sind nur die wichtigsten Felder, Methoden und Methodenbindungen dargestellt.

Dazu wird zuerst die *Wurzel* der impliziten Vererbungshierarchie gesucht, d. h. eine Rolle die keine weiteren impliziten Superrollen mehr besitzt. Es muss genau eine Wurzel existieren, da laut OTJLD §1.5 d) *Name Clashes* bei der Vererbung von Rollen verboten sind [HHM09]. Daher kann keine Vererbungshierarchie existieren, in der eine Rolle von zwei *unabhängigen* Superrollen erbt. Die impliziten Subrollen der Wurzel (ohne den Zieltyp) ergeben dann die Menge aller implizit verwandten Rollen des Zieltyps. Für eine leichtere Weiterverarbeitung wird diese Menge abschließend noch in Phantomrollen und reguläre Rollen unterteilt.

- Da auch implizit verwandten Rollen umbenannt werden müssen, müssen für jede implizit verwandte Rolle Namenskonflikte untersucht werden. Dies schließt auch die neue Vorbedingung 5 bezüglich *Shadowing* ein. Daher wird in einer Schleife für alle implizit verwandten Rollen der neue Name überprüft.
- Bei der Umbenennung von Teams müssen die in Vorbedingung 3 beschriebenen Namenskonflikte für Rollenordner ausgeschlossen werden. Dazu wird zuerst überprüft, ob das Team einen Rollenordner besitzt und ggf. eine Referenz für eine spätere Umbenennung gespeichert. Falls ein Rollenordner existiert wird überprüft, ob ein Package mit dem neuen Namen existiert und eine entsprechende Fehlermeldung[4] erzeugt.
- Bei der Umbenennung von Rollen muss implizites Overriding ausgeschlossen werden (Vorbedingung 7). Dazu wird in allen Superteams des umschließenden Teams und deren Subteams nach Rollen mit dem neuen Namen gesucht. Falls Rollen mit dem neuen Namen existieren wird eine entsprechende Fehlermeldung erzeugt.

checkRoleTypesInPackage()

Die Basismethode prüft ob Namenskonflikte innerhalb des Packages auftreten. Für OT/J Programme muss zusätzlich sichergestellt werden, dass der neue Name nicht von einer Rolle überschattet wird (Vorbedingung 1).

checkTypesInCompilationUnit()

Die Basismethode prüft ob innerhalb der gleichen Quelldatei Namenskonflikte auftreten können. Falls es sich um eine Rolle handelt muss zusätzlich im umschließenden Team überprüft werden, ob der neue Rollenname überschattet wird bzw. einen anderen Typ überschatten könnte (Vorbedingung 2 und 4).

[4] Fehlermeldungen werden immer in einem `RefactoringStatus` Objekt gekapselt.

checkNewElementName()

Die Basismethode prüft bereits, ob der neue Name ein gültiger Java Bezeichner ist. Zusätzlich werden die für Rollen verbotenen Namen ausgeschlossen (Vorbedingung 6).

initializeReferences()

Die Methode `initializeReferences()` sucht Referenzen des Zieltyps um diese umzubenennen. Da bei Rollen auch Referenzen von implizit verwandte Rollen umbenannt werden müssen, muss die Methode für jede implizit verwandte Methode aufgerufen werden. Das Problem einer solchen Umsetzung bestand darin, dass die Methode keine Parameter bekommt, sondern die Informationen direkt aus den Feldern der Refactoring Klasse liest. Eine Lösungsmöglichkeit wäre es die Methode in die adaptierende Rolle zu kopieren und für verschiedene Typen zu parametrisieren. Da die Methode sehr lang ist und Kopieren immer eine Lösung darstellt, die die Evolutionsfähigkeit des Adapters verschlechtert, wurde ein anderer Lösungsweg gewählt.

Die Adaption wurde mit einem Replace-Callin durchgeführt, der die Basismethode in einer Schleife aufruft. In Listing 6.1 ist ein Auszug der Callinmethode zu sehen. Um die Informationen für die verschiedenen Rollen bereitzustellen, wurden die von der Methode benötigten Felder in temporären Variablen gesichert (Zeilen 6-7) und vor jedem Base Call (Zeile 15) mit den für den aktuellen Aufruf benötigten Daten beschrieben (Zeile 13). Nach dem Base Call wird das Ergebnis aus dem Feld ausgelesen und gespeichert (Zeile 17). Nach der Schleife werden die Daten aus den temporären Variablen zurück in die Felder der Basis geschrieben (Zeilen 21 und 28), um wieder eine gültige Invariante herzustellen. Die resultierende Adaption führt einen Teil der neuen Änderungen durch (zusätzliche Änderung 7).

```
1  callin RefactoringStatus initializeReferences(IProgressMonitor pm) throws JavaModelException, OperationCanceledException
       {
2    //jdt strategy
3    RefactoringStatus jdtStatus = base.initializeReferences(pm);
4
5    // cache the original focus type and found jdt references
6    IType originalFocusType = getFType();
7    SearchResultGroup[] jdtReferences = getFReferences();
8
9    ArrayList<SearchResultGroup> otReferences = new ArrayList<SearchResultGroup>();
10   try{
11     // find references for implicit related types
12     for (int i = 0; i < fImplicitRelatedTypes.length; i++) {
13       setFType(fImplicitRelatedTypes[i]);
14       setFReferences(null);
15       jdtStatus.merge(base.initializeReferences(pm));
16       // add all references of the implicit related type
17       otReferences.addAll(Arrays.asList(getFReferences()));
18     }
19   }finally{
20     // ensure that the original focus type is reset
21     setFType(originalFocusType);
```

```
22
23      // combine the ot references with the jdt references
24      otReferences.addAll(Arrays.asList(jdtReferences));
25      setFReferences(otReferences.toArray(new SearchResultGroup[otReferences.size()]));
26
27      // rewrite cached data to base fields
28      ...
29    }
30    return jdtStatus;
31  }
32  initializeReferences <- replace initializeReferences;
```

Listing 6.1: Adaption mit Hilfe einer Base Call Schleife

Ein weiteres Problem stellte die Suche nach Phantomtyp Referenzen dar. Die Java Suche benötigt Informationen des Java Models, die von der Klasse `PhantomType` nicht bereit gestellt wurden. Die Klasse implementiert das JDT Interface `IType`, welches Methoden zur Beschreibung des Typs vorschreibt. Einige Methoden sind jedoch nicht implementiert und werfen eine `UnsupportedOperationException`, wenn die gewünschte Information für Phantomtypen nicht zur Verfügung steht.
Um trotzdem die Java Suche verwenden zu können, wurde das Team `PhantomTypeAdaptor` implementiert, das einem `PhantomType` die nötigen Implementierungen hinzufügt. Dazu wurden die Informationen der impliziten Superrolle[5]des Phantomtyps weitergeleitet. Um andere Teile des OTDTs nicht zu beeinträchtigen, wird der `PhantomTypeAdaptor` nur während der Java Suche von Phantomtyp Referenzen aktiviert.

Falls diese Lösung für alle Klienten der Klasse `PhantomType` verträglich ist, könnte das Verhalten des Adapters in den OTDT Kern übernommen werden.

createChanges()

Diese Methode bereitet Quelltext Änderungen vor. Die Referenzen für implizit verwandte Rollen wurden bereits angepasst. Es fehlt noch die Umbenennung ihrer Deklarationen, die in dieser Methode hinzugefügt werden (zusätzliche Änderung 6).

createChange()

Diese Methode erzeugt die fertigen Änderungen. Für eine OT Adaptierung müssen folgende Punkte ergänzt werden:

[5]Die Klasse `PhantomType` stellt dazu die Methode `getRealType()` bereit, die die erste im Sourcecode vorhandene Superrolle zurückliefert.

- Es wird ein Flag gesetzt, damit alle Quelltextdateien der implizit verwandten Rollen gespeichert werden.
- Falls beim prüfen der Vorbedingungen ein Rollenordner gefunden wurde, wird dieser mit umbenannt (zusätzliche Änderung 2).
- Wenn eine implizit verwandte Rolle in einer Rollendatei deklariert wurde, wird diese ebenfalls mit umbenannt (zusätzliche Änderung 3).

6.5. Adaptierung von Pull Up

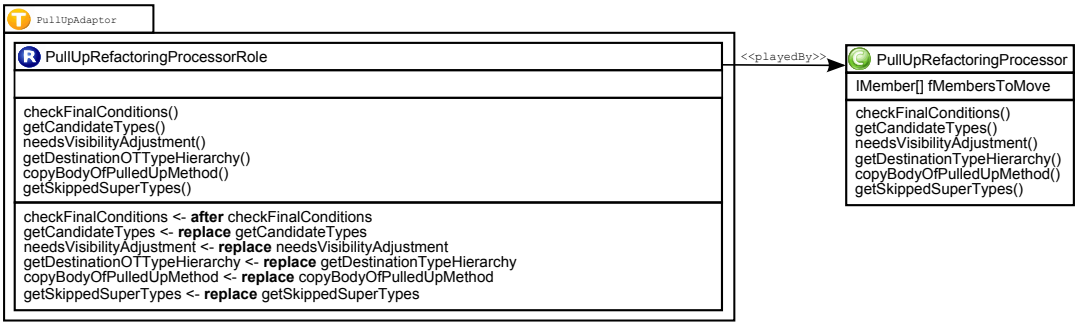

Abbildung 6.3.: Klassendiagramm des Pull Up Adapters

Auf Grundlage der in den Abschnitten 4.5.4 und 4.5.6 formulierten Vorbedingungen wurde das Pull Up Refactoring adaptiert.

6.5.1. Bereits behandelte Vorbedingungen und Änderungen

- Vorbedingung 1 muss nicht behandelt werden, da diese auch nicht vom OT/J Compiler berücksichtigt wird (siehe Abs. 6.2.3).
- Vorbedingung 2 wird bereits durch das JDT Refactoring umgesetzt, da private Methoden und Felder immer eine **`protected`** Sichtbarkeit erhalten. Der gleiche Mechanismus deckt Vorbedingung 3 und 10 ab.
- Vorbedingung 5 wird bereits durch die Basismethode `checkAccessedTypes()` überprüft. In der Methode wird für alle Zugriffe auf Typen geprüft, ob der Typ im Ziel sichtbar ist.

6.5.2. Durchgeführte Adaptionen

checkFinalConditions()

- Für OT Elemente wie Callinmethoden sind nur Rollen als Ziel möglich (Vorbedingung 4). Dazu wird in der Methode `checkDestinationForOTElements()` sichergestellt, dass Callinmethoden nur in Rollen verschoben werden. Die Methode könnte für die Umsetzung eines Pull Up Methodbinding Refactorings 4.7.6 entsprechend erweitert werden.
- Es wird die Methode `checkShadowingFieldInImplicitHierarchy()` aufgerufen, die sicherstellt, dass der Name eines verschobenen Feldes keine anderen Felder verdeckt (Vorbedingung 6).
- Die Hilfsmethode `checkOverloadingAndAmbiguity()` konnte verwendet werden, um mehrdeutige Methodenbindungen zu erkennen. Sie wurde im Rahmen der Diplomarbeit von Gregor Brcan [Brc05] entwickelt, um damit mehrdeutigen Methodenbindungen bei einem Rename Method Refactoring vorzubeugen (Vorbedingung 7).
- Wie bereits in Abschnitt 6.3 beschrieben wurde, werden Verhaltensänderungen durch Overriding durch das JDT Refactoring nicht geprüft. Daher gab es keine Basismethode, die adaptiert werden konnte und es wurde die Methode `checkOverriding()` entwickelt, die in allen Subtypen des Zieltyps nach einer Methode mit gleichem Namen und gleicher Signatur sucht. Falls diese nicht durch das Refactoring gelöscht oder verschoben[6] werden soll, wird eine Overriding Fehlermeldung erzeugt (Vorbedingung 8). Die erarbeitete Lösung behebt bereits den Mangel des JDTs und lässt sich nach einem Bugfix in der Basisklasse mit einer Replace-Callinbindung an die neue Basismethode binden, um diese zu ersetzen.

needsVisibilityAdjustment()

In dieser Methode wird geprüft ob eine verschobene Methode oder ein verschobenes Feld eine Sichtbarkeitsanpassung benötigt. Wie zuvor erwähnt löst diese Anpassung Probleme bei verschiedenen Vorbedingungen. Callinmethoden dürfen jedoch keine Sichtbarkeit deklarieren und müssen daher davor bewahrt werden. Daher wird in der Methode zusätzlich geprüft ob es sich um eine Callinmethode handelt, um dann ein **`false`** zurück zu liefern.

[6] Die JDT Implementierung des Refactorings bietet zusätzlich die Möglichkeit Methoden und Felder aus anderen Subtypen des Zieltyps zu verschieben bzw. zu löschen.

getDestinationOTTypeHierarchy()

Vorbedingung 9 lockert die Vorbedingungen des reinen Java Refactorings und erlaubt zusätzlich implizite Supertypen als Ziel. Daher tauscht diese Callinmethode die `TypeHierarchy` der Basismethode durch eine `OTTypeHierarchy` aus. Die Methode wird auch vom Wizard verwendet, um ein Auswahlmenü für den Zieltyp zu erzeugen. Da die JDT Implementierung nicht nur direkte Supertypen des deklarierenden Typs als Ziel zulässt, stellt der Wizard bereits eine Auswahlbox mit mehreren Zielen bereit.

getCandidateTypes()

Diese Methode liefert alle gültigen Ziele für das Refactoring und muss für die Umsetzung der Vorbedingung 9 ebenfalls angepasst werden.

getSkippedSuperTypes()

In dieser Methode werden Supertypen berechnet, die durch das Pull Up Refactoring übersprungen wurden. Falls die Verschiebung innerhalb der impliziten Vererbungshierarchie stattfindet, muss die Berechnung angepasst werden. Dazu wird die Schnittmenge der impliziten Subtypen der Zielrolle und der impliziten Supertypen der deklarierenden Rolle gebildet.

Für explizite Zieltypen kann weiterhin die Basismethode verwendet werden. Durch die Verwendung der `OTTypeHierarchy` entsteht jedoch ein Problem, da diese mehrere Supertypen für einen Typ bereitstellt, wird beim Aufruf der Methode `getSuperclass()` eine `UnsupportedOperationException` geworfen. Aus diesem Grund wurde ein Adapter für die Klasse `OTTypeHierarchy` entwickelt, der `getSuperclass()` Aufrufe durch `getExplicitSuperclass()` Aufrufe ersetzt. Der Adapter wurde noch für die Adaption weiterer Basismethoden verwendet, die ebenfalls die Methode `getSuperclass()` verwenden.

6.5.3. Linearisierung der OT/J Supertyphierarchie

Immer wenn die Methode `getSuperclass()` aufgerufen wird, deutet dies auf eine Diskrepanz zwischen der OT/J Vererbungshierarchie und der herkömmlichen Java Vererbungshierarchie hin. Im Wizard des Pull Up Refactorings wird zum Beispiel eine baumartige Ansicht der Vererbungshierarchie erzeugt, die nicht geeignet ist eine OT/J Vererbungshierarchie darzustellen. Da sich die Annahme, dass ein Typ nur einen direkten Supertyp hat durch weite Teile des Pull Up Refactorings zieht, konnten im Rahmen dieser Arbeit nicht alle Stellen adaptiert werden.

Daher wurde der zuvor beschriebene Adapter entwickelt, um eine Lösung bereitzustellen, die die Basisfunktionalität des Refactorings nicht beeinträchtigt.

Für eine vollständige Lösung wären weitere Adaptionen notwendig. Da die Infrastruktur des JDTs nur auf einen direkten Supertyp ausgelegt ist, stellt sich die Frage, ob es möglich wäre die OT/J Typhierarchie zu linearisieren. Als Vorbild könnten die Ansätze der Mixin Vererbung dienen [BC90], für die der *C3* Algorithmus verwendet wird um eine Linearisierung zu berechnen [Ern99].

Falls es in der OT/J Vererbungshierarchie durch Mehrfacherben zu Konflikten kommt, schreibt die OTJLD Vererbungs-Reihenfolge vor ([HHM09], §1.5. (e)). Auf Grundlage dieser Reihenfolge könnte eine Linearisierung durchgeführt werden, die implizite Supertypen vor explizite Supertypen stellt. Bei Konflikten zwischen mehreren direkten impliziten Supertypen, werden die Supertypen, die sich innerhalb des gleichen Teams befinden bevorzugt.

In Abschnitt 2.2.7 auf Seite 14 wurde in Abbildung 2.3 ein Beispiel für eine Rolle mit mehreren direkte impliziten Superrollen gegeben. Dabei handelt es sich um eine *Diamond Inheritance*, bei der ein Typ von zwei verschiedenen Supertypen erbt, die wiederum von dem gleichen Typ erben. Eine Linearisierung auf Grund von §1.5. (e) für die Supertypen der Rolle `SubTeam.T2.Role` würde die folgende Liste ergeben:

{`SubTeam.T1.Role`, `SuperTeam.T2.Role`, `SuperTeam.T1.Role`}

Im JDT wird die Methode `getSuperClass()` häufig zum Iterieren über alle Supertypen eines Typs verwendet. Dies würde bei einer Linearisierung nur funktionieren wenn der Bezugspunkt der Linearisierung bekannt ist. Andernfalls würden Supertypen ausgelassen werden, da die Linearisierung eines Supertyps nicht unbedingt eine Teilmenge der ursprünglichen Linearisierung ist. Wollte man beispielsweise über alle Supertypen der Rolle `SubTeam.T2.Role` iterieren wollen, würde die Rolle `SuperTeam.T2.Role` übersprungen werden.

1. Schritt: `getSuperClass(SubTeam.T2.Role) = SubTeam.T1.Role`
2. Schritt: `getSuperClass(SubTeam.T1.Role) = SuperTeam.T1.Role`

Die Rolle `SuperTeam.T2.Role` ist unabhängig von der Rolle `SubTeam.T1.Role` und somit nicht in der Linearisierung ihrer Supertypen enthalten. Um alle Supertypen zu erhalten müsste in jedem Aufruf der Bezugspunkt `SuperTeam.T2.Role` bekannt sein.

Um die Methode `getSuperClass()` mit einer Linearisierung der OT/J Supertypen zu ersetzen, müsste eine Möglichkeit gefunden werden der Methode den richtigen Kontext bereitzustellen, ohne ihn als Parameter zu übergeben. Eine solche für den Klienten transparente Lösung ließe

sich technisch leicht mit einem Team umsetzen. Schwierig wäre hingegen die Festlegung einer sinnvollen Semantik, die für alle Klienten funktioniert. Weitere Untersuchungen über die Umsetzbarkeit einer solchen Lösung konnten aus zeitlichen Gründen nicht mehr im Rahmen dieser Arbeit durchgeführt werden.

6.6. Adaptierung von Push Down

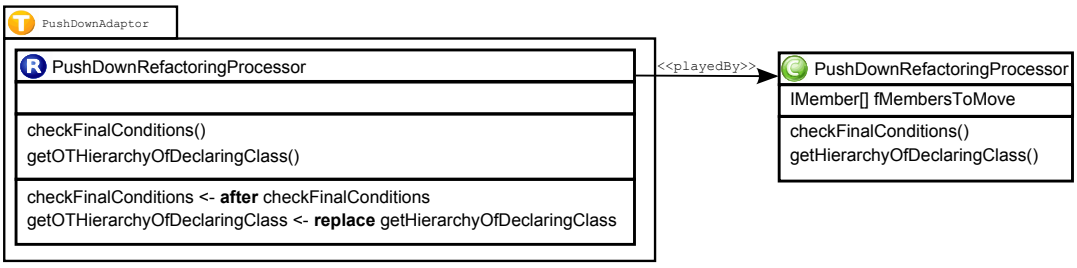

Abbildung 6.4.: Klassendiagramm des Push Down Adapters

Auf Grundlage der in den Abschnitten 4.5.5 und 4.5.6 formulierten Vorbedingungen wurde das Push Down Refactoring adaptiert.

6.6.1. Bereits behandelte Vorbedingungen und Änderungen

- In einem Push Down Refactoring werden bereits Referenzen innerhalb des deklarierenden Typs gesucht, wodurch Vorbedingung 5 abgedeckt wird.
- Ein Teil der Vorbedingung 4 muss nicht behandelt werden, da diese auch nicht vom OT/J Compiler berücksichtigt werden (siehe Abs. 6.2.3).

6.6.2. Durchgeführte Adaptionen

checkFinalConditions()

- Das Push Down Refactoring bietet die Möglichkeit für verschobene Methoden eine abstrakte Methodendeklaration im deklarierenden Typ zu erstellen. Falls keine abstrakte Deklaration erstellt wird, darf die verschobene Methode nicht in einer Methodenbindung referenziert werden. Dazu werden mit Hilfe der Java Suche alle Referenzen der zu verschiebenden Methoden und Felder gesucht. Die Methode `checkForAspectBindings()`

führt die Suche durch und erzeugt entsprechende Fehlermeldungen. Dadurch werden die Vorbedingungen 1, 2 und 3 geprüft.

- Base Guard Referenzen werden nicht durch die JDT Implementierung gefunden, da keine Referenzen außerhalb des deklarierenden Typs gesucht werden (siehe JDT Mängel in Abs. 6.3). Daher wird in den Vorbedingungen zusätzlich nach Referenzen außerhalb des deklarierenden Typs gesucht, um Vorbedingung 6 zu prüfen.
- Für Vorbedingung 7 kann die Methode `checkOverloadingAndAmbiguity()` wiederverwendet werden. Dazu wird für alle verschobenen privaten[7] Methoden geprüft, ob diese mehrdeutige Methodenbindungen im Zieltyp produzieren.
- Die Methode `checkShadowingFieldInImplicitHierarchy()` prüft Vorbedingung 4, indem für Zielrollen entlang der impliziten Vererbungshierarchie nach Feldern mit dem gleichen Namen gesucht wird.
- Um das in Vorbedingung 9 erwähnte Overriding zu verhindern, muss in allen direkten expliziten Subtypen, die eine Rolle sind, eine Überprüfung auf Overriding stattfinden. Dazu sucht die Methode `checkOverriding()` in den impliziten Superrollen des Ziels nach einer Methode mit gleichem Namen und gleicher Signatur wie die der zu verschiebenden Methode.
- Für die Implementierung wurde entschieden, dass Phantomrollen nicht durch das Refactoring materialisiert werden. Daher muss eine Fehlermeldung gegeben werden, wenn die zu verschiebenden Felder und Methoden in eine Phantomrolle geschoben werden müssten. Dazu prüft die Methode `checkForDirectPhantomSubRoles()` ob direkte Subtypen existieren, die vom Typ `PhantomType` sind. Wird ein solcher Typ gefunden, wird eine entsprechende Fehlermeldung erzeugt (Vorbedingung 10).

getOTHierarchyOfDeclaringClass()

Um auch implizite Subtypen anzupassen, ersetzt diese Rollenmethode die Basismethode `getHierarchyOfDeclaringClass()`. Dazu wird die Methode mit einer Replace-Callinbindung an die Basismethode gebunden und liefert die `OTTypeHierarchy` des deklarierenden Typs zurück (Vorbedingung 7).

[7] Bei Methoden mit einer größeren Sichtbarkeit wären die Mehrdeutigkeiten bereits vor dem Refactoring vorhanden gewesen.

6.7. Probleme bei der Adaptierung des RippleMethodFinder2

Die Klasse `RippleMethodFinder2` ist eine erweiterte Version einer Hilfsklasse, die für die Suche von verwandten Methoden eingesetzt wird. Der Unterschied zum Vorgänger besteht darin, dass als Grundlage eine auf Regionen basierende Typhierarchie verwendet wird. Eine entsprechende OT/J Implementierung existiert bisher nicht.

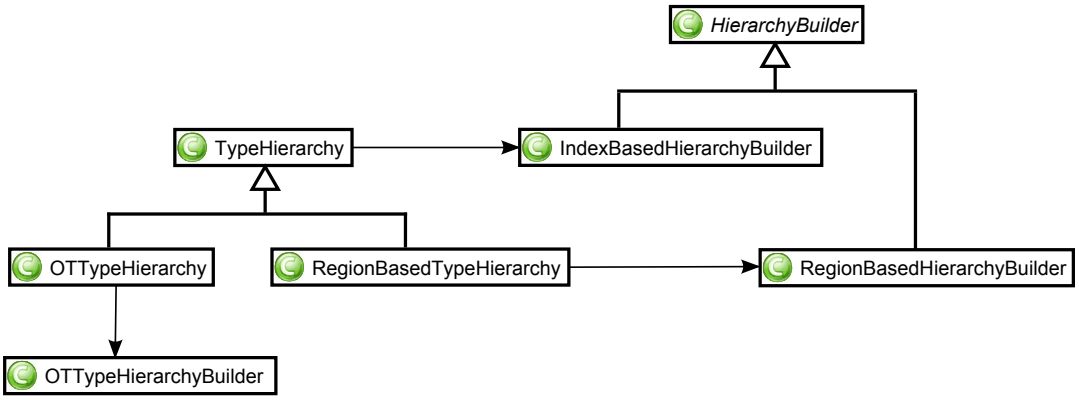

Abbildung 6.5.: Vererbungshierarchie der verschiedenen Typhierarchien

Um den `RippleMethodFinder2` auch in adaptierten Refactorings einsetzen zu können, müsste eine Regionen basierte OT/J Typhierarchie entwickelt werden. Abbildung 6.5 zeigt ein Klassendiagramm der für eine Adaption relevanten Klassen. Es ist zu sehen, dass die Typhierarchie Klassen nur die Datenstruktur bilden. Die eigentliche Konstruktion der Hierarchien ist in Klassen vom Typ `HierarchyBuilder` ausgelagert. In diesem Punkt liegt auch die Problematik, die eine schnelle Adaption verhindert. Die Klasse `OTHierarchyBuilder` verwendet eine eigene Strategie und erbt nicht von der abstrakten Klasse `HierarchyBuilder`. Dies macht eine einfache Umsetzung mit Hilfe von Vererbungsmechanismen unmöglich.

Eine Umsetzung würde eine gewisse Einarbeitungszeit in die Typhierarchie benötigen, da gewisse Teile neu Entwickelt werden müssten. Aus Zeitgründen wurde auf diese Einarbeitung verzichtet, um dafür mehr Zeit in die Entwicklung neuer OT/J spezifischer Refactorings zu investieren (siehe Kapitel 7).

Um trotzdem eine möglichst gute Lösung zu erhalten, wurde statt der Regionen basierten Hierarchie vorerst die normale OT/J Hierarchie verwendet. Dies stellt vorerst eine Notlösung dar, die jedoch besser funktioniert als die Verwendung der Regionen basierten Typhierarchie für Java. Dadurch funktionieren alle Tests für Rename Method der JDT und OT/J Testsuite.

Lediglich ein Testfall des Introduce Indirection Refactorings schlägt fehl, da keine Warnung produziert wird.

Wenn eine Regionen basierte OT/J Hierarchie entwickelt werden würde, ließe sie sich leicht im `RippleMethodFinder2` verwenden. Die `RippleMethodFinder2` Klasse wird an nur fünf Codestellen verwendet und wird von den JDT Refactorings: Rename Method, Rename Type, Introduce Indirection, Change Type und Change Signature benötigt.

6.8. Probleme bei der Adaptierung von Move Method

Ursprünglich sollte im Rahmen der Arbeit auch das Move Method Refactoring adaptiert werden. Bei genauerer Analyse des JDT Basiscodes wurde jedoch festgestellt, dass die Klasse `MoveInstanceMethodRefactoring` bisher keine Vorbedingungen prüft (siehe Abs. 6.3). Aus diesem Grund wäre eine Adaptierung schwer umsetzbar gewesen, da keine Wiederverwendung von Basismethoden möglich wäre. Die Entwicklung einer Lösung, die auch Java Vorbedingungen prüft würde den Rahmen dieser Arbeit übersteigen und bei entsprechenden Anpassungen durch die Eclipse Entwickler überflüssig werden.

6.9. Strategie für nicht adaptierte Refactorings

Da im Rahmen dieser Arbeit nur ein Teil der JDT Refactorings adaptiert werden konnte, kann es leicht zu Irritationen bei der Verwendung nicht adaptierter Refactorings kommen. Um den Benutzer vor nicht adaptierten Refactorings zu warnen, wurde ein Adapter für alle nicht adaptierten Refactorings erstellt. Dieser erzeugt beim Prüfen der finalen Bedingungen einen `RefactoringStatus` vom Typ `Info`, der darüber informiert, dass das Refactoring noch nicht alle neuen Regeln von OT/J beachtet. Der Hinweis wird nur erzeugt, wenn Teams im aktuellen Projekt oder abhängigen Projekten existieren[8]. Ohne Teams kann es auch keine Rollen und die darin möglichen OT/J Konzepte geben.

Der Vorteil eines Info Status gegenüber einer Warnung ist, dass dieser nicht die Testfälle der vorhandenen JDT Refactoring Testsuite beeinflusst. Diese erwarten für positive Testfälle, dass das Refactoring keine Warnungen produziert.

[8] Es bleibt zu überprüfen, ob dieses Vorgehen auch Plug-ins erfasst, die das betreffende Projekt über eine Aspektbindung verändern.

6.10. Bewertung der entwickelten Lösung

Im Rahmen der Diplomarbeit wurden die drei Refactorings Rename Type, Pull Up und Push Down adaptiert. Insbesondere das Rename Type Refactoring zeigt exemplarisch eine möglichst vollständige Adaption. Da die Adaptionen selbst mit OT/J durchgeführt wurden, stellt sich die Frage wie robust die durchgeführten Adaptionen sind. Dazu ist es notwendig, dass eine Evolution des Basiscodes den Aspektcode nur geringfügig beeinflusst und weiterhin funktionsfähig bleibt. Ein weiterer Aspekt ist die Erhaltung der Basisfunktionalität, die für herkömmliche Java Refactorings ein unverändertes Verhalten zeigen muss.

6.10.1. Erhaltung der Basisfunktionalität

Einige der entwickelten Callinmethoden enthalten Guards oder eine Bedingung vor dem eigentlichen Code, der prüft ob es sich um ein OT/J Element handelt und verhindert dadurch die Ausführung des Rollencodes bei der Durchführung eines reinen Java Refactorings. Zusätzlich wurde nach der Entwicklung jedes Refactoring Adapters die gesamte JDT Refactoring Testsuite ausgeführt, um sicherzustellen, dass die Adaption kein Verhalten des Basiscodes beschädigt.

Ein weiteres Kriterium stellt die Art der Adaption dar. Bei Before- und After-Callinbindungen haben die Veränderungen einen additiven Charakter, bei dem auf jeden Fall der Basiscode ausgeführt wird. Für Replace-Callinbindungen muss genauer untersucht werden, ob die Basisfunktionalität erhalten bleibt. Der Compiler hilft dabei, indem er Warnungen für fehlende Base Calls ausgibt. Die entwickelte Lösung hat keine Base Call Warnungen, bis auf die mehrfache Ausführung der in Listing 6.1 dargestellten Base Call Schleife, deren Notwendigkeit bereits erläutert wurde.
Die meisten verwendeten Replace-Callinbindungen ließen sich in eine After-Callinbindungen umwandeln, indem ein Result Parameter Mapping angegeben wird und dadurch die erste Zeile, die einen Base Call enthält ersetzt wird. Listing 6.2 und 6.3 zeigen ein Beispiel, bei dem die Adaptierung der Methode `getCandidateTypes()` in den zwei äquivalenten Varianten dargestellt ist.

```
1  callin IType[] getCandidateTypes(RefactoringStatus status, IProgressMonitor monitor) {
2    IType[] jdtCandidates = base.getCandidateTypes(jdtStatus, monitor);
3
4    // add OT candidates
5    jdtCandidates = ...
6
7    return jdtCandidates;
8  }
9  getCandidateTypes <- replace getCandidateTypes;
```

Listing 6.2: Adaptierung durch eine Replace-Callinbindung

```
1  private IType[] getCandidateTypes(RefactoringStatus status, IProgressMonitor monitor, IType[] jdtCandidates) {
2    // add OT candidates
3    jdtCandidates = ...
4
5    return jdtCandidates;
6  }
7
8  IType[] getCandidateTypes(RefactoringStatus status, IProgressMonitor monitor, IType[] jdtCandidates)
9  <- after
10 IType[] getCandidateTypes(RefactoringStatus status, IProgressMonitor monitor) with{
11   status <- status,
12   monitor <- monitor,
13   jdtCandidates <- result
14 }
```

Listing 6.3: Äquivalente Adaptierung durch eine After-Callinbindung

6.10.2. Evolutionsfähigkeit

Die wichtigsten Adaptionen betreffen die beiden öffentliche Methoden `createChange()` und `checkFinalConditions()`, die in der abstrakten Klasse `Refactoring` deklariert sind. Da diese Methoden einen Teil der öffentlichen Schnittstelle aller LTK Refactorings darstellt, ist es unwahrscheinlich, dass sich die Semantik oder Signatur der Methoden verändert. Durch den bereits erwähnten additiven Charakter der Adaptionen, wird die Funktionsweise des Aspektcodes nicht durch Änderungen im Methodenrumpf beeinträchtigt.

Um die Performanz zu verbessern und die bestehende Infrastruktur zu nutzen, wurden auch private Methoden wiederverwendet oder adaptiert. Hier muss bei einer Evolution geprüft werden, ob die Signatur und Semantik einer Methode weiterhin kompatibel zu existierenden Methodenbindungen ist. Eine Veränderung der Signatur würde schnell durch einen Fehler beim Übersetzen auffallen, wohingegen Veränderungen der Semantik schwieriger zu bestimmen sind. Da Methodennamen in der Regel beschreibende Namen besitzen, ist es unwahrscheinlich, dass die Semantik einer Methode verändert würde, ohne sie umzubenennen oder größere strukturelle Veränderungen durchzuführen, die wiederum zu Fehlern beim Übersetzen führen würden.

In jedem Fall sind kleinere Anpassungen nötig, um auf eine Evolution des Basiscodes zu reagieren. Durch eine gute Abgrenzung zwischen Basis- und Aspektcode ist es jedoch möglich, den Bedarf an Änderungen innerhalb von Methoden zu minimieren und dadurch auf reine Anpassungen von Methodenbindungen zu reduzieren. In Kapitel 8 werden die Anpassungen einer durchgeführten Migration genauer beschrieben und diskutiert.

7. Entwicklung von ObjectTeams/Java Refactorings

Der erste Teil dieses Kapitels stellt die neu entwickelten Refactorings vor. Im zweiten Teil des Kapitels wird dann die praktische Umsetzung von zwei OT/J Refactorings mit Hilfe des LTKs beschrieben.

7.1. Theoretische Grundlagen

Die in [TOHJ99] beschriebene Flexibilität durch die Zerlegung einer Software in verschiedene *Hyperslices* kann durch Refactoring unterstützt werden. Eine Rolle stellt im Prinzip eine Spezialisierung der Basis dar, da sie Methoden wiederverwendet (Calloutbindungen) und überschreibt (Callinbindungen). Ähnlich wie die Refactorings Pull Up und Push Down, wären OT/J spezifische Refactorings denkbar, die dem Entwickler dabei unterstützen Basiscode in eine Rolle zu verschieben oder umgekehrt. In den folgenden Abschnitten sind verschiedene Refactorings beschrieben, die bei solchen Umstrukturierungen hilfreich sein können.

7.1.1. Verhaltenserhaltung bei Precedence Ausdrücken

Bevor einige OT/J Refactorings im Detail vorgestellt werden, wird das Konzept der Callin Precedence genauer untersucht (siehe Abs. 2.2.4). Beim Verschieben von Code zwischen Rolle und Basis muss sichergestellt werden, dass sich die Ausführungsreihenfolge von Callinmethoden nicht verändert. Da Precedence Deklarationen nur in Rollen zur Verfügung stehen, ergeben sich Einschränkungen beim Verschieben von Code zwischen Rolle und Basis. Das folgende Beispiel enthält je zwei Callinbindungen pro Callintyp (**`before`**, **`after`** und **`replace`**), sowie eine Precedence Deklaration für jedes Paar.

```
1  public team class HWTeam {
2    protected class Role playedBy HelloWorld{
3      precedence firstBefore, secondBefore;
4      precedence firstAfter, scondAfter;
5      precedence firstReplace, secondReplace;
6
7      void b1(){
8        System.out.println("1st before");
9      }
10     void b2(){
11       System.out.println("2nd before");
12     }
13     firstBefore: b1 <- before print;
14     secondBefore: b2 <- before print;
15
16     void a1(){
17       System.out.println("1st after");
18     }
19     void a2(){
20       System.out.println("2nd after");
21     }
22     firstAfter: a1 <- after print;
23     scondAfter: a2 <- after print;
24
25     callin void r1(){
26       System.out.println("1st replace");
27       base.r1();
28       System.out.println("1st replace");
29     }
30     callin void r2(){
31       System.out.println("2nd replace");
32       base.r2();
33       System.out.println("2nd replace");
34     }
35     firstReplace: r1 <- replace print;
36     secondReplace: r2 <- replace print;
37   }
38 }
```

Listing 7.1: HWTeam.java

```
1  public class HelloWorld {
2    public static void main(String[] args) {
3      within(new HWTeam()){
4        new HelloWorld().print();
5      }
6    }
7    public void print(){
8      System.out.println("Hello World!");
9    }
10 }
```

Listing 7.2: HelloWorld.java

```
1  1st before
2  2nd before
3  1st replace
4  2nd replace
5  Hello World!
6  2nd replace
7  1st replace
8  1st after
9  2nd after
```

Listing 7.3: Ausgabe des Programms

Listing 7.2 ist ein einfaches Hello World Programm. In der Methode `main()` wird ein `HelloWorld` Objekt erzeugt und die Methode `print()` aufgerufen, die die „Hello World!“ Ausgabe erzeugt (Zeile 4). Das ganze wird innerhalb eines Within-Statements ausgeführt, das eine neue Teaminstanz vom Typ `HWTeam` erzeugt und aktiviert (Zeile 3).

Die Deklaration des aktivierten Teams ist in Listing 7.1 aufgeführt. Das Team enthält eine Rolle `Role`, die in einer playedBy-Relation an die Klasse `HelloWorld` gebunden wird (Zeile 2). Die Methoden `b1()` und `b2()` werden in den den Before-Callinbindungen `firstBefore` und `secondBefore` an die Basismethode `print()` gebunden. In Zeile 3 wird in einer Precedence Deklaration die Reihenfolge für die beiden Callinbindungen festgelegt. Analog werden die zwei After- und Replace-Callinbindungen deklariert.

Die Ausgabe des Programms ist in Listing 7.3 dargestellt. Sie visualisiert die Ausführungsreihenfolge der verschiedenen Methoden. Aus der Ausgabe können einige Regeln abgeleitet werden, die nicht explizit in der OTJLD beschrieben sind.

- Zwischen Callinbindungen verschiedenen Typs gibt es eine *natürliche Ausführungsreihenfolge*, wenn diese an die gleiche Basismethode gebunden werden. Zuerst werden die Rollenmethoden von Before-Callinbindungen ausgeführt, dann die von Replace-Callinbindungen und zuletzt die von After-Callinbindungen.
- Replace-Callinbindungen, die sich am Ende einer Precedence Deklaration befinden, werden zuletzt ausgeführt, d. h. unmittelbar vor und nach dem Aufruf der Basismethode.
- Das gleiche gilt für Before-Callinbindungen, sie werden in der gleichen Reihenfolge wie in der Precedence Deklaration ausgeführt und damit die Rollenmethode der Callinbindung an letzter Stelle direkt vor dem Aufruf der Basismethode ausgeführt, falls keine weiteren Replace-Callinbindungen existieren.
- Die erste After-Callinbindung innerhalb einer Precedence Deklaration wird als erstes und damit unmittelbar nach der Basismethode aufgerufen, falls keine weiteren Replace-Callinbindungen existieren. Es besteht ein Ticket, in dem diskutiert wird, ob dieses Verhalten beibehalten werden soll oder die Methode als letztes aufgerufen werden sollte, wie bei der Priorisierung zwischen mehreren Teams [obj09]. Da die Diskussion zum Zeitpunkt der Arbeit noch nicht beendet war, wird das hier beobachtete Verhalten als Grundlage für die Formulierung von Vorbedingungen verwendet.

Aus den Beobachtungen ergibt sich, dass das Verschieben von Rollencode in die Basismethode nur unter bestimmten Bedingungen möglich ist. Werden diese Bedingungen verletzt, verändert das Refactoring die Ausführungsreihenfolge gebundener Rollenmethoden. Die sich daraus ergebenden Bedingungen sind in den Vorbedingungen der Inline Callin und Extract Callin Refactorings genauer erläutert (siehe Abs. 7.1.2 und 7.1.3).

7.1.2. Inline Callin

Eine Callinbindung wird aufgelöst, indem der Code der gebundenen Rollenmethode in die gebundene Basismethode verschoben wird. Ein Inline Callin Refactoring kann nicht in jedem Fall eine Erhaltung der Semantik garantieren, da die Teamaktivierung nach einem solchen Refactoring keinen Einfluss mehr auf die Rollenmethode hat. Das Refactoring stellt ein Refactoring der vierten Dimension dar, da Aspektcode aus der Rolle in die Basis verschoben wird (siehe Dimensionen von aspektorientiertem Refactoring in Abs. 4.1).

Parameter

- Eine Rollenmethode R.

- Eine Callinbindung C.
- Eine Basismethode B.
- Ein Name N für die integrierte Rollenmethode (optional[1]).

Vorbedingungen

1. Die Methode R muss durch die Callinbindung C an die Basismethode B gebunden sein.
2. Falls ein Precedence Ausdruck existiert, der die Callinbindung C oder die deklarierende Rolle referenziert, müssen folgende Voraussetzungen erfüllt sein, um die Semantik des Programms zu erhalten.
   - C ist vom Typ **`before`**:
     C muss an letzter Stelle der Precedence Deklaration stehen. Nur der letzte Methodenaufruf findet unmittelbar vor Ausführung der Basisfunktionalität statt und lässt sich dadurch in die Basismethode integrieren.
   - C ist vom Typ **`after`**:
     C muss an erster Stelle der Precedence Deklaration stehen. Nur dann kann die gebundene Rollenmethode in die Basismethode integriert werden, da diese unmittelbar nach dem Basiscode ausgeführt wird.
   - C ist vom Typ **`replace`**:
     C muss an letzter Stelle der Precedence Deklaration stehen. Nur die zuletzt ausgeführte Callinmethode wird unmittelbar vor bzw. nach[2] der Basismethode ausgeführt.

   Siehe Verhaltenserhaltung bei Precedence Ausdrücken in Abs. 7.1.1.
3. Da Rollenmethoden, die in Replace Callinbindungen gebunden sind, immer vor Rollenmethoden, die in Before- oder After-Callinbindungen gebunden sind ausgeführt werden, darf B nicht in einer Replace-Callinbindung gebunden sein, wenn C vom Typ **`before`** oder **`after`** ist. Siehe Verhaltenserhaltung bei Precedence Ausdrücken in Abs. 7.1.1.
4. Wenn die Rollenmethode R oder die Callinbindung C ein Guard Predicate besitzt, kann das Refactoring nicht durchgeführt werden. Es muss zuerst ein Inline Guards Refactoring (siehe Abs. 7.1.4) durchgeführt werden.

[1] Wenn kein Name angegeben wird, kann der ursprüngliche Name von R verwendet werden
[2] Der genaue Ausführungszeitpunkt wird durch die Position des Base Calls innerhalb der Callinmethode bestimmt.

5. Alle Bezeichner, die in der Rollenmethode R verwendet werden, müssen im Sichtbarkeitsbereich der Basis (die deklarierende Klasse von B) sichtbar sein. Falls Felder oder Methoden der Rolle verwendet werden, können diese mit einem Move Field to Base bzw. Move Method to Base Refactoring (siehe Abs. 7.1.5 und 7.1.7) in die Basis integriert werden.

6. Die Rollenmethode R darf nicht in einer Callinbindung als Basismethode gebunden sein, da sonst die Verbindung zu der Basismethode B verloren geht und die an R gebundene Methode nicht mehr ausgeführt wird. Alternativ könnte R mit einem leeren Methodenrumpf bestehen bleiben, um die Verbindung des gebundenen Callins zu erhalten. Dies wäre nur möglich wenn alle Callinbindungen, in denen R als Rollenmethode gebunden ist, umgewandelt werden und R nicht explizit aufgerufen wird.

7. Der neue Name N, bzw. der Name der Rollenmethode, darf in der Basis keine mehrdeutigen Methodenbindungen erzeugen, Overriding verursachen oder von einer Methode mit der gleichen Signatur der Rollenmethode verwendet werden (OTI5. und OTI6.).

8. Die Rollenmethode R darf kein Declared Lifting enthalten (OTL8.).

Programmänderungen

1. Die Callinbindung C muss nach dem Refactoring gelöscht werden. Falls in der Callinbindung mehrere Basismethoden gebunden sind, muss nur die Basismethode B aus der Deklaration entfernt werden (OTL6.(c)).

2. Wenn die Callinbindung C in einem Precedence Ausdruck referenziert wird, muss die Referenz aus diesem entfernt werden.

3. Um Methodenreferenzen auf die Basismethode B zu erhalten, wird eine *Wrapper Methode* erzeugt, die den gleichen Namen und Signatur wie die Basismethode B hat. Um einen Namenskonflikt zu verhindern, wird die Methode B umbenannt.

4. Die Rollenmethode R wird in die Basis kopiert und bekommt den Namen N. Dabei müssen Referenzen auf Callout Methoden durch die gebundenen Basis Methoden und Felder ersetzt werden (OTL6.(b),(d)). Falls die Calloutbindungen ein Parameter Mapping enthalten, muss dieses beim Ersetzen übertragen werden (OTL7.(a)).

5. Falls es sich bei der Rollenmethode R um eine Callinmethode handelt, muss der **`callin`** Modifier von der Kopie entfernt werden[3] (OTL6.(g)).

[3]Bei der gebundenen Basis kann es sich zwar um eine Rolle handeln, jedoch muss die Methode B aufgerufen werden, was bei einer Callinmethode nicht möglich ist.

6. Außerdem können beim Kopieren einer Callinmethode *getunnelte Parameter* (siehe §4.3.(d) in [HHM09]) für einen Base Call benötigt werden. Aus diesem Grund müssen zusätzliche Parameter der Basismethode B an die Signatur der kopierten Rollenmethode R gehängt werden und an vorhandene Base Calls weitergeleitet werden.

7. Für Callinmethoden müssen zusätzlich Base Calls durch Aufrufe der umbenannten Basismethode B ersetzt werden. Falls Parameter Mappings existieren, müssen diese rückwärts übertragen werden.

8. In der Wrapper Methode wird der Typ der Callinmethode (**`before`**, **`after`** oder **`replace`**) entsprechend nachgebildet. Dazu wird die umbenannte Methode B aufgerufen und davor, danach oder stattdessen die kopierte Rollenmethode B aufgerufen. Falls Parameter Mappings existieren, müssen diese in den Aufruf der kopierten Rollenmethode B übertragen werden (OTL7.(b)).

9. Für Result Parameter Mappings oder After-Callinbindungen, die an eine Basismethode mit Rückgabewert gebunden sind, muss eine temporäre Variable zum Zwischenspeichern des Resultats der Basismethode erzeugt werden.

Beispiel

In den Listings 7.4 und 7.5 ist exemplarisch ein Inline Callin Refactoring dokumentiert. Die Rollenmethode `n()` soll für die After-Callinbindung (Zeile 4 Vorher) in die Basismethode `m()` integriert werden. Dazu wird die Rollenmethode in die Basis kopiert (Zeile 14 Nachher) und die Ausführungsreihenfolge in der Wrapper Methode in (Zeilen 8-12 Nachher) rekonstruiert. Das Beispiel zeigt auch die Verwendung einer temporären Variable um das Resultat der Basismethode zu sichern (Zeile 9 Nachher).

```
1  public team class T{
2    protected class R playedBy B{
3      protected void n(){}
4      n <- after m;
5    }
6  }
7
8  public class B{
9    public int m(){
10     return 1;
11   }
12 }
```

Listing 7.4: Vor Inline Callin

⇒

```
1  public team class T{
2    protected class R playedBy B{
3      protected void n(){}
4    }
5  }
6
7  public class B{
8    public int m(){
9      int baseResult = base_m();
10     n();
11     return baseResult;
12   }
13
14   private void n(){}
15
16   public int base_m(){
17     return 1;
18   }
19 }
```

Listing 7.5: Nach Inline Callin

7.1.3. Extract Callin

Das Extract Callin Refactoring bildet die Umkehroperation zu einem Inline Callin Refactoring. Dabei soll ein Teil einer Methode in eine Rolle verschoben werden und durch eine Callinbindung mit der Basismethode verbunden werden. Ähnlich wie bei einem Inline Callin Refactoring, kann die Erhaltung der Semantik des Programms nicht in jedem Fall garantiert werden. Die Ausführung des extrahierten Codes hängt nach dem Refactoring von der Teamaktivierung ab. Das Refactoring stellt ein Refactoring der dritten Dimension dar, da Code aus der Basis in die Rolle verschoben wird (siehe Dimensionen von aspektorientiertem Refactoring in Abs. 4.1).

Ein Extract Callin Refactoring ist in drei verschiedenen Varianten durchführbar:

1. Extraktion eines Before-Callins
   Dazu muss das erste Statement der Methode ein Methodenaufruf einer Methode sein, die in der gleichen Klasse deklariert ist.

2. Extraktion eines After-Callins
   Das letzte Statement der Methode muss ein Methodenaufruf einer Methode sein, die in der gleichen Klasse deklariert ist. Außerdem darf der Rumpf einer Methode nicht in einer Verzweigung vor der Ausführung dieses Methodenaufrufs verlassen werden.

3. Extraktion eines Replace-Callins
   Ein Replace-Callin kann immer erzeugt werden. Es sind dabei zwei verschiedene Varianten möglich:

   a) Es wird eine Callinmethode erzeugt, die einen Base Call enthält und die Basismethode in einer Replace-Callinbindung ersetzt.

   b) Der Rumpf der Basismethode wird in eine neu erzeugte Callinmethode verschoben, die die Basismethode in einer Replace-Callinbindung ersetzt.

Die beschriebenen Vorbedingungen für die verschiedenen Varianten der Extraktion mögen auf den ersten Blick sehr restriktiv erscheinen, da bereits eine Methode existieren muss, die den zu extrahierenden Code enthält. Eine solche Struktur lässt sich jedoch leicht herstellen, indem vor dem Extract Callin Refactoring ein Extract Method Refactoring durchführt wird. Dadurch lassen sich beliebige Statements zusammenfassen und anschließend in eine Rolle verschieben.

Parameter

- Eine Basismethode B.

- Eine Rolle R.
- Ein Name N für die extrahierte Basismethode.
- Ein Callintyp C (**`before`**, **`after`** oder **`replace`**).
- Falls C den Wert **`replace`** hat, muss zusätzlich ein Flag F angegeben werden, das bestimmt, ob der Basiscode in die Rollenmethode kopiert werden soll.

Vorbedingungen

1. Aus den oben beschriebenen Varianten der Extraktion lassen sich Vorbedingungen ableiten, die prüfen, ob der gewünschte Callintyp C extrahiert werden kann.
2. Falls der Callintyp C **`before`** oder **`after`** ist, dürfen keine Replace-Callinbindungen existieren, die an die Basismethode B gebunden sind, da ansonsten die dort gebundenen Rollenmethoden vor der extrahierten Basismethode ausgeführt werden würden (siehe Abs. 7.1.1).
3. Die Rolle R muss den deklarierenden Typ von B in einer playedBy-Relation als Basis binden (OTL6.(a),(c)).
4. Der gewählte Name N darf in der Rolle kein Overriding verursachen, einen Namenskonflikt mit einer existierenden Methode auslösen oder zu mehrdeutigen Methodenbindungen führen (OTI5. und OTI6.).
5. Alle Bezeichner innerhalb der zu extrahierenden Methode (die Methode des ersten oder letzten Methodenaufrufs in B oder B selbst) müssen im Sichtbarkeitsbereich der Rolle sichtbar sein. Falls weitere Basismethoden oder Felder innerhalb der extrahierten Methode referenziert werden, können diese nach dem Refactoring durch ein Callout erreicht werden. Featurezugriffe der Basis sind daher nicht von dieser Vorbedingung betroffen.
6. Bei der Extraktion mit einem **`after`** Callintyp, darf der Methodenaufruf im letzten Statement der Basismethode B keine lokalen Variablen in den Argumenten enthalten. Diese ließen sich nicht in einem Parameter Mapping nachempfinden (OTL7.(c)).

Programmänderungen

1. Wenn der Callintyp C vom Typ **`before`** oder **`after`** ist, wird die Methodendeklaration des letzten bzw. ersten Methodenaufrufs in die Rolle kopiert und erhält den Namen N.

2. Für die Extraktion eines Replace-Callins wird die Methodendeklaration der Basismethode in die Rolle kopiert und erhält den Namen N. Abhängig vom Flag F wird ggf. der Methodenrumpf durch einen Base Call ersetzt. Die Methode erhält zusätzlich einen **`callin`** Modifier und es werden alle Sichtbarkeitsmodifier entfernt (OTL6.(f)-(g)).

3. Der Methodenrumpf kann Referenzen auf Felder und Methoden des deklarierenden Typs von B enthalten. Falls Inferred Callouts für das betreffende Projekt aktiviert sind (siehe Abs. 2.2.3), stellt dies kein Problem dar. Andernfalls müssen die Callouts materialisiert werden.

4. Die kopierte Methode wird in einer Callinbindung an die Basismethode B gebunden. Dafür wird eine Callinbindung mit dem Callintyp C erzeugt. Falls B eine überladene Methode ist, würde eine mehrdeutigen Methodenbindung entstehen. In diesem Fall muss die neue Callinbindung mit vollständigen Signaturen versehen werden (OTI5.).

5. Falls die Argumente des Methodenaufrufs keine einfachen Parameter von B referenzieren oder in anderer Reihenfolge übergeben werden, muss für Before- und After-Callinbindungen ein Parameter Mapping erzeugt werden, das sich vom Methodenaufruf der extrahierten Methode ableiten lässt.

6. Falls bereits andere Callinbindungen mit dem gleichen Callintyp C in der Rolle R existieren, die die Basismethode B binden, ist die Erzeugung einer Precedence Deklaration erforderlich (OTL13.(d)). Dabei ist die Stellung der neu erzeugte Callinbindung innerhalb der Deklaration abhängig vom Callintyp C. Um die Semantik des Programms zu erhalten muss der Precedence Ausdruck wie folgt aussehen.

   - C ist vom Typ **`before`** oder **`replace`**: Die neu erzeugte Callinbindung muss am Ende der Precedence Deklaration stehen.
   - C ist vom Typ **`after`**: Die neu erzeugte Callinbindung muss an erster Stelle der Precedence Deklaration stehen.

   Siehe Verhaltenserhaltung bei Precedence Ausdrücken 7.1.1.

7. Wenn der Callintyp C vom Typ **`before`** oder **`after`** ist, wird der letzte bzw. erste Methodenaufruf in der Basismethode B entfernt.

Beispiel

In den Listings 7.6 und 7.7 ist exemplarisch ein Extract (Before) Callin Refactoring dargestellt. Die Basismethode `m()` enthält als erstes Statement (Zeile 3 Vorher) einen Methodenaufruf der Methode `log()`, die einen Logeintrag erzeugt. Das Refactoring kopiert die Methode `log()` in die Rolle `R` und bindet diese in einer Before-Callinbindung (Zeilen 9-12 Nachher) an die Basismethode `m()`. Der Aufruf und die Methodendeklaration von `log()` in Zeile 3 (Vorher) und Zeilen 6-8 (Vorher) werden entfernt.

```
1  public class B{
2    public void m(){
3      log();
4      ...
5    }
6    private void log(){
7      System.out.println("log: m called");
8    }
9  }
10
11 public team class T{
12   protected class R playedBy B{
13   }
14 }
```

Listing 7.6: Vor Extract Callin

⇒

```
1  public class B{
2    public void m(){
3      ...
4    }
5  }
6
7  public team class T{
8    protected class R playedBy B{
9      private void log(){
10       System.out.println("log: m called");
11     }
12     log <- before m;
13   }
14 }
```

Listing 7.7: Nach Extract Callin

7.1.4. Inline Guards

Das Inline Guards Refactoring ermöglicht eine semantisch äquivalente Umformung von Guard Predicates (siehe Abs. 2.2.6). Dabei wird der Ausdruck in der Guard Expression in ein If-Statement umgewandelt, das den Methodenrumpf der betreffenden Rollenmethoden umschließt. Diese Transformation ist für regular Guards semantikerhaltend. Für Base Guards kann keine Erhaltung der Semantik gewährleistet werden, da ein If-Statement in einer Rollenmethode die Basisinstanz nicht vor einem Lifting bewahren kann. Bei einem Lifting wird der Konstruktor der Rolle aufgerufen, der durch die Erzeugung der Rolle bereits einen Seiteneffekt verursacht.
Bei der Umwandlung eines Base Guards kann es außerdem notwendig sein, die Featurezugriffe in Callouts umzuwandeln, da Base Guards einen direkten Zugriff auf die Features der Basis haben (OTL12.(c)).

Ein Inline Guards Refactoring kann sinnvoll sein, um Sprachkonzepte von OT/J auf ein Java Konstrukt umzuwandeln. Dies wird insbesondere benötigt, um bei einem Inline Callin Refactoring (siehe Abs. 7.1.4) die Guards mit in die Basisklasse zu integrieren.

7.1.5. Move Field to Base

Ein Move Field to Base Refactoring verschiebt ein Feld einer Rolle in ihre Basis. Dabei weist dieses Refactoring Analogien zu einem Pull Up Field Refactoring auf, was ebenfalls eine spezielle Form des Move Field Refactorings ist. Durch den Calloutmechanismus von OT/J, bleibt das verschobene Feld in der Rolle verfügbar, wenn eine entsprechende Callout to Field Deklaration erzeugt wird (oder Inferred Callouts aktiviert sind). Falls das Feld in dem Konstruktor der Rolle initialisiert wird, wäre es sinnvoll die Initialisierung in den Konstruktor der Basisklasse zu integrieren.

Die Semantik des Programms kann durch dieses Refactoring beeinflusst werden, da das Feld nicht nur beim Aufruf des Lifting Konstruktors initialisiert wird, was möglicherweise zu Seiteneffekten führen kann. Im Zuge der Integration einer Rolle, kann eine solche Veränderung jedoch beabsichtigt sein.

Im Zuge der Integration einer Rolle in ihre Basis, kann es hilfreich sein, Felder in die Basis zu verschieben. Insbesondere bei einem Inline Callin Refactoring könnten referenzierte Felder der Rolle gemeinsam verschoben werden. Das Move Field to Base Refactoring sollte sowohl als einzelne Operation verfügbar sein, als auch Bestandteil des Inline Callin Refactorings, sowie des Move Method to Base Refactorings sein. Eine Integration des Refactorings hätte gegenüber einer separaten Ausführung folgende Vorteile:

- Abhängigkeiten zwischen der verschobenen Methode und Feldern der Rolle könnten durch das Refactoringwerkzeug analysiert werden und durch ein Move Field to Base Refactoring gelöst werden. Als Vorbild für ein solches Werkzeug kann die *Add Required* (Members) Funktionalität des Pull Up Refactorings dienen, das eine analoge Berechnung und Verschiebung innerhalb einer Vererbungshierarchie durchführt.
- Beim Verschieben von Methoden ist es außerdem möglich zyklische Strukturen zu verschieben. Eine separate Verschiebung von zwei Methoden, die sich gegenseitig referenzieren, ist ansonsten nicht möglich, ohne zwischen den einzelnen Refactoringschritten ein fehlerhaftes Programm zu erhalten. Dies gilt insbesondere für ein Move Method to Base Refactoring (siehe Abs. 7.1.7), das ebenfalls für die Realisierung der Add Required Members Funktionalität benötigt wird.

7.1.6. Move Field to Role

Bei einem Move Field to Role Refactoring wird das Feld einer Basis in eine ihrer Rollen verschoben. Dieses Refactoring ist nur möglich, wenn das Feld innerhalb der Basis unreferenziert

ist und auch nicht von anderen Rollen oder Klienten[4] referenziert wird. Eine möglicherweise existierende Initialisierung kann ähnlich wie bei einem Move Field to Base Refactoring in den Lifting Konstruktor verschoben werden. Das Java Vorbild dieses Refactorings ist das Push Down Field Refactoring.

Wie bei einem Push Down Field Refactoring, könnte die Einschränkung von Verwendungen in anderen Rollen aufgehoben werden, wenn das Feld in alle Rollen verschoben werden würde, die das Feld verwenden.

Ein solches Refactoring kann sinnvoll sein, wenn ein Feld vorwiegend in der Rolle verwendet wird, was im Zuge der Extraktion einer Rolle in der Basis überflüssig geworden ist.

7.1.7. Move Method to Base

Das Move Method to Base Refactoring stellt das OT/J Pendant zum Pull Up Method Refactoring dar. Durch Callouts kann die Methode weiterhin in der Rolle verwendet werden. Wenn die Methode selbst Ziel einer Methodenbindung ist oder von anderen Klienten referenziert wird, muss eine Callout Methodenbindung erzeugt werden, um die Referenz zu erhalten und an die Basis weiterzuleiten. Da das Move Method to Base Refactoring eine neue Methode erzeugt, müssen Probleme durch Mehrdeutigkeiten, Namenskonflikte oder Overriding ausgeschlossen werden. Wenn im Rumpf der verschobenen Methode weitere Felder oder Methoden der deklarierenden Rolle referenziert werden, müssen diese auch verschoben werden.

Ein Move Method to Base Refactoring kann ein Inline Callin Refactoring unterstützen, wenn die betreffende Rollenmethode Referenzen auf weitere Rollenmethoden enthält (siehe Add Required Members Funktionalität 7.1.5). Auch die umfassende Integration einer Rolle in ihre Basisklasse kann durch dieses Refactoring unterstützt werden.

7.1.8. Move Method to Role

Ein Move Method to Role Refactoring weist Ähnlichkeiten zu dem herkömmlichen Push Down Method Refactoring auf. Es kann auch nur durchgeführt werden, wenn die betreffende Methode nicht mehr von der Basis oder deren Klienten benötigt wird. Die zu verschiebende Rollenmethode darf außerdem nicht in vorhandenen Subtypen verwendet werden und keine Super- bzw. Tsuper-Aufrufe oder andere Zugriffe auf geerbte Features enthalten. Wenn die Rolle eine Calloutbindung für die zu verschiebende Methode deklariert, kann diese entfernt werden und

[4] Als Klienten können sowohl Klassen bezeichnet werden, die die Basisklasse referenzieren oder in einer playedBy-Relation referenzieren.

sämtliche Aufrufe der gebundenen Rollenmethode durch Aufrufe der verschobenen Methode ersetzt werden. Vorhandene Parameter Mappings können direkt in die Methodendeklaration der verschobenen Methode integriert werden.

Analog zu dem Move Field to Role Refactoring könnte die Methode in alle Rollen verschoben werden, die die Methode benötigen und dadurch die Einschränkung bezüglich der Verwendung in anderen Rollen aufgehoben werden.

7.1.9. Inline Role

Das Inline Role Refactoring integriert eine Rolle vollständig in die gebundene Basisklasse und löscht Anschließend die Rolle. Dieses Refactoring stellt im Prinzip kein neues Refactoring dar, sondern verwendet die vorher beschriebenen Refactorings Inline Callin, Inline Guards, Move Field to Base und Move Method to Base, um eine Rolle vollständig in ihre Basis zu integrieren. Dazu muss für alle Methoden, die auf der linken Seite einer Callinbindung gebundenen sind ein Inline Callin Refactoring durchgeführt werden.
Die Felder und restlichen Methoden werden mit den entsprechenden Move Refactorings in die Basis verschoben. Nur wenn alle dieser Refactorings durchführbar sind, ist das gesamte Refactoring durchführbar. Die Rolle muss unreferenziert sein und darf keine Superklassen[5] oder impliziten Subklassen besitzen.

Ein Inline Role Refactoring bewahrt nicht die Semantik des Programms. Teamaktivierung hat nach diesem Refactoring keinen Einfluss mehr auf die integrierte Rolle, da sie Teil der Basisfunktionalität geworden ist.

7.1.10. Extract Role

Wie bei einem Extract Superclass Refactoring, kann es sinnvoll sein, einen Teil einer Klasse in eine Rolle auszulagern. Der Unterschied ist jedoch, dass die extrahierten Features nicht mehr in der Basisklasse zur Verfügung stehen. Der Sinn eines solchen Refactorings ist es, Verhalten zu extrahieren, dass nur in bestimmten Situationen benötigt wird. Es ist also nicht mehr Teil des Basiscodes und soll durch Teamaktivierung ein- und abgeschaltet werden können.

Das Refactoring setzt sich aus den Refactorings Extract Callin, Move Field to Role, Move Method to Role und Create Type zusammen. Es ist daher nur durchführbar, wenn alle selektierten

[5] Eine Rolle mit Superklassen ließe sich im Prinzip trotzdem integrieren, indem zuvor alle geerbten Features in der Rolle materialisiert werden würden.

Features verschoben bzw. extrahiert werden können und setzt daher seine Vorbedingungen aus den Vorbedingungen der einzelnen Refactorings zusammen.

7.1.11. Unterschiede zu Pull/Push Refactorings

Einige der zuvor beschriebenen Move Refactorings weisen gewisse Ähnlichkeiten zu den Push Up Method/Field und Push Down Method/Field Refactorings auf. Dies ist darauf zurückzuführen, dass eine playedBy-Beziehung Ähnlichkeiten zu einer Subtyp-Beziehung aufweist. Die Rolle ist im Prinzip eine Art Spezialisierung der Basis und kann mit Hilfe von selektiven Imports (Callouts) auf deren Features zugreifen. Hier liegt auch der wesentliche Unterschied zu einer Subtyp-Beziehung, es werden nicht automatisch alle Felder und Methoden vererbt, sondern selektiv importiert. Daraus ergibt sich der Vorteil, dass Namenskonflikte durch die Umbenennung der Calloutmethode gelöst werden können.

Die JDT Implementierung der Push Down Refactorings verschiebt das Feature in alle Subklassen der deklarierenden Klasse. Bei einem Move to Role Refactoring wäre eine feinere Unterscheidung möglich, da Rollen in ihren Callouts benötigte Features deklarieren[6]. Dadurch ist es möglich das Feature nur in Rollen zu verschieben, die dieses tatsächlich benötigen.

7.1.12. Aufwandsabschätzung

Durch den beschränkten zeitlichen Rahmen der Diplomarbeit unterscheiden sich die zuvor beschriebenen OT/J Refactorings im Detailgrad. Je nach Komplexität eines Refactorings kann es bis zu einer Woche dauern, die Vorbedingungen und Beschreibungen der durchzuführenden Quelltextänderungen eines Refactorings im Detail zu untersuchen und zu beschreiben. Die Konzeption und Umsetzung des Extract Callin und Inline Callin Refactorings nahm jeweils ungefähr zwei Wochen in Anspruch. Die erarbeiteten Lösungen sind zwar nicht vollständig, jedoch muss berücksichtigt werden, dass bei der Umsetzung des ersten Refactorings eine gewisse Einarbeitungszeit notwendig war.

[6] Inferred Callouts stellen im Prinzip nur eine Kurzschreibweise dar und werden Intern genauso behandelt wie Callouts, für die eine Callout-Deklaration existiert.

	Inline Guards	Move Field to Base	Move Method to Base	Inline Callin	Inline Role	Move Field to Role	Move Method to Role	Extract Callin	Extract Role
OTL1									
(a)	-	-	-	-	-	-	-	-	-
OTL2									
(a)	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-	-
(d)	✓	✓	-	-	✓	✓	-	-	✓
(e)	✓	-	-	-	✓	-	-	-	-
(f)	-	-	-	-	-	-	-	-	-
OTL3									
(a)	✓	-	-	-	✓	-	-	-	-
(b)	✓	-	-	-	✓	-	-	-	-
(c)	✓	-	-	-	✓	-	-	-	-
OTL4									
(a)	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-
OTL5									
(a)	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-	-
OTL6									
(a)	-	-	-	-	-	-	-	✓	✓
(b)	-	-	✓	✓	✓	-	✓	-	✓
(c)	-	-	✓	✓	✓	-	✓	✓	✓
(d)	-	✓	-	✓	✓	✓	-	-	✓
(e)	-	✓	✓	✓	✓	✓	✓	✓	✓
(f)	-	-	-	✓	✓	-	-	✓	✓
(g)	-	-	✓	✓	✓	✓	-	✓	✓
OTL7									
(a)	-	-	-	✓	✓	-	-	-	-
(b)	-	-	-	✓	✓	-	-	✓	✓
(c)	-	-	-	✓	✓	-	-	✓	✓

Tabelle 7.1.: Übersicht der Regeleinflüsse auf die OT/J Refactorings (1/2)

	Inline Guards	Move Field to Base	Move Method to Base	Inline Callin	Inline Role	Move Field to Role	Move Method to Role	Extract Callin	Extract Role
OTL7.1									
(a)	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-
OTL8	-	-	✓	✓	✓	-	-	-	-
OTL9	-	✓	-	-	✓	✓	-	-	✓
OTL10									
(a)	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-	-
OTL11									
(a)	-	-	-	-	-	-	-	-	-
OTL12									
(a)	-	-	-	-	-	-	-	-	-
(b)	-	-	✓	-	✓	-	✓	-	✓
(c)	✓	-	✓	-	✓	✓	✓	-	✓
OTL13					✓				
(a)	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-
(c)	-	-	-	-	-	-	-	-	-
(d)	-	-	-	-	-	-	-	✓	✓
OTL14									
(a)	-	-	-	-	-	-	-	-	-
(b)	-	-	-	-	-	-	-	-	-
OTI1	-	-	-	-	-	-	-	-	-
OTI2	-	-	-	-	-	-	-	-	-
OTI3	-	-	-	-	-	-	-	-	-
OTI4	-	-	-	-	-	-	-	-	-
OTI5	-	-	✓	✓	✓	-	✓	✓	✓
OTI6	-	-	✓	✓	✓	-	✓	✓	✓
OTS1	-	✓	✓	-	✓	✓	✓	-	✓

Tabelle 7.2.: Übersicht der Regeleinflüsse auf die OT/J Refactorings (2/2)

7.2. Praktische Umsetzung

Von den zuvor vorgestellten OT/J Refactorings wurden im Rahmen der praktischen Arbeit das Inline Callin und Extract Callin Refactoring prototypisch umgesetzt. Im Gegensatz zu den Adaptierungen der bestehenden Refactorings, die in Kapitel 6 beschrieben wurden, erfordert die Umsetzung eines OT/J Refactorings eine vollständige Neuentwicklung der in Abschnitt 6.1.1 beschriebenen Bestandteile eines Refactorings.

Die Mechanismen und Techniken, die eingesetzt werden können, um die Vorbedingung eines OT/J Refactorings zu prüfen, sind die gleichen, wie in einem JDT Refactoring oder dessen Adapter. Auch hier ist es möglich Methoden der Utility Klasse zur Überprüfung von Vorbedingungen wiederzuverwenden. Aus diesem Grund liegt der Fokus der folgenden Abschnitte auf der Erzeugung der Quellcodeänderungen.

7.2.1. AST versus Java Model

Quellcodeänderungen werden mit Hilfe des OT/J *ASTs*[7] erzeugt, wohingegen für die Überprüfung der meisten Vorbedingungen das *Java Model* ausreicht. Der wesentliche Unterschied der beiden Datenstrukturen, die beide ein Programm darstellen, liegt im Detailgrad. Das Java Model stellt nur die grobe Struktur des Programms dar, dies sind z. B. Projekte, Packages, Typen, Felder, Methoden etc. Der AST stellt für jedes Element der Sprachspezifikation einen Knoten bereit und enthält dadurch auch Methodenrümpfe, Methodenaufrufe, Variablendeklarationen, Zuweisungen etc. Zu Gunsten der Performanz sollte jedoch nach Möglichkeit das Java Model verwendet werden, da die Erzeugung und Verarbeitung des ASTs wesentlich aufwändiger ist.

Um aus Manipulationen an einem AST Quelltextänderungen zu erzeugen, bietet Eclipse `ASTRewrite` Objekte an, die zu einem AST angefordert werden können und durchgeführte Manipulationen protokollieren. Zum Abschluss des Refactorings kann aus dem `ASTRewrite` Objekt ein `TextChange` Objekt erzeugt werden, das Zeilen- und Spaltenangaben der Änderungen enthält. Diese Trennung von Inhalt und Repräsentation ermöglicht eine komfortable Verarbeitung des ASTs, die unabhängig vom eigentlichen Quelltext ist.

7.2.2. Einschränkungen bezüglich Precedence Deklarationen

Aus Zeitgründen wurden nicht alle Konzepte von OT/J in der praktischen Umsetzung der neuen Refactorings berücksichtigt. So wurden Regeln zu Precedence vorerst ignoriert, um stattdessen

[7] Abstrakter Syntaxbaum (vom engl. Abstract Syntax Tree)

intensiver auf Parameter Mappings einzugehen. Die fehlende Umsetzung betrifft die Inline Callin Vorbedingungen 2 und 3, Programmänderung 2, sowie die Programmänderung 6 und Vorbedingung 2 des Extract Callin Refactorings.

7.2.3. Inline Callin Refactoring

Die Implementierung des Inline Callin Refactorings beruht auf den theoretischen Vorüberlegungen aus Abschnitt 7.1.2. Insgesamt wurden für die Umsetzung des Refactorings die Klassen `InlineCallinAction`, `InlineCallinWizard`, `InlineCallinInputPage`, `CallinBaseMethodInfo` und `InlineCallinRefactoring` implementiert.

InlineCallinAction

In dieser Klasse wird die Verfügbarkeit des Refactorings geprüft und eine Instanz der eigentlichen Refactoring Klasse erzeugt. Ein Inline Callin Refactoring ist verfügbar, wenn eine Methode oder eine Callinbindung selektiert wurde. Die selektierte Methode bzw. die gebundene Rollenmethode der Callinbindung bestimmen dabei den ersten Parameter des Refactorings, die Rollenmethode R (siehe Abs. 7.1.2).

Es wurde bewusst eine Rollenmethode als primärer Eingabeparameter gewählt, da dies eine größere Flexibilität für das Refactoring ermöglicht. Erst durch die Möglichkeit alle vorhandenen Callins einer Rollenmethode zu behandeln, kann die Rollenmethode im Anschluss gelöscht werden (siehe weitere Parameter in InlineCallinInputPage 7.2.3). Außerdem bietet die Selektion eines Callins nicht die nötige Granularität, da Callins mehrere Basismethoden binden können. Wenn die Rollenmethode als Ursprung gewählt wird, können vorhandenen Callins unabhängig von der Darstellung (einzeln oder mit mehreren Basismethoden) gleichwertig behandelt werden und separat selektiert werden.

InlineCallinWizard

Der Wizard soll verschiedene Eingabeseiten verwalten. In diesem Fall gibt es nur eine Seite, die durch die Klasse `InlineCallinInputPage` repräsentiert wird.

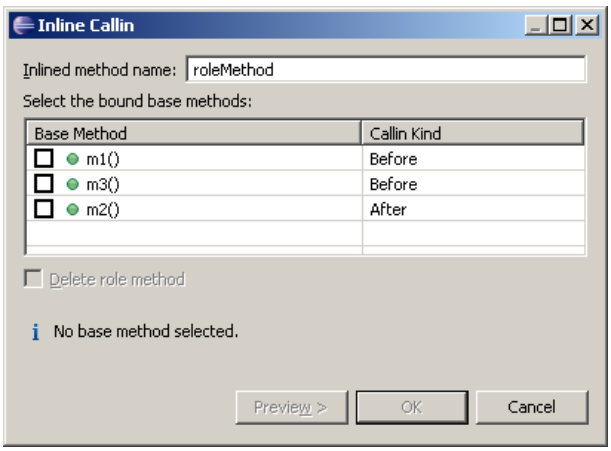

Abbildung 7.1.: Inline Callin Refactoring Wizard

InlineCallinInputPage

Die Klasse erzeugt die Elemente der grafischen Benutzeroberfläche zur Eingabe der Parameter und leitet diese an die Instanz des Refactorings weiter. Abbildung 7.1 zeigt die verschiedenen Elemente zur Eingabe der Parameter. Für die Kopie der Rollenmethode kann ein neuer Name gewählt werden. Da eine Rollenmethode an verschiedene Basismethoden gebunden sein kann, wird eine Liste der gebundenen Basismethoden angezeigt, sowie die zugehörigen Callinarten. In der Liste können die Basismethoden ausgewählt werden, in die die Rollenmethode integriert werden soll. Wenn alle Basismethoden ausgewählt wurden, wird die Checkbox zum Löschen der Rollenmethode verfügbar. Es bleibt zwar zu untersuchen, ob die Rollenmethode wirklich gelöscht werden kann, jedoch ist die Selektion aller Basismethoden eine notwendige Bedingung, da andernfalls Callinbindungen zurückbleiben, die die Rollenmethode referenzieren. Am unteren Rand der Seite wird der Benutzer über Fehleingaben informiert.

CallinBaseMethodInfo

Diese Klasse fasst alle Informationen zusammen, die für die Integration der Rollenmethode in eine bestimmte Basismethode benötigt werden. Dies beinhaltet die Basismethode, die Art der Callinbindung und den neuen Namen für die Basismethode, der benötigt wird, da der aktuelle Name der Basismethode für die Wrapper Methode verwendet wird (Änderung 3).

InlineCallinRefactoring

Hier wird das eigentliche Refactoring umgesetzt. Die Methode `inlineCallin()` erzeugt die Programmänderungen, die sowohl den AST der Rolle, als auch den AST der Basis betreffen. Für jede Basismethode, in die die Rollenmethode integriert werden soll, werden folgende Aktionen durchgeführt:

1. Die Basismethode wird umbenannt, wobei ein „base_“ Präfix vor den Namen gehängt wird. Falls ein Namenskonflikt besteht, wird eine Zahl an den Namen gehängt, die so lange hochgezählt wird, bis ein freier Name gefunden wird[8] (Änderung 3).
2. Es wird eine Wrapper Methode mit dem Namen und der Signatur der Basismethode erzeugt (Änderung 3).
3. Abhängig von der Art der Callinbindung wird in der Wrapper Methode eine Kombination aus Basismethodenaufruf und Rollenmethodenaufruf erzeugt. Der Rollenmethodenaufruf bezieht sich dabei auf den Namen und die Signatur der anschließend erzeugten Rollenmethodenkopie (Änderung 8).

Nach der Bearbeitung aller Basismethoden werden Änderungen erzeugt, die nur einmal notwendig sind. Das sind zum einen die Anpassungen der Callinbindungen, die entweder komplett gelöscht werden oder bei denen die Namen der behandelten Basismethoden entfernt werden (Änderung 1). Dann muss einmalig die Rollenmethode in die Basis kopiert werden und in der Rolle gelöscht werden, falls dies gefordert ist (Änderung 4).

Sonderfälle einer After-Callinbindung

Wenn die Basismethode einen Rückgabetyp hat, muss eine lokale Variable erzeugt werden, die diesen zwischenspeichert und nach dem Aufruf der Rollenmethode zurückliefert. Falls ein Result Parameter Mapping existiert, muss der Rückgabewert an die Rollenmethode übergeben werden. Die Position des Result Arguments wird dabei aus dem Parameter Mapping ermittelt[9] (Änderung 9).

[8] Diese Strategie wird für alle erzeugten Namen innerhalb des Refactorings verwendet.
[9] Result Parameter Mappings für After-Callins wurden im Rahmen der praktischen Arbeit nicht berücksichtigt. Der Vollständigkeit halber wurde der Punkt trotzdem erwähnt.

Sonderfälle einer Replace-Callinbindung

Beim Kopieren der Rollenmethode muss der **`callin`** Modifier entfernt werden, da eine Callinmethode nicht aufgerufen werden dürfte und nur in Rollen verwendet werden darf. Außerdem kann der Rumpf einer Callinmethode Base Calls enthalten, die durch den Aufruf der Basismethode ersetzt werden müssen (Änderungen 5 und 7).

Die Signatur einer Callinmethode reicht nicht in jedem Fall aus, um der Basismethode alle benötigten Parameter zur Verfügung zu stellen. Falls die Signatur der Rollenmethode kürzer als die der Basismethode ist, findet ein sogenanntes *Parameter Tunneling* statt (siehe OTJLD §4.3.(d) [HHM09]). Das heißt, die überschüssigen Parameter werden unverändert an die Basismethode weitergeleitet. Um dies zu realisieren, müssen überschüssige Parameter an die Signatur der Kopie angehängt werden[10], die dann in der Ersetzung der Base Calls verwendet werden können (Änderung 6).

Eine kompliziertere Variante des Parameter Tunnelings kann auftreten, wenn ein Parameter Mapping angegeben ist, das nicht alle Parameter der Basismethode an einen Parameter der Rolle zuweist (siehe OTJLD §4.4.(b) [HHM09]). Solche Parameter müssen nicht zwangsläufig am Ende der Signatur stehen, werden jedoch ebenfalls an die Signatur der Rollenmethodenkopie gehängt, um sie dann unverändert an die Basismethode weiterzuleiten.

Eine weitere Besonderheit tritt auf, wenn die Basismethode einen Rückgabetyp deklariert, die Callinmethode jedoch nicht. Dann entsteht ein Result Tunneling (siehe OTJLD §4.3.(d) [HHM09]), das durch die Deklaration einer lokalen Variable gelöst wird. Bei jedem Aufruf der Basismethode wird der lokalen Variable das Ergebnis zugewiesen, um am Ende der Methode den gespeicherten Wert zurück zu liefern.
Die lokale Variable wird nicht benötigt, falls ein Result Parameter Mapping existiert (siehe OTJLD §4.4.(b) [HHM09]).
Daraus ergibt sich außerdem, dass der Rückgabetyp der Kopie vom Rückgabetyp der Basismethode abhängt und nicht von der Rollenmethode, wie bei einer Before- oder After-Callinbindung.

Bugfix für Callinbindungen

Wenn ein Result Tunneling stattfindet, befindet sich beim Kopieren der Rollenmethode am Ende des Methodenrumpfes folgendes Statement:

```
return _OT$result;
```

[10] Auch hierbei werden die Parameternamen erzeugt, damit keine Konflikte mit Parameternamen oder lokalen Variablen der Rollenmethode auftreten.

Das Statement ist ein Teil der internen Repräsentation, der durch einen Bug im OTDT (Version 1.2.8) sichtbar wird. Um trotzdem eine brauchbare Kopie zu erhalten wird in diesem Fall das letzte Statement entfernt.

Parameter Mappings

Falls in der Callinbindung ein Parameter Mapping angegeben wurde, müssen die Expressions in die Argumentliste des Rollenmethodenaufrufs übertragen werden. Da die Parameternamen innerhalb eines Parameter Mappings nicht mit den Parameternamen der referenzierten Methode übereinstimmen müssen, wird zuerst eine Abbildung vom Typ `HashMap<String,String>` angelegt, die den Parameternamen auf der rechten Seite des Parameter Mappings auf den Parameternamen der Basismethode abbildet. Die Abbildung ergibt sich aus der Positionen innerhalb der Signatur.

Anschließend werden für alle Parameter der Rollenmethode die Expressions aus dem Parameter Mapping gesucht und kopiert. In der Kopie müssen dann die Vorkommen des Basisparameters durch den in der Signatur der Basismethode angegebenen Parameternamen ersetzt werden. Dazu wird die zuvor erzeugte Abbildung verwendet.

Wenn alle Expressions kopiert und angepasst wurden, können sie in den Aufruf der Rollenmethode gesetzt werden (Änderung 8).

Für Replace-Callins müssen die Parameter Mappings zusätzlich in die ersetzten Base Calls übertragen werden. Dazu wird analog eine Abbildung verwendet, die die Namen auf der linken Seite des Parameter Mapping auf die Parameternamen der Rollenmethode abbildet (Änderung 7).

Callouts

Beim Kopieren der Rollenmethode müssen Referenzen auf Calloutmethoden beachtet werden. Dazu werden **`get`** und **`set`** Callout to Field Bindungen durch Feldzugriffe oder Feldzuweisungen ersetzt. Callouts, die eine Methode betreffen, werden durch den entsprechenden Methodenaufruf der Basismethode ersetzt. Falls die Callouts Parameter Mappings deklarieren, müssen diese bei der Ersetzung ebenfalls übertragen werden. Dies geschieht analog zur Übertragung der Parameter Mappings von Callinbindungen[11] (Änderung 4).

[11] Die Behandlung von Callout Parameter Mappings wurde im Rahmen der praktischen Arbeit noch nicht umgesetzt, wird hier aber der Vollständigkeit halber erwähnt.

Import Organisation

Sowohl die Expressions in vorhandenen Parameter Mappings, als auch der Methodenrumpf der kopierten Rollenmethode können Typen referenzieren, die in der Basis noch nicht bekannt sind. Das LTK stellt dafür die Methode `collectImports()` bereit, die für einen neu hinzugefügten AST Knoten die benötigten Imports bestimmt.

Analog können nicht mehr benötigte Imports in der Rolle entfernt werden, wenn Callinbindungen mit Parameter Mappings oder die Rollenmethode selbst entfernt wird.

Schnittstellen zu weiteren Refactorings

Da im Rahmen dieser Arbeit die Refactorings Inline Guards, Move Method to Base und Move Field to Base nicht umgesetzt wurden, stehen diese nicht zur Verfügung. Die Querbezüge in den theoretischen Grundlagen sind daher in der prototypischen Umsetzung nicht realisiert worden (siehe Vorbedingungen 4 und 5).

7.2.4. Extract Callin Refactoring

Die Implementierung des Extract Callin Refactorings beruht auf den theoretischen Vorüberlegungen aus Abschnitt 7.1.3. Insgesamt wurden für die Umsetzung des Refactorings die Klassen `ExtractCallinAction`, `ExtractCallinWizard`, `ExtractCallinInputPage` und `ExtractCallinRefactoring` implementiert.

ExtractCallinAction

Wie bei dem Inline Callin Refactoring, wird hier die Verfügbarkeit geprüft und das Refactoring erzeugt. Ein Extract Callin Refactoring ist verfügbar, wenn eine Methode selektiert wurde. Die Existenz einer Rolle, die die deklarierende Klasse in einer playedBy-Beziehung bindet, wird erst nach der Erzeugung des Refactorings geprüft. Die Prüfung der Verfügbarkeit beschränkt sich auf minimale Voraussetzungen, da sie nach jeder Änderung in der Selektion durchgeführt wird und daher sehr kurz sein sollte.

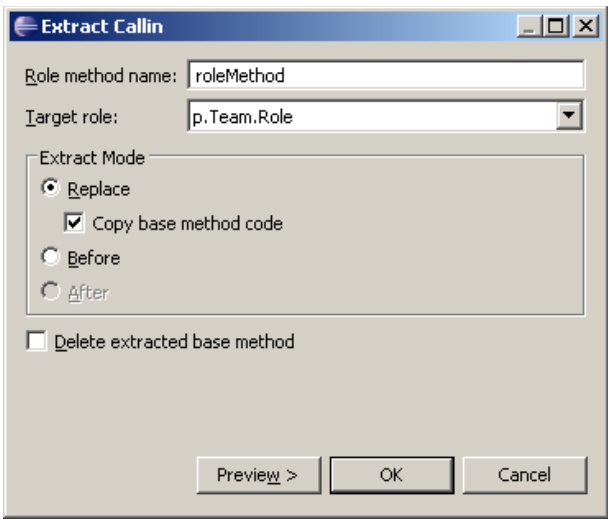

Abbildung 7.2.: Extract Callin Refactoring Wizard

ExtractCallinWizard

Auch der Wizard des Extract Callin Refactorings enthält nur eine Seite für die Eingabe von Parametern. Bei der Erzeugung des Wizards wird die Seite `ExtractCallinInputPage` angezeigt.

ExtractCallinInputPage

Abbildung 7.2 zeigt die verschiedenen Elemente der Eingabeseite. Für die extrahierte Rollenmethode kann ein Name angeben werden. Die Auswahl der Zielrolle wird durch eine Auswahlliste realisiert, da eine Basis von mehreren Rollen gebunden werden kann.

In Abschnitt 7.1.3 wurden die verschiedenen Varianten der Extraktion beschrieben. Die Radiobuttons in der Gruppe *Extract Mode* spiegeln die verschiedenen Varianten wieder. Die zusätzliche Checkbox für die Extraktion eines Replace-Callins unterscheidet dabei die zwei Möglichkeiten einer Replace-Callin Extraktion (siehe Variante 3).

Die Verfügbarkeit der Radiobuttons hängt von der Struktur der Basismethode ab. Die Klasse `ExtractCallinRefactoring` stellt die Methoden `isExtractBeforeAvailable()` und `isExtractAfterAvailable()` zur Verfügung, die die in den Varianten beschriebenen Voraussetzungen prüfen. Die Extraktion eines Replace-Callins ist hingegen immer möglich.

Die untere Checkbox entscheidet, ob die extrahierte Basismethode gelöscht werden soll. Für Replace-Callins ist das Löschen der Basismethode nicht möglich, da die extrahierte Methode selbst für die Callinbindung verwendet wird. In diesem Fall wird die Checkbox deaktiviert.

Im untersten Bereich der Seite werden vorhandene Probleme angezeigt (im Beispiel sind keine vorhanden).

ExtractCallinRefactoring

Hier wird das eigentliche Refactoring umgesetzt und weitere Vorbedingungen geprüft, die in Abschnitt 7.1.3 beschrieben wurden. Die Methode `extractCallin()` erzeugt die Quelltextänderungen für die Rolle und Basis. Abhängig von der gewählten Callinart, wird die Methode des ersten bzw. letzten Methodenaufrufs der Basismethode als Ziel für die Extraktion festgelegt. Der Methodenaufruf wird anschließend aus dem Rumpf der Basismethode entfernt (Änderung 7). Für Replace-Callins wird die Basismethode selbst als Ziel für die Extraktion verwendet.

Die zu extrahierende Methode wird kopiert und in die Rolle eingefügt. Falls es sich um die Extraktion eines Replace-Callins handelt und der Methodenrumpf der Basismethoden nicht kopiert werden soll, wird der Methodenrumpf in der Kopie durch einen Base Call ersetzt. Außerdem wird für **`replace`** Callins ein **`callin`** Modifier erzeugt und vorhandene Sichtbarkeitsmodifier entfernt (Änderungen 1 und 2).

Anschließend wird in der Rolle eine Callinbindung mit entsprechendem Typ erzeugt und an die Basis Methode gebunden[12] (Änderung 4).

Parameter Mappings

Für Before- und After-Callins kann es notwendig sein ein Parameter Mapping zu erzeugen. Dazu wird der Methodenaufruf der extrahierten Methode auf folgende Bedingungen geprüft:

- Die Argumente im Methodenaufruf sind nicht die einfachen Parameternamen der umschließenden Methode.
- Die Signatur der extrahierten Methode ist länger als die der Basismethode.
- Die Reihenfolge der übergebenen Parameter entspricht nicht der Reihenfolge in der Signatur der Basismethode.

Trifft eine der oben genannten Bedingungen zu, ist die Erzeugung eines Parameter Mappings erforderlich. Dazu werden die Ausdrücke des Methodenaufrufs der Reihenfolge nach an die Parameternamen der extrahierten Basismethode gebunden. Bei der Erzeugung der Callinbindung werden die gleichen Parameternamen wie in den Signaturen verwendet. Daher wird keine Abbildung benötigt, die die Parameternamen übersetzt (Änderung 5).

[12] Im Fall eines Replace-Callins sind der Ursprung der Kopie und die Basismethode identisch.

Inferred Callouts

Der Rumpf der extrahierten Basismethode kann Feldzugriffe oder Methodenaufrufe der Basis beinhalten. Wenn Inferred Callouts deaktiviert sind (siehe Abs. 2.2.3), führt dies zu Fehlern in der Rolle. Das OTDT stellt eine Operation zum Materialisieren von Callouts zur Verfügung, die verwendet werden kann, um die Fehler zu beheben. In einer Verfeinerung des Refactorings könnte dieser Mechanismus in das Refactoring integriert werden, um in beiden Fällen ein fehlerfreies Programm zu erzeugen (Änderung 3).

Import Organisation

Die kopierte Basismethode kann Typreferenzen enthalten, die in der Rolle nicht bekannt sind. Das gleiche gilt für Ausdrücke im Argument des Methodenaufrufs der extrahierten Methode, die dann in Form eines Parameter Mappings in die Rolle übertragen werden. Für beide Konstrukte wird der für das Inline Callin Refactoring erwähnte `collectImports()` Mechanismus verwendet (siehe Abs. 7.2.3).

Wenn die Basismethode gelöscht werden soll, können außerdem nicht mehr benötigte Imports aus der Basis entfernt werden.

7.3. Zusammenfassung

Die in den theoretischen Grundlagen (siehe Abs. 7.1) vorgestellten OT/J Refactorings geben einen Einblick in die Möglichkeiten, die sich durch OT/J eröffnen. Die Liste der OT/J Refactorings ist sicherlich nicht vollständig und lässt sich beliebig erweitern. Das liegt daran, dass Refactorings (ähnlich wie Design Patterns) aus Erfahrungen entstehen. Erst mit der Zeit lassen sich häufig durchgeführte Umstrukturierungen verallgemeinern und als Refactoring formulieren. Die vorgestellten Refactorings wurden aus Beobachtungen, die während der praktischen Arbeit gesammelt wurden, abgeleitet. Der Fokus lag dabei bewusst auf Refactorings, die sich der dritten und vierten Dimension von aspektorientiertem Refactoring zuordnen lassen. Das umfasst Refactorings, die Code aus der Basis in eine Rolle verschieben oder Code aus der Rolle in die Basis integrieren (siehe Dimensionen von aspektorientiertem Refactoring in Abs. 4.1).

Mit der Implementierung der OT/J Refactorings ist die Entwicklungsphase abgeschlossen. Die beiden OT/J Refactorings, sowie die entwickelten Adapter (siehe Kapitel 6) können für die neuste Eclipse Version migriert werden. Das folgende Kapitel befasst sich mit der durchgeführten Migration.

8. Migration der Ergebnisse von Eclipse 3.4 nach Eclipse 3.6

Die Refactorings und Refactoring Adapter wurden bewusst auf Basis der Eclipse Version 3.4 entwickelt[1], um im Abschluss der Arbeit eine Migration durchzuführen. Mittlerweile gibt es bereits eine Vorabversion von Eclipse 3.6, die als Ziel der Migration verwendet wird und somit die Version 3.5 gänzlich übersprungen wird. In diesem Kapitel werden die Anpassungen beschrieben, die notwendig waren, um die entwickelte Lösung an die Eclipse Version 3.6 anzupassen. Da der Refactoring Adapter ein Bestandteil des OTDTs ist, mussten auch die Änderungen zwischen der Version 1.2.8 (Eclipse 3.4) und 1.4.0 M1 (Eclipse 3.6) des OTDTs berücksichtigt werden. Die notwendigen Änderungen können Aufschluss darüber geben, wie evolutionsfähig die erarbeitete Lösung ist.

8.1. Allgemeines Vorgehen bei einer Migration

In diesem Abschnitt wird der allgemeine Ablauf der durchgeführten Migration beschrieben.

8.1.1. Aktualisierung von benötigten Plug-ins

Bei einer Migration werden alle verwendeten Plug-ins durch die Version der Zielversion ersetzt. Dies betrifft alle Plug-ins, die in einer *required*-Beziehung von den entwickelten Plug-ins referenziert werden (siehe Kapitel 5).

[1] Zu Beginn der Arbeit gab es bereits eine Vorabversion von Eclipse 3.5, die auch als Grundlage hätte verwendet werden können.

8.1.2. Wiederherstellung eines fehlerfreien Programms

Durch Strukturänderungen oder Umbenennungen, können Fehler beim Übersetzen des Programms entstehen. In der Regel treten diese Änderungen nur in internen Klassen auf, da exportierte Klassen oder Schnittstellen nicht geändert werden. Bevor die Semantik des Programms untersucht werden kann, müssen diese Fehler behoben werden.

8.1.3. Wiederherstellung der Semantik

Nachdem alle statischen Fehler behoben wurden, kann das Programm wieder ausgeführt werden. Es muss sichergestellt werden, dass das Verhalten des Programms nicht durch die Migration beeinträchtigt wurde. Die Wiederherstellung der Semantik ist wesentlich schwieriger, als die Behebung von statischen Fehler. Nur durch eine gute Testsuite kann gewährleistet werden, dass die Semantik des Programms erhalten werden kann. Für Verhalten, das nicht in der Testsuite dokumentiert ist, kann keine Äquivalenz der Semantik garantiert werden. Das Programm bzw. die Testfälle[2] müssen so lange überarbeitet werden, bis alle Laufzeitfehler behoben wurden und die Testsuite wieder ein positives Ergebnis liefert.

8.1.4. Vereinigung von parallelen Änderungen

Falls neben dem *Branch*[3], an dem entwickelte wird ein weiterer Branch existiert, müssen Änderungen vereinigt und Konflikte behoben werden. Dieser Fall kann auch eintreten, wenn Änderungen an der Basis vorgenommen wurden, von der der Branch erstellt wurde oder die Zielversion selbst Änderungen für die bearbeiteten Klassen enthält[4].

Versionsverwaltungssysteme bieten zur Reintegration eines Branchs die *Merge* Operation an. Hier gilt es nach Möglichkeit die Änderungen beider Versionen zu übernehmen. Nach der Vereinigung der Änderungen muss erneut die Semantik des Programms überprüft und ggf. wiederhergestellt werden.

[2] Testfälle unterliegen den gleichen Einflüssen, wie das eigentliche Programm. In einem fehlschlagenden Testfall muss daher auch überprüft werden, ob sich die Semantik der verwendeten Hilfsmethoden geändert hat.

[3] Branch ist ein Begriff aus der Versionsverwaltung. Ein Branch bezeichnet eine Kopie der ursprünglichen Version, die unabhängig weiterentwickelt wird.

[4] Ein solcher Konflikt wurde vermieden, indem Änderungen nicht direkt in bestehenden Klassen des JDTs durchgeführt wurden (siehe Abs. 6.2).

8.2. Durchgeführte Änderungen

In den Adaptern für die JDT Refactorings musste nichts angepasst werden. Nach der Integration der Adapter in die aktuelle OTDT Version musste ein Testfall der JDT Refactoring Testsuite deaktiviert werden. Dabei handelte es sich um den bereits erwähnten Testfall für das Introduce Indirection Refactoring, der auf Grund der fehlenden Regionen basierten OT/J Typhierarchie fehlschlägt (siehe Abs. 6.7).
Im Gegenzug konnten zwei Tests für das Move Instance Method Refactorings reaktiviert werden. In den Tests traten zuvor Probleme auf, da das JDT Refactoring keine Überprüfung für Namenskonflikte durchführte (siehe JDT Mängel 6.3). Der Bugreport war erfolgreich und wurde in der Eclipse Version 3.6 behoben.

In den neu entwickelten OT/J Refactorings traten mehr Probleme auf, was im wesentlichen auf Änderungen im OT/J AST zurückzuführen ist.

8.2.1. Änderungen im AST

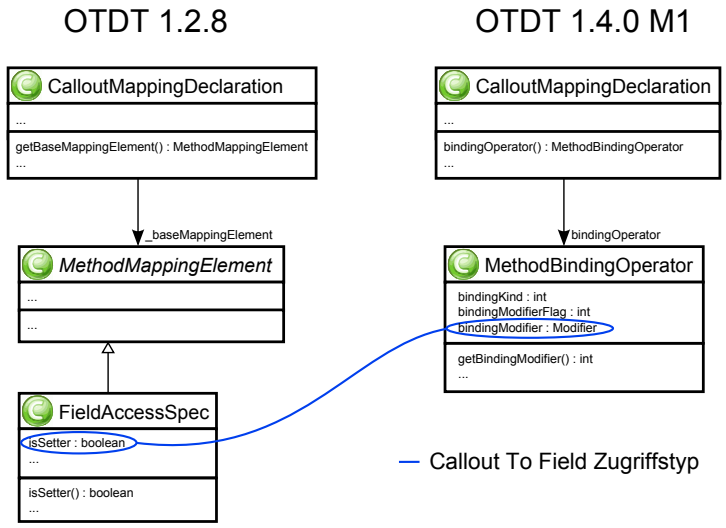

Abbildung 8.1.: Änderungen im AST Knoten für Calloutbindungen

Das Inline Callin Refactoring muss bei der Ersetzung von Aufrufen von Calloutmethoden die verschiedenen Callout to Field Varianten unterscheiden (**`get`** und **`set`**). Abbildung 8.1 stellt die Veränderung im AST Knoten für Calloutbindungen dar. Auf Grund der Änderungen musste die Unterscheidung der Callout to Field Varianten im Quelltext folgendermaßen angepasst werden:

```
if ((FieldAccessSpec)calloutDecl.getBaseMappingElement()).isSetter(){...
```

```
if (Modifier.isSet(calloutDecl.bindingOperator().getBindingModifier())){...
```

Eine weitere Änderung betrifft den AST Knoten für Parameter Mappings. Die Parameter auf der linken Seite des Mappings waren zuvor in Form von Strings repräsentiert. In der neusten Version wird stattdessen ein `SimpleName` Objekt verwendet. Daher mussten einige Stellen im Code wie folgt angepasst werden:

```
mappingDecl.getIdentifier()
```

```
mappingDecl.getIdentifier().getIdentifier()
```

Da die Änderungen im AST die Struktur und nicht nur die Semantik betrafen, wurden für alle anzupassenden Stellen im Quelltext Fehlermeldungen beim Übersetzen erzeugt. Dadurch konnte die Anpassung schnell durchgeführt werden.

Trotzdem lieferten nicht alle Testfälle ein positives Ergebnis. Ein Testfall schlug fehl, da sich die Semantik des Programms geändert hatte. Das Problem lag in dem in Abschnitt 7.2.3 erwähnten Bugfix für Callinbindungen. Da dieser Bug mittlerweile im OTDT behoben wurde, ließ sich das Problem durch das Entfernen des Workarounds lösen. Da bereits während der Entwicklung bekannt war, dass der Fehler in der neusten Version des OTDT behoben wurde, war die Codestelle bereits mit einer *Fixme*-Marke markiert worden und konnte leicht gefunden werden. Listing 8.1 zeigt die betroffene Stelle im Quellcode. Es wurde das Fixme-Kommentar, sowie der Bugfix selbst entfernt (Zeilen 2-4).

```
1 if (fRoleMethod.getReturnType().equals(Character.toString(Signature.C_VOID))) {
2   // FIXME(jogeb):The last statement of callins with void return
3   // is a "return _OT$result" statement(internal representation)
4   statements.remove(statements.size() - 1);
5   ...
6 }
```

Listing 8.1: Entfernter Bugfix Code

8.3. Fazit

Die Migration verlief bis auf kleinere Anpassungen in den neu entwickelten OT/J Refactorings ohne größere Probleme. Der Einsatz von OT/J ermöglichte trotz geringer Abhängigkeiten zwischen Rolle und Basis die Wiederverwendung der benötigten Basisfunktionalität. Die meisten Adaptionen haben einen rein additiven Charakter (siehe Abs. 6.10.1) und werden daher von Änderungen in der Basis nicht beeinflusst. Größere Anpassungen wären zu erwarten, wenn sich die Struktur der adaptierten Refactorings in größerem Maße ändert. Dies ist in nächster Zeit nicht zu erwarten, da die Refactorings mittlerweile recht ausgereift sind und die Signaturen der wichtigsten Methoden durch das LTK vorgegeben werden.

9. Fazit und Ausblick

Die untersuchten Konzepte ObjectTeams/Java und Refactoring stellen beide Techniken zur Verbesserung der Softwarestruktur dar. Dabei verfolgen beide Techniken unterschiedliche Ansätze, die sich miteinander kombinieren lassen. So unterliegen OT/J Programme dem gleichen Evolutionsprozess wie klassische objektorientierte Programme und die Struktur muss dabei stetig durch Refactoring an neue Anforderungen angepasst werden. Dazu ist es notwendig, dass die Refactorings die neuen Sprachkonzepte berücksichtigen, um das beobachtbare Verhalten des Programms nicht zu verändern.

Dazu wurden in der Arbeit 18 atomare Refactorings untersucht und die von Opdyke in [Opd92] formulierten Vorbedingungen um OT/J Vorbedingungen ergänzt. Dabei wurde der von Gregor Brcan erarbeitete Regelsatz zur Verhaltenserhaltung von OT/J Programmen wiederverwendet [Brc05] und auf den neusten Stand gebracht. Außerdem konnte die Einschränkung, dass die Ziele der Refactorings nur Elemente im Basiscode betreffen durften, aufgehoben werden. Das Ergebnis sind 5 vollständig beschriebene Refactorings mit ergänzten Vor- und Nachbedingungen[1], sowie 13 grob skizzierte Refactorings (siehe Abs. 4.5 und 4.6). Außerdem wurden 8 weitere Refactorings beschrieben, die zum Teil erste Umstrukturierungen von OT/J Elementen ermöglichen (siehe Abs. 4.7).

Refactorings können jedoch auch von den neuen Modularisierungskonzepten von OT/J profitieren und diese zur Behebung von Problemen in der Programmstruktur verwenden, die durch rein objektorientierte Sprachen nicht zufriedenstellend gelöst werden können. Refactoring kann dabei helfen nachträglich entdeckte oder entstandene Crosscutting Concerns zu modularisieren. In einigen Fällen ist es sogar möglich mit Hilfe von Refactoring ein OT/J Programm in ein äquivalentes Java Programm umzuformen, das dann von einem herkömmlichen Java Compiler übersetzt werden kann.

In der Arbeit konnten erste Einblicke in die Möglichkeiten für neue Refactorings gegeben werden. Es wurden 9 neue Refactorings beschrieben und 2 davon exemplarisch im Detail untersucht (siehe Abs. 7.1).

[1] Nachbedingungen sind in dieser Arbeit in Form von Quelltextänderungen beschrieben worden.

Die theoretischen Erkenntnisse der Arbeit konnten verwendet werden um Werkzeuge zur Unterstützung bei der Durchführung eines Refactorings zu entwickeln. Dazu wurden 3 Refactorings des JDTs mit Hilfe von OT/Equinox adaptiert und 2 OT/J Refactorings von Grund auf neu entwickelt (siehe Abs. 6.4-6.6 und 7.2).

9.1. Theorie und Praxis

Das Ergebnis der theoretischen Untersuchungen der Refactorings bildete eine gute Grundlage für eine praktische Umsetzung. Bei den erweiterten Vor- und Nachbedingungen für die JDT Refactorings konnten alle Sprachregeln von OT/J berücksichtigt werden. Es gab keine Komplikationen mit Regeln, deren Einhaltung durch ein Refactoring nicht gewährleistet werden könnte.

In der praktischen Arbeit kamen jedoch weitere Anforderungen hinzu, die in den theoretischen Vorüberlegungen nicht bedacht wurden, da sie nicht aus der Sprachdefinition hervorgehen. Das waren z. B. fehlende Schreibrechte oder Probleme beim Versuch durch ein Refactoring eine binäre Klasse zu verändern. Außerdem musste bei der Überprüfung von Vorbedingungen auf die Performanz des Programms geachtet werden. Bedingungen, die für Mengen gelten oder die Nichtexistenz einer Struktur fordern, können schon bei Programmen mittlerer Größe zu einem erheblichen Berechnungsaufwand führen. Hier galt es möglichst effiziente Algorithmen zu verwenden und Ergebnisse, die häufiger benötigt werden, zwischenzuspeichern.

9.1.1. Besonderheiten bei der Entwicklung von Adaptern

Ein weiterer Aspekt, der für die Refactoring-Adapter wichtig war, war die Entwicklung einer gut strukturierten Lösung, die selbst evolutionsfähig ist. Dies beeinflusst vor allem die Wahl der Callinbindungen, die möglichst unabhängig von der internen Struktur des zu adaptierenden Refactorings sein sollten. Das LTK und die JDT Implementierung des Refactorings gaben dabei die Rahmenbedingungen vor. Oft gab es nicht die optimale Lösung, sondern es musste zwischen verschiedenen Anforderungen abgewogen werden, die durch die Wahl der Callinbindung beeinflusst wurden.

- Der Adapter sollte übersichtlich und leicht verständlich sein.

- Die Verwendung weniger Callinbindungen verbessert die Robustheit des Adapters, da Änderungen in der Basis weniger Callinbindungen betreffen[2].
- Die Adaptionen sollten nach Möglichkeit rein additiv sein und unabhängig von den internen Strukturen der JDT Implementierung.
- Um Redundanzen zu vermeiden, sollte trotzdem möglichst viel der vorhandenen Infrastruktur wiederverwendet werden. Dazu gehört auch der Verzicht auf das Kopieren von Basiscode, der möglicherweise bei Änderungen in der Basis ebenfalls angepasst werden müsste.
- Um die Performanz nicht zu beeinträchtigen, sollten Referenzsuchen nach Möglichkeit nur einmal durchgeführt werden. Auch vorhandene Zwischenergebnisse sollten nicht ein zweites Mal berechnet werden, dies betrifft insbesondere die Erzeugung einer Typhierarchie die in großen Vererbungshierarchien relativ aufwändig sein kann.

9.1.2. Neu entwickelte OT/J Refactorings

Bei der theoretischen Ausarbeitung der neuen OT/J Refactorings wurden einige Einschränkungen festgestellt, da nicht immer eine Erhaltung des Verhaltens gewährleistet werden kann. Die Schwierigkeiten werden durch die Teamaktivierung verursacht, die zur Laufzeit verändert werden kann und damit nur schwer statisch analysierbar ist. Auch das Konzept der Callin Precedence bringt einige Einschränkungen mit sich, die jedoch gut in Regeln übertragen werden können (siehe Abs. 7.1.1).

Bei der Neuentwicklung traten andere Schwierigkeiten als bei der Entwicklung der Adapter auf. Die Sprache OT/J bietet dem Entwickler viel Flexibilität und viele neue Konzepte, die sich beliebig miteinander kombinieren lassen. Diese Freiheit und Flexibilität führt dazu, dass die Analyse des Programms recht kompliziert wird und viele Randfälle oder Kombinationen verschiedener Konzepte eine Sonderbehandlung benötigen. Dies wirkte sich vor allem auf die Erzeugung der Quellcodeänderungen aus, die bei dem Inline Callin Refactoring den größten Teil der Implementierung ausmachen. Dabei mussten verschiedene Kombinationsmöglichkeiten der Signaturen von Rollen- und Basismethode, Parameter Mappings und Sonderfälle für die verschiedenen Callintypen (**`before`**, **`after`** und **`replace`**) behandelt werden (siehe Abs. 7.2.3).

[2] Im Idealfall könnten Änderungen in der Basis durch Refactorings auch in die Rolle übertragen werden. Im Fall des JDTs ist den Entwicklern jedoch nicht bewusst, dass ihre Klasse als Basis verwendet wird, da das JDT unabhängig vom OTDT entwickelt wird.

9.2. Erfahrung mit Refactoring

Refactoring ist ein mächtiges Werkzeug, das die Struktur von Software erheblich verbessern kann. Der Einsatz eines Refactoringwerkzeugs kann vor allem in großen Programmen helfen die Semantik des Programms zu erhalten und das Refactoring konsequent an allen nötigen Stellen durchzuführen. Dabei sollte jedoch bedacht werden, dass die Korrektheit des Refactorings von der Implementierung abhängt und Fehler innerhalb des Werkzeugs zu Fehlern im Programm führen können. Die aktuellen Refactoringwerkzeuge sind zwar sehr ausgereift, trotzdem werden immer wieder Randfälle gefunden, die von den Werkzeugen übersehen werden. Daher sollte man einem Refactoringwerkzeug nicht blind vertrauen und die Änderungen selbst begutachten.

Innerhalb der Arbeit habe ich mich selbst ertappt, wie ich vermeintlich kleine Refactorings per Hand durchführen wollte. Dies ist darauf zurückzuführen, dass die Durchführung eines Refactorings in einem großen Projekt mehrere Sekunden in Anspruch nehmen kann. Das Nachbessern von Flüchtigkeitsfehlern dauerte dann aber doch oft länger als die Laufzeit des Werkzeugs.
Das zeigt, dass die Performanz eines Refactoringwerkzeug sehr wichtig ist. Es gilt allen möglichen Fehlern vorzubeugen, jedoch nur einen minimalen Teil des Programms zu analysieren, auf das das Refactoring eine Auswirkung haben kann. Dies ist nicht in allen Fällen trivial, da die Vorbedingungen eines Refactorings ein syntaktisch korrektes Programm und zusätzlich die Erhaltung des beobachtbaren Verhaltens garantieren müssen.

9.3. Fazit

In der Arbeit konnten fast alle festgelegten Ziele erreicht werden. Es wurden einige Refactorings theoretisch untersucht und in die Praxis umgesetzt. Es sind dabei alle wesentlichen Konzepte der Sprache OT/J berücksichtigt worden und nach Einschätzung keine größeren Probleme offen geblieben, die der Adaption weiterer Refactorings im Weg stehen könnten.

Mit den neu entwickelten OT/J Refactorings konnten erste Möglichkeiten aufgezeigt werden, das Feld ist jedoch relativ neu und bietet viele weitere Möglichkeiten und Probleme, die in der Arbeit nur kurz skizziert werden konnten. Die Implementierung zeigt prototypisch die Umsetzbarkeit von zwei der vorgestellten Refactorings.

Die durchgeführte Migration war erfolgreich und verlief ohne größere Probleme. Die entwickelten Adapter scheinen robust zu sein und wurden von den durchgeführten Änderungen im JDT nicht beeinträchtigt.

Die angestrebte Adaption des `RippleMethodFinder2` zum Auffinden aller verwandten Methoden bei einem Rename Method konnte leider nicht realisiert werden. Die Adaption hätte die Entwicklung einer neuen Typhierarchie erfordert, die umfangreicher als erwartet gewesen wäre (siehe Abs. 6.7).

9.4. Ausblick

Auf Basis der skizzierten Regelerweiterungen für Refactorings ließen sich weitere JDT Refactorings adaptieren, bis eine vollständige Adaption aller Refactorings erreicht ist.

Die neuen OT/J Refactorings müssten in der Praxis erprobt werden um zu beurteilen wie nützlich sie im alltäglichen Gebrauch sind. Außerdem wären noch einige Verfeinerungen notwendig, sowie eine Implementierung der vollständigen Überprüfung aller Vorbedingungen und Beachtung aller Sprachkonzepte. Mit der Implementierung weiterer OT/J Refactorings könnten die verschiedenen Refactorings miteinander kombiniert werden und dadurch auch zyklische Strukturen verarbeiten, die mehrere Refactoringschritte benötigen würden (siehe Abs. 7.1.5). Ein paar sinnvolle Kombinationen wurden bereits im Abschnitt 7.1 vorgestellt.

Interessant wäre auch ein Vergleich mit Refactoringwerkzeugen, die für andere aspektorientierte Programmiersprachen entwickelt wurden. Hier könnten eventuell Erkenntnisse übertragen und ein Vergleich im Umgang mit der Erhaltung der Semantik gezogen werden.

Der interessantes Punkt für weitere Forschungen wäre die Zusammenarbeit zwischen Refactoring und der sich zur Zeit in der Entwicklung befindlichen Joinpoint Sprache für OT/J. Hier wird an einer logischen Beschreibung für Joinpoints gearbeitet, die robuster als herkömmliche Joinpoint Sprachen ist, da diese mit dem expliziten Namen auf eine Methode verweisen. Eine logische Joinpoint Sprache könnte dem Refactoringwerkzeug präzisere Informationen bereitstellen und dadurch die Wahrung der Joinpoints erleichtern oder sogar automatische Ergänzungen ermöglichen.

A. Anhang

A.1. Abkürzungsverzeichnis

JDT Java Development Tooling
OTDT Object Teams Development Tooling
LTK Refactoring Language Toolkit
OT/J ObjectTeams/Java
OTJLD ObjectTeams/Java Language Definition
PDE Plug-in Development Environment

A.2. Überarbeitete Regeln zur Verhaltenserhaltung in ObjectTeams/Java

Die Regeln zur Verhaltenserhaltung stammen aus der Diplomarbeit von Gregor Brcan[Brc05] und enthalten die in Abschnitt 4.2.1 und 4.3.2 formulierten Ergänzungen. Die referenzierten Paragraphen verweisen dabei auf die OTJLD [HHM09].

A.2.1. Sprachregeln für ObjectTeams/Java

OTL1. Regeln für Teamklassen:

(a) Eine Teamklasse kann nur von einer anderen Teamklasse erben (§1.3). Eine Klasse, die von einer Teamklasse erbt, muss den Modifikator **`team`** haben (§A.1(b)).

OTL2. Regeln für Rollenklassen:

(a) Eine Rollenklasse darf nur **`public`** oder **`protected`** sein (§1.2.1(a)).

(b) Eine Rollenklasse muss mindestens die Sichtbarkeit ihrer impliziten Superrolle haben (§1.3.1(h)).

(c) Eine Rollenklasse kann explizit (unter Verwendung von **`extends`** von einer regulären Klasse oder einer Rollenklasse eines umschließenden Teams erben (§1.3.2(a)).

(d) Eine Rollenklasse darf nicht denselben Namen haben wie eine Methode oder ein Feld ihres umschließenden Teams. Außerdem darf der Name einer Rolle keine sichtbaren Typen des umschließenden Teams verdecken (*Shadowing*) (§1.4(a)).

(e) Eine gebundene Rollenklasse (mit **`playedBy`**), darf nicht **`static`** deklariert werden, und muss direkt in einer Teamklasse enthalten sein (§A.1(b)).

(f) Rollenklassen dürfen nicht die Namen „IConfined“, „Confined“ oder „ILowerable“ haben (§7.).

OTL3. Regeln für Team- und Rollenverschachtelung:

(a) Ist eine Rollenklasse mit dem Team-Modifikator **`team`** markiert (verschachteltes Team bzw. Rolle), darf diese Klasse nur von einer Teamklasse erben (siehe L1.(a)).

(b) Eine reguläre Rollenklasse (das heißt, nicht mit **`team`** markiert), kann lokale und anonyme Typen, jedoch keine Member-Typen enthalten (§1.5(b)).

(c) Ein verschachteltes Team darf nicht von seinem eigenen, umschließenden Team erben (§1.5(c)).

OTL4. Regeln für externe Rollen (*Externalized Roles*):

(a) Eine Rollenklasse, auf die außerhalb ihres Teams zugegriffen wird, muss **`public`** sein (§1.2.2(a)).

(b) In der Deklaration mit einem Anchored Type muss die Expression auf der linken Seite auf eine Instanz einer Teamklasse verweisen und der Typ auf der rechten Seite muss der einfache Name einer Rolle, die in diesem Team enthalten ist, sein. Dies beinhaltet auch implizit geerbte Rollenklassen (§1.2.2(b)).

OTL5. Regeln für Rollendateien (*Role Files*):

(a) Der Name des Verzeichnisses, in dem Rollendateien abgelegt werden, entspricht dem Namen des Teams (ohne .java Endung) (§1.2.5(a)).

(b) Der Name einer Rollendatei entspricht dem Namen der darin gespeicherten Rollenklasse (um .java erweitert, §1.2.5(b)).

(c) Eine Rollenklasse in einer Rollendatei muss als Package den voll qualifizierten Namen der umschließenden Teamklasse sowie den Modifikator **`team`** deklarieren (§1.2.5(c)).

OTL6. Regeln für Methoden-Bindungen (Callout- und Callinbindungen):

(a) Callout- und Callinbindungen dürfen nur in gebundenen Rollenklassen vorkommen (§A.3(a)).

(b) In einer Calloutbindung muss auf der linken Seite eine Rollenmethode und auf der rechten Seite eine Methode aus der gebundenen Basisklasse (Basismethode) referenziert sein.

(c) In einer Callinbindung muss auf der linken Seite eine Rollenmethode und auf der rechten Seite eine (oder mehrere) Methode(n) aus der gebundenen Basisklasse referenziert sein.

(d) In einer Callout to Field-Bindung muss auf der linken Seite eine Rollenmethode und auf der rechten Seite ein Feld der gebundenen Basisklasse referenziert sein.

(e) In einer Callout- oder Callinbindung dürfen Methodennamen und vollständige Signaturen für die Methoden-Bezeichner nicht vermischt werden (§A.3(b)).

(f) In einer Callin Replace-Bindung muss der Methoden-Bezeichner auf der linken Seite auf eine Methode mit **`callin`**-Modifikator verweisen (§A.3(c)).

(g) Der Modifikator **`callin`** ist nur für Methoden von Rollen erlaubt. Eine Callinmethode darf keine Sichtbarkeit definieren (§4.2(d)).

OTL7. Regeln für Parameter Mappings:

(a) In einem Callout Parameter Mapping muss auf der linken Seite eine Expression und auf der rechten Seite der Name eines Parameters der Basismethode angegeben sein.

(b) In einem Callin Parameter Mapping muss auf der linken Seite der Name eines Parameters der Rollenmethode und auf der rechten Seite eine Expression angegeben sein.

(c) Jeder Bezeichner, der innerhalb einer Expression eines Parameter Mappings vorkommt, muss im Scope der Rolleninstanz sichtbar sein (§3.2(d)).

OTL7.1. Implizite Parameter Mappings (§3.2(e)):

(a) Jeder Parameter der Rollenmethode muss konform sein zu dem korrespondierenden Parameter der Basismethode.

(b) Der Rückgabetyp der Basismethode muss konform sein zu dem Rückgabetyp der Rollenmethode.

OTL8. Beim *Declared Lifting* muss der zweite Typ in der Deklaration (die Rollenklasse) eine Rolle des ersten Typs (die Basisklasse) sein. Darüber hinaus muss der deklarierte Rollen-Typ eine Rolle der umschließenden Teamklasse sein, die die Lifting-Methode definiert (§2.3.2(a)).

OTL9. Entlang impliziter Vererbung dürfen die Namen von Rollenmethoden oder Feldern nicht irgendwelche vorher sichtbaren Namen verstecken bzw. verdecken (§1.4(b)).

OTL10. Regeln für **`playedBy`** (Typ-Bindung):

(a) Die Basisklasse irgendeiner Rollenklasse darf nicht eine Rolle desselben Teams sein (§2.1.2(a)).

(b) Eine Basisklasse, die hinter dem Schlüsselwort **`playedBy`** angegeben ist, darf von keiner Rollenklasse des umschließenden Teams verdeckt werden. Das heißt, Rollenklasse und Basisklasse müssen verschiedene Namen haben (§2.1.2.(a)). Eine Ausnahme bilden Basisklassen, die mit einem Base Import importiert wurden (§2.1.2.(d)).

(c) Die hinter **`playedBy`** angegebene Basisklasse, darf nicht ein umschließender Typ (auf jeglicher Tiefe) der definierten Rollenklasse sein (§2.1.2(b)).

OTL11. Regeln für **`within`** (Team-Aktivierung):

(a) Die hinter dem Schlüsselwort **`within`** angegebene Expression muss eine Team-Instanz bezeichnen (§5.2(a)).

OTL12. Regeln für Guard Predicates:

(a) Der Ausdruck in einem Guard muss vom Typ `boolean` sein (§A.7.(a)).

(b) Die Bezeichner in dem Ausdruck eines Guards müssen innerhalb des *Scopes* des deklarierenden Teams oder der deklarierenden Rolle sichtbar sein (§5.4.1.(a)–(d)).

(c) Die Bezeichner in dem Ausdruck eines Base Guards dürfen Features der Basis referenzieren, jedoch keine Features der Rolle (§5.4.2.(d)).

OTL13. Regeln für Precedence:

(a) Der Ausdruck in einer Precedence Deklaration besteht aus einer Liste von Namen, die auf eine Callinbindung verweisen. Eine Precedence Deklaration befindet sich in einem Team oder einer Rolle (§4.8.(a)).

(b) In einem Team muss der qualifizierte Name einer Callinbindung angegeben werden. Innerhalb einer Rolle können Callinbindung mit ihrem unqualifiziertem Namen referenziert werden (§4.8.(b)).

(c) In einem Team können alle Callinbindungen einer Rolle mit dem Namen der Rolle referenziert werden (§4.8.(c)).

(d) Falls innerhalb eines Teams mehrere Callinbindungen eine Basismethode referenzieren und dabei den gleichen Modifier (**`before`**, **`after`** oder **`replace`**) verwenden, muss eine Precedence Ausdruck angegeben werden.

OTL14. Regeln für Rollenkapselung:

(a) Auf Features von Rollen, die das Interface `IConfined` implementieren, darf außerhalb des umschließenden Teams nicht zugegriffen werden (§7.1.).

(b) Auf Rollen, die von `Team.Confined` erben, darf es außerhalb des umschließenden Teams keine Referenzen geben (§7.2.).

A.2.2. Regeln für die Erhaltung von Vererbungsbeziehungen in ObjectTeams/Java

OTI1. Wenn eine von einer regulären Klasse oder einem Interface geerbte Methode in einer Rollenklasse überschrieben bzw. implementiert wird, und die überschriebene bzw. implementierte Methode verändert wird, müssen diese Änderungen an der Methode in regulären Subklassen und in expliziten und impliziten Subrollen als auch in Supertypen (Klassen und Interfaces) berücksichtigt werden. Das heißt, die Signaturen von überschreibenden Methoden in regulären Subklassen und in Subrollen sowie die Signaturen von überschriebenen (geerbten) Methoden in Supertypen (Klassen und Interfaces) müssen ebenfalls geändert werden.

OTI2. Wenn eine implizite Subrolle die Extends-Relation der Superrolle verändert, muss die neue Superklasse eine Subklasse der Klasse sein, die in der Extends-Relation der impliziten Superrolle deklariert ist (§1.3.2(b)).

OTI3. Wenn eine explizite Subrolle die playedBy-Relation der Superrolle verändert, muss die neue Basisklasse eine Subklasse der Klasse sein, die in der playedBy-Relation der expliziten Superrolle deklariert ist (§2.1(c)).

OTI4. Eine implizite Subrolle darf die geerbte playedBy-Relation nicht ver- ändern (§2.1(d)).

OTI5. Existiert in einer Rollenklasse eine Methoden-Bindung, deren Methoden- Bezeichner nur aus den Methodennamen (ohne vollständige Signatur) bestehen, darf nach dem Refactoring keine (geerbte) Methode existieren, die die in der Methoden-Bindung referenzierte Rollen- oder Basis-Methode überlädt (kein Overloading).

OTI6. Rollen können zusätzlich zu den expliziten Super- und Subklassen implizite Super- und Subrollen haben. Eine Rolle erbt von einer Rolle eines Superteams ihres umschließenden Teams, wenn sie den gleichen Namen hat. Um die Vererbungsbeziehungen zu erhalten, müssen die Namen der Super- und Subrollen konsistent gehalten werden (§1.3.1.(c)).

A.2.3. Regeln für die Erhaltung semantischer Äquivalenz in ObjectTeams/Java

OTS1. Unreferenzierte Variablen (Felder), Methoden und Typen können im OO-Code (Basis-Code) hinzugefügt oder entfernt werden. Beim Entfernen dürfen keine Referenzen auf diese Elemente in Teamklassen, in Rollenklassen oder implizit geerbte Referenzen existieren. Rollen dürfen außerdem keine impliziten Subrollen haben, es sei denn sie haben einen leeren Rumpf. Beim Hinzufügen von Methoden und Typen dürfen außerdem keine Mehrdeutigkeiten durch *Shadowing* (OTL2. (d)) oder *Overloading* (OTI5.) produziert werden. Neu hinzugefügte Methoden dürfen keine in der impliziten Vererbungshierarchie existierende Methoden überschreiben oder von einer solchen überschrieben werden.

Literaturverzeichnis

[BC90] Bracha, Gilad und William Cook: *Mixin-based Inheritance*. In: *OOPSLA/ECOOP '90*, Seiten 303–311, New York, NY, USA, 1990. ACM Press.

[Bol03] Bolour, Azad: *Notes on the Eclipse Plug-in Architecture*, 2003. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html, [Online; Stand 14. Juli 2009].

[Brc05] Brcan, Gregor: *Erweiterung von objektorientiertem Refactoring für die aspektorientierte Sprache ObjectTeams/Java*. Diplomarbeit, TU-Berlin, 2005.

[Dij74] Dijkstra, Edsger W.: *On the role of scientific thought*, August 1974.

[DW02] Dudziak, Thomas und Jan Wloka: *Tool-Supported Discovery and Refactoring of Structural Weaknesses in Code*. Diplomarbeit, TU-Berlin, 2002.

[ecl09a] eclipse.org: *JDT Bug 139144*, 2009. https://bugs.eclipse.org/bugs/show_bug.cgi?id=139144, [Online; Stand 8. Oktober 2009].

[ecl09b] eclipse.org: *JDT Bug 286221*, 2009. https://bugs.eclipse.org/bugs/show_bug.cgi?id=286221, [Online; Stand 8. Oktober 2009].

[ecl09c] eclipse.org: *JDT Bug 286224*, 2009. https://bugs.eclipse.org/bugs/show_bug.cgi?id=286224, [Online; Stand 8. Oktober 2009].

[ecl09d] eclipse.org: *JDT Bug 290430*, 2009. https://bugs.eclipse.org/bugs/show_bug.cgi?id=290430, [Online; Stand 8. Oktober 2009].

[ecl09e] eclipse.org: *JDT Bug 290698*, 2009. https://bugs.eclipse.org/bugs/show_bug.cgi?id=290698, [Online; Stand 8. Oktober 2009].

[EF09] Eclipse-Foundation: *aspectj — crosscutting objects for better modularity*, 2009. http://www.eclipse.org/aspectj/, [Online; Stand 1. Juni 2009].

[Ern99] Ernst, Erik: *Propagating Class and Method Combination*. In: *In Proceedings ECOOP '99, volume 1628 of LNCS, 67–91*, Seiten 67–91. Springer-Verlag, 1999.

[FF01] Filman, Robert E. und Daniel P. Friedman: *Aspect-Oriented Programming is Quantification and Obliviousness*, 2001.

[Fow99] Fowler, Martin: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fre06] Frenzel, Leif: *The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs*. Eclipse Magazine, Januar 2006.

[GHJV95] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, 1995.

[GJSB05] Gosling, James, Bill Joy, Guy Steele und Gilad Bracha: *Java(TM) Language Specification*. Addison-Wesley Professional, 3. Auflage, 2005.

[Her02] Herrmann, Stephan: *Object Teams: Improving Modularity for Crosscutting Collaborations*. In: *Net.ObjectDays*, Seiten 248–264. Springer, 2002.

[HHM07] Herrmann, Stephan, Christine Hundt und Marco Mosconi: *ObjectTeams/Java Language Definition — version 1.0*. Technischer Bericht, Fak. IV, Technical University Berlin, 2007.

[HHM09] Herrmann, Stephan, Christine Hundt und Marco Mosconi: *ObjectTeams/Java Language Definition — version 1.2*, 2009. http://www.objectteams.org/def/OTJLDv1.2-final.pdf, [Online; Stand 10. August 2009].

[HM07] Herrmann, Stephan und Marco Mosconi: *Integrating Object Teams and OSGi: Joint Efforts for Superior Modularity*. Technischer Bericht, TU-Berlin, 2007.

[HO93] Harrison, William und Harold Ossher: *Subject-Oriented Programming: A Critique of Pure Objects*. In: *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, Band 28, Seiten 411–428. ACM Press, October 1993.

[KIL+97] Kiczales, Gregor, John Irwin, John Lamping, Jean Marc Loingtier, Cristina Videira Lopes, Chris Maeda und Anurag Mendhekar: *Aspect-Oriented Programming*, 1997.

[OA09] OSGi-Alliance: *OSGi – The Dynamic Module System for Java*, 2009. http://www.osgi.org, [Online; Stand 5. Oktober 2009].

[obj] objectteams.org: *Patterns of good design with OT/J*. http://trac.objectteams.org/ot/wiki/OtPatterns, [Online; Stand 11. September 2009].

[obj09] objectteams.org: *Improve specification of callin precedence*, 2009. http://trac.objectteams.org/ot/ticket/328, [Online; Stand 25. November 2009].

[Opd92] Opdyke, William F.: *Refactoring Object-Oriented Frameworks*. Dissertation, University of Illinois, 1992.

[Rob99] Roberts, Donald B: *Practical Analysis for Refactoring*. Dissertation, University of Illinois at Urbana-Champaign, 1999.

[Rur03] Rura, Shimon: *Refactoring Aspect-Oriented Software*. Diplomarbeit, Williams College, Williamstown, Massachusetts, 2003.

[SMHS06] Sokenou, Dehla, Katharina Mehner, Stephan Herrmann und Henry Sudhof: *Patterns for Re-usable Aspects*, 2006. Net.ObjectDays, Erfurt.

[ST09] Steimann, Friedrich und Andreas Thies: *From Public to Private to Absent: Refactoring JAVA Programs under Constrained Accessibility*. In: *ECOOP 2009 - Object-Oriented Programming, 23nd European Conference*, Seiten 419—443. Springer, 6.-10. Juli, 2009.

[SVEdM09] Schäfer, Max, Mathieu Verbaere, Torbjörn Ekman und Oege de Moor: *Stepping Stones over the Refactoring Rubicon Lightweight Language Extensions to Easily Realise Refactorings*. In: *ECOOP 2009 - Object-Oriented Programming, 23nd European Conference*, Seite 369–393. Springer, Juli 6-10, 2009.

[TKB03] Tip, Frank, Adam Kieżun und Dirk Bäumer: *Refactoring for generalization using type constraints*. In: *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2003)*, Seiten 13–26, Anaheim, CA, USA, November 6–8, 2003.

[TOHJ99] Tarr, Peri, Harold Ossher, William Harrison und Jr: *N degrees of separation: multi-dimensional separation of concerns*. In: *ICSE '99: Proceedings of the 21st international conference on Software engineering*, Seiten 107–119, New York, NY, USA, 1999. ACM.

[Wid06] Widmer, Tobias: *Unleashing the Power of Refactoring*. Eclipse Magazine, Juli 2006.

[Wlo06] Wloka, Jan: *Aspect-aware Refactoring tool support*. Technischer Bericht, Frauenhofer FIRST, 2006.