Erweiterung von objektorientiertem Refactoring für die aspektorientierte Sprache ObjectTeams/Java

Diplomarbeit

vorgelegt von

Gregor Brcan
laibach@cs.tu-berlin.de

Gutachter:
Prof. Dr. Stefan Jähnichen
Dr. Stephan Herrmann

Betreuer:
Dipl.-Inf. Jan Wloka

Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik
Technische Universität Berlin

Berlin, Oktober 2005

Zusammenfassung

Refactoring und aspektorientierte Programmierung (AOP) sind Konzepte, die die gemeinsamen Ziele verfolgen, die Modularität eines Softwaresystems zu erhöhen und die Verständlichkeit und Wartbarkeit von objektorientiertem Code zu verbessern.
Refactoring ist eine Technik, mit der existierender Code umstrukturiert werden kann, ohne dabei das Verhalten der Anwendung zu verändern.
Die aspektorientierte Programmierung ist ein Ansatz, um die Modularisierung von sogenannten *crosscutting concerns* zu erleichtern.
Aspektorientierte Programmiersprachen stellen neue, aspektorientierte Sprachelemente zur Verfügung, mit denen *crosscutting concerns* gekapselt werden können bzw. mit denen vorhandene Programme nachträglich erweitert werden können.
Um Refactoring und aspektorientierte Sprachen gemeinsam verwenden zu können, müssen objektorientierte Refactorings derart angepasst werden, dass sie die neuen Sprachelemente berücksichtigen und das Verhalten eines aspektorientierten Programms erhalten.
In dieser Arbeit werden bestehende, objektorientierte Refactorings für die aspektorientierte Sprache ObjectTeams/Java adaptiert.
Es wird gezeigt, welche Auswirkungen die neuen Sprachelemente in ObjectTeams/Java auf existierende Refactorings haben.
Es werden neue Regeln aufgestellt, die die Erhaltung des Verhaltens in einem ObjectTeams/Java Programm garantieren.
Die Einhaltung der neuen Regeln wird anhand zusätzlicher Vor- und Nachbedingungen überprüft, die für jedes Refactoring definiert werden.
Drei ausgewählte, konzeptuell erweiterte Refactorings werden exemplarisch in der Entwicklungsumgebung Eclipse implementiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Software Evolution

Software, die verwendet wird, muss laufend den sich ändernden Anforderungen angepasst werden, das heißt sie muss gewartet werden. Leider führt Software-Wartung in der Regel zur Degeneration der Struktur der Software. Dadurch sind weitere Änderungen immer schwieriger durchzuführen, bis schließlich nur noch eine Neuimplementierung sinnvoll ist.
In [16] hat Lehman diese Entwicklung beobachtet und festgestellt, dass es sich bei diesem Phänomen um eine Art Naturgesetz handelt. Durch kontinuierliche Änderungen an einem Softwaresystem steigt seine Komplexität (dies wird auch als *Software Entropie* bezeichnet) und seine Struktur zerfällt, falls nicht Maßnahmen unternommen werden, um die Komplexität zu reduzieren und die Struktur zu verbessern. Eine Möglichkeit, dieser „natürlichen" Degeneration der Software-Struktur entgegenzuwirken, ist die kontinuierliche Software-Restrukturierung, die dafür sorgt, dass die Software jederzeit verständlich und änderbar bleibt. Bei objektorientierter Software wird anstelle von Software-Restrukturierung auch von **Software-Refactoring** gesprochen.

1.2 Refactoring

Fowler definiert den Begriff Refactoring in [7] folgendermaßen:

> „Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure."

Das Grundprinzip von Refactoring ist, die Struktur (das Design) des Codes zu verbessern, ohne dabei das externe, beobachtbare Verhalten der Applikation zu verändern. Beim Refactoring geht es um die Restrukturierung des

Systems und nicht um Fehlerkorrektur oder die Erweiterung der Anwendung.
Es ist wichtig zu definieren, was mit Verhaltenserhaltung gemeint ist. Die Definition von Verhaltenserhaltung ist system- bzw. anwendungsabhängig (siehe Abschnitt 3.1). Es gibt auch Änderungen an der Struktur eines Systems, die zwar gewünscht sind, aber nicht verhaltenserhaltend sind und somit kein Refactoring darstellen.
Es gibt bereits zahlreiche Refactorings (auch *Refactoring-Patterns* genannt), die ein Codefragment, ausgehend von einem definierten Anfangszustand, mittels einer Folge von wohl formulierten Transformationen in einen definierten Endzustand überführen. Fowler hat insgesamt über siebzig solcher Refactorings in einem Katalog[7] definiert, der auch online[6] verfügbar ist. Darin beschreibt er, neben dem Namen des Refactorings, in welchen Fällen das Refactoring angewendet wird, was es leistet und warum es eingesetzt werden sollte. Zusätzlich gibt er für jedes Refactoring eine Anleitung für die schrittweise Durchführung des Refactorings sowie ein einfaches Beispiel an.

1.2.1 Ziele von Refactoring

Fowler nennt als primäre Ziele von Refactoring, eine erhöhte Überschaubarkeit und Verständlichkeit des Codes und eine vereinfachte Wartbarkeit. Davon abgeleitet nennt Wloka in [31] noch speziellere Ziele wie wohldefinierte Verantwortlichkeiten, Code mit niedriger Komplexität, die Eliminierung duplizierten Codes (Entfernung von Redundanz) und das Auffinden von Fehlern. Er führt an, dass Refactoring im Allgemeinen die Struktur in kleinere Typen und Methoden zerlegt, sie entkoppelt und vereinfacht. Dies verbessert die Lesbarkeit, Modularität und damit die *Evolvability*[1] eines Softwaresystems.

1.2.2 Automatisiertes, werkzeuggestütztes Refactoring

Manuell durchgeführtes Refactoring erfordert ein schrittweises, iteratives Vorgehen, um sicherzustellen, dass durch die Änderungen an der Struktur keine neuen Fehler in die Software eingeführt werden. Das Vorhandensein von umfangreichen und stabilen Tests war und ist auch heute noch eine wesentliche Vorbedingung für die Durchführung von Refactoring.
Roberts stellt in [7] fest, dass manuell durchgeführtes Refactoring sehr zeitaufwendig, fehlerträchtig und damit auch kostenintensiv ist. Dies war in der Vergangenheit eines der größten Hindernisse bei der Umstrukturierung von Code.
Durch den Einsatz heutiger Werkzeuge kann der Arbeitsaufwand, der beim manuellen Refactoring anfällt, sowie die Kosten erheblich reduziert werden. Werkzeuge können den Entwickler bei der Durchführung von Refactorings

unterstützen und Refactorings automatisiert durchführen.
Ein Werkzeug ist wesentlich schneller und effizienter und kann manuelle Fehler beim Ändern von Code vermeiden. Es leistet wichtige Hilfestellung bei der vorausgehenden Analyse des Codes und kann automatisch überprüfen, ob ein Refactoring anwendbar ist. Solche Überprüfungen beinhalten Dinge wie syntaktische Korrektheit und semantische Aspekte. Darüber hinaus kann ein Werkzeug auch bestimmte sprachspezifische Eigenschaften, wie zum Beispiel Polymorphismus und Redefinition, berücksichtigen.
Die Erhaltung des Verhaltens ist einfacher mit Werkzeugen zu bewerkstelligen, da ein Werkzeug die Einführung von Fehlern effektiver verhindert und garantiert, dass das Verhalten erhalten bleibt.
Es gibt bereits zahlreiche Werkzeuge, die automatisierte Refactoring-Unterstützung anbieten. In den integrierten Entwicklungsumgebungen Eclipse[4] und IDEA[11] sind zahlreiche der in Fowlers Standardwerk[7] und im Online-Katalog[6] aufgestellten Refactorings implementiert.

1.2.3 Anwendung von Refactoring

Der schwierigste bzw. problematischste Teil beim Refactoring ist nicht das Refactoring selbst, sondern die Frage, wo und an welchen Stellen im Code Refactoring eingesetzt werden soll.
Wie bereits erwähnt, führen häufige Änderungen an einem Softwaresystem mit der Zeit zum Zerfall der initialen Struktur, wodurch weitere Änderungen immer schwieriger werden. Dudziak und Wloka bezeichnen in ihrer Arbeit[3] den Teil eines Systems, dessen Struktur den Entwickler daran hindert das System zu ändern, als „strukturelle Schwäche“ (*Structural Weakness*).
Häufig werden Stellen, die einen Schwachpunkt im System darstellen, aufgrund der Erfahrung und der Intuition des Entwicklers entdeckt. Gleichwohl gibt es auch Indikatoren für strukturelle Schwächen im Code. Die Anzeichen, die auf strukturelle Schwachstellen im Design bzw. Code hinweisen, werden als *Bad Smells* bezeichnet. Beck und Fowler haben einige solcher „Bad Smells“ in [7] zusammengetragen, die Eigenschaften von Systemteilen benennen, die eine Schwachstelle darstellen und die von einem Refactoring entfernt werden können.
Dudziak und Wloka beschreiben in ihrer Arbeit verschiedene Ansätze, um strukturelle Schwächen (automatisch) zu entdecken (sie bezeichnen dies als *Weakness Analysis*), und präsentieren entsprechende Werkzeugarten.

1.2.4 Probleme mit Refactoring

Auf einigen Gebieten kann Refactoring zu Problemen führen oder es kann gar nicht eingesetzt werden. Einer der von Fowler in seinem Buch[7] beschriebenen Problembereiche sind Datenbanken. Viele Anwendungen sind eng an

ein Datenbankschema gekoppelt, wodurch eine Änderung an der Datenbank sehr schwierig ist. Ein weiteres Problem stellen öffentliche Schnittstellen dar, die nicht mehr geändert werden können. Fowler stellt fest, dass Refactoring auch negativen Einfluss auf die Leistung und Geschwindigkeit von Software haben kann. Weitere Probleme im Zusammenhang mit Refactoring sind von ihm in [7] ausführlich beschrieben.

1.2.5 Grenzen von Refactoring

Die mit Refactoring durchgeführten Verbesserungen des Designs sind begrenzt durch objektorientierte Modularisierungskonzepte.
Wloka folgert in [31] daraus, dass deshalb einige Designkonflikte nicht gelöst werden können und Komplexität häufig nur lokal reduziert wird. In solchen Fällen werden komplexe Strukturen nicht entkoppelt oder entfernt, sie werden nur verschoben. Präziser gesagt, wann immer eine bestimmte Struktur an einer Stelle vereinfacht wird, wird noch mehr Komplexität in anderen Strukturen eingeführt. Die globalen Abhängigkeiten zwischen verschiedenen Teilen eines Systems sind häufig schwierig umzustrukturieren.
Solche Designkonflikte werden oft durch sogenannte *crosscutting concerns* verursacht. In [13] beschreibt Laddad einen *concern* als ein bestimmtes Ziel, Konzept, oder Interessensgebiet. Der Begriff *crosscutting* deutet darauf hin, dass ein *concern* mehrere Implementationsmodule beeinflusst.
In objektorientierten Systemen ist die Implementierung eines solchen *crosscutting concern* über viele Typen und Methoden verteilt. Typische Beispiele für *crosscutting concerns* in einem Softwaresystem sind Logging, Synchronisation, Exception-Handling, kontext-sensitive Fehlerbehandlung und Leistungsoptimierung.
Wloka kommt in [31] zu dem Ergebnis, dass globale Designverbesserungen anhand von Refactoring durch *crosscutting concerns* erschwert werden. Daher könnten sie als ein Indikator für die Schwäche von objektorientierten Modularisierungskonzepten angesehen werden. Ein neues Konzept für die Kapselung dieser *crosscutting concerns* ist erforderlich.

1.3 Aspektorientierte Programmierung (AOP)

Durch die Verwendung objektorientierter Programmiermethodiken und Techniken erstrecken sich *crosscutting concerns* über mehrere Module, was zu Systemen führt, die schwieriger zu entwerfen, zu verstehen, zu implementieren und zu erweitern sind. In [13] nennt Laddad einige Symptome, die darauf hindeuten, dass eine Implementierung von *crosscutting concerns* unter Benutzung von aktuellen Methodiken problematisch ist. Er teilt diese Symptome allgemein in zwei Kategorien ein: Code tangling und code scattering.

Code tangling (Verflechtung) bedeutet, dass bestimmte Module in einem Softwaresystem gleichzeitig mehrere Anforderungen implementieren, was zu einer Vermischung von unterschiedlichen Funktionalitäten in einem Implementierungsmodul führt.
Code scattering (Zerstreuung) heißt, dass die Implementierung eines *crosscutting concern*, wie der *concern* selbst, sich über mehrere Module erstreckt bzw. zerstreut ist.
Laddad schlussfolgert, dass *code tangling* und *code scattering* zusammen zu geringerer Produktivität, weniger Code-Wiederverwendung, schlechter Code-Qualität und schwierigerer Evolution führen.
Die neue Methodik der **aspektorientierten Programmierung (AOP)** trennt *concerns* besser als vorherige Programmieransätze und erleichtert dadurch die Modularisierung von *crosscutting concerns* (*separation of concerns*). Laddad ist der Auffassung, dass mit AOP modularisierte Implementierungen von *crosscutting concerns* erzeugt werden können, die einfacher zu entwerfen, zu verstehen und zu warten sind. Sie weisen außerdem eine minimale Kopplung und weniger duplizierten Code auf. AOP hilft, die durch *code tangling* und *code scattering* verursachten Probleme zu überwinden. Weitere Vorteile, die AOP bietet, sind zum einen eine höhere Produktivität und verbesserte Qualität und zum anderen eine einfachere Erweiterung von Systemen sowie mehr Code-Wiederverwendung.
Einen guten Überblick und eine Einführung in die aspektorientierte Programmierung findet man in einem Standard-Werk zu diesem Thema[5].

1.3.1 Aspektorientierte vs. objektorientierte Programmierung

Die objektorientierte Programmierung (OOP) zeigt ihre Stärken, sobald es darum geht, gemeinsames Verhalten zu modellieren. Sie stößt jedoch, wie bereits erwähnt, an ihre Grenzen, wenn es nötig ist Verhalten zu erfassen, dass sich über mehrere, oft nicht in Zusammenhang stehende Module, erstreckt.
In [13] hält Laddad fest, dass OOP Systeme durch Benutzung von lose gekoppelten, modularisierten Implementationen von *common concerns* erzeugt. AOP hingegen erzeugt Systeme unter Verwendung von lose gekoppelten, modularisierten Implementationen von *crosscutting concerns*.
OOP kapselt *common concerns*, also Dinge, die ein gemeinsames Verhalten aufweisen, in **Klassen**. Die Modularisierungseinheit in AOP wird als **Aspekt** bezeichnet.
AOP ermöglicht es, individuelle, lose gekoppelte *concerns* als Aspekte zu implementieren, und diese zu einem Gesamtsystem zu integrieren, oder mit einem bereits existierenden objektorientiertem System zu verknüpfen.

1.4 Aspektorientierte Programmiersprachen

Es existieren bereits mehrere aspektorientierte Sprachen, die AOP implementieren und explizite Sprachkonstrukte zur Modularisierung und Zusammensetzung von *crosscutting concerns* zur Verfügung stellen.
In den folgenden Abschnitten werden drei ausgewählte aspektorientierte Sprachen kurz vorgestellt.

1.4.1 AspectJ

AspectJ[9] ist eine aspektorientierte Erweiterung für Java. Um AOP zu unterstützten, fügt AspectJ folgende neue Sprachkonzepte bzw. Sprachkonstrukte zu Java hinzu: *Join-Points*, *Pointcuts*, *Advices* und *Aspekte*.
Join-Points, ein zentrales Konzept in AspectJ, sind wohldefinierte Punkte in der Ausführung eines Programms. Diese beinhalten unter anderem Konstruktor- und Methodenaufrufe, Ausführungen von Konstruktoren und Methoden und lesende bzw. schreibende Feldzugriffe. Ein Join-Point zum Beispiel, der einen Methodenaufruf darstellt, schließt alle Aktionen ein, aus denen ein Methodenaufruf besteht, beginnend nachdem alle Argumente ausgewertet sind bis einschließlich der Rückgabe.
Pointcuts sind Programmkonstrukte, die Join-Points auswählen bzw. eine Sammlung von Join-Points spezifizieren. Es gibt mehrere primitive Pointcut-Typen bzw. *pointcut designators* (z.B. `call`, `execution`, `get`, `set`, `handler`, `within`, `cflow`, `if`). Andere Pointcuts können anhand der `pointcut`-Deklaration benannt und definiert werden. Pointcuts können auch bestimmte Werte im Ausführungskontext eines Join-Points bereitstellen, die in einer Advice-Implementierung benutzt werden können.
Advices spezifizieren den Code, der an jedem Join-Point ausgeführt wird, wenn der jeweilige Pointcut erreicht wird. AspectJ bietet drei verschiedene Arten von Advices an: *Before-Advice*, *After-Advice*, und *Around-Advice*.
Ein Before-Advice wird vor dem Join-Point ausgeführt, während ein After-Advice nach dem Join-Point ausgeführt wird. Ein Around-Advice umschliesst einen Join-Point und hat die Kontrolle darüber, ob die Ausführung des Join-Points fortgesetzt werden soll.
Pointcuts und Advices bilden den dynamischen Teil von AspectJ. AspectJ hat auch verschiedene Arten von **Inter-Type Deklarationen**, die die statischen *crosscutting* Features in AspectJ bilden, mit denen die statische Struktur eines Programms modifiziert werden kann. Mit Inter-type Deklarationen können neue Methoden und Felder als Member von bestehenden Klassen oder in Interfaces deklariert werden (dies wird auch als *introduction* bezeichnet). Es können auch Vererbungsbeziehungen zwischen Klassen verändert werden, indem Super-Klassen und Interfaces von bestehenden Klassen deklariert werden (`declare parents`).

Aspekte sind die Modularisierungseinheit in AspectJ und stellen Pointcuts, Advices und Inter-type Deklarationen zusammen. Aspekte werden ähnlich wie Klassen definiert. Sie können neben den erwähnten Konstrukten auch Methoden und Felder enthalten, von anderen Klassen oder Aspekten erben und Interfaces implementieren.
AspectJ ist mittlerweile die bekannteste unter den aspektorientierten Sprachen und stellt den Standard dar. Eine ausführliche Beschreibung von AspectJ, sowie eine Einführung in die Sprache und zahlreiche Beispiele, finden sich im *AspectJ Programming Guide*[23] und in [14].

1.4.2 CaesarJ

CaesarJ[10] ist eine neue aspektorientierte Programmiersprache, die die wichtigsten Ziele im Software-Design anspricht: Modularität, Wiederverwendbarkeit, Flexibilität und Korrektheit. Sie basiert auf objektorientierten Konzepten und ergänzt Java, indem sie neue Konstrukte zur Verfügung stellt, die die Implementierung von *crosscutting concerns* ermöglichen. Darüber hinaus stellt CaesarJ noch andere wichtige Eigenschaften von Modularität sicher: Abstraktion, Kapselung, und die Minimierung von Abhängigkeiten. Aspekte sind als Komponenten entworfen, die eine klare Abstraktion aufweisen und wiederverwendbar sind.
CaesarJ verbessert die Trennung von *concerns* auf die gleiche Weise wie AspectJ (siehe 1.4.1). Es übernimmt das Join-Point-Modell, welches in AspectJ eingeführt wurde, als Basisgrundlage für die Modularisierung von *crosscutting concerns*. Pointcuts und Advices im Stil von AspectJ können benutzt werden, um Punkte abzufangen, an denen Komponenten-Funktionalität integriert werden soll.
CaesarJ modularisiert Komponenten, die aus mehreren kollaborierenden Klassen bestehen. Ein wichtiges Element ist das *collaboration interface*, welches aus einer Menge von zusammenhängenden Java Interfaces besteht und von der implementierten Komponente verwendet wird.
Das *Binding* stellt die Verknüpfung der Komponente mit der Anwendung dar und wird in einem separaten Modul definiert.
Eine Kollaboration wird verwendet, um eine Menge von zusammenarbeitenden Klassen zu gruppieren. Eine Kollaboration ist eine sogenannte **Caesar-Klasse**, die wiederum mindestens eine innere Caesar-Klasse enthält.
In CaesarJ ist jede innere (Caesar)-Klasse eine *virtuelle Klasse*[18], die in Sub-Klassen der umschließenden Klasse redefiniert bzw. überschrieben werden kann[1]. Mit virtuellen Klassen kann eine Basis-Kollaboration definiert werden, die in Sub-Kollaborationen mit neuen Features verfeinert werden kann.
Unabhängig voneinander entwickelte Features können durch Anwendung der

[1] Im Gegensatz zu Java, wo dies nicht möglich ist.

sogenannten **Mixin-Komposition** verschmolzen werden.
Ein **Mixin** ist eine Klasse mit einer parametrisierbaren Super-Klasse, das heißt, eine Caesar-Klasse kann von mehreren Super-Klassen erben. Dies ist der Unterschied zu einer einfachen Java Klasse, die eine feste Elternklasse hat und nicht in verschiedene Vererbungshierarchien umgeordnet werden kann[2].
Mixin-Komposition kann sowohl auf individuelle Klassen als auch auf Kollaborationen angewendet werden und hat einen propagierenden Effekt. Das bedeutet, wenn zwei Kollaborationen komponiert werden, dann werden ihre virtuellen Klassen mit denselben Namen wiederum anhand der Mixin-Komposition zusammengesetzt.
Eine ausführliche Beschreibung von CaesarJ und der darin eingeführten Konzepte sowie einiger Beispiele enthält die CaesarJ Sprachdefinition[10].

1.4.3 ObjectTeams/Java (OT/J)

1.4.3.1 Motivation

Die Frage, die sich im Vorfeld der Entwicklung von ObjectTeams/Java stellte, war, ob es Modulkonzepte für das Erfassen von Kollaborationen von Objekten gibt, die effektiv genutzt werden können, um *crosscutting concerns* in wiederverwendbaren und unabhängig voneinander entwickelten Modulen zu implementieren, und diese nachträglich in existierende Systeme zu integrieren. Obwohl es bereits einige Vorschläge und Ansätze für solche Modulkonstrukte gab, löst jeder dieser Vorschläge eine andere Teilmenge von Problemen bezüglich wiederverwendbarer Kollaborationsmodule.
Aufgrund dieser unzureichenden Lösungen wurde eine neue Art von Kollaborationsmodul, das sogenannte *Object Team* vorgeschlagen, welches die besten Eigenschaften mehrerer existierender Ansätze ([21], [17], [20], [27], und [9]) kombiniert. Es ergänzt diese mit Konzepten zum Beschreiben von Beziehungen zwischen unabhängigen Kollaborationen und ermöglicht so eine nachträgliche Integration dieser Kollaborationen in bestehende Systeme.

1.4.3.2 Sprachkonzepte

ObjectTeams/Java[28] ist eine aspektorientierte Programmiersprache die auf Java basiert und ein mächtiges Werkzeug für die Definition und Komposition von Kollaborationsmodulen (Object Teams) darstellt.
Ein solches Modul kombiniert die Eigenschaften von Klassen und Paketen und ist *das* Schlüsselkonzept in Object Teams.
Ein *Object Team* ist eine instanziierbare Aggregation von eingeschlossenen Objekten, **Rollen** genannt.

[2] In Java gibt es keine mehrfache Vererbung.

Das zusammengesetzte Objekt, *Team-Instanz* genannt, repräsentiert das **Team** und enthält alle teilnehmenden Rollen. Neben Rollen kann eine Team-Instanz auch eigene Features wie Attribute und Methoden haben.
Teams können durch Benutzung von Vererbung verfeinert werden. Die Vererbung zwischen Teams führt eine *implizite Vererbung* zwischen den enthaltenen Rollen ein: Alle Rollen eines Super-Teams werden vererbt. Wenn das Sub-Team eine Rolle mit dem gleichen Namen hat wie eine geerbte Rolle, dann wird letztere implizit überschrieben. Die Methoden und Attribute der überschriebenen Rolle werden von der neuen Rolle implizit geerbt.
Diese Features können wiederum anhand der bekannten Regeln überschrieben werden.
Die Komposition von einem Team (Aspekt-Code) mit einem bereits existierenden Paket der Applikation (Basis-Code) wird unter Verwendung objektbasierter Vererbung (auch als *Delegation* bekannt) realisiert. Dies geschieht durch die Deklarierung einer Rolle-Basis Beziehung mit dem Schlüsselwort `playedBy`. Dadurch ist es möglich, eine abstrakte Methode einer Rollenklasse durch eine existierende Methode einer Basisklasse zu realisieren. Ein Rollenobjekt bindet eine Methode, die lokal nicht verfügbar ist, durch das Weiterleiten an das assoziierte Basisobjekt. Diese Art von Methoden-Bindung wird als **Callout-Bindung** bezeichnet.
Im Gegensatz dazu wird bei einer **Callin-Bindung** eine Basismethode durch eine Rollenmethode „überschrieben“. Diese Art von Überschreiben wird durch das Einfügen von neuem Code in bestehende Klassen ermöglicht (*advice weaving*). Dabei kann der neue Code entweder vor oder nach der ursprünglichen Methode eingesetzt werden, oder diese sogar ersetzen (analog zum Before-, After- und Around-Advice in AspectJ).
Für eine ausführliche Beschreibung und Erläuterung aller in ObjectTeams/-Java vorhandenen Konzepte sowie technischer Details siehe Kapitel 2, [8], und die ObjectTeams/Java Sprachdefinition[2].

1.5 Aspektorientiertes Refactoring

Laddad stellt in [15] fest, dass Refactoring und aspektorientierte Programmierung, individuell gesehen, das Ziel haben Systeme zu erzeugen, die verständlicher und wartbarer sind, ohne größere vorherige Designanstrengungen zu fordern. Beides sind Konzepte für die Entkopplung, Dekomposition und Vereinfachung von objektorientiertem Code.
Aspektorientiertes Refactoring (AO-Refactoring), eine Kombination beider Konzepte, hilft dabei, Code bezüglich *crosscutting concerns* zu reorganisieren, um die Modularisierung noch weiter zu verbessern, und um die üblichen Problemsymptome wie *code tangling* und *code scattering* (siehe Abschnitt 1.3) zu überwinden.
Neben der Möglichkeit *crosscutting concerns* in einen Aspekt zu extrahieren

oder neue aspektorientierte Sprachelemente (z.B. *Push Down Advice*) umzustrukturieren, gibt es noch weitere Dimensionen von aspektorientiertem Refactoring, die im nächsten Abschnitt diskutiert werden.

1.5.1 Dimensionen von aspektorientiertem Refactoring

In [32] identifiziert Wloka folgende vier Dimensionen von aspektorientiertem Refactoring:

- Refactoring von Basis-Code (Aspect-aware Refactoring)
- Refactoring von Aspekt-Code
- Refactoring von Basis-Code nach Aspekt-Code
- Refactoring von Aspekt-Code nach Basis-Code

Wird ein Teil des Basis-Codes, das bedeutet bereits bestehender objektorientierter Code, refaktorisiert, so wird von **Aspect-aware Refactoring** gesprochen. Dem angewendeten Refactoring müssen dabei die existierenden Aspekt-Bindungen bekannt sein. Jeder Aspekt, der das umstrukturierte Element referenziert, muss aktualisiert werden.
Die zweite Dimension beinhaltet alle Fälle, in denen aspektorientierte Sprachelemente (z.B. Pointcuts) refaktorisiert werden sollen. Es werden also **aspektorientierte Refactorings** (AO-Refactorings) benötigt, die die neuen aspektorientierten Sprachelemente umstrukturieren, ähnlich wie objektorientierte Refactorings dies für Felder, Methoden, und Klassen durchführen.
Die dritte Dimension stellt die natürlichste Erweiterung von objektorientiertem Refactoring dar. Es können neue Refactorings eingeführt werden, die vorhandenen Basis-Code, der über mehrere Module verteilt ist, in einen Aspekt auslagern.
Falls ein Aspekt nicht häufig, an mehreren Stellen, benutzt wird, wäre es unter Umständen nützlich, seine Funktionalität in den Basis-Code zu verschieben und den Aspekt zu entfernen. Für solche Fälle, in denen die Verwendung von reinem objektorientiertem Code angebrachter ist, müssen ebenfalls neue AO-Refactorings entwickelt werden.
Jede dieser Dimensionen ist auf ein bestimmtes Ziel von AO-Refactoring gerichtet. In dieser Arbeit wird die erste Dimension, das Aspect-aware Refactoring, betrachtet.

1.6 Refactoring unter Berücksichtigung von Aspekten (Aspect-aware Refactoring)

1.6.1 Problemstellung

Per Definition muss ein angewendetes Refactoring das externe, beobachtbare Verhalten einer Anwendung erhalten (siehe 1.2). Die neuen Sprachkonzepte in AspectJ (1.4.1), CaesarJ (1.4.2) und ObjectTeams/Java (1.4.3) sowie die Mechanismen dieser Sprachelemente, die verwendet werden um ein bestehendes Java-Programm anzupassen, haben allerdings Auswirkungen auf das Verhalten einer aspektorientierten Anwendung und damit auch auf das Refactoring, welches die Verhaltenserhaltung garantieren soll.
Aufgrund der Verlinkung zwischen dem Aspekt-Code und dem Basis-Code (**Aspekt-Bindungen**), können neue implizite und explizite Referenzen auf den Basis-Code innerhalb des Aspekt-Codes (**Aspekt-Basis Referenzen**) existieren.
In AspectJ (siehe 1.4.1) werden Elemente aus dem Basis-Code in Inter-Type Deklarationen, in Deklarationen von neuen Vererbungsbeziehungen und in Pointcuts referenziert. Wird demnach ein Feld oder eine Methode als neues Member einer bestehenden Klasse deklariert, dann wird diese Klasse explizit anhand ihres Namens selektiert. Auf die gleiche Weise können neue Vererbungsbeziehungen spezifiziert werden.
Pointcuts in AspectJ und CaesarJ können auf viele unterschiedliche Arten Join-Points spezifizieren. Join-Points können direkt anhand ihrer Namen, ihrer strukturellen Eigenschaften oder aufgrund von Laufzeit-Informationen selektiert werden.
In ObjectTeams/Java werden Elemente aus dem Basis-Code in der `playedBy`-Relation und in den Methoden-Bindungen (Callin- und Callout-Bindungen) referenziert. Diese Elemente werden ebenfalls direkt anhand ihrer Namen selektiert.
Die explizite Referenzierung bestimmter Elemente aus dem Basis-Code im Aspekt-Code anhand ihrer Namen ist allerdings nur ein Bereich, der die Programmeigenschaften und damit das Verhalten einer aspektorientierten Anwendung beeinflusst wenn ein objektorientiertes Refactoring angewendet wird.
Jede der drei beschriebenen aspektorientierten Sprachen führt auch neue Vererbungsbeziehungen ein oder ermöglicht es, bestehende Vererbungshierarchien in einem Programm zu verändern.
In CaesarJ und ObjectTeams/Java gibt es das Konzept der virtuellen Klassen[3], wodurch innere Klassen unterschiedlicher umschließender Klassen von-

[3]Eine virtuelle Klasse in CaesarJ ist eine innere überschreibbare Caesar Klasse einer Kollaboration, während sie in ObjectTeams/Java eine überschreibbare Rolle des umschließenden Teams ist.

einander erben bzw. überschrieben werden können.
In AspectJ kann innerhalb eines Aspekts, durch die Deklaration einer neuen Super-Klasse einer existierenden Klasse, die Vererbungshierarchie dieser Klasse verändert werden.
Wendet man bestehende Refactorings an, so wie sie bisher definiert sind, dann könnte das Verhalten der aspektorientierten Anwendung verändert werden, da den Refactorings die neuen aspektorientierten Sprachelemente nicht bewusst sind. Diese Refactorings gehen von reinem objektorientierten Code aus. Daher sind ihnen die zusätzlichen Referenzen in den neuen aspektorientierten Sprachkonstrukten nicht bekannt. Veränderte Vererbungssemantiken, beispielsweise bezüglich Overriding, die sich aus den neuen bzw. geänderten Vererbungsbeziehungen ergeben können, werden ebenfalls nicht berücksichtigt.
Aufgrund dieser Probleme ist es daher zunächst notwendig diejenigen Spracheigenschaften und strukturellen Elemente der betrachteten aspektorientierten Sprache zu identifizieren, die Refactoring beeinflussen.
Es ist zu prüfen, welche Auswirkungen die Spracherweiterungen auf existierende objektorientierte Refactorings haben und welche Programmeigenschaften betroffen sind. Anschließend müssen die bestehenden Refactorings auf der Grundlage von syntaktischen und semantischen Sprachregeln so erweitert bzw. angepasst werden, dass bei deren Einsatz das Verhalten des aspektorientierten Programms beibehalten wird.

1.7 Beiträge dieser Arbeit

Im Rahmen dieser Arbeit soll untersucht werden, welche Auswirkungen die aspektorientierte Sprache bzw. die Sprachkonzepte in ObjectTeams/Java auf objektorientiertes Refactoring haben. Auf der Grundlage der gewonnenen Erkenntnisse sollen anschließend objektorientierte Refactorings für ObjectTeams/Java konzeptuell erweitert werden und einige davon beispielhaft implementiert werden.
Um diese Ziele zu erreichen, sollen erstens die vorhandenen Sprachkonzepte und Mechanismen in ObjectTeams/Java beschrieben werden sowie deren Auswirkungen auf bestehende Refactorings dargestellt werden.
Zweitens sollen Regeln für die Verhaltenserhaltung (syntaktische Korrektheit, Vererbungsbeziehungen, semantische Äquivalenz) in ObjectTeams/Java Programmen auf Basis bestehender Regeln für objektorientierte Sprachen (C++ und Java) definiert werden.
Drittens sollen, durch Anwendung der definierten ObjectTeams/Java Sprachregeln, die wichtigsten und am häufigsten verwendeten objektorientierten Refactorings konzeptuell erweitert werden. Dazu ist eine Überprüfung, Überarbeitung und Erweiterung der Vor- und Nachbedingungen der Refactorings notwendig.

Viertens sollen exemplarisch drei ausgewählte adaptierte Refactorings (OT/J-aware Refactorings) in der Entwicklungsumgebung Eclipse implementiert werden.
Abschließend sollen die wichtigsten Erkenntnisse bezüglich ObjectTeams/Java und Refactoring zusammengefasst präsentiert werden, sowie ein Vergleich zu anderen aspektorientierten Sprachen gezogen werden.

1.8 Aufbau der Arbeit

Die übrigen Kapitel dieser Arbeit setzen sich folgendermaßen zusammen:

Kapitel 2 behandelt die Sprachkonzepte des Programmiermodells Object Teams, sowie die Umsetzung dieser Konzepte und ihrer Mechanismen in ObjectTeams/Java. Anhand von Beispielen wird beschrieben, welche Auswirkungen ObjectTeams/Java Code auf objektorientiertes Refactoring hat.

In **Kapitel 3** wird ausführlich auf Verhaltenserhaltung eingegangen, der wichtigsten Bedingung beim Refactoring. Es werden zunächst Regeln für die Erhaltung des Verhaltens von objektorientierten Programmiersprachen (C++ und Java) präsentiert. Darauf aufbauend werden Regeln für die Sprache ObjectTeams/Java definiert, die die Verhaltenserhaltung in ObjectTeams/-Java Programmen garantieren.

Kapitel 4 umfasst einen Katalog von konzeptuell adaptierten objektorientierten Refactorings für ObjectTeams/Java (OT/J-aware Refactorings). Für jedes Refactoring sind die überarbeiteten und erweiterten Vor- und Nachbedingungen angegeben, die auf der Grundlage der in Kapitel 3 eingeführten Sprachregeln erstellt worden sind.

In **Kapitel 5** wird die Implementierung dreier ausgewählter, erweiterter Refactorings in der Entwicklungsumgebung Eclipse detailliert beschrieben.

Kapitel 6 liefert eine Zusammenfassung der geleisteten Beiträge. Es beschreibt verwandte Arbeiten auf diesem Gebiet und skizziert Richtungen für zukünftige Arbeiten.

Kapitel 2

Sprachkonzepte in ObjectTeams/Java und deren Einfluss auf Refactoring

Die Sprache ObjectTeams/Java enthält neben den bekannten Konzepten aus Java auch eine Reihe von neuen aspektorientierten Konzepten und Sprachelementen.
In diesem Kapitel werden zunächst die aspektorientierten Konzepte in dem Programmiermodell Object Teams betrachtet. Anschließend werden diese Konzepte und deren Mechanismen, die in ObjectTeams/Java umgesetzt werden, beschrieben.
Am Ende dieses Kapitels werden die Auswirkungen dieser neuen Sprachelemente auf objektorientierte Refactorings dargestellt.

2.1 Konzepte in Object Teams

Das Programmiermodell Object Teams[8] führt eine Reihe von neuen Konzepten ein, mit denen ein bestehendes Programm nachträglich erweitert werden kann.
Das Schlüsselkonzept in Object Teams ist eine Art Modul, welches Eigenschaften von Klassen und Paketen in sich vereint. Eine **Teamklasse** (oder einfach *Team*) dient als Container für *Rollenklassen* (siehe unten). Teams können durch Benutzung von Vererbung verfeinert werden. Die Vererbung zwischen Teams führt außerdem eine neue Art von Vererbung zwischen den enthaltenen Rollenklassen ein, und zwar die **implizite Vererbung**. Abgesehen von einigen Unterschieden sind Teamklassen reguläre Java Klassen, die Methoden und Felder enhalten und deren Instanzen reguläre Java Objekte sind.

Jede direkte innere Klasse eines Teams ist eine **Rollenklasse** (oder einfach *Rolle*). Im Allgemeinen ist eine Rollen-Instanz durch ihr umschließendes Team begrenzt. Die Ausnahme bilden *externe Rollen*, die auch außerhalb des Teams sichtbar sind.
Die Komposition von einem Team (Aspekt-Code) mit einem bereits existierenden Paket der Applikation (Basis-Code), geschieht in Object Teams durch die Deklarierung einer Rolle-Basis Beziehung mit dem Schlüsselwort `playedBy`. Diese Relation deklariert, dass jede Rolleninstanz mit einer Basis-Instanz assoziiert ist. Die Klasse, an die eine Rolle gebunden ist, nennt man ihre *Basisklasse*. Eine Rollenklasse, die eine `playedBy`-Relation deklariert, nennt man eine *gebundene Rolle*.
Rollen-Instanzen können Features aus ihre Basis-Instanz erben und überschreiben. Object Teams stellt dafür zwei Arten von **Methoden-Bindungen** zur Verfügung, und zwar Callout-Bindungen und Callin-Bindungen.
Eine **Callout-Bindung** bindet eine abstrakte Rollenmethode („erwartete Methode“) an eine konkrete Basismethode („bereitgestellte Methode“). Der Effekt ist, dass ein Aufruf der Rollenmethode an das assoziierte Basisobjekt weitergeleitet wird und dort die angegebene Basismethode aufgerufen wird. Eine Rollenmethode kann auch an ein Feld der Basisklasse der Rolle gebunden werden (*Callout To Field*). Hierbei wird eine Implementierung für diese Rollenmethode generiert, durch die sie als Getter und Setter für das gegebene Feld des assoziierten Basisobjekts fungiert.
Eine **Callin-Bindung** komponiert eine existierende Rollenmethode mit einer gegebenen Basismethode. Dabei werden Aufrufe von einer oder mehrerer Basismethoden durch die Rollenmethode „abgefangen“. Die Rollenmethode kann dabei entweder vor oder nach der Basismethode aufgerufen werden, oder diese komplett ersetzen.
In sogenannten **Parameter Mappings** kann darüber hinaus für eine Callout- oder Callin-Bindung (bei vollständig angegebenen Signaturen der referenzierten Methoden) deklariert werden, welche Parameterwerte jeweils an die Basismethode bzw. Rollenmethode übergeben werden sollen und welcher Rückgabewert (falls vorhanden) verwendet werden soll (*Result Mapping*).

2.2 Sprachkonzepte und Mechanismen in ObjectTeams/Java

Die aspektorientierte Sprache ObjectTeams/Java ist eine vollständige Umsetzung des Programmiermodells Object Teams auf Grundlage der objektorientierten Sprache Java. In den folgenden Abschnitten soll gezeigt werden, wie die Object Teams Konzepte in ObjectTeams/Java realisiert sind und welche Mechanismen es gibt. Außerdem wird erläutert, welche Java-Konzepte durch die neuen Sprachelemente verändert oder erweitert werden.

2.2.1 Teamklassen

In ObjectTeams/Java ist eine Klasse, die mit dem Modifikator `team`[1] deklariert ist, eine **Teamklasse**. Teamklassen sind, bis auf zwei Unterschiede, reguläre Java Klassen (siehe 2.1). Die Unterschiede zu Java-Klassen sind die *Team-Aktivierung* und das *Declared Lifting* in Team-Methoden.
Die *Team-Aktivierung* ([2], §5.) ist Voraussetzung für die Ausführung von Callin-Bindungen (siehe 2.2.9.4). Callin-Bindungen haben nur einen Effekt, wenn die korrespondierende Team-Instanz aktiviert ist.
Beim *Declared Lifting* ([2], §2.3.2) kann eine nicht-statische Team-Methode oder ein Konstruktor einen Parameter mit zwei Typen deklarieren, um explizit einen Ort für *Lifting* ([2], §2.3) zu benennen. Der zweite Typ in der Deklaration (Rollenklasse) ist dabei eine Rolle des ersten Typs (Basisklasse). Diese Basisklasse kann eine reguläre Klasse oder eine Teamklasse sein. Darüber hinaus muss der deklarierte Rollentyp eine Rolle der umschließenden Teamklasse sein, die die Lifting-Methode definiert.
Die spezielle Vererbungsbeziehung zwischen Rollen, die durch die Vererbung zwischen Teams eingeführt wird, wird in Abschnitt 2.2.5 näher beschrieben.

2.2.2 Rollenklassen

Eine **Rollenklasse** ist eine innere Klasse eines Teams (siehe 2.1). Wie innere Klassen in Java hat auch jede Instanz einer Rollenklasse eine implizite (unveränderbare) Referenz auf ihre umschließende Team-Instanz (Zugriff auf diese Referenz mit `Team.this`). Im Gegensatz zu einer inneren Klasse in Java kann eine Rollenklasse in ObjectTeams/Java nur einen der Modifikatoren `public` oder `protected` haben. Auf eine Rolle, die als `protected` deklariert ist, kann nur innerhalb des umschließenden Teams oder in einem der Sub-Teams zugegriffen werden. Rollen, die `public` deklariert sind, sind auch außerhalb ihrer Team-Instanz sichtbar.
Die Methoden einer Rollenklasse umfassen direkte (in der Rollenklasse deklarierte) Methoden sowie Methoden, die durch Vererbung erworben werden. Neben der expliziten Vererbung (siehe 2.2.6) kann eine Rollenklasse Methoden auch über *implizite Vererbung* (siehe 2.2.5) erwerben.
Die Regeln in Java, die die Sichtbarkeit bzw. den Zugriff auf Klassen und deren Member kontrollieren ([12], 6.6), sind in ObjectTeams/Java teilweise verändert worden. So ist ein privates Feld oder eine private Methode einer Rolle auch in jeder impliziten Sub-Rolle sichtbar.
Im Gegensatz zu inneren Klassen in Java sind private Member einer Rolle im umschließenden Team nicht sichtbar. Die Member einer Rolle, die keinen Modifikator besitzen, sind in der deklarierenden Rollenklasse und in ihren Sub-Klassen (explizit und implizit) sichtbar. Rollen-Member, die `protected`

[1] Der Modifikator `team` wird noch in einem anderen Kontext verwendet (siehe 2.2.2.2)

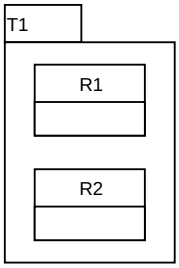

Abbildung 2.1: Beispiel für eine Teamklasse und Rollenklassen.

sind, sind nur im umschließenden Team sichtbar. Die `public` Member in Rollen hingegen gewähren unbeschränkten Zugriff.
Abbildung 2.1 zeigt ein Beispiel für eine Teamklasse `T1`, die zwei Rollen `R1` und `R2` enthält.

2.2.2.1 Externe Rollen

Normalerweise kapselt ein Team seine Rollen gegen ungewollten Zugriff von außen. Sind Rollen auch außerhalb der umschließenden Team-Instanz sichtbar (also als `public` deklariert), handelt es sich um **externe Rollen** (*Externalized Roles*[2], §1.2.2). Der Rollentyp wird dabei relativ zu einer existierenden (unveränderbaren) Team-Instanz angegeben (auch *Anchored Type* genannt).

2.2.2.2 Rollendateien

Den regulären inneren Klassen entsprechend, können Rollenklassen direkt im Quellcode des umschließenden Teams angegeben werden. Als Alternative können Rollenklassen auch in separaten **Rollendateien** (*Role Files*[2], §1.2.5) abgelegt werden. Die Rollenklassen werden dabei in einem Verzeichnis gespeichert, das den gleichen Namen hat wie das Team (ohne `.java`). Die `package`-Deklaration einer Rollenklasse in einem Role File beinhaltet als erste Angabe den Modifikator `team` und dahinter den vollständig qualifizierten Namen der umschließenden Teamklasse. Die Angabe des Teamnamens in der Paket-Deklaration bedeutet, dass die in diesem Compilation Unit ([12], 7.3) deklarierte Rollenklasse eine Rolle des angegebenen Teams ist. Die in einem Compilation Unit (Rollendatei) deklarierte Rollenklasse ist streng gesehen keine *Top-level* Typ-Deklaration ([12], 7.6), wie es in der Java-Sprachbeschreibung[12] in Abschnitt 7.3 gefordert wird.

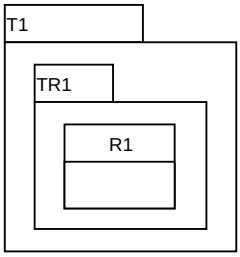

Abbildung 2.2: Beispiel für ein Nested Team.

2.2.3 Typ-Bindung (`playedBy`)

Eine Rollenklasse kann unter Verwendung einer `playedBy`-Relation an eine Basisklasse gebunden werden (siehe 2.1). Diese neue **Typ-Bindung** bringt Einschränkungen bezüglich der referenzierten Basisklasse und der Sichtbarkeit dieser Basisklasse mit sich, die technische Gründe haben (siehe [2], §2.1.2). Die nach dem Schlüsselwort `playedBy` angegebene Basisklasse stellt eine explizite Typ-Referenz dar. Die Basisklasse kann entweder eine reguläre Klasse oder eine Teamklasse sein.

2.2.4 Verschachtelte Team- und Rollenklassen (Nested Teams)

Teams und Rollen in ObjectTeams/Java können auch verschachtelt sein. Enthält eine Rollenklasse den Modifikator `team`, dann kann sie selbst wiederum Rollen enthalten und auch an eine Basisklasse gebunden werden. Ist dies der Fall, dann ist diese Klasse einerseits eine Rolle des umschließenden Teams und andererseits ein Team, welches weitere Rollen enthält. Eine solche gemischte Klasse besitzt alle Eigenschaften von beiden Arten von Klassen (Teams und Rollen) und wird auch als **verschachteltes Team** (*Nested Team*[2], §1.5) bezeichnet. Ein Beispiel für ein verschachteltes Team ist in Abbildung 2.2) gegeben.

2.2.5 Implizite Vererbung von Rollenklassen

Die Vererbung zwischen Teamklassen in ObjectTeams/Java führt eine spezielle Vererbungsbeziehung zwischen ihren enthaltenen Rollen ein, die **implizite Vererbung**. Dabei erwirbt ein Team alle Rollen seines Super-Teams. Der Erwerb von Rollenklassen kann entweder *direkt* erfolgen oder er kann implizite Vererbung beinhalten.
Beim direkten Erwerb ist innerhalb eines Sub-Teams `T` jede Rolle `S.R` seines Super-Teams `S` anhand des Namens `T.R` ohne weitere Deklaration verfügbar.

Implizite Vererbung ist folgendermaßen definiert: Wenn ein Team eine Rollenklasse mit demselben Namen definiert wie eine Rolle, die in seinem Super-Team definiert ist, dann überschreibt die neue Rolle die korrespondierende Rolle aus dem Super-Team und *erbt implizit* alle ihre Features[2] ([2], §1.3.1(c)). Diese Rolle darf die Sichtbarkeit der impliziten Super-Rolle nicht einschränken (siehe Regeln in [12] in Abschnitt 8.4.6.3). Die geerbten Features dieser Rolle, die auch private Member der impliziten Super-Rolle umfassen (siehe 2.2.2), können wiederum unter Beachtung derselben Regeln überschrieben werden.
In Java gibt es keine implizite Vererbung, das heißt, innere Klassen unterschiedlicher umschließender Klassen können nicht voneinander erben. Eine Klasse in Java erbt nur alle nicht-privaten Felder und Methoden ihrer direkten Super-Klasse und Super-Interfaces ([12], 8.3 und 8.4.6). Daher können in Java private Felder oder Methoden auch nicht überschrieben werden.
Im Gegensatz zur regulären Vererbung (`extends`) werden auch Konstruktoren entlang impliziter Vererbung vererbt und können wie normale Methoden überschrieben werden.
Das neue Schlüsselwort `tsuper` wird für Superaufrufe entlang impliziter Vererbung benutzt. Dabei kann `tsuper` nur dazu verwendet werden, eine korrespondierende Version der umschließenden Methode (bzw. des umschließenden Konstruktors) aufzurufen. Mit anderen Worten, der Name der hinter `tsuper` angegebenen Methode der korrespondierenden Rolle muss gleich dem Namen der umschließenden Methode sein, in welcher sich der `tsuper`-Ausdruck befindet.

2.2.6 Explizite (reguläre) Vererbung von Rollenklassen

Neben der impliziten Vererbung können Rollen auch explizit erben (unter Verwendung von `extends`). Hierbei erbt die Rolle ebenfalls alle Features ihrer Super-Klasse. Es bestehen jedoch Restriktionen bezüglich der expliziten Super-Klasse einer Rolle ([2], §1.3.2).

2.2.7 Implizite und explizite Rollenvererbung

2.2.7.1 Vererbung von `extends`

Besitzt eine Rollenklasse eine explizite Super-Klasse (`extends`), wird diese Relation entlang impliziter Vererbung vererbt. Die vererbte `extends`-Relation kann von einer implizit erbenden Rolle verändert werden, mit der Einschränkung, dass die neue Super-Klasse eine Sub-Klasse der Klasse in der überschriebenen `extends`-Relation ist.

[2] Dies wird technisch durch *Copy Inheritance*[30] realisiert.

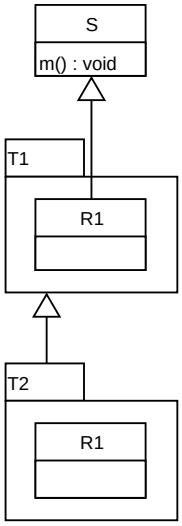

Abbildung 2.3: Beispiel für implizite und explizite Vererbung.

2.2.7.2 Vererbung von `playedBy`

Die `playedBy`-Relation (siehe 2.1) wird entlang expliziter und impliziter Rollenvererbung vererbt. Eine *explizite* Sub-Rolle (Sub-Klasse, die `extends` verwendet), kann die `playedBy`-Relation durch Angabe einer spezielleren Basisklasse verfeinern. Das heißt, die neue Basisklasse muss eine Sub-Klasse der Klasse sein, die in der `playedBy`-Relation der expliziten Super-Klasse deklariert ist. Eine *implizite* Sub-Rolle hingegen, darf die `playedBy`-Relation nicht verändern.

2.2.7.3 Vorrangigkeit der impliziten Vererbung

Erbt eine Rollenklasse irgendwelche Features (z.B. Methoden) aus Super-Klassen sowohl implizit als auch explizit, dann werden die explizit geerbten Features durch die implizit geerbten „überschrieben“, denn implizite Vererbung hat immer Vorrang vor expliziter Vererbung (implizit bindet stärker als explizit).
In Abbildung 2.3 erbt die Rolle `R1` in Team `T1` die Methode `m` aus der expliziten Super-Klasse `S`. Die Rolle `R1` in Team `T2` erbt die Methode `m` einerseits implizit von der Rolle `R1` aus dem Super-Team `T1`, und andererseits explizit von der Klasse `S` (siehe 2.2.7.1). Da implizite Vererbung stärker bindet, erbt `R1` in `T2` die Version von `m` aus der impliziten Super-Rolle `T1.R1`.

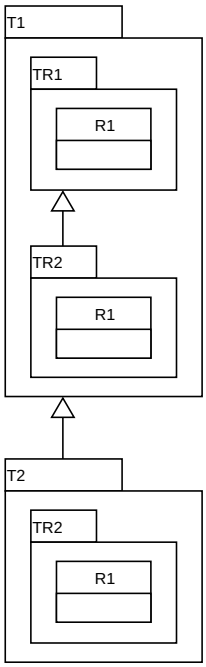

Abbildung 2.4: Beispiel für Vererbung zwischen Nested Teams.

2.2.8 Vererbung zwischen verschachtelten Team- und Rollenklassen

Verschachtelte Teams (siehe 2.2.4) können ebenfalls sowohl implizit als auch explizit voneinander erben. Im Beispiel in Abbildung 2.4 erbt das Nested Team `TR2` in `T1` explizit von dem Nested Team `TR1` aus demselben Team. Dabei überschreibt die Rolle `R1` in `TR2` die implizit geerbte Super-Rolle `R1` aus `TR1`. Weiterhin erbt das Nested Team `TR2` in `T2` implizit von dem gleichnamigen Nested Team aus `T1`. Die Rolle `R1` in `T2.TR2` überschreibt dabei wiederum die implizite Super-Rolle `R1` aus `T1.TR2`.

2.2.9 Methoden-Bindungen

Rollenklassen können **Methoden-Bindungen** deklarieren, um Features aus der Basisklasse zu erben und zu überschreiben. Es gibt zwei Arten von Methoden-Bindungen in ObjectTeams/Java: Callin-Bindungen und Callout-Bindungen.

2.2.9.1 Callout-Bindung

Eine **Callout-Bindung** (siehe 2.1) stellt eine *Aspekt-Basis Referenz* dar. Auf der linken Seite einer Callout-Bindung ist entweder der (einmalige) Na-

me oder die vollständige Signatur (inklusive Parametern und Rückgabewert) einer abstrakten Rollenmethode angegeben[3]. Die Rollenmethode, die durch eine Callout-Bindung gebunden wird, kann in der selben Klasse wie die Bindung deklariert sein oder sie kann von einer Super-Klasse oder einem Super-Interface geerbt werden. Die Super-Klasse kann entweder eine Rollenklasse eines umschließenden Teams oder eine reguläre Klasse sein.
Auf der rechten Seite einer Callout-Bindung ist der Name bzw. die vollständige Signatur einer Basismethode angegeben[4].
Eine Callout-Bindung kann auch Methoden einer Basisklasse referenzieren, die normalerweise nicht sichtbar sind[5]. Hier setzt sich ObjectTeams/Java über die Regeln in [12] in Abschnitt 6.6 hinweg. Callout-Bindungen und Callin-Bindungen (siehe 2.2.9.4) sind die jedoch die einzigen Stellen in einem ObjectTeams/Java Programm, die Methoden referenzieren können, auf die sonst nicht zugegriffen werden kann.
Durch Verwendung einer **Callout To Field-Bindung** (siehe 2.1) kann auch auf normalerweise nicht sichtbare Felder einer Basisklasse zugegriffen werden. Dabei ist auf der rechten Seite statt einer Methode ein Feld der gebundenen Basisklasse referenziert[6], sowie ein zusätzlicher *Callout-Modifikator* (`get` oder `set`) angegeben.

2.2.9.2 Vererbung und Überschreiben von Callout-Bindungen

Callout-Bindungen werden entlang expliziter und impliziter Vererbung vererbt. Geerbte Callout-Bindungen (auch Callout To Field-Bindungen) können überschrieben werden (unter Verwendung von „=>“). Dies wird als *Callout Override* bezeichnet.

2.2.9.3 Callout Parameter Mappings

Falls in einer Callout-Bindung auf der linken und rechten Seite vollständige Signaturen anstelle von Methodennamen deklariert sind, können die Parameter und Rückgabewerte anhand des `with{}`-Konstrukts gemappt werden. In einem **Callout Parameter Mapping** (siehe 2.1) steht auf der linken Seite des Pfeils (->) eine Expression und auf der rechten Seite der Name eines Parameters der Basismethode. Die Expression kann einen Parameter der Rollenmethode oder ein innerhalb der Rolleninstanz sichtbares Feature enthalten.

[3]Dies ist eine Referenz, die technisch durch einen Methodenbezeichner (*MethodSpec*) realisiert ist.

[4]Dies ist eine Referenz, die technisch durch einen Methodenbezeichner (*MethodSpec*) realisiert ist.

[5]Dieses Konzept wird als *Decapsulation* bezeichnet.

[6]Diese Referenz ist technisch durch einen Feldbezeichner (*FieldAccessSpec*) realisiert.

In einer Callout-Bindung kann anhand eines *Result Mapping* der Rückgabewert einer Basismethode auf den Rückgabewert der Rollenmethode (unter Verwendung des speziellen Bezeichners `result`) gemappt werden.
Die Werte in einer Callout To Field-Bindung können ebenfalls gemappt werden (bei komplett angegebenen Signaturen). In einem solchen *Value Mapping* steht auf der linken Seite des Pfeils eine Expression und auf der rechten Seite der Name des referenzierten Feldes.

2.2.9.4 Callin-Bindung

Eine **Callin-Bindung** (siehe 2.1) stellt ebenfalls eine *Aspekt-Basis Referenz* dar.
Auf der linken Seite einer Callin-Bindung ist entweder der (einmalige) Name oder die vollständige Signatur (inklusive Parametern und Rückgabewert) einer existierenden Rollenmethode angegeben[7]. Die Rollenmethode, die durch eine Callin-Bindung gebunden wird, kann in der selben Klasse wie die Bindung deklariert sein oder von einer Super-Klasse oder einem Super-Interface geerbt werden. Die Super-Klasse ist entweder eine Rollenklasse eines umschließenden Teams oder eine reguläre Klasse.
Auf der rechten Seite einer Callin-Bindung ist der Name bzw. die vollständige Signatur einer bzw. mehrerer Basismethode(n) angegeben[8].
Eine Callin-Bindung enthält auf der rechten Seite zusätzlich noch einen *Callin-Modifikator* (`before, after` oder `replace`), der angibt, wie die Rollenmethode und die Basismethode komponiert werden sollen. Ist in einer Callin-Bindung der Modifikator `replace` angegeben, dann handelt es sich bei der Rollenmethode um eine *Callin-Methode* (Methode mit dem neuen Modifikator `callin`). Callin-Methoden enthalten einen Basis-Aufruf (*base call*[2], §4.3), der den speziellen Namen `base` verwendet, um die originale Basismethode aufzurufen.
Eine Callin-Bindung kann wie eine Callout-Bindung auch versteckte Methoden einer Basisklasse referenzieren.

2.2.9.5 Callin Parameter Mappings

Sind in einer Callin-Bindung auf der linken und rechten Seite vollständige Signaturen anstelle von Methodennamen deklariert, können die Parameter und Rückgabewerte anhand des `with{}`-Konstrukts gemappt werden.
In einem **Callin Parameter Mapping** (siehe 2.1) steht auf der linken Seite des Pfeils (<-) der Name eines Parameters der Rollenmethode, und auf der

[7] Dies ist eine Referenz, die technisch durch einen Methodenbezeichner (*MethodSpec*) realisiert ist.

[8] Dies ist eine Referenz, die technisch durch einen Methodenbezeichner (*MethodSpec*) realisiert ist.

rechten Seite eine Expression.
In einer Callin Replace-Bindung (Callin-Bindung mit dem Modifikator `replace`) kann die rechte Seite eines Parameter Mappings entweder aus dem Namen eines Parameters der Basismethode bestehen oder aus einer Expression, die keinen Parameter der Basismethode enthält.
In einer Callin-Bindung, welche den Modifikator `after` verwendet, kann die rechte Seite eines Parameter Mappings den Bezeichner `result` verwenden, um auf das Ergebnis der Basismethode zu verweisen.
Callin-Bindungen, die den Modifier `before` verwenden, haben kein Result Mapping. Im Parameter Mapping einer Callin Replace-Bindung, muss `result` auf sich selbst gemappt sein, falls Basis- und Rollenmethode ein Ergebnis zurückliefern. Deklariert nur die Basismethode ein Ergebnis, dann besteht die Möglichkeit eine beliebige Expression auf `result` zu mappen.

2.2.10 Explizite Team-Aktivierung

Mit dem Block-Konstrukt `within`(*Expression*){} kann in ObjectTeams/Java ein Team explizit aktiviert werden. Die *Expression* muss dabei eine Team-Instanz bezeichnen. Dieser Block kann nur in einer Teamklasse deklariert werden. Solange die Anweisungen in diesem Block ausgeführt werden ist diese Team-Instanz aktiviert.

2.2.11 Guards

Neben der expliziten und impliziten Team-Aktivierung (siehe [2], §5.2 und §5.3) gibt es ein weiteres Konzept in ObjectTeams/Java, mit dem die Wirkung von Callin-Bindungen kontrolliert werden kann. Dies sind die sogenannten **Guards**. Es gibt reguläre Guards ([2], §5.4.1) und Basis-Guards ([2], §5.4.2).
Guards können an vier Stellen in einem ObjectTeams/Java Programm auftreten: im Header einer Teamklasse, Rollenklasse oder Methode und hinter einer Methoden-Bindung.
Reguläre Guards enthalten das Schlüsselwort `when` und einen Prädikatenausdruck vom Typ `boolean`. Basis-Guards enthalten zusätzlich noch den Modifikator `base` (vor dem Schlüsselwort `when`).
Abhängig von der Art des Guards können unterschiedliche Objekte bzw. Werte, zum Beispiel die Team- oder Rollen-Instanz oder Features des Teams oder der Rolle, im Prädikaten-Ausdruck referenziert werden.
Wird während der Programmausführung der Ausdruck zu `true` ausgewertet, werden die Callin-Bindungen aktiviert, wird er zu `false` ausgewertet, werden sie deaktiviert.

2.3 Auswirkungen von ObjectTeams/Java Code auf objektorientiertes Refactoring

Die beschriebenen neuen Sprachkonzepte bzw. Sprachkonstrukte in ObjectTeams/Java haben Auswirkungen auf die meisten der existierenden Refactorings[7]. Diese Refactorings analysieren und transformieren nur objektorientierten Code (OO-Code). Eine ObjectTeams/Java Anwendung besteht jedoch aus objektorientiertem Code (reguläre Java-Klassen, Interfaces, etc.) *und* aspektorientiertem Code (Teamklassen, Rollenklassen, Methoden-Bindungen, usw.).
Innerhalb des aspektorientierten Codes (AO-Code) werden Elemente aus dem OO-Code (Basis-Code) explizit anhand von Namen referenziert (siehe 2.2.3, 2.2.9.1, und 2.2.9.4).
Wendet man zum Beispiel ein Refactoring an, welches ein Element (Paket, Typ, Methode oder Feld) im OO-Code eines ObjectTeams/Java Programms umbenennt, würde es die Referenzen auf dieses Element im AO-Code nicht aktualisieren, weil das Refactoring nur auf OO-Code operiert. Dem objektorientierten Code sind die aspektorientierten Code-Teile des ObjectTeams/Java Programms nicht bekannt.
In Abbildung 2.5 ist beispielhaft ein Ausschnitt eines ObjectTeams/Java Programms angegeben. Das Beispiel besteht aus zwei regulären Klassen, `S1` und `B1`, und einer Teamklasse (`T1`).
Würde man beispielsweise die Methode `sm1` in der Klasse `S1` umbenennen (*Rename Method*[7], S.273)), dann würde die Referenz auf `sm1` in der Callin-Bindung in Rolle `R1` nicht aktualisiert werden. Entsprechend würden auch die Referenzen auf die Basis-Methode `bm1` in der Callout- und der Callin-Bindung in Rolle `R1` nicht umbenannt werden, wenn der Name von `bm1` in der Basisklasse `B1` verändert werden würde.
Wird die Basisklasse `B1` umbenannt (*Rename Type*[6]), hätte dies zur Folge, dass die Referenzen auf `B1` in der Signatur der Team-Methode (Lifting-Methode) und in der Typ-Bindung (`playedBy`) der Rollenklasse vom Refactoring nicht berücksichtigt würden.
Die Folge dieser durchgeführten Refactorings wäre, dass die im AO-Code referenzierten und nicht aktualisierten Namen im OO-Code nicht mehr existieren würden und das ObjectTeams/Java Programm nicht mehr kompilieren würde.
Die Rollenklassen einer ObjectTeams/Java Applikation können bestehende Vererbungshierarchien verändern, indem sie explizit von einer existierenden regulären Klasse erben (siehe 2.2.6) und darüber hinaus eventuell noch implizite Super- bzw. Sub-Klassen besitzen (siehe 2.2.5). Diese (nachträglich) veränderten bzw. neuen Vererbungsbeziehungen können beim Einsatz von bestehenden Refactorings zu diversen Problemen führen. Einer dieser Problembereiche ist *Overriding* ([12], 8.4.6.1).

```
public class S1 implements I {
    public void im(){}
    public void sm1(){}
}

public class B1 {
    private int bf;

    public void bm1(){}
}

public team class T1 {
    private int threshold;

    //declared lifting
    public void tm1(B1 as R1 r0) {
        activate(); //Methode aus org.objectteams.Team
    }

    public class R1 extends S1 playedBy B1 {
        public void im(){}
        public abstract int rm1();
        //callout binding
        rm1 -> bm1;
        //callin binding
        sm1 <- after bm1;
    }
}
```

Abbildung 2.5: Ein ObjectTeams/Java Beispiel

Wird ein Refactoring angewendet, welches die Signatur einer Methode in einer regulären Klasse verändert (z.B. *Rename Method*[7], S.273, *Add Parameter*[7], S.275, *Remove Parameter*[7], S.277), würde es die Signaturen von überschreibenden Methoden in expliziten Sub-Klassen (in diesem Fall Rollenklassen) dieser Klasse und die Signaturen von überschriebenen bzw. überschreibenden Methoden in impliziten Super- bzw. Sub-Rollen dieser Sub-Klassen nicht verändern.
Der Grund ist wiederum der gleiche wie bei den expliziten Namensreferenzen: Das Refactoring analysiert und verändert ausschließlich den objektorientierten Code der ObjectTeams/Java Anwendung. Der OO-Code seinerseits hat keine Kenntnisse von den aspektorientierten Teilen des Codes. Daher sind ihm auch die veränderten bzw. neuen (expliziten und impliziten) Vererbungsbeziehungen nicht bewusst.
Abbildung 2.6 zeigt einen weiteren Ausschnitt aus einem ObjectTeams/Java Programm. Das Beispiel umfasst ein Interface (`I`), eine reguläre Klasse (`S1`)

```
public interface I {
    public void im();
}

public class S1 implements I {
    public void im(){}
    public void sm1(){}
}

public team class T1 {
    private int threshold;

    //declared lifting
    public void tm1(B1 as R1 r0) {
        activate(); //Methode aus org.objectteams.Team
    }

    public class R1 extends S1 playedBy B1 {
        public void im(){}

public team class T2 extends T1 {
    public class R1 playedBy B1 {
        public void im(){}
```

Abbildung 2.6: Ein weiteres ObjectTeams/Java Beispiel

sowie zwei Teamklassen (`T1` und `T2`).
Wird die Methode `im` in Klasse `S1` umbenannt, dann würde nur die Methode `im` in Interface `I` aktualisiert werden. Die überschreibende Methode in der expliziten Sub-Klasse `T1.R1` und die wiederum überschreibende Methode in der impliziten Sub-Rolle `T2.R1` würden jedoch nicht umbenannt werden. Diese überschreibenden Methoden im AO-Code würden auch nicht aktualisiert werden, wenn beispielsweise der Methode `im` in `S1` ein Parameter hinzugefügt oder entfernt worden wäre.
Jede der genannten Änderungen hätte unter Umständen zur Folge, dass das Verhalten des ObjectTeams/Java Programms verändert worden wäre.
Ein weiteres Java-Sprachfeature, dass bei der Anwendung von einigen Refactorings zu Problemen in den Methoden-Bindungen (siehe 2.2.9) führen kann, ist *Overloading* ([12], 8.4.7).
Würde ein Refactoring (z.B. *Move Method*[7], S.142, *Extract Method*[7], S.110 oder *Rename Method*) in der Klasse `S1` eine Methode erzeugen, die die bestehende Methode `sm1` überlädt, dann würde dies in einem Fehler in der Callin-Bindung in Rolle `T1.R1` resultieren, da der auf der linken Seite referenzierte Methodenname jetzt mehrdeutig (*ambiguous*) wäre. Der gleiche Fehler könnte auf der rechten Seite der Callin-Bindung auftreten, wenn nach einem Refactoring die referenzierte Methode `bm1` aus der gebundenen

Basisklasse überladen wäre. Das beschriebene Problem könnte auch in der Callout-Bindung der Rolle vorkommen.

Alle diese Beispiele zeigen, dass es nicht ausreicht, bestehende Refactorings auf ObjectTeams/Java Programme anzuwenden, da diese nur objektorientierten Code analysieren und transformieren. Der Aspekt-Code eines ObjectTeams/Java Programms wird nicht berücksichtigt. Die neuen AO-Sprachelemente und Mechanismen in ObjectTeams/Java referenzieren jedoch Elemente aus dem objektorientierten Code-Teil einer ObjectTeams/Java Anwendung (Basis-Code) und verändern bzw. erweitern bestehende Vererbungsbeziehungen.
Werden existierende Refactorings, so wie sie bisher definiert sind, eingesetzt, dann kann das Verhalten einer ObjectTeams/Java Applikation verändert werden.
Um diese Refactorings korrekt anwenden zu können ist es daher notwendig jedes Refactoring so zu erweitern, dass es auch den aspektorientierten Code einer ObjectTeams/Java Anwendung analysiert und ihn, wenn notwendig, implizit transformiert. Diese erweiterten Refactorings müssen das Verhalten eines ObjectTeams/Java Programms erhalten.
Dazu ist zunächst zu klären, was Verhaltenserhaltung bedeutet und wie diese für eine ObjectTeams/Java Anwendung sichergestellt werden kann.
Dies ist Thema des nächsten Kapitels. In Kapitel 4 werden dann bestehende Refactorings für ObjectTeams/Java konzeptuell erweitert.

Kapitel 3

Erhaltung des Verhaltens während des Refactorings

Die wichtigste Bedingung beim Refactoring ist, dass das Verhalten eines Programms bzw. einer Software erhalten werden muss.
In diesem Kapitel werden Regeln beschrieben, die sicherstellen, dass Refactorings das Verhalten einer Applikation beibehalten. Diese Regeln sind in drei Kategorien eingeteilt: Sprachanforderungen, Vererbungsbeziehungen und semantische Äquivalenz.
Zunächst werden die von Opdyke in seiner Dissertation[22] und die in Ruras Diplomarbeit[25] definierten Regeln für Verhaltenserhaltung in objektorientierten Programmen (C++ und Java) betrachtet. Anschließend werden diese Regeln überarbeitet und neue Regeln für die Sprache ObjectTeams/Java definiert.

3.1 Erhaltung des Programmverhaltens

Per Definition sollte ein Refactoring nicht das „beobachtbare, von außen sichtbare, Verhalten eines Programms oder einer Software verändern“ (siehe 1.2).
Die Erhaltung des Verhaltens zu garantieren bzw. zu beweisen ist eine schwierige Aufgabe bei der Durchführung von Refactoring. Die konkrete Definition von Verhalten hängt von unterschiedlichen Aspekten ab, zum Beispiel dem jeweiligen Anwendungsbereich der betrachteten Software oder bestimmten benutzerspezifischen Aspekten.
In ihrer Übersicht zu Software Refactoring[19] werden von Mens und Tourwé einige wichtige Anwendungsbereiche von Software genannt, die unterschiedliche Anforderungen an das Verhalten stellen: Für *Echtzeit-Software* ist ein wichtiger Aspekt des Verhaltens die Ausführungszeit von bestimmten Operationen. Mit anderen Worten, Refactorings sollten alle Arten von tempo-

ralen Constraints erhalten.
Für *eingebettete Software* sind Constraints bezüglich Speicher und Leistungsverbrauch auch wichtige Aspekte des Verhaltens, die von einem Refactoring erhalten werden sollten.
Für *sicherheitskritische Software* gibt es bestimmte Sicherheitsaspekte, die von einem Refactoring beibehalten werden sollten.
Ein grundlegender Ansatz, um zu garantieren, dass das Verhalten erhalten wird, ist, formal zu beweisen, dass Refactorings die vollständige Programmsemantik erhalten. Für komplexe Sprachen wie C++, für die eine formale Semantik schwer zu definieren ist, müssen bestimmte Einschränkungen für die erlaubten Sprachkonstrukte oder Refactorings gemacht werden.
In seiner Dissertation[22] verfolgt Opdyke teilweise diesen Ansatz, indem er sich bei seiner Definition von Verhaltenserhaltung auf bestimmte Programmeigenschaften, die von einem Refactoring nicht verletzt werden dürfen, beschränkt und semi-formal nachweist, dass die von ihm definierten Refactorings diese Eigenschaften erfüllen. Einige Sprachfeatures von C++ wie Mehrfachvererbung, Overloading, Typ-Casts und Typ-Parameter (*Generics*) werden von Opdyke in seiner Analyse, aufgrund der zuvor genannten Schwierigkeit, nicht berücksichtigt.

3.1.1 Programmeigenschaften und Regeln für Verhaltenserhaltung nach Opdyke

Die von Opdyke in seiner Arbeit vorgeschlagene Definition von Verhaltenserhaltung ist, dass „Refactorings immer legale, syntaktisch korrekte, Programme produzieren, die die gleichen Operationen durchführen wie vor dem Refactoring“.
Um dies zu gewährleisten, wurden von Opdyke sieben Programmeigenschaften bzw. Programminvarianten für die objektorientierte Sprache C++ beschrieben, die, wie sich in der Praxis herausgestellt hat, durch Refactoring am häufigsten verletzt werden. Sie beziehen sich auf Vererbung, Scoping, Typ-Kompatibilität und semantische Äquivalenz.
Diese Eigenschaften sind:

1. *Einmalige Super-Klasse.* Nach dem Refactoring darf eine Klasse höchstens eine direkte Super-Klasse haben, und ihre Super-Klasse darf auch nicht eine ihrer Sub-Klassen sein.

2. *Verschiedene Klassennamen.* Nach dem Refactoring muss jede Klasse einen einmaligen Namen haben.

3. *Verschiedene Membernamen.* Nach dem Refactoring haben alle Membervariablen (Felder) und Funktionen *innerhalb einer Klasse* verschiedene Namen. Es ist jedoch erlaubt eine Funktion aus einer Super-Klasse in einer Sub-Klasse zu überschreiben.

4. *Geerbte Membervariablen nicht redefiniert.* Eine Membervariable, die von einer Super-Klasse geerbt wurde, wird in keiner ihrer Sub-Klassen redefiniert.

5. *Kompatible Signaturen in Redefinition von Memberfunktion.* Nach dem Refactoring, wenn eine in einer Super-Klasse definierte Memberfunktion in einer Sub-Klasse redefiniert wird, müssen alle Attribute (außer dem Funktionsrumpf) der beiden Funktionen kompatibel sein.

6. *Typsichere Zuweisungen.* Nach einem Refactoring, muss der Typ jeder Expression, die einer Variablen zugewiesen ist, eine Instanz des definierten Typs der Variable sein oder eine Instanz eines seiner Sub-Typen.

7. *Semantisch gleichwertige Referenzen und Operationen.*

Ohne Regeln, die überprüfen, dass diese Programmeigenschaften nicht verletzt werden, besteht die Gefahr ein Programm zu erzeugen, das syntaktisch nicht korrekt ist oder, noch schlimmer, korrekt kompiliert, sich aber nach dem Refactoring anders verhält. Durch Rekompilieren des Programms nach dem Refactoring könnten Verletzungen der ersten sechs Eigenschaften entdeckt werden, aber Verletzungen der siebten Eigenschaft könnten eventuell nicht gefunden werden.
In Abbildung 3.1 ist die Umbenennung der Methode `m2` in `m1` in der Klasse `B` dargestellt. Angenommen `A.m1` und die ursprüngliche Methode `B.m2` verhalten sich unterschiedlich, dann erhält dieses „Refactoring“ nicht das Verhalten, weil die Methode `k` nun die umbenannte Methode, die in ihrer lokalen Klasse definiert ist, aufrufen würde, anstatt der vorher geerbten Methode aus der Super-Klasse. Dieser Fehler würde von einem Compiler unentdeckt bleiben.
Damit Refactorings verhaltenserhaltend sind, müssen neben der Erzeugung von legalen Programmen und der Beibehaltung von Vererbungsbeziehungen, die Versionen des Programms vor und nach dem Refactoring auch *semantisch äquivalente Referenzen und Operationen* produzieren (siehe 7.). Semantische Äquivalenz wird von Opdyke folgendermaßen definiert:

> „Die externe Schnittstelle zu dem Programm sei über die Funktion *main*. Wenn die Funktion *main* zweimal (einmal vor und einmal nach einem Refactoring) mit derselben Menge von Eingabewerten aufgerufen wird, dann muss die resultierende Menge von Ausgabewerten die gleiche sein.“

Diese Definition von semantischer Äquivalenz erlaubt Änderungen innerhalb des gesamten Programms, solange dieses Mapping von Eingabe- nach Ausgabewerten das gleiche bleibt.

Fehlerhaftes Refactoring

Vorher:

```
public class A
{
  public void m1(String str)
  {...}
}
public class B extends A
{
  public void m2(String str)
  {...}
  public void k(String str)
  { m1(str) ...}
}
```

Nachher:

```
public class A
{
  public void m1(String str)
  {...}
}
public class B extends A
{
  public void m1(String str)
  {...}
  public void k(String str)
  { m1(str) ...}
}
```

Abbildung 3.1: Umbenennung der Methode m2 nach m1 in Klasse B

Ein Refactoring kann unterschiedliche Teile eines Programms beeinflussen. Das von außen beobachtbare Verhalten ändert sich dabei nicht. Zum Beispiel, wenn eine Variable an verschiedenen Stellen im Programm referenziert wird, würde ein Refactoring, dass den Namen der Variable ändert, praktisch das gesamte Programm beeinflussen. Andererseits würde nur ein Teil eines Methodenrumpfs betroffen sein, wenn ein bestimmter Methodenaufruf darin durch den Block dieser Methode ersetzt werden würde (*Inline Method*[7], S.117). In beiden Fällen ist die Grundidee, dass sich die Ergebnisse von aufgerufenen Operationen und die Referenzen, die von außerhalb der betroffenen Stellen gemacht werden, nicht ändern.
Auf der Grundlage dieser Definition von semantischer Äquivalenz werden von Opdyke folgende wichtige Änderungen genannt, die die Äquivalenz nicht beeinflussen:

OS1. Expressions können vereinfacht werden.

OS2. Toter Code kann entfernt werden.

OS3. Unreferenzierte Variablen, Funktionen und Klassen können hinzugefügt oder entfernt werden.

OS4. Der Typ einer Variable kann verändert werden, solange jede Operation auf der Variablen äquivalent für den neuen Typ definiert wird und alle Zuweisungen, die diese Variable beinhalten, typsicher bleiben.

OS5. Referenzen auf eine Variable oder Funktion, die in einer Klasse definiert sind, können durch Referenzen auf eine äquivalente Variable oder Funktion in einer anderen Klasse ersetzt werden. Eine Folge davon ist, dass lokal definierte Member durch geerbte Member (und umgekehrt)

ersetzt werden können, vorausgesetzt, die Member-Deklarationen sind äquivalent.

Um zu zeigen, dass die von ihm untersuchten Refactorings keine der genannten Programmeigenschaften verletzen, das Programmverhalten folglich nicht verändern, gibt Opdyke zu jedem Refactoring *Vorbedingungen* an. Diese Vorbedingungen müssen erfüllt sein bevor ein Refactoring angewendet werden kann. Die Vorbedingungen werden dabei in Form von speziellen Funktionen, die ein Programm analysieren, überprüft. Mit Hilfe dieser formal spezifizierten Vorbedingungen wird von Opdyke nachgewiesen, dass die Transformationen semantikerhaltend sind, sofern ihre Vorbedingungen erfüllt sind.

In dieser Arbeit sollen einige der von Opdyke definierten als auch weitere objektorientierte Refactorings aus Fowlers Katalog[7] so adaptiert und erweitert werden, dass sie bei der Anwendung auf ObjectTeams/Java Programme das Verhalten dieser Programme nicht verändern. Um die Erhaltung des Verhaltens zu garantieren, sollen die von Opdyke aufgestellten Programmeigenschaften von C++ bzw. die Regeln für Verhaltenserhaltung in C++ für ObjectTeams/Java aktualisiert und entsprechend erweitert werden. Die Einhaltung dieser Regeln soll, wie bei Opdyke, anhand von Vorbedingungen überprüft werden (siehe Kapitel 4).
Da die Sprache ObjectTeams/Java eine Erweiterung von Java ist, ist der erste erforderliche Schritt die Programmeigenschaften von C++ für Java zu überarbeiten und zu erweitern. Dies wurde, wie nachfolgend beschrieben, in der Arbeit von Rura[25] bereits durchgeführt.

3.1.2 Programmeigenschaften und Regeln für Verhaltenserhaltung nach Rura

Die von Opdyke aufgestellten Programmeigenschaften bzw. Regeln wurden von Rura überarbeitet und für die Sprache Java aktualisiert. Das Sprachfeature *Overloading* wird von Rura, im Gegensatz zu Opdyke, bei der Definition der Regeln miteinbezogen.
Um eine bessere Klassifizierung und Abgrenzung zu erreichen, wurden die von ihm definierten Regeln in folgende drei Kategorien eingeteilt:

1. Sprachanforderungen
2. Vererbungsbeziehungen (Sub-Typ-Beziehungen und Overriding)
3. Semantische Äquivalenz

3.1.2.1 Sprachanforderungen von Java

Die von Rura aufgestellten Sprachanforderungen von Java, die auch nach der Durchführung eines Refactorings erfüllt sein müssen, sind folgende:

JL1. Jede Klasse muss eine direkte Super-Klasse haben, die nicht gleichzeitig eine direkte oder indirekte Sub-Klasse ist.

JL2. Jeder Typ (Klasse oder Interface) muss einen einmaligen Namen haben.

JL3. Jede Variable muss einen einmaligen Namen in ihrem Scope haben.

JL4. Jede Methode in einem Typ muss eine einmalige Signatur haben.

JL5. Das Programm muss typsicher sein.

JL6. Geerbte Felder dürfen nicht überschrieben werden (kein *Hiding*, siehe [12], 8.3)

JL7. Regeln für `extends` und `implements`:

(a) Ein Interface kann nur von anderen Interfaces erben. Ein Interface kann nicht ein anderes Interface implementieren.

(b) Eine Klasse kann nur von einer anderen Klasse erben.

(c) Eine Klasse kann nur Interfaces implementieren.

JL8. Wenn eine Methode eine andere überlädt und beide haben die gleiche Signatur (gleiche Namen und gleiche Anzahl von Parametern), dann muss die eine *spezifischer* sein als die andere (siehe [12], 15.12.2.2).

3.1.2.2 Erhaltung von Vererbungsbeziehungen in Java

Die Erhaltung von Vererbungsbeziehungen (Sub-Typ-Beziehungen und Overriding) während eines Refactorings wird von Rura anhand folgender Regeln sichergestellt:

JI1. Eine Klasse, die Interfaces oder abstrakte Klassen implementiert, sollte nach dem Refactoring weiterhin die Anforderungen erfüllen, die ihr von ihren Interfaces und abstrakten Klassen auferlegt werden.

JI2. Wenn eine geerbte Methode (inklusive abstrakte Methoden in Interfaces) überschrieben wird, und diese durch ein Refactoring verändert wird, müssen diese Änderungen an der Methode sowohl in Sub-Typen als auch in Super-Typen berücksichtigt werden. Das bedeutet, bei der Anwendung eines Refactorings, welches die Signatur einer Methode verändert, ist es notwendig entsprechend die Signaturen von überschreibenden Methoden in Sub-Typen (Klassen und Interfaces) und die Signaturen von überschriebenen (geerbten) Methoden in Super-Typen (Klassen und Interfaces) zu ändern.

3.1.2.3 Erhaltung semantischer Äquivalenz in Java

Die von Rura beschriebenen Änderungen, die die Semantik bzw. das Verhalten von Java-Programmen erhalten, entsprechen, bis auf die programmiersprachlich unterschiedlichen Bezeichnungen für dieselben Sprachkonstrukte (z.B. Methode vs. Funktion), denen von Opdyke (siehe Seite 39).

Die genannten Programmeigenschaften von Java bilden die Basis für die im Folgenden beschriebenen Regeln für Verhaltenserhaltung in ObjectTeams/Java Programmen, da sämtliche Sprachfeatures, die in Java vorhanden sind, auch von ObjectTeams/Java unterstützt werden. Dazu werden einige Regeln, dort wo es notwendig ist, ergänzt bzw. überarbeitet. Darüber hinaus kommen weitere Eigenschaften von ObjectTeams/Java Anwendungen hinzu, die sich aus den neuen aspektorientierten Sprachkonzepten ergeben.

3.2 Programmeigenschaften und Regeln für Verhaltenserhaltung in ObjectTeams/Java

Im Folgenden werden die in Abschnitt 3.1.2 beschriebenen Regeln für die Sprache ObjectTeams/Java erweitert und es werden zusätzliche Regeln definiert, um die Erhaltung des Verhaltens während des Refactorings von ObjectTeams/Java Programmen zu gewährleisten. Die Regeln werden, wie bei Rura, in folgende drei Bereiche eingeteilt:

1. Sprachanforderungen von ObjectTeams/Java
2. Vererbungsbeziehungen (Sub-Typ-Beziehungen und Overriding) in ObjectTeams/Java
3. Semantische Äquivalenz in ObjectTeams/Java

3.2.1 Sprachanforderungen von ObjectTeams/Java

Aufgrund der neuen Sprachkonzepte in ObjectTeams/Java, die in Kapitel 2 beschrieben sind, ergeben sich zahlreiche neue Anforderungen von ObjectTeams/Java Programmen. Diese Sprachanforderungen müssen erfüllt sein damit ein ObjectTeams/Java Programm korrekt kompiliert. Um dies zu garantieren müssen die Sprachanforderungen von Java aus Abschnitt 3.1.2.1 überarbeitet und erweitert werden sowie neue Regeln aufgestellt werden.

3.2.1.1 Sprachregeln für ObjectTeams/Java

Die folgenden Regeln stellen sicher, dass ein ObjectTeams/Java Programm syntaktisch korrekt ist und kompiliert. Sie umfassen sämtliche in Kapitel 2 beschriebenen Sprachelemente in ObjectTeams/Java und entsprechen größtenteils den Regeln in der ObjectTeams/Java Sprachdefinition[2]:

OTL1. Regeln für Teamklassen:

(a) Eine Teamklasse kann nur von einer anderen Teamklasse erben ([2], §1.3). Eine Klasse, die von einer Teamklasse erbt, muss den Modifikator `team` haben ([2], §A.1(b)).

OTL2. Regeln für Rollenklassen:

(a) Eine Rollenklasse darf nur `public` oder `protected` sein ([2], §1.2.1(a)).

(b) Eine Rollenklasse muss mindestens die Sichtbarkeit ihrer impliziten Super-Rolle haben ([2], §1.3.1(h)).

(c) Eine Rollenklasse kann explizit (unter Verwendung von `extends` von einer regulären Klasse oder einer Rollenklasse eines umschließenden Teams erben ([2], §1.3.2(a)).

(d) Eine Rollenklasse darf nicht denselben Namen haben wie eine Methode oder ein Feld ihres umschließenden Teams ([2], §1.4(a)).

(e) Eine gebundene Rollenklasse (mit `playedBy`), darf nicht `static` deklariert werden, und muss direkt in einer Teamklasse enthalten sein ([2], §A.1(b)).

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 ([2], §1.5(b)).

(c) Ein verschachteltes Team darf nicht von seinem eigenen, umschließenden Team erben ([2], §1.5(c)).

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

(a) Eine Rollenklasse, auf die außerhalb ihres Teams zugegriffen wird, muss `public` sein ([2], §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 ([2], §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) ([2], §1.2.5(a)).

(b) Der Name einer Rollendatei entspricht dem Namen der darin gespeicherten Rollenklasse (um .java erweitert, [2], §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 ([2], §1.2.5(c)).

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

(a) Callout- und Callin-Bindungen dürfen nur in gebundenen Rollenklassen vorkommen ([2], §A.3(a)).

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

(c) In einer Callin-Bindung 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 Callin-Bindung dürfen Methodennamen und vollständige Signaturen für die Methoden-Bezeichner nicht vermischt werden ([2], §A.3(b)).

(f) In einer Callin Replace-Bindung muss der Methoden-Bezeichner auf der linken Seite auf eine Methode mit `callin`-Modifikator verweisen ([2], §A.3(c)).

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 ([2], §3.2(d)).

OTL7.1. Implizite Parameter Mappings ([2], §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], §2.3.2(a)).

OTL9. Entlang impliziter Vererbung dürfen die Namen von Rollenmethoden oder Feldern nicht irgendwelche vorher sichtbaren Namen verstecken bzw. verdecken ([2], §1.4(b)).

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

(a) Die Basisklasse irgendeiner Rollenklasse darf nicht eine Rolle desselben Teams sein ([2], §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], §2.1.2(a)).

(c) Die hinter `playedBy` angegebene Basisklasse, darf nicht ein umschließender Typ (auf jeglicher Tiefe) der definierten Rollenklasse sein ([2], §2.1.2(b)).

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

(a) Die hinter dem Schlüsselwort `within` angegebene Expression muss eine Team-Instanz bezeichnen ([2], §5.2(a)).

Die Regeln aus Abschnitt 3.1.2.1 (JL1. bis JL8.) gelten uneingeschränkt auch für ObjectTeams/Java.

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

Durch die Einführung der impliziten Vererbung (siehe 2.2.5) und die Möglichkeit der Kombination bzw. Vermischung dieser mit der expliziten Vererbung

(siehe 2.2.6), können neue, zum Teil sehr komplexe Vererbungshierarchien in ObjectTeams/Java Programmen auftreten. Dies hat auch neue Sub-Typ-Beziehungen zur Folge. Diese neuen impliziten und expliziten Vererbungsbeziehungen müssen von einem Refactoring überprüft und beibehalten werden damit das Verhalten einer ObjectTeams/Java Anwendung nicht verändert wird. Um dies zu gewährleisten müssen die Regeln zur Erhaltung von Vererbungsbeziehungen in Java aus Abschnitt 3.1.2.2 überarbeitet und erweitert werden sowie zusätzlich neue Regeln aufgestellt werden, damit die Vererbungsbeziehungen in einem ObjectTeams/Java Programm nach dem Refactoring erhalten bleiben.

3.2.2.1 Regeln für die Erhaltung von Vererbungsbeziehungen in ObjectTeams/Java

Aufgrund der beiden vorhandenen Vererbungsarten in ObjectTeams/Java und den daraus resultierenden Vererbungsbeziehungen, ergeben sich folgende Regeln:

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 Sub-Klassen und in expliziten und impliziten Sub-Rollen als auch in Super-Typen (Klassen und Interfaces) berücksichtigt werden. Das heißt, die Signaturen von überschreibenden Methoden in regulären Sub-Klassen und in Sub-Rollen sowie die Signaturen von überschriebenen (geerbten) Methoden in Super-Typen (Klassen und Interfaces) müssen ebenfalls geändert werden.

OTI2. Wenn eine implizite Sub-Rolle die `extends`-Relation der Super-Rolle verändert, muss die neue Super-Klasse eine Sub-Klasse der Klasse sein, die in der `extends`-Relation der impliziten Super-Rolle deklariert ist ([2], §1.3.2(b)).

OTI3. Wenn eine explizite Sub-Rolle die `playedBy`-Relation der Super-Rolle verändert, muss die neue Basisklasse eine Sub-Klasse der Klasse sein, die in der `playedBy`-Relation der expliziten Super-Rolle deklariert ist ([2], §2.1(c)).

OTI4. Eine implizite Sub-Rolle darf die geerbte `playedBy`-Relation nicht verändern ([2], §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*).

Regel OTI1. folgt aus der Tatsache, dass Rollen sowohl explizit (2.2.6) als auch implizit (2.2.5) erben können und geerbte Features sowohl in expliziten als auch in impliziten Sub-Rollen überschrieben werden können.
Die Regeln OTI2., OTI3., und OTI4. sind explizite Anforderungen von ObjectTeams/Java.
Regel OTI5. erfüllt die Forderung, dass jeder Methoden-Bezeichner in einer Methoden-Bindung genau eine Methode selektieren muss ([2], §3.1(c)). Um diese Selektion zu ermöglichen, müssen die Methoden-Signaturen immer genau zueinander passen.
Die Regel JI1. aus Abschnitt 3.1.2.2 gilt auch für Team- und Rollenklassen in ObjectTeams/Java.

3.2.3 Erhaltung semantischer Äquivalenz in ObjectTeams/Java

Ein angewendetes Refactoring erhält das Verhalten eines ObjectTeams/Java Programms nur dann, wenn neben der Erzeugung von legalen, syntaktisch korrekten Programmen und der Beibehaltung von Vererbungsbeziehungen, die Versionen des Programms vor und nach dem Refactoring auch *semantisch äquivalente Referenzen und Operationen* produzieren (siehe 3.1.1).

3.2.3.1 Semantisch äquivalente Referenzen

Durch die Einführung der neuen Sprachkonzepte in ObjectTeams/Java (siehe 2.2) können im aspektorientierten Code eines ObjectTeams/Java Programms, also in den Team- und Rollenklassen, zusätzliche Referenzen auf Attribute, Methoden und Typen aus dem objektorientierten Code vorkommen. Elemente aus dem Basis-Code können zum Beispiel in Typ-Bindungen (2.2.3), Methoden-Bindungen (2.2.9.1, 2.2.9.4), Lifting-Methoden (2.2.1) und Guards (2.2.11) referenziert werden. Weitere explizite Referenzen auf Elemente aus dem OO-Code können auch an anderen Stellen in Team- und Rollenklassen vorkommen, wie in Import-Deklarationen oder nach dem Schlüsselwort `extends` in der Deklaration einer Rollenklasse.
Werden diese oder andere Referenzen durch ein Refactoring verändert, dann darf sich dabei nicht das Verhalten des ObjectTeams/Java Programms ändern. Das heißt, das Programm muss vor und nach dem Refactoring die gleichen Ausgaben produzieren (siehe Opdykes Definition in 3.1.1).

3.2.3.2 Semantisch äquivalente Operationen

Eine Operation[1] ist semantisch äquivalent zu einer anderen Operation, falls beide aufgerufenen Operationen das gleiche Ergebnis liefern. Ein Refactoring kann auch den Rumpf einer Methode verändern (*Extract Method*[7], S.110 oder *Inline Method*[7], S.117). Solange sich das Ergebnis eines Aufrufs dieser veränderten Methode im Vergleich zum Ergebnis der aufgerufenen Methode vor dem Refactoring nicht verändert hat, handelt es sich um semantisch äquivalente Methoden.
Ein Problem bei der Einhaltung von semantischer Äquivalenz könnten existierende Guards in einer ObjectTeams/Java Anwendung darstellen. Nach einem Refactoring könnte sich die Bedingung eines Guards derart verändert haben, dass möglicherweise vor dem Refactoring ausgeführte Callin-Bindungen nach dem Refactoring nicht mehr aktiviert sind. Dadurch kann sich die Reihenfolge der aufgerufenen Methoden ändern und damit auch das Verhalten des Programms.[2]

3.2.3.3 Regeln für die Erhaltung semantischer Äquivalenz in ObjectTeams/Java

Folgende Änderungen können von einem Refactoring durchgeführt werden, ohne dass dadurch die Äquivalenz von Referenzen und Operationen in einem ObjectTeams/Java Programm beeinflusst wird:

OTS1. Unreferenzierte Variablen (Felder), Methoden und Typen können im OO-Code (Basis-Code) hinzugefügt oder entfernt werden. Dies beinhaltet, dass keine Referenzen auf diese Elemente in Teamklassen oder in Rollenklassen existieren.

Die Regeln OS1., OS2., OS4. und OS5. von Seite 39 gelten auch für ObjectTeams/Java[3].

3.3 Annahmen und Einschränkungen

In den zuvor beschriebenen Regeln für Verhaltenserhaltung sowie in den Definitionen der Vor- und Nachbedingungen der adaptierten Refactorings, die Bestandteil des nächsten Kapitels sind, werden einige wichtige Annahmen gemacht.

[1] In Java und ObjectTeams/Java ist mit Operation eine Methode gemeint.
[2] Ein repräsentatives Code-Beispiel hierfür konnte jedoch nicht gefunden werden.
[3] Die beschriebenen Änderungen in diesen Regeln beziehen sich nur auf den Basis-Code einer ObjectTeams/Java Anwendung.

Die erste Annahme ist, dass jedes Programm, auf das ein Refactoring angewendet werden soll, syntaktisch und semantisch gültig ist, dass heißt, es kompiliert. Diese Annahme vereinfacht die Vorbedingungen, die jedes Refactoring verlangt.
Die zweite Annahme, die erforderlich ist um die Analysen und Transformationen während eines Refactorings korrekt auszuführen, ist, dass auf das gesamte Programm zugegriffen werden kann („Closed-world assumption“). Mit anderen Worten, der gesamte Quellcode eines Programms ist vorhanden, er kann gelesen und modifiziert werden und es muss nur das Verhalten dieses Programms erhalten werden.
Eine bedeutende Einschränkung ist den in dieser Arbeit beschriebenen Refactorings durch *Reflection* auferlegt. Die Verwendung von *Reflection* könnte Abhängigkeiten zwischen Programmstrukturen bilden, die die Refactorings verändern würden. Da diese Abhängigkeiten durch dynamische Werte ausgedrückt werden, können diese Abhängigkeiten nur schwer oder unmöglich statisch ermittelt werden.
Schließlich werden in dieser Arbeit aspektorientierte Konstrukte wie Join-Points und Pointcuts (siehe 1.4.1) bei der Definition von Regeln für Verhaltenserhaltung in ObjectTeams/Java Programmen und bei der Adaption der Refactorings nicht betrachtet, da zum Zeitpunkt der Erstellung dieser Arbeit noch keine Join-Point Sprache für ObjectTeams/Java existierte.

Kapitel 4

Konzeptuelle Erweiterung objektorientierter Refactorings für ObjectTeams/Java

In diesem Kapitel werden einige elementare Refactorings, die Opdyke in seiner Dissertation[22] definiert hat, sowie weitere, häufig verwendete Refactorings aus Fowlers Katalog[7] für die Sprache ObjectTeams/Java erweitert. Die bestehenden Refactorings werden derart adaptiert, dass sie bei der Anwendung auf den objektorientierten Teil (Basis-Code) eines ObjectTeams/Java Programms das Verhalten dieser Anwendung erhalten.
Dazu werden im Einzelnen die von Opdyke und die von Dudziak und Wloka in ihrer Arbeit[3] aufgestellten Vorbedingungen der Refactorings überarbeitet und um neue ObjectTeams/Java-spezifische Vorbedingungen ergänzt. Darüber hinaus werden die von ihnen für jedes Refactoring beschriebenen Nachbedingungen, falls notwendig, um zusätzliche ObjectTeams/Java-spezifische Nachbedingungen erweitert.

4.1 Struktur der Refactoring-Beschreibungen

Für jedes beschriebene Refactoring wird die folgende Struktur verwendet:

Beschreibung der Änderungen
Zunächst wird kurz beschrieben, welche Änderungen das Refactoring durchführt.

Herkunft
Es folgt die Angabe der Herkunft des Refactorings (z.B. ein Refactoring aus dem Refactoring-Katalog[7]).

Parameter
Die Refactorings benötigen in der Regel zusätzliche Informationen, wie zum Beispiel einen neuen Namen für das umzustrukturierende Element oder die Zielklasse, in die eine Methode verschoben werden soll. Diese werden entweder vom Benutzer zur Verfügung gestellt (durch Benutzer-Interaktion in einem Werkzeug) oder durch ein anderes Refactoring, welches das aktuelle verwendet. Diese Parameter werden hier angegeben.

Einfluss von ObjectTeams/Java Code
In diesem Abschnitt wird kurz dargelegt, ob und welche der neuen Sprachelemente bzw. Sprachmechanismen in ObjectTeams/Java (siehe Kapitel 2) das Refactoring beeinflussen und ob der Aspekt-Code Auswirkungen auf bestimmte Sprachanforderungen oder Vererbungsbeziehungen hat.

Vorbedingungen
Damit ein Refactoring bei der Anwendung auf ein ObjectTeams/Java Programm das Verhalten dieser Anwendung erhält müssen die in Kapitel 3.1 überarbeiteten und erweiterten Regeln für ObjectTeams/Java sowie bestimmte Java-Regeln vom Refactoring eingehalten werden. Um die Einhaltung dieser Regeln zu garantieren werden hier für jedes Refactoring Vorbedingungen definiert, die erfüllt sein müssen bevor das Refactoring angewendet werden kann. Hinter jeder Vorbedingung ist in Klammern die entsprechende Regel, falls vorhanden[1], angegeben, deren Einhaltung durch diese Vorbedingung überprüft wird. Neue, ObjectTeams/Java-spezifische Vorbedingungen sind *kursiv* hervorgehoben.

Probleme (Optional)
Während eines Refactorings können Situationen auftreten, die problematisch werden können. All diese Probleme werden in diesem Abschnitt erwähnt.

Nachbedingungen
Die Nachbedingungen repräsentieren den Zustand eines ObjectTeams/Java Programms nach der Durchführung eines Refactorings. Diese Nachbedingungen werden hier definiert. Falls eine Nachbedingung eine vorhandene Regel erfüllt, wird diese in Klammern dahinter angegeben.

[1]Für einige Vorbedingungen, die trivial sind bzw. die sich nicht auf bestimmte Programmeigenschaften beziehen, existieren keine Regeln.

4.2 Einteilung der Refactorings

Die nachfolgend erweiterten Refactorings sind in folgende zwei Kategorien eingeteilt:

- **Atomare Refactorings** und
- **Zusammengesetzte Refactorings**

Die atomaren Refactorings sind wiederum in drei Sub-Kategorien aufgeteilt:

- Refactorings, die Code-Elemente erzeugen.
- Refactorings, die Code-Elemente entfernen.
- Refactorings, die Code-Elemente verändern.

4.3 Atomare Refactorings

Atomare Refactorings sind grundlegende Refactorings, die elementare Änderungen durchführen. Dazu gehört das Erzeugen von neuen Code-Elementen und das Entfernen und Ändern von bereits existierenden Elementen aus bzw. in einem Typ. Die atomaren Refactorings werden häufig in den zusammengesetzten Refactorings (siehe 4.4) verwendet.

4.3.1 Refactorings, die Code-Elemente erzeugen (Create Refactorings)

Diese Refactorings erzeugen einen neuen Typ (reguläre Klasse oder Interface) oder ein neues Element in einem bestehenden Typ. Jedes Refactoring ist verhaltenserhaltend, da die Einheit (Typ oder Member einer Klasse), die es hinzufügt, entweder unreferenziert ist oder semantisch äquivalent ist zu dem ersetzten Member. Unreferenziert bedeutet hier, dass das entsprechende Element auch in Team- und Rollenklassen nicht referenziert wird.
Semantische Äquivalenz spielt beim Erzeugen und beim Entfernen einer Methode eine entscheidende Rolle.
Wird beispielsweise in einer Rollenklasse eine Methode aus einer indirekten Super-Klasse aufgerufen und wird diese Methode durch eine neu erzeugte Methode in der direkten Super-Klasse überschrieben, muss die neue Methode semantisch äquivalent sein zu der redefinierten Methode.

4.3.1.1 Create Type

Beschreibung der Änderungen
Erzeugt einen neuen Top-level Typ (Klasse oder Interface).

Herkunft
Aus [[22], S. 39] und [3].

Parameter

- Der Name des neuen Typs.
- Das Paket, dass den neuen Typ enthalten soll.
- Angabe, ob der neue Typ eine Klasse oder ein Interface ist.
- Die Super-Interfaces (optional).
- Falls der neue Typ eine Klasse ist, die Super-Klasse (optional).
- Falls der neue Typ eine Klasse ist, ob die Klasse abstrakt sein soll oder nicht.
- Falls der neue Typ eine abstrakte Klasse ist, welche Methoden aus den Super-Typen implementiert werden sollen (alle Methoden werden automatisch implementiert, wenn der neue Typ eine nicht-abstrakte Klasse ist).

Einfluss von ObjectTeams/Java Code
Eine existierende Teamklasse (siehe 2.2.1) kann den gleichen qualifizierten Namen haben wie der neue Typ. Wenn eine neue Klasse erzeugt wird gibt es jedoch weder Instanzen von ihr, noch können irgendwelche Sub-Klassen, also auch Rollenklassen, von ihr erben. Wird ein neues Interface erzeugt können ebenfalls keine Klassen (reguläre Klassen oder Rollenklassen) dieses Interface implementieren. Der Aspekt-Code hat daher bei diesem Refactoring Einfluss auf einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*.

Vorbedingungen

1. Der neue Name ist gültig.
2. Es existiert kein Typ (*einschließlich Teamklassen*) oder Paket mit diesem qualifizierten Namen (JL2.).

Nachbedingungen

- Es existiert ein Typ mit dem qualifizierten Namen.

- Der Typ erbt von den spezifizierten Super-Typen.
- Alle angegebenen Methoden sind unter Verwendung einer Standard-Implementierung implementiert.

4.3.1.2 Create Field

Beschreibung der Änderungen
Erzeugt ein neues, unreferenziertes Feld in einem Typ (Klasse oder Interface).

Herkunft
Aus [[22], S. 40] und [3].

Parameter

- Der Name und Typ des Feldes.
- Der Typ zu dem das Feld hinzugefügt werden soll.
- Falls der Ziel-Typ eine Klasse ist, die Sichtbarkeit.
- Falls der Ziel-Typ eine Klasse ist, ob das Feld eine Konstante (`static final`), statisch (`static`), oder eine Instanzvariable ist.

Einfluss von ObjectTeams/Java Code
Da das Feld neu erzeugt wird und es daher in Team- und Rollenklassen nicht referenziert werden kann, hat der Aspekt-Code eines ObjectTeams/Java Programms keinen Einfluss auf dieses Refactoring.

Vorbedingungen

1. *Falls der Ziel-Typ eine Klasse ist, muss es eine reguläre OO-Klasse sein.*
2. Der neue Name ist gültig.
3. Es existiert kein Feld mit dem Namen des neuen Feldes im Ziel-Typ (JL3.).
4. Es existiert kein Feld mit dem Namen des neuen Feldes in einem Super-Typ des Ziel-Typs (JL6.).
5. Falls ein umschliessender Typ des Ziel-Typs ein Feld mit dem Namen des neuen Feldes definiert, dann wird es nicht im Ziel-Typ verwendet (*Shadowing*, siehe [12], 6.3.1).

Probleme
Existiert im Ziel-Typ ein Typ mit dem gleichen unqualifizierten Namen, der unqualifiziert referenziert werden kann, dann kann dies zu Problemen führen, wenn das Feld oder der Typ verwendet werden, um auf Features zuzugreifen.

Nachbedingungen

- Der Typ definiert ein Feld mit der angegebenen Sichtbarkeit, Typ und Namen.

4.3.1.3 Create Method

Beschreibung der Änderungen
Erzeugt eine neue Methode in einem Typ (Klasse oder Interface).

Herkunft
Aus [[22], S. 40] und [3].

Parameter

- Der Typ zu dem die Methode hinzugefügt werden soll.
- Der Name der Methode.
- Die Namen und Typen der Parameter.
- Der Rückgabetyp (optional).
- Falls der Ziel-Typ eine Klasse ist, die Sichtbarkeit der Methode.
- Falls der Ziel-Typ eine Klasse ist, ob die Methode statisch (`static`) ist.

Einfluss von ObjectTeams/Java Code
Rollenklassen können explizit von einer regulären Klasse erben (siehe 2.2.6) und somit auf Methoden dieser Klasse und der indirekten Super-Klassen zugreifen und diese auch überschreiben. Eine Rollenklasse kann unter Umständen eine Methode besitzen, deren Signatur zuweisungskompatibel[2] ist zu der Signatur der neuen Methode. Aufgrund eventuell vorhandener Methoden-Bindungen (siehe 2.2.9) in Rollenklassen kann es auch zu Problemen bezüglich *Overloading* kommen. Somit werden bei diesem Refactoring ein Teil der Sprachanforderungen, und zwar bezüglich *Naming*, sowie *Vererbungsbeziehungen* vom Aspekt-Code beeinflusst.

[2]Eine Signatur ist zuweisungskompatibel zu einer anderen Signatur, wenn ein Aufruf möglich ist, der auf beide Signaturen angewendet werden kann (siehe auch [12], 15.12.2).

Vorbedingungen

1. *Falls der Ziel-Typ eine Klasse ist, muss es eine reguläre OO-Klasse sein.*
2. Der neue Name ist gültig.
3. Es existiert keine Methode mit der gleichen Signatur im Ziel-Typ (JL4.).
4. Es existiert keine Methode im Ziel-Typ, einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*) des Ziel-Typs, deren Signatur zuweisungskompatibel ist zu der Signatur der neuen Methode.
5. Falls die neue Methode eine Redefinition einer geerbten Methode aus einem Super-Typ ist, dann wird die redefinierte Methode im Ziel-Typ und in den Sub-Typen (*inklusive Rollenklassen*) nicht benutzt (sie ist unreferenziert), oder die redefinierende Methode ist semantisch äquivalent zu der redefinierten Methode (OS5.).
6. *Es existiert keine Methode im Ziel-Typ, einem Super-Typ oder einem Sub-Typ, die in einer Methoden-Bindung ohne vollständige Signatur referenziert wird und die von der neuen Methode überladen wird (OTI5.).*

Probleme
Die Erzeugung einer abstrakten Methode in einer Klasse wird nicht erlaubt, da hierdurch auch die Klasse (und eventuell auch die Subklassen) als abstrakt deklariert werden müssten.

Nachbedingungen

- Der Typ definiert eine Methode mit der spezifizierten Signatur.
- Falls die Methode eine Redefinition ist, dann hat sie die gleichen Exception-Typen wie die redefinierte Methode und sie enthält einen Aufruf der redefinierten Methode.

4.3.2 Refactorings, die Code-Elemente entfernen (Delete Refactorings)

Die Refactorings dieser Gruppe entfernen einen unreferenzierten Typ oder ein unreferenziertes Element eines Typs. Jedes Refactoring erhält das Verhalten, weil die Einheit (Typ oder Member einer Klasse), die es entfernt, unreferenziert ist. Dies schließt mit ein, dass der entsprechende Typ oder das Element nicht in Team- und Rollenklassen referenziert wird.

4.3.2.1 Delete Type

Beschreibung der Änderungen
Entfernt einen unreferenzierten Top-level Typ.

Herkunft
Aus [[22], S. 41] und [3].

Parameter

- Der Typ, der entfernt werden soll.

Einfluss von ObjectTeams/Java Code
Da der Typ im Aspekt-Code nicht referenziert wird, beeinflusst der Aspekt-Code eines ObjectTeams/Java Programms dieses Refactoring nicht.

Vorbedingungen

1. *Falls der Typ eine Klasse ist, muss es eine reguläre OO-Klasse sein.*
2. Der Typ wird nicht referenziert. *Dies beinhaltet, dass auch in Team- und Rollenklassen keine Referenzen auf den Typ existieren* (OTS1.).
3. Der Typ hat keine Sub-Klassen. *Dies schließt mit ein, dass er keine Rollenklasse als Sub-Klasse hat.*

Nachbedingungen

- Es existiert kein Typ mit diesem qualifizierten Namen.

4.3.2.2 Delete Field

Beschreibung der Änderungen
Entfernt ein unreferenziertes Feld aus einem Typ.

Herkunft
Aus [[22], S. 41] und [3].

Parameter

- Das Feld, das entfernt werden soll.

Einfluss von ObjectTeams/Java Code
Da das Feld im Aspekt-Code nicht referenziert wird, hat der Aspekt-Code eines ObjectTeams/Java Programms keine Auswirkungen auf dieses Refactoring.

Vorbedingungen

1. *Falls das Feld in einer Klasse deklariert ist, muss es eine reguläre OO-Klasse sein.*
2. Das Feld wird nicht referenziert. *Dies beinhaltet, dass auch in Team- und Rollenklassen keine Referenzen auf das Feld existieren* (OTS1.).

Nachbedingungen

- Der Typ definiert kein Feld mit diesem Namen.

4.3.2.3 Delete Method

Beschreibung der Änderungen
Entfernt eine unreferenzierte Methode aus einem Typ.

Herkunft
Aus [[22], S. 41] und [3].

Parameter

- Die Methode, die entfernt werden soll.

Einfluss von ObjectTeams/Java Code
Da die Methode im Aspekt-Code nicht referenziert wird, hat der Aspekt-Code eines ObjectTeams/Java Programms keinen Einfluss auf dieses Refactoring.

Vorbedingungen

1. *Falls die Methode in einer Klasse deklariert ist, muss es eine reguläre OO-Klasse sein.*
2. Die Methode wird nicht referenziert. *Dies beinhaltet, dass auch in Team- und Rollenklassen keine Referenzen auf die Methode existieren* (OTS1.).
3. Die Methode ist entweder abstrakt oder keine Redefinition.

Nachbedingungen

- Der Typ definiert keine Methode mit dieser Signatur.

4.3.3 Refactorings, die Code-Elemente verändern (Change Refactorings)

Zu dieser Kategorie gehören Refactorings, die den Namen eines Typen oder den Namen eines Elements einer Klasse verändern. Weitere Refactorings verändern die Signatur einer bzw. mehrerer Methoden, indem sie Parameter hinzufügen oder entfernen.
Auf Refactorings, die die Signatur einer Methode verändern (den Namen oder die Parameteranzahl der Methode), hat der Aspekt-Code in Object-Teams/Java große Auswirkungen, da durch die Existenz von Rollenklassen nun neue Sub-Typ-Beziehungen entstehen können. Das bedeutet, dass Methoden in regulären Java-Klassen durch Methoden, die in explizit erbenden Rollenklassen als auch in weiteren impliziten Sub-Rollen definiert sind, überschrieben werden können.
Das Überladen (*Overloading*) von Methoden kann aufgrund der Existenz von Methoden-Bindungen ebenfalls zu Problemen führen.
Alle Refactorings dieser Gruppe müssen auch vorhandene *Aspekt-Basis Referenzen* sowie weitere Referenzen auf Elemente aus dem Basis-Programm (z.B. Methodenaufrufe) innerhalb des Aspekt-Codes beachten.

4.3.3.1 Rename Type

Beschreibung der Änderungen
Ändert den Namen eines Top-level Typs sowie alle Referenzen auf diesen Typ.

Herkunft
Aus [[22], S. 42] und [3].

Parameter

- Der Typ, der umbenannt werden soll.
- Der neue, unqualifizierte Name.

Einfluss von ObjectTeams/Java Code
Der Typ, der umbenannt werden soll, kann an verschiedenen Stellen im Aspekt-Code explizit anhand seines Namens referenziert werden: in der Import-Deklaration einer Teamklasse, in der `extends`-Relation einer Rollenklasse, in einer Lifting-Methode (*Declared Lifting*, siehe 2.2.1), in einer Typ-Bindung (`playedBy`-Relation, siehe 2.2.3) und in einer Callout- oder Callin-Bindung (siehe 2.2.9.1 und 2.2.9.4) mit vollständig angegebenen Signaturen (als Rückgabe-Typ oder Parameter-Typ einer referenzierten Rollen- oder Basismethode oder als Typ eines referenzierten Feldes in einer Callout To Field-Bindung). Weitere mögliche Verwendungen für einen Typen sind: als

ein Exception-Typ, als ein Typ einer lokalen Variablen, in einem Cast, in einer Instanziierung, als ein Array-Typ, als ein `instanceof`-Typ und in einem `class`-Zugriff. Alle diese Typ-Referenzen können auch in Rollenklassen, die ihrerseits in separaten Dateien (siehe 2.2.2.2) abgespeichert seien können, und in Teamklassen vorkommen.
Somit beeinflussen vor allem die *Aspekt-Basis Referenzen* sowie andere Typ-Referenzen innerhalb des Aspekt-Codes dieses Refactoring.

Vorbedingungen

1. *Der Typ muss eine reguläre OO-Klasse sein.*
2. Der neue Name ist gültig.
3. Es existiert kein Typ (*einschließlich Teamklassen*) oder Paket mit dem neuen, qualifizierten Namen (JL2.).
4. *Wird der Typ in einer Typ-Bindung (`playedBy`) referenziert, dann darf die gebundene Rollenklasse nicht den neuen, unqualifizierten Namen haben (OTL10.(b)).*
5. Entweder befindet sich der Typ nicht im default-Paket oder er wird nicht in einem Typ oder dem Scope eines Feldes oder einer Variable mit dem neuen, unqualifizierten Namen referenziert.

Probleme
Es können verschiedene Probleme und Fehler beim Umgang mit Importen auftreten, wenn ein Typ A umbenannt wird und ein Typ B mit dem gleichen, unqualifizierten Namen in einem anderen Paket existiert. Die einfachste Strategie ist, immer die qualifizierten Namen (Paketname + Typname) von A und B zu verwenden und direkte Importe der beiden Typen zu entfernen.

Nachbedingungen

- Der Typ hat den neuen, unqualifizierten Namen.
- Alle Referenzen auf diesen Typ verwenden den neuen Namen. *Dies trifft auch auf Referenzen auf diesen Typ in Team- und Rollenklassen zu.*

4.3.3.2 Rename Field

Beschreibung der Änderungen
Ändert den Namen eines Feldes sowie alle Referenzen auf dieses Feld. Zusätzlich werden vorhandene Zugriffsmethoden (Getter- und Setter-Methoden)

umbenannt (wenn möglich).

Herkunft
Aus [[22], S. 42] und [3].

Parameter

- Das Feld, das umbenannt werden soll.
- Der neue Name.

Einfluss von ObjectTeams/Java Code
Das Feld, das umbenannt werden soll, kann auf der rechten Seite einer Callout To Field-Bindung (siehe 2.2.9.1) explizit referenziert werden. Diese Callout To Field-Bindung kann auch in einer Rollendatei (siehe 2.2.2.2) vorkommen. Bei diesem Refactoring hat der Aspekt-Code daher Auswirkungen auf einen Teil der Sprachanforderungen, und zwar bezüglich *Naming*.

Vorbedingungen

1. *Die Klasse, die das Feld deklariert, muss eine reguläre OO-Klasse sein.*
2. Der neue Name ist gültig.
3. Es existiert kein Feld mit dem neuen Namen im selben Typ (JL3.).
4. Es existiert kein Feld mit dem neuen Namen in einem Super-Typ (JL6.).
5. Falls ein umschliessender Typ ein Feld mit dem neuen Namen definiert, dann wird es nicht im deklarierenden Typ verwendet (*Shadowing*, siehe [12], 6.3.1).
6. Das Feld wird nicht ohne explizite Angabe von `this` im Scope eines Feldes, einer lokalen Variablen, eines Parameters oder eines Typen mit diesem (neuen) Namen referenziert.

Probleme
Das Umbenennen der Getter- und Setter-Methoden unter Verwendung von *Rename Method* kann aus diversen Gründen fehlschlagen (siehe 4.3.3.3).

Nachbedingungen

- Das Feld hat den neuen Namen.
- Alle Referenzen auf das Feld verwenden den neuen Namen. *Dies trifft auch auf Referenzen auf dieses Feld in Callout To Field-Bindungen zu.*

4.3.3.3 Rename Method

Beschreibung der Änderungen
Ändert den Namen einer Methode und aller redefinierten/redefinierenden Methoden sowie alle Referenzen auf diese Methode.

Herkunft
Aus [[7], S. 273] und [[22], S. 43].

Parameter

- Die Methode, die umbenannt werden soll.
- Der neue Name.

Einfluss von ObjectTeams/Java Code
Die Methode, die umbenannt werden soll, kann von einer Methode in einer Rollenklasse überschrieben (redefiniert) werden. Diese Methode kann wiederum in expliziten und impliziten Sub-Rollen redefiniert werden (siehe 2.2.5). Die überschreibende Methode kann auch in einem `tsuper`-Aufruf innerhalb einer Methode in einer impliziten Sub-Rolle referenziert werden. Die Rollenklassen können unter Umständen Methoden besitzen, deren Signaturen zuweisungskompatibel zu der neuen Signatur sind. Es ist möglich, dass die Methode (oder eine der redefinierenden Methoden) auf der linken Seite (gebundene Rollenmethode) oder auf der rechten Seite (gebundene Basismethode) einer Methoden-Bindung (siehe 2.2.9.1 und 2.2.9.4) explizit anhand ihres Namens referenziert (selektiert) wird. Handelt es sich bei der umbenannten Methode um eine private Methode, so kann diese ebenfalls auf der rechten Seite einer Methoden-Bindung referenziert werden. Existieren in irgendeiner Rollenklasse Methoden-Bindungen (siehe 2.2.9), können bei diesem Refactoring auch Probleme bezüglich *Overloading* auftreten. Referenzen auf die umzubenennende Methode können auch in einer Rollendatei (siehe 2.2.2.2) oder in Guards (siehe 2.2.11) vorkommen. Folglich hat bei diesem Refactoring der Aspekt-Code Einfluss auf einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*, und auf *Vererbungsbeziehungen*.

Vorbedingungen

1. *Falls die Methode in einer Klasse deklariert ist, muss es eine reguläre OO-Klasse sein.*
2. Der neue Name ist gültig.
3. Es existiert keine Methode mit der gleichen Signatur in dem Typ, der die Methode definiert (JL4.).

4. Es existiert keine Methode im gleichen Typ, einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*), deren Signatur zuweisungskompatibel zu der neuen Signatur ist, (dies gilt auch für jede überschriebene/überschreibende Methode).

5. *Es existiert keine Methode im Ziel-Typ, einem Super-Typ oder einem Sub-Typ, die in einer Methoden-Bindung ohne vollständige Signatur referenziert wird und die von der umbenannten Methode überladen wird (OTI5.).*

Nachbedingungen

- Die Methode hat den neuen Namen.
- Alle redefinierten/redefinierenden Methoden haben den neuen Namen. *Dies trifft auch auf redefinierende Methoden in Rollenklassen zu (OTI1.).*
- Alle Referenzen auf die Methode verwenden den neuen Namen. *Dies trifft auch auf Referenzen auf die Methode und redefinierende Methoden in Team- und Rollenklassen zu (Referenzen in Methoden-Bindungen,* `tsuper`*-Aufrufen, Guards, etc.).*

4.3.3.4 Add Parameter

Beschreibung der Änderungen
Fügt einer Methode und allen redefinierten/redefinierenden Methoden einen neuen Parameter hinzu.

Herkunft
Aus [[7], S. 275] und [[22], S. 45].

Parameter

- Die Methode, deren Parameter-Liste verändert werden soll.
- Der Name und Typ des neuen Parameters.
- Falls die Parameter-Liste nicht leer ist, dann die Position des neuen Parameters in der Liste.

Einfluss von ObjectTeams/Java Code
Die Methode, die einen neuen Parameter erhalten soll, kann von einer Methode in einer Rollenklasse überschrieben (redefiniert) werden, die wiederum von Methoden in expliziten und impliziten Sub-Rollen überschrieben werden kann (siehe 2.2.5). Die überschriebene Methode kann auch in einem

`tsuper`-Aufruf innerhalb einer Methode in einer impliziten Sub-Rolle referenziert werden. In den Rollenklassen können auch Methoden vorkommen, deren Signaturen zuweisungskompatibel zu der veränderten Signatur sind. Es ist auch möglich, dass eine oder mehrere der zu verändernden Methoden in einer Methoden-Bindung referenziert wird bzw. werden (siehe 2.2.9). In den Methoden-Bindungen müssen bestimmte Bedingungen bezüglich des Mappings von Parametern erfüllt sein (siehe Vorbedingungen). Der Aspekt-Code beeinflusst bei diesem Refactoring somit die *Vererbungsbeziehungen* und einen Teil der Sprachanforderungen, und zwar bezüglich *Naming*.

Vorbedingungen

1. *Falls die Methode in einer Klasse deklariert ist, muss es eine reguläre OO-Klasse sein.*
2. Es existiert kein Parameter mit demselben Namen in den Methoden, die verändert werden sollen (JL3.).
3. Keine Methode, die verändert werden soll, verwendet eine Variable oder ein Feld mit demselben Namen (JL3.).
4. Es existiert weder eine Methode im Typ, der die Methode definiert, noch in einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*), deren Signatur zuweisungskompatibel zu der veränderten Signatur ist.
5. *Wird die veränderte Methode bzw. eine veränderte redefinierende Methode in einer Methoden-Bindung ohne vollständige Signatur referenziert, dann muss jeder Parameter der Rollenmethode konform sein zu dem korrespondierenden Parameter der Basismethode (OTL7.1(a)).*

Nachbedingungen

- Die Methode hat einen neuen Parameter mit dem angegebenen Namen und Typ an der spezifizierten Position.
- Alle redefinierten/redefinierenden Methoden besitzen den neuen Parameter. *Dies trifft auch auf redefinierende Methoden in Rollenklassen zu (OTI1.).*
- *Die vollständig angegebene Signatur der Methode oder einer redefinierenden Methode in den Methoden-Bindungen enthält den neuen Parameter mit dem angegebenen Namen und Typ an der spezifizierten Position.*

- Alle Aufrufe der Methode und der redefinierten/redefinierenden Methoden enthalten als Argument für den neuen Parameter den Standard-Wert des Typs des Parameters. *Dies trifft auch auf Aufrufe der Methode und der redefinierenden Methoden in Team- und Rollenklassen zu (z.B. in* `tsuper`*-Aufrufen).*

4.3.3.5 Remove Parameter

Beschreibung der Änderungen
Entfernt einen unreferenzierten Parameter aus der Signatur einer Methode sowie aus den Signaturen aller redefinierten/redefinierenden Methoden.

Herkunft
Aus [[7], S. 277] und [[22], S. 46].

Parameter

- Die Methode, deren Parameter-Liste verändert werden soll.
- Der Parameter, der entfernt werden soll.

Einfluss von ObjectTeams/Java Code
Die Methode, aus deren Signatur ein Parameter entfernt werden soll, kann von einer Methode in einer Rollenklasse überschrieben (redefiniert) werden. Diese kann wiederum von Methoden in expliziten und impliziten Sub-Rollen überschrieben werden (siehe 2.2.5). Die überschriebene Methode kann auch in einem `tsuper`-Aufruf innerhalb einer Methode in einer impliziten Sub-Rolle referenziert werden. In den Rollenklassen können auch Methoden existieren, deren Signaturen zuweisungskompatibel zu der veränderten Signatur sind. Darüber hinaus kann es vorkommen, dass eine oder mehrere der zu verändernden Methoden in einer Methoden-Bindung referenziert wird bzw. werden (siehe 2.2.9). In den Methoden-Bindungen müssen bestimmte Bedingungen bezüglich des Mappings von Parametern erfüllt sein (siehe Vorbedingungen). Der Parameter kann auch in einem Parameter Mapping (siehe 2.2.9.3 und 2.2.9.5) einer Methoden-Bindung benutzt werden. Die Folge ist, dass bei diesem Refactoring der Aspekt-Code Auswirkungen auf *Vererbungsbeziehungen* und einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*, hat.

Vorbedingungen

1. *Falls die Methode in einer Klasse deklariert ist, muss es eine reguläre OO-Klasse sein.*
2. Der Parameter wird nicht in der Methode verwendet.

3. Der Parameter wird nicht in den redefinierten/redefinierenden Methoden verwendet und alle Expressions (Argumente), die an diesen Parameter übergeben werden, haben keine Seiten-Effekte.

4. *Der Parameter wird nicht in einem Parameter Mapping verwendet.*

5. Es existiert weder eine Methode im Typ, der die Methode definiert, noch in einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*), deren Signatur zuweisungskompatibel zu der veränderten Signatur ist.

6. *Wird die veränderte Methode bzw. eine veränderte redefinierende Methode in einer Methoden-Bindung ohne vollständige Signatur referenziert, dann muss jeder Parameter der Rollenmethode konform sein zu dem korrespondierenden Parameter der Basismethode (OTL7.1(a)).*

Nachbedingungen

- Die Methode hat den angegebenen Parameter nicht mehr.
- Alle redefinierten/redefinierenden Methoden besitzen den Parameter nicht mehr. *Dies trifft auch auf redefinierende Methoden in Rollenklassen zu (OTI1.).*
- *Die vollständig angegebene Signatur der Methode oder einer redefinierenden Methode in den Methoden-Bindungen enthält den Parameter nicht mehr.*
- Alle Aufrufe der Methode und der redefinierten/redefinierenden Methoden besitzen nicht mehr das Argument, das dem entfernten Parameter entspricht. *Dies trifft auch auf Aufrufe der Methode und der redefinierenden Methoden in Team- und Rollenklassen zu (z.B. in* `tsuper`*-Aufrufen).*

4.4 Zusammengesetzte Refactorings

Die zusammengesetzten Refactorings verwenden jeweils ein oder mehrere atomare Refactorings (siehe 4.3). Zu den Änderungen, die von Refactorings dieser Gruppe durchgeführt werden, zählt das Verschieben von Klassen in ein anderes Paket, das Verschieben von Membern einer Klasse in eine andere Klasse, und das Erzeugen und/oder Entfernen von (Super-/Sub-)Klassen. Die Member einer Klasse können dabei entlang einer existierenden Vererbungshierarchie verschoben werden oder in Klassen ausserhalb dieser Vererbungshierarchie[3].

[3]Diese Klassen können wiederum Teil einer anderen Vererbungshierarchie sein.

4.4.1 Refactorings, die Code-Elemente verschieben (Move Refactorings)

Refactorings, die Elemente aus einer Klasse bzw. aus einem Paket in eine andere Klasse bzw. ein anderes Paket verschieben, erzeugen ein neues Element in der Ziel-Klasse bzw. im Ziel-Paket und entfernen in der Regel das ursprüngliche Element in der Original-Klasse bzw. im Original-Paket. Die Refactorings dieser Gruppe verwenden somit diejenigen atomaren Refactorings, die Code-Elemente erzeugen bzw. entfernen.
Das Refactoring *Move Class* verwendet *Create Type* und *Delete Type*. Die Refactorings *Move Field*, *Pull Up Field* und *Push Down Field* verwenden *Create Field* und *Delete Field*. Die Refactorings *Move Method*, *Pull Up Method* und *Push Down Method* verwenden *Create Method* und *Delete Method*.
In den Refactorings, die Methoden verschieben, hat der Aspekt-Code wieder Einfluss auf Vererbungsbeziehungen. Die meisten Refactorings, die hier definiert sind, müssen auch bestehende Referenzen auf die zu verschiebenden Elemente in Team- und Rollenklassen berücksichtigen.

4.4.1.1 Move Class

Beschreibung der Änderungen
Verschiebt eine Klasse in ein anderes Paket.

Herkunft
Aus [6] und [[22][4], S. 53].

Parameter

- Die Klasse, die verschoben werden soll.
- Das Ziel-Paket.

Einfluss von ObjectTeams/Java Code
Eine Teamklasse (siehe 2.2.1) mit dem gleichen unqualifizierten Namen wie die Klasse, die verschoben werden soll, kann im Ziel-Paket existieren. Der Aspekt-Code beeinflusst somit bei diesem Refactoring einen Teil der Sprachanforderungen, und zwar bezüglich *Naming*.

Vorbedingungen

1. *Die Klasse, die verschoben werden soll, ist eine reguläre OO-Klasse.*
2. Es existiert kein Typ (*einschließlich Teamklassen*) oder Sub-Paket mit dem gleichen unqualifizierten Namen im Ziel-Paket (JL2.).

[4] Anmerkung: Opdyke definiert dieses Refactoring im Sinne von *Change Superclass.*

Probleme
Es können Probleme entstehen, wenn es Beziehungen zwischen der zu verschiebenden Klasse und einer anderen Klasse im ursprünglichen Paket gibt. Eine solche Beziehung ist gegeben, falls beide Klassen sich im selben Paket befinden und eine der Klassen ein Feature mit *Paket-Sichtbarkeit* (`protected` oder friendly) der anderen Klasse referenziert, oder wenn eine der Klassen nur innerhalb des Pakets sichtbar ist und von der anderen Klasse in diesem Paket benutzt wird.
Entweder wird in solchen Fällen das Verschieben der Klasse nicht unterstützt oder die Sichtbarkeit der referenzierten Features oder Typen wird auf `public` gesetzt (bei Methoden eventuell auch die Sichtbarkeit der Redefinitionen der jeweiligen Methode).

Nachbedingungen

- Die Klasse befindet sich im anderen Paket.
- Alle Importe und Referenzen auf die Klasse (*auch in Teamklassen*) sind aktualisiert.

4.4.1.2 Move Field

Beschreibung der Änderungen
Verschiebt ein Feld und seine Zugriffsmethoden (Getter- und Setter-Methoden) in eine andere Klasse.

Herkunft
Aus [[7], S. 146].

Parameter

- Das Feld, das verschoben werden soll.
- Die Ziel-Klasse.

Einfluss von ObjectTeams/Java Code
Eine Rollenklasse kann eine Sub-Klasse der Ziel-Klasse sein (siehe 2.2.6) und Methoden definieren, die die verschobenen Getter- und Setter-Methoden überschreiben. Diese Methoden können wiederum in expliziten und impliziten Sub-Rollen überschrieben werden (siehe 2.2.5). Außerdem kann die Signatur einer Rollenmethode zuweisungskompatibel zu der Signatur einer verschobenen Methode sein. Das Vorhandensein von Methoden-Bindungen (siehe 2.2.9) in Rollenklassen kann bei diesem Refactoring zu Problemen bezüglich *Overloading* führen. Weiterhin kann das zu verschiebende Feld auf der rechten Seite einer Callout-Bindung (siehe 2.2.9.1) referenziert sein.

Daher hat der Aspekt-Code bei diesem Refactoring Auswirkungen auf einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*, sowie auf *Vererbungsbeziehungen.*

Vorbedingungen

1. *Die Klasse, die das Feld deklariert, ist eine reguläre OO-Klasse.*
2. *Die Ziel-Klasse ist eine reguläre OO-Klasse*, die keine Super- oder Sub-Klasse der Klasse ist, die das Feld definiert[5].
3. Das Feld ist statisch und verwendet ausschließlich Features, die auch in der Ziel-Klasse sichtbar sind, oder es wird nur in seinen Zugriffsmethoden verwendet, die wiederum nirgendwo referenziert werden. *Das Feld darf auch nicht in einer Callout-Bindung referenziert werden.*
4. Es existiert kein Feld mit dem Namen des verschobenen Feldes in der Ziel-Klasse (JL3.).
5. Es existiert kein Feld mit dem Namen des verschobenen Feldes in einer Super-Klasse der Ziel-Klasse (JL6.).
6. Falls eine umschliessende Klasse der Ziel-Klasse ein Feld mit diesem Namen definiert, dann wird es nicht in der Ziel-Klasse verwendet (*Shadowing*, siehe [12], 6.3.1).
7. Es existiert keine Methode in der Ziel-Klasse, einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*) der Ziel-Klasse, deren Signatur zuweisungskompatibel zu irgendeiner Signatur der Zugriffsmethoden ist.
8. *Es existiert keine Methode in der Ziel-Klasse, einer Super-Klasse oder einer Sub-Klasse (inklusive Rollenklassen) der Ziel-Klasse, die in einer Methoden-Bindung ohne vollständige Signatur referenziert wird und die von einer der verschobenen Methoden überladen wird (OTI5.).*

Nachbedingungen

- Die Original-Klasse definiert das Feld nicht mehr.
- Die Ziel-Klasse definiert das Feld.
- Alle Referenzen auf das Feld werden aktualisiert (falls notwendig).

[5]Für das Verschieben eines Feldes in eine Super- oder Subklasse, siehe 4.4.1.4 und 4.4.1.5.

4.4.1.3 Move Method

Beschreibung der Änderungen
Verschiebt eine Methode in eine andere Klasse.

Herkunft
Aus [[7], S. 142].

Parameter

- Die Methode, die verschoben werden soll.
- Die Ziel-Klasse.
- Falls die Methode nicht statisch ist und sie nicht-statische Features ihrer umschliessenden Typen verwendet, dann ein Feld oder eine Methode in der Ziel-Klasse oder eine ihrer umschliessenden Klassen für jeden der verwendeten umschliessenden Original-Typen, dessen Typ dieser umschliessende Original-Typ oder einer seiner Sub-Typen ist.
- Falls die Methode nicht statisch ist und aufgerufen wird, dann ein Feld oder eine Methode, auf die lokal in der Original-Klasse zugegriffen werden kann und deren Rückgabewert die Ziel-Klasse ist.

Einfluss von ObjectTeams/Java Code
Die Ziel-Klasse kann als Sub-Klasse eine Rollenklasse haben (siehe 2.2.6), die eine Methode enthält, die die verschobene Methode überschreibt. Diese Methode kann wiederum in expliziten und impliziten Sub-Rollen überschrieben werden (siehe 2.2.5). Es ist auch möglich, dass die Signatur einer Methode in diesen Rollen zuweisungskompatibel zu der Signatur der verschobenen Methode ist. Darüber hinaus können auch Fehler im Zusammenhang mit *Overloading* auftreten, falls in den Rollenklassen Methoden-Bindungen existieren. Folglich hat bei diesem Refatoring der Aspekt-Code Einfluss auf einen Teil der Sprachanforderungen, und zwar bezüglich *Naming*, sowie auf *Vererbungsbeziehungen*.

Vorbedingungen

1. *Die Klasse, die die Methode deklariert, ist eine reguläre OO-Klasse.*
2. *Die Ziel-Klasse ist eine reguläre OO-Klasse*, die keine Super- oder Sub-Klasse der Klasse ist, die die Methode definiert[6].
3. Die Methode ist nicht abstrakt.

[6]Für das Verschieben einer Methode in eine Super- oder Subklasse, siehe 4.4.1.6 und 4.4.1.7.

4. Es existiert keine Methode mit der gleichen Signatur in der Ziel-Klasse (JL4.).

5. Es existiert keine Methode in der Ziel-Klasse, einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*) der Ziel-Klasse, deren Signatur zuweisungskompatibel ist zu der Signatur der verschobenen Methode.

6. *Es existiert keine Methode in der Ziel-Klasse, einer Super-Klasse oder einer Sub-Klasse (inklusive Rollenklassen) der Ziel-Klasse, die in einer Methoden-Bindung ohne vollständige Signatur referenziert wird und die von der verschobenen Methode überladen wird (OTI5.).*

7. Die Methode referenziert keine privaten Features eines ursprünglich umschliessenden Typen oder sie wird nicht ausserhalb des Scopes der verwendeten Features verschoben.

8. Die Methode referenziert keine Features, die `protected` deklariert sind, eines ursprünglich umschliessenden Typen oder sie wird in eine Klasse im selben Paket verschoben.

9. Die Methode referenziert keine Features, die nur innerhalb eines Pakets sichtbar sind, eines ursprünglich umschliessenden Typen oder sie wird in eine Klasse im selben Paket verschoben.

Nachbedingungen

- Die Methode befindet sich in der Ziel-Klasse.
- Alle Zugriffe auf statische Features von den ursprünglich umschliessenden Typen innerhalb der Methode sind vollständig qualifiziert.
- Falls die Methode referenziert wird, dann ruft die Original-Methode die verschobene Methode auf (delegierende Methode). Ein spezifiziertes Feld oder eine Methode wird für den Zugriff verwendet.

4.4.1.4 Pull Up Field

Beschreibung der Änderungen
Verschiebt ein identisches Feld aus den Sub-Klassen in die Super-Klasse.

Herkunft
Aus [[7], S. 320] und [[22], S. 51].

Parameter

- Das Feld, das verschoben werden soll.

Einfluss von ObjectTeams/Java Code
Eine oder mehrere der Sub-Klassen kann eine Rollenklasse sein (siehe 2.2.6), die das zu verschiebende Feld definiert. Bei diesem Refactoring hat der Aspekt-Code folglich Einfluss auf einen Teil der Sprachanforderungen, und zwar bezüglich *Naming*, und auf *Vererbungsbeziehungen*.

Vorbedingungen

1. *Die Sub-Klassen, aus denen das Feld verschoben werden soll, sind reguläre OO-Klassen.*
2. Das Feld ist in allen Sub-Klassen, in denen es definiert ist, identisch definiert (gleiche Typen und Namen).
3. In der Initialisierung des Feldes werden keine Features verwendet, die nicht in der Super-Klasse oder einer ihrer Super-Klassen deklariert sind (inklusive polymorpher Methoden).
4. *Es existiert keine Rollenklasse, die explizit von der Super-Klasse erbt und die ein Feld definiert, welches nach dem Refactoring das Feld aus der Super-Klasse überschreiben würde (JL6.).*
5. Es existiert kein Feld mit dem Namen des verschobenen Feldes in der Super-Klasse (JL3.).
6. Es existiert kein Feld mit dem Namen des verschobenen Feldes in einer Super-Klasse der Ziel-Klasse (JL6.).

Nachbedingungen

- Die Sub-Klassen definieren das Feld nicht mehr.
- Die Super-Klasse definiert das Feld.

4.4.1.5 Push Down Field

Beschreibung der Änderungen
Verschiebt ein Feld aus der Super-Klasse in die Sub-Klassen.

Herkunft
Aus [[7], S. 329] und [[22], S. 51].

Parameter

- Das Feld, das verschoben werden soll.

Einfluss von ObjectTeams/Java Code
Unter den Sub-Klassen können sich Rollenklassen befinden (siehe 2.2.6). Somit beeinflusst bei diesem Refactoring der Aspekt-Code die *Vererbungsbeziehungen*.

Vorbedingungen

1. *Die Super-Klasse, aus der das Feld verschoben werden soll, ist eine reguläre OO-Klasse.*
2. *Die Sub-Klasse bzw. Sub-Klassen, in die das Feld verschoben werden soll, ist eine bzw. sind reguläre OO-Klasse(n).*
3. Es existieren keine Referenzen auf das Feld in der Super-Klasse (das Feld kann jedoch in der Sub-Klasse bzw. den Sub-Klassen referenziert sein).

Nachbedingungen

- Die Super-Klasse definiert das Feld nicht mehr.
- Die Sub-Klasse(n) definiert/definieren das Feld.

4.4.1.6 Pull Up Method

Beschreibung der Änderungen
Verschiebt eine identische Methode aus den Sub-Klassen in die Super-Klasse.

Herkunft
Aus [[7], S. 322].

Parameter

- Die Methode, die verschoben werden soll.

Einfluss von ObjectTeams/Java Code
Eine oder mehrere der Sub-Klassen kann eine Rollenklasse sein (siehe 2.2.6), die die zu verschiebende Methode definiert. Der Aspekt-Code hat bei diesem Refactoring daher Auswirkungen auf einen Teil der Sprachanforderungen, und zwar bezüglich *Naming*, und auf *Vererbungsbeziehungen*.

Vorbedingungen

1. *Die Sub-Klassen, aus denen die Methode verschoben werden soll, sind reguläre OO-Klassen.*

2. Die Methode ist in allen Sub-Klassen, in denen sie definiert ist, identisch definiert (gleiche Signaturen und Methodenrümpfe).

3. Im Rumpf der Methode werden keine Features verwendet, die nicht in der Super-Klasse oder einer ihrer Super-Klassen deklariert sind (inklusive polymorpher Methoden).

4. Es existiert keine Methode in der Super-Klasse, einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*), deren Signatur zuweisungskompatibel zu der Signatur der verschobenen Methode ist.

5. *Es existiert keine Methode in der Ziel-Klasse, einer Super-Klasse oder einer Sub-Klasse (Rollenklasse) der Ziel-Klasse, die in einer Methoden-Bindung ohne vollständige Signatur referenziert wird und die von der verschobenen Methode überladen wird (OTI5.).*

Nachbedingungen

- Die Sub-Klassen definieren die Methode nicht mehr.
- Die Super-Klasse definiert die Methode.

4.4.1.7 Push Down Method

Beschreibung der Änderungen
Verschiebt eine Methode aus der Super-Klasse in die Sub-Klassen.

Herkunft
Aus [[7], S. 328].

Parameter

- Die Methode, die verschoben werden soll.

Einfluss von ObjectTeams/Java Code
Unter den Sub-Klassen können sich Rollenklassen befinden (siehe 2.2.6). Daher beeinflusst bei diesem Refactoring der Aspekt-Code die *Vererbungsbeziehungen*.

Vorbedingungen

1. *Die Super-Klasse, aus der das Feld verschoben werden soll, ist eine reguläre OO-Klasse.*

2. *Die Sub-Klasse bzw. Sub-Klassen, in die das Feld verschoben werden soll, ist eine bzw. sind reguläre OO-Klasse(n).*

3. Die Methode wird nur in der Sub-Klasse oder einer ihrer Sub-Klassen referenziert.

4. Es existiert keine Methode in irgendeiner der Sub-Klassen (*inklusive Rollenklassen*), einem Super-Typ oder einer Sub-Klasse dieser Sub-Klassen, deren Signatur zuweisungskompatibel zu der Signatur der verschobenen Methode ist.

Nachbedingungen

- Die Super-Klasse enthält eine nicht-abstrakte Template-Methode mit einer Standard-Implementierung.
- Die Sub-Klasse bzw. Sub-Klassen definiert bzw. definieren die Methode.

4.4.2 Andere zusammengesetzte Refactorings

Die Refactorings dieser Kategorie verwenden die zuvor definierten Refactorings aus Abschnitt 4.4.1 (mit Ausnahme von *(Self) Encapsulate Field*, *Extract Method* und *Inline Method*).
Die Refactorings *Extract Subclass*, *Extract Superclass* und *Collapse Hierarchy* verwenden *Create Type*, *Push Down Field* und *Push Down Method* bzw. *Pull Up Field* und *Pull Up Method*. Die Refactorings *Extract Class* und *Inline Class* verwenden *Move Field*, *Move Method* und `Create Type` bzw. *Delete Type*. *Extract Interface* verwendet *Create Type* und *Create Method*. *(Self) Encapsulate Field* und *Extract Method* benutzen ebenfalls *Create Method*. *Inline Method* macht Gebrauch von `Delete Method`.
Da sich die Refactorings dieser Gruppe aus den zuvor beschriebenen Refactorings zusammensetzen und diese wiederum die atomaren Refactorings verwenden, hat der Aspekt-Code einer ObjectTeams/Java Anwendung bei diesen Refactorings die gleichen Auswirkungen, die bereits in den jeweiligen Refactorings beschrieben worden sind.

4.4.2.1 (Self) Encapsulate Field

Beschreibung der Änderungen
Erzeugt Getter- und Setter-Methoden für ein Feld, ersetzt alle Referenzen auf das Feld durch Aufrufe dieser Methoden und versteckt das Feld.

Herkunft
Aus [[7], S. 171], [[22][7], S. 52] und [3].

[7] Anmerkung: Opdyke nennt dieses Refactoring *Abstract Access To Member Variable*.

Parameter

- Das Feld, dass gekapselt werden soll.
- Der Name der Getter-Methode.
- Der Name der Setter-Methode.

Einfluss von ObjectTeams/Java Code
Die neu erzeugten Methoden können von Methoden in einer Rollenklasse überschrieben werden und diese können wiederum in weiteren expliziten und impliziten Sub-Rollen redefiniert werden (siehe 2.2.5). Die Signatur einer Methode in einer Rollenklasse kann unter Umständen auch zuweisungskompatibel zu der Signatur einer der neuen Methoden sein. Aufgrund eventuell vorhandener Methoden-Bindungen (siehe 2.2.9) in Rollenklassen kann es auch zu Problemen bezüglich *Overloading* kommen. Das Feld kann ausserdem auf der rechten Seite einer Callout To Field-Bindung (siehe 2.2.9.1) explizit referenziert werden. Somit werden bei diesem Refactoring ein Teil der Sprachanforderungen, und zwar bezüglich *Naming*, sowie *Vererbungsbeziehungen* vom Aspekt-Code beeinflusst.

Vorbedingungen

1. *Das Feld ist in einer regulären OO-Klasse definiert.*
2. Das Feld ist nicht `final`.
3. Es existiert keine Methode in der Klasse, die das Feld definiert, einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*) dieser Klasse, deren Signatur zuweisungskompatibel zu der Signatur der neuen Getter-Methode ist.
4. Es existiert keine Methode in der Klasse, die das Feld definiert, einem Super-Typ oder einem Sub-Typ (*inklusive Rollenklassen*) dieser Klasse, deren Signatur zuweisungskompatibel zu der Signatur der neuen Setter-Methode ist.
5. *Es existiert keine Methode in der Klasse, die das Feld definiert oder einem Super-Typ oder einem Sub-Typ dieser Klasse, die in einer Methoden-Bindung ohne vollständige Signatur referenziert wird und die von einer der neuen Methoden überladen wird (OTI5.).*

Probleme
Eine möglicherweise existierende Referenz auf das Feld in einer Callout To Field-Bindung (siehe 2.2.9.1) darf nicht durch die neue Getter- oder Setter-Methode ersetzt werden, da dies zu einem Compiler-Fehler führen würde.

Darüber hinaus würde dies auch keinen Sinn machen, da durch eine solche Callout To Field-Bindung eine Getter- oder Setter-Methode für das referenzierte Feld in einer Rollenklasse generiert wird.

Nachbedingungen

- Das Feld ist `private`.
- Alle Referenzen auf das Feld, *ausser die Referenzen in Callout To Field-Bindungen (siehe vorherigen Abschnitt)*, werden durch Aufrufe der Getter- und Setter-Methoden ersetzt.

4.4.2.2 Extract Method

Beschreibung der Änderungen
Extrahiert ein Code-Fragment (eine selektierte Expression oder Block-Anweisungen in einer Methode, Konstruktor oder Initializer) in eine neue, private Methode in derselben Klasse.

Herkunft
Aus [[7], S. 110], [[22][8], S. 53] und [3].

Parameter

- Die Block-Anweisungen oder die Expression, die extrahiert werden sollen bzw. soll.
- Der Name der neuen Methode.

Einfluss von ObjectTeams/Java Code
Die Methoden-Bindungen in Rollenklassen können auch Methoden einer Basisklasse referenzieren, die normalerweise nicht sichtbar sind (siehe 2.2.9.1 und 2.2.9.4). Wird die private Methode in einer Basisklasse (reguläre OO-Klasse) oder in einer ihrer Super-Klassen extrahiert, dann ist diese Methode in den Methoden-Bindungen der Rollenklassen, die an diese Basisklasse gebunden sind, sichtbar. Dies kann zu Problemen in den Methoden-Bindungen führen, wenn die neue, extrahierte Methode eine bereits existierende Methode überlädt. Der Aspekt-Code hat daher bei diesem Refactoring Auswirkungen auf einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*, und auf *Vererbungsbeziehungen*.

Vorbedingungen

1. *Der umschliessende Typ des selektierten Code-Fragments ist eine reguläre OO-Klasse (diese kann lokal oder anonym sein).*

[8] Anmerkung: Opdyke nennt dieses Refactoring *Convert Code Segment To Function.*

2. Es existiert keine Methode in der Klasse, einem Super-Typ oder einer Sub-Klasse, die die gleiche Signatur hat wie die extrahierte Methode.

3. Höchstens eine lokale Variable, die außerhalb der Selektion definiert ist, wird innerhalb der Selektion modifiziert (schreibender Zugriff).

4. Alle lokalen Typen, die in der Selektion verwendet werden, sind entweder auch in der neuen Methode sichtbar, oder sie werden nur innerhalb der Selektion definiert und verwendet.

5. Alle lokalen Variablen, die innerhalb der Selektion definiert sind, werden nicht ausserhalb der Selektion verwendet.

6. *Es existiert keine Methode in der Klasse, einem Super-Typ oder einer Sub-Klasse, die in einer Methoden-Bindung ohne vollständige Signatur referenziert wird und die von der extrahierten Methode überladen wird (OTI5.).*

Nachbedingungen

- Eine neue, private Methode mit dem angegebenen Namen existiert in der gleichen Klasse, die die Selektion enthält.
- Die neue Methode hat einen Parameter für jede lokale Variable bzw. jeden Parameter, die bzw. der in der Original-Selektion referenziert aber nicht definiert ist.
- Die Selektion wird durch einen Aufruf der neuen Methode ersetzt.

4.4.2.3 Inline Method

Beschreibung der Änderungen
Ersetzt alle Aufrufe einer Methode durch ihren Methodenrumpf und entfernt die Methoden-Deklaration.

Herkunft
Aus [[7], S. 117], [[22][9], S. 49] und [3].

Parameter

- Die Methode, deren Aufrufe durch den Methodenrumpf ersetzt werden sollen.

[9] Anmerkung: Opdyke nennt dieses Refactoring *Inline Function Call*.

Einfluss von ObjectTeams/Java Code
Die Methode kann in Rollenklassen überschrieben werden. Ausserdem ist es möglich, dass die Methode in Methoden-Bindungen (siehe 2.2.9) referenziert wird. Folglich hat der Aspekt-Code bei diesem Refactoring Einfluss auf einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*, und auf *Vererbungsbeziehungen*.

Vorbedingungen

1. Die Methode ist nicht abstrakt.
2. Die Methode ist nicht polymorph, d.h. es gibt keine überschriebenen/überschreibenden Methoden in regulären Klassen. *Es gibt auch keine redefinierenden Methoden in Rollenklassen.*
3. *Die Methode wird nicht in einer Methoden-Bindung referenziert.*
4. Die Methode hat keinen Rückgabewert oder ihr Rumpf enthält nur eine Return-Anweisung mit einer Rückgabewert-Expression, die keine Seiten-Effekte hat bezüglich der Parameter der Methode.
5. Alle Features, die innerhalb der Methode referenziert werden, sind auch innerhalb des Elements sichtbar, dass den Methodenaufruf enthält.
6. Jedes Argument in allen Aufrufen der Methode ist frei von Seiten-Effekten.

Nachbedingungen

- Die Methode existiert nicht mehr.
- Falls die Methode einen Rückgabewert hat, dann sind alle ehemaligen Aufrufe der Methode (*inklusive der Aufrufe in Team- und Rollenklassen*) ersetzt durch die Rückgabewert-Expression.
- Falls die Methode keinen Rückgabewert hat, dann sind alle ehemaligen Expression-Anweisungen, die Aufrufe der Methode enthalten (*inklusive der Aufrufe in Team- und Rollenklassen*), ersetzt durch Block-Anweisungen, die den Rumpf der Methode enthalten.
- In den Rümpfen der aufrufenden Methoden sind alle Zugriffe auf Parameter der ehemaligen Methode ersetzt durch die entsprechenden Argumente der aktuellen Aufrufe.
- In den Rümpfen der aufrufenden Methoden sind alle Referenzen auf Features aktualisiert (falls notwendig).

4.4.2.4 Extract Subclass

Beschreibung der Änderungen
Erzeugt für eine gegebene Klasse eine neue Sub-Klasse und verschiebt alle ausgewählten Felder und Methoden in diese Klasse.

Herkunft
Aus [[7], S. 330] und [3].

Parameter

- Die Klasse, aus der die neue Sub-Klasse extrahiert werden soll.
- Der qualifizierte Name der neuen Klasse.
- Die Felder und Methoden, die verschoben werden sollen.
- Angabe, ob abstrakte Methoden in konkrete Methoden umgewandelt werden sollen (ansonsten wird die neue Klasse abstrakt sein).

Einfluss von ObjectTeams/Java Code
Es ist möglich, dass eine Teamklasse mit dem qualifizierten Namen der neuen Sub-Klasse bereits existiert. Darüber hinaus kann es eine oder mehrere Rollenklassen geben, die von der Klasse erben, für die eine neue Sub-Klasse erzeugt werden soll. Somit hat der Aspekt-Code bei diesem Refactoring Auswirkungen auf einen Teil der Sprachanforderungen, und zwar bezüglich *Naming*, sowie auf *Vererbungsbeziehungen*.

Vorbedingungen

1. *Die Klasse, aus der die neue Sub-Klasse extrahiert werden soll, ist eine reguläre OO-Klasse.*
2. Der qualifizierte Name ist gültig.
3. Es existiert kein Typ (*einschließlich Teamklassen*) oder Paket mit diesem qualifizierten Namen (JL2.).
4. Alle Typen, Felder und Methoden, die in den selektierten Feldern und Methoden verwendet werden, können in der neuen Sub-Klasse entweder direkt (gleiches Paket) oder qualifiziert (anderes Paket) referenziert werden.
5. Kein Feld und keine Methode in der gegebenen Klasse, das bzw. die nicht verschoben wird, referenziert ein Feld oder eine Methode, das bzw. die verschoben werden soll.

6. Keine Methode, die verschoben werden soll, ist eine Implementierung einer abstrakten Methode.

Nachbedingungen

- Es existiert eine Klasse mit dem angegebenen qualifizierten Namen.
- Alle ausgewählten Felder und Methoden befinden sich in der neuen Sub-Klasse.
- Die neue Klasse hat als Super-Klasse die gegebene Klasse.
- Alle Klassen (*inklusive Rollenklassen*), die zuvor von der gegebenen Klasse geerbt haben, erben nun stattdessen von der neuen Klasse.
- Die Klasse ist abstrakt, falls eine der verschobenen Methoden abstrakt ist.

4.4.2.5 Extract Superclass

Beschreibung der Änderungen
Erzeugt für eine gegebene Klasse eine neue Super-Klasse und verschiebt alle ausgewählten Felder und Methoden in diese Klasse.

Herkunft
Aus [[7], S. 336] und [3].

Parameter

- Die Klasse, aus der die neue Super-Klasse extrahiert werden soll.
- Der qualifizierte Name der neuen Klasse.
- Die Felder und Methoden, die verschoben werden sollen.

Einfluss von ObjectTeams/Java Code
Es kann eine Teamklasse (siehe 2.2.1) geben, die den gleichen qualifizierten Namen hat wie die neue Super-Klasse. Bei diesem Refactoring beeinflusst der Aspekt-Code folglich einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*.

Vorbedingungen

1. *Die Klasse, aus der die neue Super-Klasse extrahiert werden soll, ist eine reguläre OO-Klasse.*
2. Der qualifizierte Name ist gültig.

3. Es existiert kein Typ (*einschließlich Teamklassen*) oder Paket mit diesem qualifizierten Namen (JL2.).

4. Alle Typen, Felder und Methoden, die in den selektierten Feldern und Methoden verwendet werden, können in der neuen Super-Klasse entweder direkt (gleiches Paket) oder qualifiziert (anderes Paket) referenziert werden.

5. Die Felder und Methoden, die verschoben werden sollen, referenzieren nur Felder und Methoden, die entweder auch verschoben werden oder die in der neuen Klasse sichtbar sind.

Nachbedingungen

- Es existiert eine Klasse mit dem angegebenen qualifizierten Namen.
- Alle ausgewählten Felder und Methoden befinden sich in der neuen Super-Klasse.
- Die neue Klasse hat dieselben Super-Interfaces und dieselbe Super-Klasse wie die gegebene Klasse.
- Die neue Klasse ist die Super-Klasse der gegebenen Klasse.
- Die Klasse ist abstrakt, falls eine der verschobenen Methoden abstrakt ist.

4.4.2.6 Extract Interface

Beschreibung der Änderungen
Erzeugt für eine gegebene Klasse ein neues Interface mit abstrakten Methoden für alle ausgewählten Methoden.

Herkunft
Aus [[7], S. 341] und [3].

Parameter

- Die Klasse, aus der das Interface extrahiert werden soll.
- Der qualifizierte Name des neuen Interfaces.
- Die Methoden, für die abstrakte Methoden im neuen Interface erzeugt werden sollen.

Einfluss von ObjectTeams/Java Code
Eine vorhandene Teamklasse (siehe 2.2.1) kann den gleichen qualifizierten Namen haben wie das neue Interface. Der Aspekt-Code hat daher bei diesem Refactoring Einfluss auf einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*.

Vorbedingungen

1. *Die Klasse, aus der das neue Interface extrahiert werden soll, ist eine reguläre OO-Klasse.*
2. Der qualifizierte Name ist gültig.
3. Es existiert kein Typ (*einschließlich Teamklassen*) oder Paket mit diesem qualifizierten Namen (JL2.).
4. Keine der selektierten Methoden ist `private`.
5. Keine der selektierten Methoden ist eine Redefinition einer konkreten Methode.
6. Alle Typen, die in den Signaturen der ausgewählten Methoden verwendet werden, können in dem neuen Interface entweder direkt (gleiches Paket) oder qualifiziert (anderes Paket) referenziert werden.

Probleme
Die Aktualisierung von Typ-Deklarationen in Elementen wie Variablen, Feldern, Parametern und Rückgabe-Typen von Methoden bei diesem Refactoring erfordert eine umfangreiche Analyse. Der in diesen Elementen deklarierte Typ (die Klasse, aus der das Interface extrahiert werden soll) kann nämlich nicht in allen Fällen durch das neu erzeugte Interface ersetzt werden. Eine ausführliche Diskussion des Problems sowie ein Lösungsansatz ist in [29] zu finden.

Nachbedingungen

- Es existiert ein Interface mit dem angegebenen qualifizierten Namen.
- Die gegebene Klasse implementiert das neue Interface.
- Für jede ausgewählte Methode existiert eine abstrakte Methode im neuen Interface.
- Das neue Interface hat dieselben Super-Interfaces wie die gegebene Klasse.
- In Deklarationen von lokalen Variablen, Feldern, Parametern und Rückgabe-Typen von Methoden wird, wenn möglich, statt dem ursprünglich deklarierten Typ (die gegebene Klasse) nun das Interface verwendet (siehe vorherigen Abschnitt).

4.4.2.7 Extract Class

Beschreibung der Änderungen
Erzeugt eine neue Klasse und verschiebt die ausgewählten Felder und Methoden aus der gegebenen Klasse in die neue Klasse.

Herkunft
Aus [[7], S. 149] und [3].

Parameter

- Die Klasse, aus der die neue Klasse extrahiert wird.
- Der qualifizierte Name der neuen Klasse.
- Die Felder udn Methoden, die verschoben werden sollen.

Einfluss von ObjectTeams/Java Code
Es kann bereits eine Teamklasse (siehe 2.2.1) vorhanden sein, die den qualifizierten Namen der neuen Klasse hat. Folglich hat der Aspekt-Code bei diesem Refactoring Auswirkungen auf einen Teil der Sprachanforderungen, und zwar bezüglich *Naming*.

Vorbedingungen

1. *Die Klasse, aus der die neue Klasse extrahiert werden soll, ist eine reguläre OO-Klasse.*
2. Der qualifizierte Name ist gültig.
3. Es existiert kein Typ (*einschließlich Teamklassen*) oder Paket mit diesem qualifizierten Namen (JL2.).
4. Keine Methode, die verschoben werden soll, ist eine Implementierung einer abstrakten Methode.

Nachbedingungen

- Es existiert eine Klasse mit dem angegebenen qualifizierten Namen.
- Alle ausgewählten Felder und Methoden befinden sich in der neuen Klasse.
- Eine neues Feld existiert in der gegebenen Klasse, falls notwendig. Der Typ des Feldes ist die neue Klasse und sie wird mit einer Instanz der neuen Klasse initialisiert.

- Ein Feld existiert in der neuen Klasse, falls irgendein verschobenes Feld oder eine Methode ein anderes Feld oder eine andere Methode in der gegebenen Klasse verwendet, dass nicht verschoben wurde.
- Für alle verschobenen Methoden, die außerhalb der beiden Typen referenziert werden, existiert eine Methoden-Deklaration in der gegebenen Klasse, die wiederum die verschobene Methode aufruft. Der Zugriff erfolgt unter Verwendung des neuen Feldes.
- Alle Felder in der gegebenen Klasse, die nicht verschoben wurden und die von verschobenen Features referenziert werden, sind in der gegebenen Klasse gekapselt (es existieren Getter- und Setter-Methoden).
- Alle Methoden in der gegebenen Klasse, die nicht verschoben wurden und die von verschobenen Features referenziert werden, sind `protected`, falls die neue und die gegebene Klasse sich im selben Paket befinden oder `public`, falls die beiden Klassen sich in unterschiedlichen Paketen befinden.
- Alle Methoden in der neuen Klasse, die von Methoden in der gegebenen Klasse referenziert werden, sind `protected`, falls die neue und die gegebene Klasse sich im selben Paket befinden oder `public`, falls die beiden Klassen sich in unterschiedlichen Paketen befinden.

4.4.2.8 Inline Class

Beschreibung der Änderungen
Verschiebt alle Felder und Methoden aus einer Klasse in eine andere Klasse und entfernt die Klasse anschließend.

Herkunft
Aus [[7], S. 154] und [3].

Parameter

- Die Ziel-Klasse, in die die Felder und Methoden aus der anderen Klasse verschoben werden sollen.

Einfluss von ObjectTeams/Java Code
Eine Rollenklasse kann von der zu entfernenden Klasse erben. Darüber hinaus können Referenzen auf die Klasse, die entfernt werden soll, an verschiedenen Stellen in Team- und Rollenklassen vorkommen, beispielsweise in Import-Deklarationen einer Teamklasse, in der `extends`-Relation einer Rollenklasse, in einer Lifting-Methode (*Declared Lifting*, siehe 2.2.1), in einer Typ-Bindung (`playedBy`-Relation, siehe 2.2.3) und in einer Callout- oder

Callin-Bindung (siehe 2.2.9.1 und 2.2.9.4) mit vollständig angegebenen Signaturen. Entsprechend hat bei diesem Refactoring der Aspekt-Code Einfluss auf einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*, und auf *Vererbungsbeziehungen*.

Vorbedingungen

1. *Die Klasse, die entfernt werden soll, ist eine reguläre OO-Klasse.*
2. *Die Klasse, in die die Felder und Methoden verschoben werden sollen, ist eine reguläre OO-Klasse.*
3. Die Klasse, die entfernt werden soll, ist nicht abstrakt.
4. Die direkte Super-Klasse der Klasse, die entfernt werden soll, ist auch eine Super-Klasse der Ziel-Klasse.
5. Die eine Klasse verwendet jeweils keine Felder und Methoden der anderen Klasse und umgekehrt.

Nachbedingungen

- Alle Felder und Methoden der Original-Klasse befinden sich in der Ziel-Klasse.
- Es existiert kein Typ mit dem qualifizierten Namen der Original-Klasse.
- Alle Super-Interfaces der Original-Klasse sind nun Super-Interfaces der Ziel-Klasse.
- Alle Sub-Klassen (*inklusive Rollenklassen*) der Original-Klasse sind nun Sub-Klassen der Ziel-Klasse.
- Alle Referenzen auf die Original-Klasse sind durch Referenzen auf die Ziel-Klasse ersetzt. *Dies beinhaltet auch sämtliche Referenzen auf die Original-Klasse in Team- und Rollenklassen (z.B. in Lifting-Methoden, Typ-Bindungen, Callout- und Callin-Bindungen usw.).*

4.4.2.9 Collapse Hierarchy

Beschreibung der Änderungen
Verschiebt alle Features (Felder und Methoden) aus einer Sub-Klasse in die Super-Klasse (oder umgekehrt) und entfernt die leere Klasse anschließend.

Herkunft
Aus [[7], S. 344] und [3].

Parameter

- Die Super- oder Sub-Klasse, die entfernt werden soll.

Einfluss von ObjectTeams/Java Code
Es ist möglich, dass eine Rollenklasse von der zu entfernenden Klasse erbt. Andererseits können Referenzen auf die Klasse, die entfernt werden soll, an verschiedenen Stellen in Team- und Rollenklassen vorkommen, beispielsweise in Import-Deklarationen einer Teamklasse, in der `extends`-Relation einer Rollenklasse, in einer Lifting-Methode (*Declared Lifting*, siehe 2.2.1), in einer Typ-Bindung (`playedBy`-Relation, siehe 2.2.3) und in einer Callout- oder Callin-Bindung (siehe 2.2.9.1 und 2.2.9.4) mit vollständig angegebenen Signaturen. Bei diesem Refactoring beeinflusst der Aspekt-Code daher einen Teil der Sprachanforderungen, und zwar hinsichtlich *Naming*, und die *Vererbungsbeziehungen*.

Vorbedingungen

1. *Die Super- oder Sub-Klasse, die entfernt werden soll, ist eine reguläre OO-Klasse.*
2. *Werden die Features in die Sub-Klasse verschoben, dann ist die Sub-Klasse eine reguläre OO-Klasse.*
3. Die Klasse, die entfernt werden soll, ist nicht abstrakt.
4. Wird die Super-Klasse entfernt, dann ist die direkte Super-Klasse dieser Klasse auch eine Super-Klasse der Ziel-Klasse.
5. Die eine Klasse verwendet jeweils keine Felder der anderen Klasse und umgekehrt.

Nachbedingungen

- Alle Features der Original-Klasse befinden sich in der Ziel-Klasse.
- Es existiert kein Typ mit dem qualifizierten Namen der Original-Klasse.
- Alle Super-Interfaces der Original-Klasse sind nun Super-Interfaces der Ziel-Klasse.
- Alle Sub-Klassen (*inklusive Rollenklassen*) der Original-Klasse sind nun Sub-Klassen der Ziel-Klasse.
- Alle Referenzen auf die Original-Klasse sind durch Referenzen auf die Ziel-Klasse ersetzt. *Dies beinhaltet auch sämtliche Referenzen auf die Original-Klasse in Team- und Rollenklassen (z.B. in Lifting-Methoden, Typ-Bindungen, Callout- und Callin-Bindungen usw.).*

4.5 Weitere Refactorings

Es existieren noch zahlreiche andere Refactorings, die in den meisten Fällen ähnliche Änderungen durchführen und dieselben atomaren Refactorings verwenden wie die zuvor beschriebenen zusammengesetzten Refactorings. Die Auswirkungen des Aspekt-Codes eines ObjectTeams/Java Programms auf diese Refactorings sind die gleichen, die bereits in den anderen Refactorings beschrieben wurden.
Preserve Whole Object (siehe [7], S. 288) und *Replace Parameter with Method* (siehe [7], S. 292) sind Refactorings, die die Signatur einer Methode verändern. Sie gehören somit zu den Refactorings, die in Abschnitt 4.3.3 definiert sind.
Preserve Whole Object ersetzt die Parameter einer Methode, die bestimmte Werte eines Objekts repräsentieren, durch das gesamte Objekt.
Replace Parameter with Method entfernt den Parameter einer Methode, der als Argument ein Ergebnis eines Methodenaufrufs übergeben bekommt. Die Methode wird stattdessen vom Receiver aufgerufen.
Diese Refactorings verwenden das atomare Refactoring *Add Parameter* (siehe 4.3.3.4) bzw. *Remove Parameter* (siehe 4.3.3.5).
Somit hat der Aspekt-Code auf die erwähnten Refactorings dieselben Auswirkungen wie auf die beiden atomaren Refactorings. Für die genannten Refactorings können also vergleichbare Vorbedingungen aufgestellt werden wie für die jeweiligen atomaren Refactorings.
Die Refactorings *Separate Query from Modifier* (siehe [7], S. 279), *Parameterize Method* (siehe [7], S. 283) und *Replace Parameter with Explicit Methods* (siehe [7], S. 285) erzeugen alle neue Methoden. Sie können daher in die Gruppe der Refactorings, die in Abschnitt 4.4.2 definiert sind, eingeordnet werden.
Separate Query from Modifier erzeugt aus einer Methode, die einen Wert zurückliefert und gleichzeitig den Zustand eines Objekts verändert (Methode mit Seiten-Effekten), zwei Methoden: Eine, die den Wert liefert, und eine für die Modifikation.
Parameterize Method ersetzt mehrere Methoden, die Ähnliches tun aber auf unterschiedlichen Werten arbeiten, durch eine einzige Methode mit einem Parameter für die verschiedenen Werte.
Replace Parameter with Explicit Methods ist die Umkehrung von *Parameterize Method*. Dieses Refactoring erzeugt für jeden Wert eines Parameters eine separate Methode.
Alle diese Refactorings verwenden die beiden atomaren Refactorings *Create Method* (siehe 4.3.1.3) und *Delete Method* (siehe 4.3.2.3), um die neuen Methoden zu erzeugen bzw. die alten zu entfernen.
Daher hat der Aspekt-Code auf die genannten Refactorings denselben Einfluss wie auf *Create Method* und *Delete Method*. Die Vorbedingungen der erwähnten Refactorings setzen sich somit unter anderem aus den Vorbedin-

gungen dieser atomaren Refactorings zusammen.
Zu den Refactorings, die neue Klassen erzeugen, gehören unter anderem auch *Replace Method with Method Object* (siehe [7], S. 135), *Replace Array with Object* (siehe [7], S. 186) und *Introduce Parameter Object* (siehe [7], S. 295). Diese Refactorings können somit ebenfalls in die Kategorie der in Abschnitt 4.4.2 definierten Refactorings eingeteilt werden.
Replace Method with Method Object wandelt eine Methode in ein Methoden-Objekt um, indem es für die Methode eine neue Klasse erstellt und alle lokalen Variablen und Parameter der Methode als Felder der neuen Klasse deklariert sowie einen Konstruktor und zusätzlich benötigte Methoden erzeugt.
Replace Array with Object ersetzt ein Array, dass aus unterschiedlichen Elementen besteht, durch ein Objekt, indem es eine Klasse erzeugt, die ein Feld für jedes Element in dem Array enthält.
Introduce Parameter Object ersetzt eine Gruppe von zusammengehörenden Parametern in Methoden durch ein Objekt. Dazu wird eine Klasse erzeugt, die die Gruppe von Parametern repräsentiert. Anschließend werden die Parameter der betroffenen Methoden durch das neue Objekt ersetzt.
Diese Refactorings verwenden ebenfalls atomare Refactorings, und zwar *Create Type* (siehe 4.3.1.1), *Create Field* (siehe 4.3.1.2) und *Create Method* (siehe 4.3.1.3). Das Refactoring *Introduce Parameter Object* verwendet zusätzlich noch *Add Parameter* und *Remove Parameter*.
Folglich hat der Aspekt-Code auf die erwähnten Refactorings die gleichen Auswirkungen wie auf diese atomaren Refactorings. Die Vorbedingungen der genannten Refactorings bestehen daher unter anderem aus den Vorbedingungen dieser atomaren Refactorings.

Auf Refactorings, die lokale Variablen oder Anweisungen in Methodenrümpfen (z.B. *Split Temporary Variable*[7], S. 128, *Remove Control Flag*[7], S. 245 und *Replace Nested Conditional With Guard Clauses*[7], S. 250) umstrukturieren, hat der Aspekt-Code einer ObjectTeams/Java Anwendung keine Auswirkungen, da diese Elemente nicht außerhalb der deklarierenden Klasse referenziert werden können.

Kapitel 5

Implementierung von adaptierten Refactorings in Eclipse

In diesem Kapitel wird die Implementierung dreier ausgewählter adaptierter Refactorings aus Kapitel 4 in der integrierten Entwicklungsumgebung Eclipse[4] beschrieben.
Im ersten Teil wird das Object Teams Development Tooling (OTDT) beschrieben, welches die aspektorientierte Sprache ObjectTeams/Java (OT/J) in Eclipse integriert und die notwendigen Voraussetzungen für die Durchführung und Erweiterung existierender Refactorings bereitstellt. Es folgt ein kurzer Überblick über die interne Programmrepräsentation in Eclipse anhand syntaktischer Syntaxbäume und die Erweiterung dieser um die neuen Sprachkonstrukte in ObjectTeams/Java.
Im zweiten Teil wird der vollständige Ablauf eines Refactorings in Eclipse beschrieben. Es wird auf die zentralen Methoden in den Refactoring-Implementierungen eingegangen, die die Vorbedingungen jedes Refactorings überprüfen. Anschließend werden die zwei unterschiedlichen Implementierungsarten von Refactorings in Eclipse behandelt. Darüber hinaus wird erläutert, welche Funktion ein Refactoring-Wizard hat und welche Datenstrukturen für die Transformationsbeschreibungen in Eclipse verwendet werden.
Im letzten Teil des Kapitels werden schließlich die Implementierungen der ausgewählten Refactorings ausführlich beschrieben.

5.1 Object Teams Development Tooling (OTDT)

Im Rahmen des Projekts Topprax[24] wird zur Zeit eine umfassende und integrierte Werkzeugunterstützung für ObjectTeams/Java entwickelt. Das

Object Teams Development Tooling (OTDT) stellt eine Erweiterung für die Eclipse Plattform dar. Es baut auf dem Java Development Tooling (JDT) auf und bettet die aspektorientierte Sprache ObjectTeams/Java in die integrierte Entwicklungsumgebung ein, indem es diverse Entwicklungswerkzeuge zur Verfügung stellt. Diese umfassen einen inkrementellen Compiler[30], einen Quell-Code Editor und verschiedene strukturelle Ansichten.
Im Rahmen der Entwicklung des OTDT sind, unter anderem, die internen Datenstrukturen von Eclipse, welche im nächsten Abschnitt beschrieben sind, um die neuen ObjectTeams/Java-spezifischen Sprachelemente erweitert und entsprechend angepasst worden.

5.1.1 Voraussetzungen für die Durchführung und Erweiterung von Refactorings in Eclipse

5.1.1.1 Interne Programmrepräsentation und Datenstrukturen in Eclipse

Damit ein Programm während eines Refactorings statisch analysiert und manipuliert werden kann, muss eine geeignete Struktur bzw. Repräsentation des Quell-Codes vorhanden sein. Diese Datenstruktur wird im Allgemeinen durch sogenannte *abstrakte Syntaxbäume* (AST) zur Verfügung gestellt.
Abstrakte Syntaxbäume reflektieren die logische Struktur des Quell-Codes. Jedes Element im Quell-Code wird repräsentiert durch einen Knoten im Baum. Quell-Code-Elemente sind zum Beispiel Klassen, deren Methoden und Felder und auch Anweisungen und Ausdrücke. Der Baum wird konstruiert, indem eine Assoziation zwischen zwei Knoten erzeugt wird, falls sich das Quell-Code-Element des einen Knotens innerhalb des Quell-Code-Elements des anderen Knotens befindet. Zum Beispiel enthält ein Knoten für eine Methoden-Deklaration Subknoten für jede lokale Variable und jeden Ausdruck, die im Rumpf dieser Methode deklariert sind.
In Eclipse wird jede Java Quell-Code-Datei (Compilation Unit) intern durch einen solchen abstrakten Syntaxbaum repräsentiert. Dabei erzeugt ein Parser aus einem String, der Java Quell-Code enthält, den entsprechenden AST. Es gibt zwei verschiedene Arten von ASTs in Eclipse: Einen internen *Compiler-AST*, der beim Kompilieren aufgebaut wird, und einen öffentlichen *DOM-AST*, der aus dem Compiler-AST anhand eines Konvertierers (Klasse `ASTConverter`) erzeugt wird. Dieser DOM-AST wird in Eclipse unter anderem für die statische Analyse während eines Refactorings benötigt.

5.1.1.2 Erweiterung des abstrakten Syntaxbaums (DOM-AST)

Um bei der Implementierung der adaptierten Refactorings die neuen ObjectTeams/Java Sprachelemente (siehe Kapitel 2) berücksichtigen zu können

wurden diverse Klassen wie `AST, ASTNode, ASTConverter, ASTMatcher`, u.a. um die neuen Sprachkonstrukte erweitert bzw. eine Unterstützung für diese Sprachelemente bereitgestellt. Für jedes Sprachelement (z.B. Rollenklasse, Callout- oder Callin-Bindung oder Parameter Mapping) wurde eine eigene Klasse angelegt, um das Element als Knoten in einem abstrakten Syntaxbaum repräsentieren zu können.
Um einen solchen Knoten im abstrakten Syntaxbaum zu erzeugen, wurde die Klasse `AST` um neue Factory Methoden ergänzt. Für jedes neue Element existiert eine Factory Methode, die bei einem Aufruf den entsprechenden Knoten im aktuellen AST erzeugt.
Damit die neuen Elemente im aufgebauten AST traversiert werden können, musste die Visitor-Klasse `ASTVisitor` um zusätzliche `visit` und `endVisit` Methoden ergänzt werden. Für jeden neuen Knotentyp existiert ein solches Methodenpaar.
Die Klassen `ASTRewrite` und `ASTRewriteAnalyzer`, die die Infrastruktur für Code-Modifikationen zur Verfügung stellen, wurden ebenfalls angepasst, um die Änderungen im Code, welche die neuen Knoten beinhalten können, beschreiben zu können.
Die genannten Änderungen und die Klassen, die dadurch betroffen sind, stellen lediglich einen kleinen Ausschnitt der durchgeführten Erweiterungen während der Entwicklung des OTDT dar. Die detaillierte Beschreibung sämtlicher Änderungen würde über den Rahmen dieser Arbeit hinausgehen und ist für die weiteren Betrachtungen und Ausführungen nicht erforderlich.

5.2 Refactoring in Eclipse

In der integrierten Entwicklungsumgebung Eclipse, deren Refactoring-Unterstützung im Rahmen des Topprax-Projekts[24] vollständig für ObjectTeams/-Java adaptiert werden soll, sind viele der wichtigsten und am häufigsten verwendeten Refactorings implementiert. Eine vollständige Übersicht aller in Eclipse in der Version 3.0 existierenden Refactorings ist in Tabelle A.1 in Anhang A angegeben.
Im Folgenden wird zunächst der Lebenszyklus eines Refactorings in Eclipse beschrieben sowie zentrale Methoden, die während jedes Refactorings ausgeführt werden. Anschließend werden die beiden Implementierungsarten der Refactorings in Eclipse verglichen. Außerdem werden die graphische Benutzerschnittstelle (GUI) eines Refactorings und die Datenstrukturen, die von Eclipse für die Code-Transformationen verwendet werden, beschrieben.

5.2.1 Zentrale Methoden und Lebenszyklus eines Refactorings

Jedes Refactoring in Eclipse bietet zwei unterschiedliche Arten von Methoden an:

- Methoden, die Bedingungen überprüfen um festzustellen, ob ein Refactoring überhaupt ausgeführt werden kann und ob die Transformation verhaltenserhaltend sein wird (`checkInitialConditions` und `checkFinalConditions`).
- Eine Methode, die ein Objekt erzeugt, welches die tatsächlichen Modifikationen im Workspace darstellt (`createChange`).

Der Lebenszyklus eines Refactorings sieht folgendermaßen aus (siehe Abbildung A.1 in Anhang A):
Zunächst wird das Refactoring erzeugt. Anschließend wird das Refactoring mit den Elementen, die umstrukturiert werden sollen, initialisiert. Danach wird die Methode `checkInitialConditions` aufgerufen, die einige Vorbedingungen basierend auf dem umzustrukturierenden Element überprüft und feststellt, ob das Refactoring überhaupt anwendbar ist (zum Beispiel, ob die betroffenen Dateien modifizierbar sind). Daraufhin werden vom Entwickler zusätzliche Argumente zur Verfügung gestellt um das Refactoring durchzuführen (zum Beispiel der neue Name eines Elements im Falle eines *Rename Refactorings*). Nun wird die Methode `checkFinalConditions` aufgerufen, die die übriggebliebenen Vorbedingungen überprüft und ob die Transformationen das Programmverhalten erhalten werden und durch das Refactoring keine Fehler eingeführt werden (beispielsweise könnte im Falle von *Rename Method* eine Methode mit dem vom Entwickler angegebenen neuen Namen in der betroffenen Klasse bereits existieren). Zuletzt wird die Methode `createChange` ausgeführt, welche ein `Change`-Objekt erzeugt, dass sämtliche Transformationen im Workspace darstellt (siehe auch 5.2.4).
Alle genannten Methoden liefern als Rückgabewert ein Objekt vom Typ `RefactoringStatus`. Ein `RefactoringStatus`-Objekt repräsentiert das Ergebnis einer Bedingungsüberprüfung. Es verwaltet unter anderem den Schweregrad eines Problems.
Liefert die Methode `checkInitialConditions` nach ihrer Ausführung einen Status mit der Schwere `RefactoringStatus.FATAL` zurück, dann ist das Refactoring nicht durchführbar. Die Methode `checkFinalConditions` darf in diesem Fall nicht mehr aufgerufen werden. Das gleiche gilt, wenn die Methode `checkFinalConditions` einen Status mit der Schwere `RefactoringStatus.FATAL` zurückliefert. Die Methode `createChange` darf nicht aufgerufen werden, falls eine der beiden erwähnten Methoden einen RefactoringStatus der Schwere `RefactoringStatus.FATAL` zurückgibt.

Wird hingegen in allen Methoden der Status `RefactoringStatus.OK` zurückgeliefert, dann ist kein Problem aufgetreten und das Refactoring ist durchführbar.
Es existieren darüber hinaus noch weitere Schweregrade für ein Problem. Liegt nach der Überprüfung der Vorbedingungen eines Refactorings ein Problem mit dem Schweregrad `RefactoringStatus.WARNING` vor, dann ist es dem Entwickler überlassen, ob er das Refactoring trotzdem durchführen möchte oder nicht. Ein Problem mit der Schwere `RefactoringStatus.WARNING` tritt zum Beispiel auf, wenn nach der Umbenennung einer Methode diese Methode eine andere Methode in einer Super-Klasse überschreibt oder von einer Methode in einer Sub-Klasse überschrieben wird.

5.2.2 Implementierungsarten

Die bestehenden Refactorings in Eclipse sind auf zwei unterschiedliche Arten implementiert und zwar entweder *prozessorbasiert* oder *nicht-prozessorbasiert*.

5.2.2.1 Prozessorbasierte Refactorings

Prozessorbasierte Refactorings sind Refactorings, die eine spezielle Prozessor/Participant-Architektur verwenden. Sie sind aufgeteilt in einen Prozessor und 0 bis n Participants.
Ein Refactoring-Prozessor ist erstens verantwortlich für die Umstrukturierung des eigentlichen Elements. Wird zum Beispiel ein Rename Method Refactoring ausgeführt, dann stellt der assoziierte Prozessor die Überprüfung der Vorbedingungen für das Umbenennen einer Methode bereit und erzeugt das `Change`-Objekt, welches die Modifikationen im Workspace beschreibt.
Zweitens ist ein Refactoring-Prozessor für das Laden aller Participants verantwortlich, die an dem Refactoring teilnehmen wollen. Ein *Rename Method Processor* muss beispielsweise alle Participants laden, die sich an der Umbenennung einer Methode beteiligen wollen.
Ein Refactoring Participant nimmt, wie bereits erklärt wurde, an der Überprüfung der Vorbedingungen und der Erzeugung der Änderungen eines Prozessors teil. Wichtig dabei ist, dass die von einem Participant generierten Änderungen nicht mit Änderungen die von anderen Participants oder dem Refactoring(-prozessor) selbst bereitgestellt werden, in Konflikt geraten. Um dies sicherzustellen darf ein Participant nur Resourcen manipulieren, die zu seinem Bereich gehören. Zum Beispiel darf ein *Rename Method Participant*, der den Breakpoint einer Methode aktualisiert, nur diesen Breakpoint aktualisieren und nicht irgendwelche Java Resourcen oder Resourcen, die nicht zu seinem Bereich gehören.

Jede konkrete Prozessor-Klasse implementiert die bereits bekannten Methoden `checkInitialConditions, checkFinalConditions` und `createChange` aus der abstrakten Super-Klasse `RefactoringProcessor`. Die Participant-Klassen wiederum implementieren die Methoden `checkConditions` und `createChange` aus der abstrakten Super-Klasse `RefactoringParticipant`.
Refactorings, die eine Prozessor/Participant Architektur verwenden, sind zum Beispiel sämtliche *Rename Refactorings* (ausser *Rename Temp*).
Die abstrakte Basis-Implementierung für diese Refactorings ist die Klasse `ProcessorBasedRefactoring`.
Die Prozessor/Participant-Architektur ist exemplarisch in Abbildung A.2 in Anhang A dargestellt.

5.2.2.2 Nicht-prozessorbasierte Refactorings

Nicht-prozessorbasierte Refactorings verwenden keine spezielle Prozessor/-Participant-Architektur. Diese Refactorings erben direkt von der abstrakten Super-Klasse `Refactoring`, und implementieren die bereits beschriebenen Methoden `checkInitialConditions, checkFinalConditions` und `createChange` um sämtliches Verhalten zu realisieren. Einige Beispiele für solche Refactorings sind *Extract Method*, *Pull Up Field* und *Push Down Field*.

5.2.3 Wizard: Die graphische Benutzerschnittstelle eines Refactorings

Jedes Refactoring in Eclipse bietet nach der Überprüfung der initialen Vorbedingungen dem Entwickler einen Dialog an, in den er die gewünschten Parameter, zum Beispiel den neuen Namen einer Methode im Falle eines *Rename Method Refactorings*, eintragen kann bzw. muss. Dieser Dialog wird auch als *Wizard* bezeichnet.
Ein Wizard besteht aus einer oder mehreren *Pages*. Eine Page wiederum setzt sich aus unterschiedlichen graphischen Elementen zusammen wie Checkboxen, Schaltknöpfen und Textfeldern, die der Entwickler benutzt, um sämtliche Angaben zu machen, die für die Durchführung eines Refactorings benötigt werden. Für jedes Refactoring existiert ein eigener Wizard, da jedes Refactoring andere Parameter verlangt und jeweils auf unterschiedlichen Daten operiert.
Die graphischen Komponenten eines Wizards mussten für die Implementierung der adaptierten Refactorings nicht verändert oder erweitert werden, da keine zusätzlichen Informationen bzw. Auswahlmöglichkeiten, neben den bereits angebotenen, für die Ausführung der Refactorings erforderlich sind.

5.2.4 Datenstrukturen für Code-Transformationen

Aufgrund der neuen Sprachelemente und Mechanismen in ObjectTeams/Java (siehe 2) können sich Code-Elemente, die von einem Refactoring verändert werden müssen, auch in Team- und Rollenklassen eines ObjectTeams/Java Programms befinden. Die Sammlung aller Änderungen im Workspace, die in der Methode `createChange` des jeweiligen Refactorings durchgeführt wird, muss daher für einige Refactorings (unter anderem auch für *Rename Virtual Method*, siehe 5.3.4.1) erweitert werden.
Die eigentliche Transformation des Codes, dass bedeutet, die Durchführung der beschriebenen Änderungen, wird durch die Methode `executeChange` in der Klasse `PerformChangeOperation` realisiert.
Alle Änderungen innerhalb einer Compilation Unit werden durch ein `Change`- bzw. `TextChange`-Objekt repräsentiert. Das `Change`-Objekt wiederum enthält sogenannte *TextEdits*. Ein `TextEdit` beschreibt jeweils eine elementare Textmanipulationsoperation. Während eines *Rename Refactorings* müssen zum Beispiel alle Textstellen, die auf das Element verweisen, das umbenannt werden soll, durch den neuen Namen ersetzt werden. Jede textuelle Änderung wird durch ein solches `TextEdit`-Objekt repräsentiert.
In einigen Refactorings müssen neue AST-Knoten erzeugt oder bestehende Knoten in einem AST entfernt oder verändert werden. Die Klasse `ASTRewrite` beschreibt Modifikationen an Knoten und übersetzt diese Beschreibungen in *TextEdits*, die dann auf den Original-Code angewendet werden können.
Die Klassen, die die erwähnten Datenstrukturen realisieren, mussten für die Implementierung der nachfolgend beschriebenen Refactorings nicht angepasst werden, da sämtliche Code-Manipulationen auf Textbasis durchgeführt werden. Das bedeutet, dass auch im Aspekt-Code (Team- und Rollenklassen), falls dieser durch das Refactoring betroffen ist, die erforderlichen Änderungen bereits ausgeführt werden.

5.3 Implementierte Refactorings (OT/J-aware Refactorings)

Im Folgenden werden die Implementierungen dreier erweiterter Refactorings in Eclipse beschrieben. Diese exemplarischen Implementierungen basieren auf den entsprechenden Definitionen der konzeptuell erweiterten Refactorings in Kapitel 4.
Die getroffene Auswahl umfasst das *Extract Method Refactoring* (siehe 4.4.2.2), bei dem keine bzw. nur geringe Erweiterungen notwendig sind und zwei Refactorings bei denen der Umfang der Erweiterungen wesentlich grösser ausfällt, und zwar *Move Instance Method* (siehe 4.4.1.3) und *Rename Method* (siehe 4.3.3.3).

5.3.1 Initiale Vorbedingung aller Refactorings

Die initiale Vorbedingung, die für jedes erweiterte Refactoring gelten muss, ist, dass das selektierte Code-Fragment, auf dem das entsprechende Refactoring ausgeführt werden soll, Teil einer regulären OO-Klasse sein muss (siehe Vorbedingungen der Refactorings in Kapitel 4). Diese Vorbedingung muss für jedes Refactoring in der Methode `checkInitialConditions` als erstes abgeprüft werden. Handelt es sich bei dem selektierten Element bzw. Code-Fragment nicht um ein Basis-Element bzw. um einen Teil einer regulären Java-Klasse, sondern um ein AO-Konstrukt (z.B. Teamklasse, Rollenklasse oder Rollenmethode), dann ist das Refactoring grundsätzlich nicht durchführbar.

5.3.2 Extract Method

Beim *Extract Method Refactoring* handelt es sich um ein nicht-prozessorbasiertes Refactoring. Es ist auf Seite 77 definiert. Die komplette Erweiterungsstruktur für dieses Refactoring ist in Abbildung A.3 in Anhang A dargestellt. Die Adaptierung ist in der Klasse `OTExtractMethodRefactoring` implementiert. Diese Klasse erbt von der abstrakten Super-Klasse `Refactoring` und ist als Wrapper für die existierende Klasse `ExtractMethodRefactoring` realisiert.
Im Rahmen der Erweiterung sind darüber hinaus die Klassen `OTExtractMethodAction, OTExtractMethodWizard` und `OTExtractMethodInputPage` implementiert worden.
In der Klasse `OTExtractMethodAction`, die vom Eclipse-Framework aufgerufen wird wenn eine Methode extrahiert werden soll, wird das Refactoring `OTExtractMethodRefactoring` erzeugt. Darüber hinaus wird dort der verwendete Wizard (`OTExtractMethodWizard`) mit der zugehörigen Page (`OTExtractMethodInputPage`) erstellt.

Überprüfung der Vorbedingungen
Die Vorbedingungen Nr. 2 bis 5 des *Extract Method Refactorings* (siehe 4.4.2.2) werden bereits von Eclipse unter Verwendung spezieller Hilfsklassen (u.a. `ExtractMethodAnalyzer, InOutFlowAnalyzer, InputFlowAnalyzer` und `Checks`) überprüft. Eine weitere triviale Vorbedingung, die von Eclipse kontrolliert wird, ist, dass die Klasse modifizierbar sein muss (Datei ist nicht *read-only* oder binär).
Die Vorbedingungen Nr. 1 und Nr. 6 stellen neue, ObjectTeams/Java-spezifische Vorbedingungen dar.
Bei der Vorbedingung Nr. 1 handelt es sich um die initiale Vorbedingung, die von allen Refactorings erfüllt sein muss (siehe 5.3.1). Diese wird in der Methode `checkInitialConditions` in der Klasse `OTExtractMethodRefactoring` als erstes überprüft.

Die Vorbedingung Nr. 6 wurde in der initialen Implementierung des erweiterten Refactorings nicht berücksichtigt. Die Überprüfung dieser Vorbedingung ist jedoch in einer späteren Version nachträglich implementiert worden, und zwar in der Methode `checkOverloadingAndAmbiguity`, die ebenfalls in `checkInitialConditions` aufgerufen wird.
Im Gegensatz zur Definition von *Extract Method* in Abschnitt 4.4.2.2, in der ausdrücklich eine private Methode erzeugt wird, ist es in Eclipse auch möglich die extrahierte Methode als `public`, `protected` oder ohne Zugriffs-Modifikator (Standard-Sichtbarkeit) zu deklarieren. Die Vorbedingung Nr. 6 wird für diese Fälle ebenfalls überprüft.

Änderungen im Aspekt-Code
Da sich sämtliche Änderungen in *Extract Method* (Erzeugung der neuen Methode und Ersetzung der Selektion durch einen Aufruf der Methode) lokal auf die Klasse beschränken, in der sich auch die selektierten Elemente befinden, sind während der Transformation eines ObjectTeams/Java Programms in den Team- und Rollenklassen keine weiteren Modifikationen notwendig (siehe auch Nachbedingungen dieses Refactorings in Abschnitt 4.4.2.2).

5.3.3 Move Instance Method

Das *Move Instance Method Refactoring* ist ebenfalls ein nicht-prozessorbasiertes Refactoring. Es ist auf Seite 70 definiert. Die vollständige Erweiterungsstruktur ist in Abbildung A.4 in Anhang A dargestellt. Die Erweiterung ist in der Klasse `OTMoveInstanceMethodRefactoring` implementiert. Diese Klasse erbt von der abstrakten Super-Klasse `Refactoring` und stellt einen Wrapper für die Klasse `MoveInstanceMethodRefactoring` dar.
Weiterhin sind die Klassen `OTMoveAction, OTMoveInstanceMethodAction, OTMoveInstanceMethodWizard` (inklusive der zugehörigen Page) und `OTInstanceMethodMover` implementiert worden.
Wenn eine Methode verschoben werden soll wird zunächst die Klasse `OTMoveAction` vom Eclipse-Framework aufgerufen. Diese ruft wiederum die Klasse `OTMoveInstanceMethodAction` auf, wo das Refactoring (`OTMoveInstanceMethodRefactoring`) und der Wizard (`OTMoveInstanceMethodWizard`) erzeugt und aktiviert werden (nach der Überprüfung der initialen Vorbedingungen).

Überprüfung der Vorbedingungen
In der Klasse `OTInstanceMethodMover` sind sämtliche Methoden implementiert, die die Überprüfung der Vorbedingungen realisieren.
Da dieses Refactoring in Eclipse stark eingeschränkt ist und, wie der Name bereits aussagt, nur das Verschieben von Instanz-Methoden berücksichtigt[1],

[1]Das Verschieben von statischen Methoden ist in Eclipse in der Klasse `MoveStaticMembersProcessor` implementiert.

gibt es neben den Vorbedingungen in Abschnitt 4.4.1.3 noch weitere spezielle Vorbedingungen, die in Eclipse geprüft werden.
Diese Vorbedingungen sind: Die zu verschiebende Methode ist kein Konstruktor, sie ist nicht statisch, `native`, `synchronized`, rekursiv und sie enthält keinen `super`-Aufruf und keine Referenzen auf die umschließende Instanz.
Als mögliche Ziel-Klasse, in die die Methode verschoben werden soll, kommt in Eclipse nur der Typ eines Parameters der Methode oder der Typ eines Feldes, das in derselben Klasse deklariert ist wie die Methode, in Frage. Die Klasse, aus der die Methode verschoben werden soll, als auch die Ziel-Klasse müssen modifizierbar sein (Datei ist nicht *read-only* oder binär).
Die einzige der Vorbedingungen des *Move Method Refactorings* (siehe 4.4.1.3), die von Eclipse überprüft wird, ist die Vorbedingung Nr. 3. Die Überprüfungen der Vorbedingungen Nr. 4 und Nr. 7 bis 9 sind in Eclipse nicht implementiert. Um jedoch die durchgeführte Erweiterung dieses Refactorings ausprobieren und testen zu können, musste die Prüfung dieser Vorbedingungen eingebaut werden. Dazu sind die Methoden `checkForDuplicateMethodInNewReceiver(NewReceiver, IJavaProject)`, `checkFieldAccesses()` sowie weitere Hilfsmethoden implementiert worden.
Die Vorbedingung Nr. 5 wird weder in der Eclipse-Implementierung noch in der Adaptierung dieses Refactorings berücksichtigt.
Die Vorbedingungen Nr. 1, Nr. 2 und Nr. 6 sind neue, ObjectTeams/Java-spezifische Vorbedingungen. Die initiale Vorbedingung Nr. 1 (siehe 5.3.1) wird in der Methode `checkInitialConditions` in der Klasse `OTMoveInstanceMethodRefactoring` kontrolliert.
Die Ziel-Klasse, in die die Methode verschoben werden soll, muss eine reguläre OO-Klasse sein. Diese Vorbedingung (Nr. 2) wird in der Methode `canAddAsPossibleNewReceiver(ITypeBinding)` überprüft.
In der Methode `checkForOverridingAndAmbiguity()` sowie den dazugehörigen Hilfsmethoden wird Vorbedingung Nr. 6 geprüft.

Änderungen im Aspekt-Code
Wie schon beim *Extract Method Refactoring* sind auch bei diesem Refactoring während der Transformation eines ObjectTeams/Java Programms keine weiteren Änderungen in den Team- und Rollenklassen notwendig, da sich die Modifikationen (Entfernen der Methode in der ursprünglichen Klasse und Erzeugung der Methode in der Ziel-Klasse) auf die Klasse, die die Methode definiert und auf die Klasse, in welche die Methode verschoben werden soll, beschränken (siehe auch Nachbedingungen dieses Refactorings in Abschnitt 4.4.1.3).

5.3.4 Rename Method

In der Implementierung des *Rename Method Refactorings* unterscheidet Eclipse zwischen dem Umbenennen einer virtuellen Methode, mit anderen Worten, einer Methode, die in einer Sub-Klasse überschrieben bzw. redefiniert werden darf (`RenameVirtualMethodProcessor`) und dem Umbenennen einer nicht-virtuellen, das heißt, einer privaten oder statischen Methode (`RenameNonVirtualMethodProcessor`). In beiden Fällen handelt es sich um ein prozessorbasiertes Refactoring. Das Rename Method Refactoring ist auf Seite 62 definiert. Eine komplette Übersicht der Erweiterungsstruktur für beide Varianten ist in Abbildung A.5 in Anhang A abgebildet.

5.3.4.1 Rename Virtual Method

Die Erweiterung für diese Variante des *Rename Method Refactorings* ist in der Klasse `OTRenameVirtualMethodProcessor` implementiert. Diese Klasse erbt von `RenameVirtualMethodProcessor` und redefiniert die Methoden `checkInitialConditions` und `checkFinalConditions` (siehe unten)
Darüber hinaus sind noch folgende Klassen implementiert worden: `OTRenameAction, OTRenameJavaElementAction, OTRenameSupport, OTRenameUserInterfaceManager` und `OTRenameMethodUserInterfaceStarter`.
Die Klasse `OTRenameAction` wird von Eclipse aufgerufen, wenn eine Methode umbenannt werden soll. In der Klasse `OTRenameJavaElementAction`, die von `OTRenameAction` verwendet wird, wird das entsprechende Refactoring mit Hilfe der Klasse `OTRenameSupport` erzeugt. In `OTRenameSupport` wird anschließend, unter Verwendung der Klassen `OTRenameUserInterfaceManager` und `OTRenameMethodUserInterfaceStarter`, das Refactoring aktiviert und der Wizard geöffnet (nach Prüfung der initialen Vorbedingungen).

Überprüfung der Vorbedingungen
In der Eclipse-Implementierung werden in den Methoden `checkInitialConditions` und `checkFinalConditions` in den Klassen `RenameVirtualMethodProcessor` und `RenameMethodProcessor` bereits die Vorbedingungen Nr. 2 und Nr. 3 des *Rename Method Refactorings* (siehe 4.3.3.3) überprüft. Außerdem werden dort noch folgende Vorbedingungen geprüft: Die Methode, die umbenannt werden soll, darf nicht *native*, *read-only* oder binär sein.
Die Vorbedingungen Nr. 1 und Nr. 5 sind neue, ObjectTeams/Java-spezifische Vorbedingungen. Die initiale Vorbedingung Nr. 1 wird wieder in der Methode `checkInitialConditions` in der Klasse `OTRenameVirtualMethodProcessor` überprüft.
Vorbedingung Nr. 5 wurde in der initialen Implementierung des erweiterten Refactorings nicht berücksichtigt. Die Prüfung dieser Vorbedingung ist jedoch in einer späteren Version nachträglich implementiert worden.

Die Vorbedingung Nr. 4 ist weder in der Eclipse-Implementierung noch in der Adaptierung dieses Refactorings implementiert.

Änderungen im Aspekt-Code

Damit bei diesem Refactoring auch Referenzen auf die umzubenennende Methode in den Methoden-Bindungen (Callout- und Callin-Bindungen) innerhalb einer Rollenklasse gefunden und aktualisiert werden (siehe auch Nachbedingungen dieses Refactorings in Abschnitt 4.3.3.3), sind einige Klassen angepasst worden, die Teil der Suchmaschine (*search engine*) in Eclipse sind. Dazu gehören die Klassen `MatchLocator, MatchLocatorParser, PatternLocator, OrLocator`, und `MethodLocator`.

Die Klasse `MatchLocatorParser` repräsentiert einen *Parser*, der AST-Knoten lokalisiert, die zu einem bestimmten Suchmuster (z.B. ein voll qualifizierter Methodenname) passen.

Um auch Methoden-Referenzen in Callout- und Callin-Bindungen lokalisieren zu können, muss der neue AST-Knoten `MethodSpec`[2] bei der Suche nach referenzierten Elementen berücksichtigt werden. In der Klasse `MatchLocatorParser` sind daher zusätzlich folgende Methoden implementiert worden: Die Methode `visit(MethodSpec)` in der Member-Klasse `ClassButNoMethodDeclarationVisitor` sowie die Methoden `consumeMethodSpecLong()` und `consumeMethodSpecShort()`.

Die `visit`-Methode wird während der Suche nach Referenzen für das Traversieren einer `MethodSpec` in einem vorhandenen AST verwendet. Die `consume`-Methoden werden vom Parser zur Lokalisierung von Knoten vom Typ `MethodSpec` benötigt.

Den Klassen `PatternLocator, OrLocator, und MethodLocator` ist die Methode `match(MethodSpec, MatchingNodeSet)` hinzugefügt worden. Diese `match`-Methode wird in jeder der zuvor genannten Methoden aufgerufen. In der `match`-Methode der Klasse `MethodLocator` werden die Attribute der übergebenen `MethodSpec` mit den Attributen des Suchmusters verglichen und auf diese Weise geprüft, ob es sich bei dem Knoten (`MethodSpec`) um eine passende Referenz auf die umzubenennende Methode handelt oder nicht. Die abstrakte Klasse `PatternLocator` enthält lediglich eine Standard-Implementierung der `match`-Methode.

Die Klasse `OrLocator` wird verwendet, falls das Suchmuster aus mehreren verschiedenen Suchmustern besteht. Dies ist beispielsweise der Fall, wenn Methoden innerhalb einer Hierarchie (in Interfaces und Klassen) existieren, die alle den gleichen Namen haben und die alle umbenannt werden müssen.

Schließlich sind in der Klasse `MatchLocator` noch einige Methoden adaptiert worden um spezielle Probleme zu lösen, die während des Parse-Vorgangs (Erzeugung der Compilation Unit, die mögliche Referenzen enthält) und bei der Auflösung von Elementen (Erzeugung der *Bindings* für diese Compilation

[2] Eine `MethodSpec` repräsentiert jeweils die Referenz auf eine Methode auf der linken bzw. rechten Seite einer Methoden-Bindung.

Unit) auftraten.
Die Methoden in expliziten und impliziten Rollenklassen, die die umzubenennende Methode überschreiben, werden bereits in der Eclipse-Implementierung lokalisiert und entsprechend umbenannt. Diese Methoden werden jedoch erst über die Suche nach Referenzen gefunden. Eine Implementierung, die der in Eclipse entspricht, wäre, sämtliche redefinierende bzw. redefinierte Methoden in einer Rollenhierarchie (Klassen und Interfaces) zu sammeln und direkt umzubenennen. Dieser Ansatz war jedoch nicht möglich, da zum Zeitpunkt der Adaptierung dieses Refactorings noch kein Algorithmus und keine geeignete Datenstruktur für den Aufbau einer vollständigen (expliziten und impliziten) Rollenhierarchie existierte.
Methoden-Referenzen in einem *Guard* sind bei der Erweiterung dieses Refactorings nicht berücksichtigt worden, da zu jenem Zeitpunkt dieses Feature noch nicht Bestandteil des Compilers war.

5.3.4.2 Rename Non-Virtual Method

In der Klasse `OTRenameNonVirtualMethodProcessor` ist die Erweiterung für diese Variante des *Rename Method Refactorings* implementiert. Diese Klasse erbt von `RenameNonVirtualMethodProcessor` und redefiniert die Methode `checkInitialConditions` aus `RenameMethodProcessor` und die Methode `checkFinalConditions` aus der direkten Super-Klasse (siehe unten). Diese Variante verwendet für die Erzeugung und Aktivierung des Refactorings sowie des Wizards die gleichen Klassen wie die zuvor beschriebene Variante (siehe 5.3.4.1).

Überprüfung der Vorbedingungen
Die Vorbedingungen aus Abschnitt 4.3.3.3 gelten auch für diese Variante des *Rename Method Refactorings* und werden genauso überprüft wie bei *Rename Virtual Method* (siehe 5.3.4.1). Der einzige Unterschied hier ist, dass die initiale Vorbedingung in der Methode `checkInitialConditions` in der Klasse `OTRenameNonVirtualMethodProcessor` geprüft wird.

Änderungen im Aspekt-Code
Bei diesem Refactoring muss eine auf der rechten Seite einer Methoden-Bindung vorhandene Referenz auf eine nicht-virtuelle, private Methode, die umbenannt werden soll, ebenfalls aktualisiert werden. Die notwendigen Anpassungen an der Suchmaschine, die dafür notwendig sind, wurden für die Adaptierung von *Rename Virtual Method* (siehe oben) bereits durchgeführt.

Kapitel 6

Zusammenfassung

6.1 Zusammenfassung und Fazit

Refactoring und aspektorientierte Programmierung sind Techniken, die gemeinsame Ziele verfolgen. Dazu gehören eine Vereinfachung und eine verbesserte Modularisierung von objektorientiertem Code, die die Lesbarkeit, Verständlichkeit und Wartbarkeit des Codes erhöhen. Werden beide Konzepte innerhalb desselben Softwareentwicklungsprozesses eingesetzt kann dies jedoch zu Problemen führen.
Aspektorientierte Programmiersprachen erweitern objektorientierte Sprachen um neue aspektorientierte Sprachelemente. Diese aspektorientierten Sprachkonstrukte können in einer aspektorientierten Anwendung direkt oder indirekt strukturelle Elemente aus dem Basis-Programm anhand von Namen referenzieren oder bestehende Vererbungshierarchien verändern.
Die zentrale Bedingung beim Refactoring ist, dass das Verhalten eines Programms erhalten wird. Die Modifikation von Elementen innerhalb einer Basis-Anwendung durch ein Refactoring führt jedoch dazu, dass die im Aspekt-Code spezifizierten bzw. referenzierten Elemente aus dem Basis-Code nicht angepasst werden. Der Grund ist, dass dem Refactoring die neuen, aspektorientierten Sprachelemente nicht bewusst sind. Dadurch verändert sich das Verhalten der Applikation und das Refactoring ist folglich nicht mehr verhaltenserhaltend.
Um Refactoring und aspektorientierte Programmiersprachen gemeinsam verwenden zu können, müssen die existierenden Refactorings auch die neuen, aspektorientierten Sprachelemente berücksichtigen. Es ist notwendig bestehende Refactorings so zu adaptieren, dass veränderte Elemente im Basis-Code, die vom Aspekt-Code referenziert werden, dort ebenfalls aktualisiert werden. Das Verhalten einer aspektorientierten Anwendung muss beim Einsatz von Refactoring beibehalten werden.

In dieser Arbeit werden existierende Refactorings für die aspektorientierte Sprache ObjectTeams/Java erweitert. Hierzu wird zunächst untersucht, welche Auswirkungen die aspektorientierten Konzepte und Mechanismen in ObjectTeams/Java auf objektorientiertes Refactoring haben. Anschließend werden Regeln definiert, die erfüllt sein müssen, damit ein Refactoring das Verhalten eines ObjectTeams/Java Programms erhält. Diese Regeln beziehen sich auf Sprachanforderungen, Vererbungsbeziehungen und semantische Äquivalenz. Für jedes erweiterte Refactoring werden Vor- und Nachbedingungen angegeben, die die Einhaltung bestimmter Regeln überprüfen. Drei der konzeptuell erweiterten Refactorings sind im Rahmen dieser Arbeit in der Entwicklungsumgebung Eclipse implementiert worden.

Die Analyse der Auswirkungen von Sprachkonzepten und Mechanismen in ObjectTeams/Java auf Refactoring und die durchgeführte Adaptierung existierender Refactorings in dieser Arbeit liefern mehrere Erkenntnisse bezüglich der Anwendung von Refactoring in ObjectTeams/Java Programmen.
In vielen der betrachteten Refactorings hat der Aspekt-Code einer ObjectTeams/Java Anwendung Auswirkungen auf Vererbungsbeziehungen, die erhalten werden müssen. Sämtliche Refactorings, die Signaturen von Methoden verändern oder die neue Methoden erzeugen, sind durch die neuen Sub-Typ-Beziehungen betroffen.
Dabei spielen insbesondere die Mechanismen *Overriding* und *Overloading* eine wichtige Rolle. Werden im Basis-Code veränderte oder neu erzeugte Methoden in explizit und implizit erbenden Rollenklassen überschrieben oder überladen, muss dies von einem Refactoring berücksichtigt werden. Überladene Methoden in einer Vererbungshierarchie können zu Mehrdeutigkeiten in existierenden Methoden-Bindungen führen.
Bedeutende Auswirkungen auf einige Refactorings haben auch die *Aspekt-Basis Referenzen* in einem ObjectTeams/Java Programm. Diese expliziten Referenzen auf strukturelle Elemente aus dem Basis-Code innerhalb des Aspekt-Codes beeinflussen alle Refactorings, die die Namen von Elementen verändern. Die Namen von Elementen eines Basis-Programms können an diversen Stellen im Aspekt-Code vorkommen. Werden diese Elemente umbenannt, müssen auch die Referenzen in Team- und Rollenklassen aktualisiert werden.
Refactorings, die neue Felder oder Typen erzeugen oder verschieben, werden vom Aspekt-Code lediglich bezüglich der Sprachanforderungen *Naming* und *Scoping* beeinflusst.
Auf Refactorings, die Elemente im Basis-Code entfernen, hat der Aspekt-Code insofern Auswirkungen, dass die zu entfernenden Elemente nicht in ihm referenziert werden dürfen.
Die Einhaltung der meisten Sprachanforderungen bzw. Sprachregeln in ObjectTeams/Java (siehe 3.2.1.1) wird von den in dieser Arbeit definierten Refactorings nur selten beeinflusst.

Viele dieser Sprachregeln müssten jedoch von neuen, aspektorientierten Refactorings (siehe 6.3.2) beachtet werden, die die aspektorientierten Sprachelemente wie Team- und Rollenklassen, Methoden-Bindungen, Parameter Mappings oder `callin`-Methoden umstrukturieren.

6.2 Vergleich: Adaptierung von Refactorings in ObjectTeams/Java und CaesarJ

Für die aspektorientierte Sprache CaesarJ (siehe 1.4.2), für die bislang noch keine automatisierte Refactoring-Unterstützung angeboten wird, könnten objektorientierte Refactorings auf ähnliche Weise adaptiert werden wie dies in dieser Arbeit für ObjectTeams/Java durchgeführt wird. Beide Sprachen besitzen einige ähnliche Konzepte und Mechanismen (z.B. virtuelle Klassen), die jedoch unterschiedlich realisiert sind.
In CaesarJ gibt es die sogenannte Caesar-Klasse (vergleichbar mit einer Teamklasse), die wiederum weitere, innere Caesar-Klassen (vergleichbar mit Rollenklassen) enthalten kann. Eine solche Caesar-Klasse kann nur von einer anderen Caesar-Klasse erben, jedoch nicht von einer Java-Klasse. Die Vererbungshierarchien von Caesar-Klassen und Java-Klassen sind in CaesarJ streng voneinander getrennt.
Dies ist der bedeutende Unterschied zu Rollenklassen in ObjectTeams/Java, die, im Gegensatz zu Caesar-Klassen, auch von Java-Klassen erben können. Dadurch wird eine Adaptierung von Refactorings, deren Änderungen sich auch auf Sub-Klassen auswirken (z.B. *Rename Method*), deutlich vereinfacht, da durch die neuen Sprachelemente in CaesarJ die bestehenden Vererbungshierarchien einer Basis-Anwendung nicht erweitert bzw. verändert werden können.
Die Ausnahme bilden Situationen, in denen die Signatur einer abstrakten Methode in einem Interface verändert wird. Da Caesar-Klassen Interfaces implementieren können, müssten auch die Methoden in den implementierenden Caesar-Klassen von einem Refactoring berücksichtigt werden.
Caesar-Klassen können auch Referenzen auf Elemente aus dem Basis-Programm enthalten. Dadurch sind alle Refactorings betroffen, die diese Elemente verändern oder entfernen.
Andere Refactorings, die zum Beispiel Typen oder Felder erzeugen oder verschieben, werden vom Aspekt-Code in CaesarJ ähnlich beeinflusst wie vom Aspekt-Code in ObjectTeams/Java.
Das in CaesarJ vorhandene Konzept der *Mixin-Komposition*, wodurch Caesar-Klassen von mehreren Super-Klassen erben können, würde insbesondere Auswirkungen auf neue, aspektorientierte Refactorings haben.

6.3 Ausblick

6.3.1 Adaptierung weiterer Refactorings in Eclipse

Im Rahmen des Topprax-Projekts[24] sind neben den in Kapitel 5 beschriebenen Refactorings in der Zwischenzeit bereits weitere Refactorings in Eclipse adaptiert worden. Diese Refactorings sind derart implementiert, dass sie nun nicht mehr ausschließlich auf den OO-Code einer ObjectTeams/Java-Anwendung angewendet werden können, sondern auch auf den Aspekt-Code selbst (falls dies möglich und sinnvoll ist).
Die im Rahmen dieser Arbeit konzeptuell erweiterten Refactorings (siehe Kapitel 4) können als Grundlage für die praktische Adaptierung weiterer Refactorings in Eclipse (siehe Tabelle A.1 in Anhang A) verwendet werden.

6.3.2 Entwicklung und Implementierung von neuen Refactorings für ObjectTeams/Java

Neben den in dieser Arbeit definierten OT/J-aware Refactorings könnten weitere Dimensionen von aspektorientiertem Refactoring (siehe 1.5.1) berücksichtigt werden.
So könnten neue, spezielle *ObjectTeams/Java Refactorings* entwickelt und implementiert werden, die Team- und Rollenklassen umstrukturieren (z.B. *Move Team Class* oder *Extract Role Class*) oder Methoden-Bindungen refaktorisieren (z.B. *Push Down Callin Binding*).
Eine weitere Möglichkeit wäre Refactorings einzuführen, die vorhandenen Basis-Code in einen Aspekt (Teamklasse) auslagern, oder umgekehrt, Refactorings zu definieren, die Funktionalität aus einer Team- oder Rollenklasse in eine Basisklasse verschieben.

6.3.3 Refactoring und Join-Points in ObjectTeams/Java

In dieser Arbeit sind bei der Definition von Regeln für Verhaltenserhaltung in ObjectTeams/Java Programmen und bei der Adaptierung der Refactorings keine aspektorientierten Konzepte bzw. Konstrukte wie Join-Points bzw. Pointcuts berücksichtigt worden, da bis zum heutigen Zeitpunkt noch keine Join-Point Sprache für ObjectTeams/Java existiert.
Eine Join-Point Sprache spezifiziert bestimmte Join-Points (siehe 1.4.1), mit deren Hilfe Aspekte in eine Basis-Anwendung hineingewebt werden.
Der Pointcut `call(* *.myMethod(..))` in AspectJ zum Beispiel selektiert alle Join-Points, die einen Aufruf der Methode `myMethod` in irgendeiner Klasse darstellen. Wird die Methode `myMethod` anhand von *Rename Method* (siehe 4.3.3.3) umbenannt, dann wird dadurch das Verhalten des AspectJ Programms verändert, da die Join-Point Spezifikation in obigem Pointcut nicht

angepasst wird. Alle Join-Points in der Pointcut-Spezifikation, die sich vor dem Refactoring auf `myMethod` bezogen haben, verweisen nach dem Refactoring nicht mehr auf denselben Join-Point. Refactorings können die Menge der Join-Points verändern, die in Pointcuts selektiert werden.
Hanenberg, Oberschulte und Unland stellen in [26] allgemein fest, dass das Transformieren einer Applikation unvermeidlich zu einer Modifikation von Join-Points führt, die in Aspekten bzw. Pointcuts spezifiziert werden können. Folglich sind Refactorings nicht länger verhaltenserhaltend, falls die Join-Point Spezifikationen nicht adaptiert werden.
Damit die in Kapitel 4 definierten OT/J-aware Refactorings auch in Zukunft, wenn eine Join-Point Sprache für ObjectTeams/Java existiert, das Verhalten erhalten, müssen diese so erweitert werden, dass sie auch alle notwendigen Join-Point Spezifikationen innerhalb eines ObjectTeams/Java Programms transformieren. Dazu muss für jedes Refactoring unter anderem ermittelt werden, *welche* Join-Points jeweils betroffen sind und *wie* diese vom Refactoring beeinflusst werden.
Einen Ansatz beschreiben Hanenberg u.a. in [26], in welchem sie zusätzliche Bedingungen bezüglich Join-Points angeben, die erfüllt sein müssen damit beim Einsatz von Refactorings in einem aspektorientiertem System das Verhalten erhalten wird. Die Berücksichtigung von Join-Point Spezifikationen während der Durchführung eines Refactorings ist auch ein Thema in Ruras Arbeit (siehe 6.4).

6.4 Verwandte Arbeiten

In seiner Arbeit „Refactoring aspect-oriented software“ [25] beschreibt Rura, welche Auswirkungen neue aspektorientierte Konstrukte wie Pointcuts und Join-Points, die in aspektorientierten Sprachen verfügbar sind, auf das Programmverhalten haben wenn existierende Refactorings eingesetzt werden. Um diese Refactorings auf ein aspektorientiertes Programm anwenden zu können, sind von ihm die objektorientierten Refactorings, die Opdyke in seiner Dissertation[22] definiert hat, für die Sprache AspectJ (siehe 1.4.1) überarbeitet worden, ähnlich wie dies in dieser Arbeit für die Sprache ObjectTeams/Java getan wird.
Rura aktualisiert dazu die von Opdyke aufgestellten Regeln für die Sprache Java. In Analogie zu den hier erweiterten Sprachanforderungen und Regeln für Verhaltenserhaltung in ObjectTeams/Java werden von ihm die Sprachanforderungen von Java um neue Anforderungen der Sprache AspectJ erweitert sowie neue Regeln für die Einhaltung von Sub-Typ-Beziehungen und semantischer Äquivalenz in einem AspectJ Programm aufgestellt.
Darüber hinaus definiert Rura neue, aspektorientierte Refactorings, die speziell die aspektorientierten Programmelemente (Aspekte, Pointcuts und Advices) betreffen.

6.5 Nicht berücksichtigte Sprach-Features in ObjectTeams/Java

In den Definitionen der Sprachregeln für ObjectTeams/Java in Kapitel 3.1 und in den adaptierten Refactorings in Kapitel 4 wird das Sprachfeature *callin precedence* und das Binden von statischen Methoden in Callin-Bindungen nicht berücksichtigt. Diese Features waren noch nicht Bestandteil der initialen Version 0.8 (Stand: Juni 2005) der ObjectTeams/Java Sprachdefinition, die in dieser Arbeit verwendet wurde.

Anhang A

Name des Refactorings	Klassifikation	Implementierungsart
Convert Anonymous Class To Nested	Typ	Standard
Convert Local Variable To Field	Feld/Variable	Standard
Change Signature	Methode	Standard
Extract Constant	Feld/Variable	Standard
Extract Interface	Generalisierung	Standard
Extract Method	Methode	Standard
Extract Temp (Local Variable)	Feld/Variable	Standard
Generalize Type	Generalisierung	Standard
Inline Constant	Feld/Variable	Standard
Inline Method	Methode	Standard
Inline Temp (Local Variable)	Feld/Variable	Standard
Introduce Factory	Methode	Standard
Introduce Parameter	Methode	Standard
Move Inner To Top	Typ	Standard
Move Instance Method	Methode	Standard
Move Static Members	Feld/Methode/Typ	Prozessorbasiert
Pull Up (Field, Method, Type)	Generalisierung	Standard
Push Down (Field, Method)	Spezialisierung	Standard
Rename Compilation Unit	Compilation Unit	Prozessorbasiert
Rename Field	Feld/Variable	Prozessorbasiert
Rename Java Project	Projekt	Prozessorbasiert
Rename Non-Virtual Method	Methode	Prozessorbasiert
Rename Package	Package	Prozessorbasiert
Rename Resource	Datei	Prozessorbasiert
Rename Source Folder	Verzeichnis	Prozessorbasiert
Rename Temp (Local Variable)	Feld/Variable	Prozessorbasiert
Rename Type	Typ	Prozessorbasiert
Rename Virtual Method	Methode	Prozessorbasiert

Tabelle A.1: Implementierte Refactorings in Eclipse

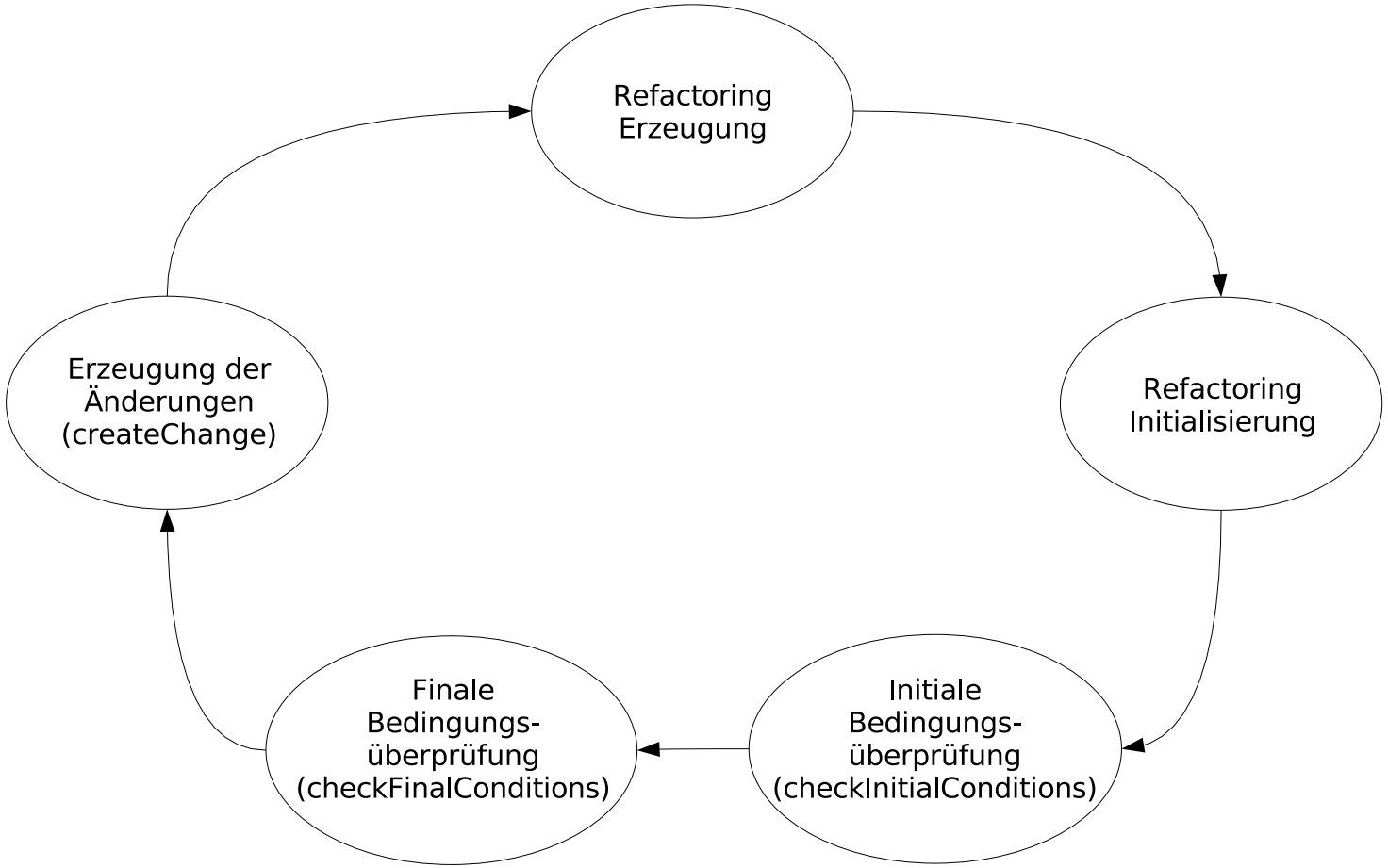

Abbildung A.1: Lebenszyklus eines Refactorings in Eclipse

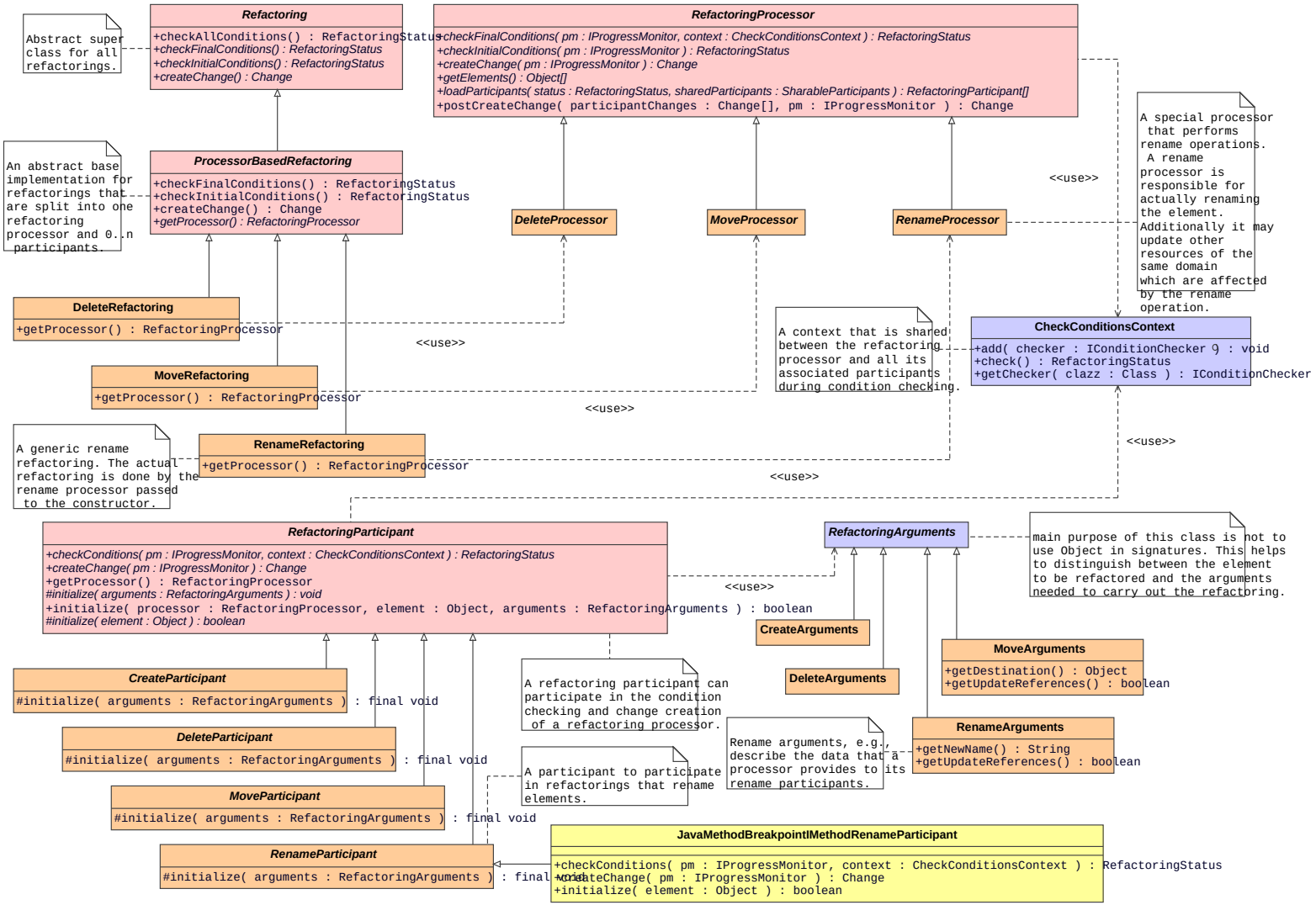

Abbildung A.2: Die Prozessor/Participant-Architektur

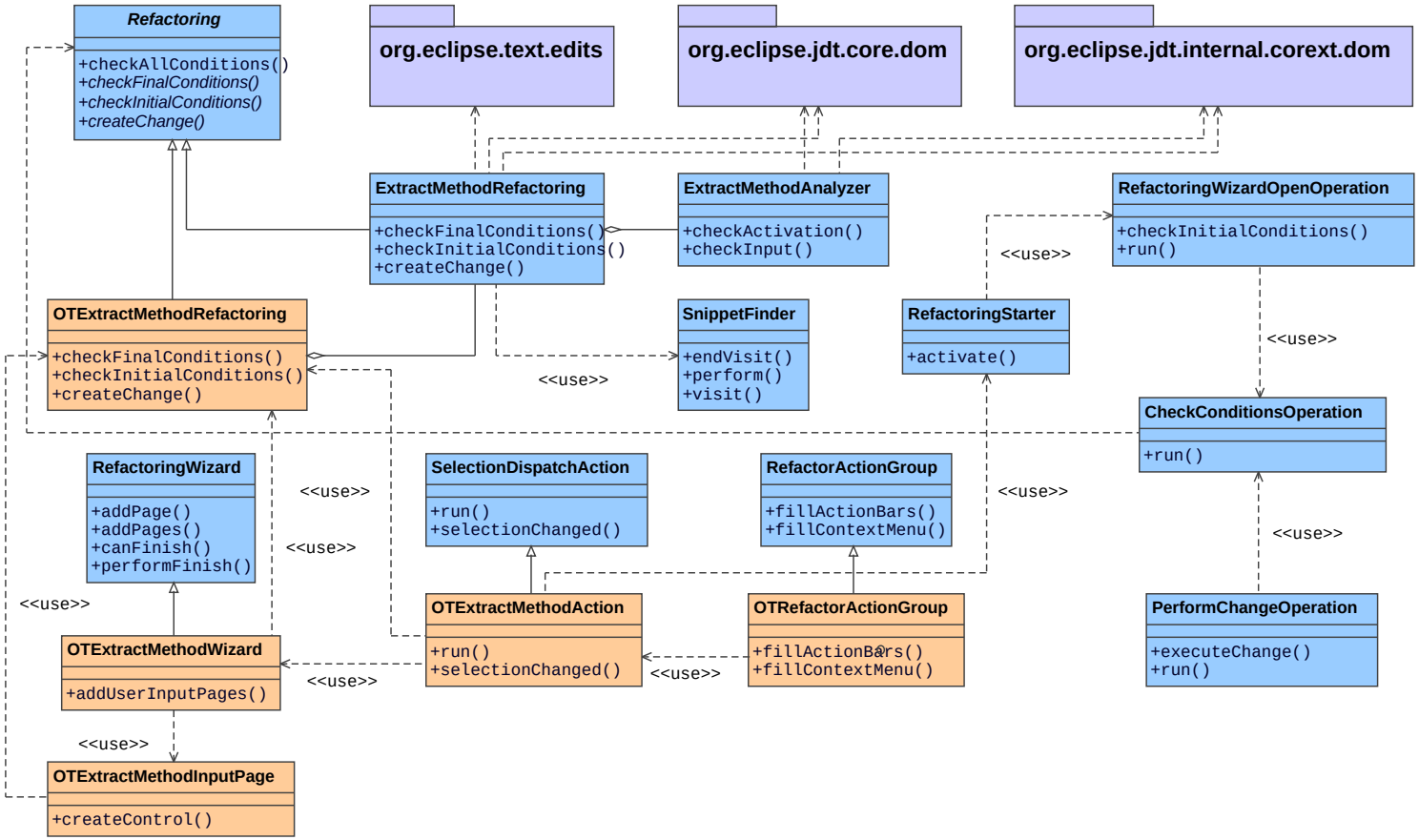

Abbildung A.3: Erweiterung des Extract Method Refactorings

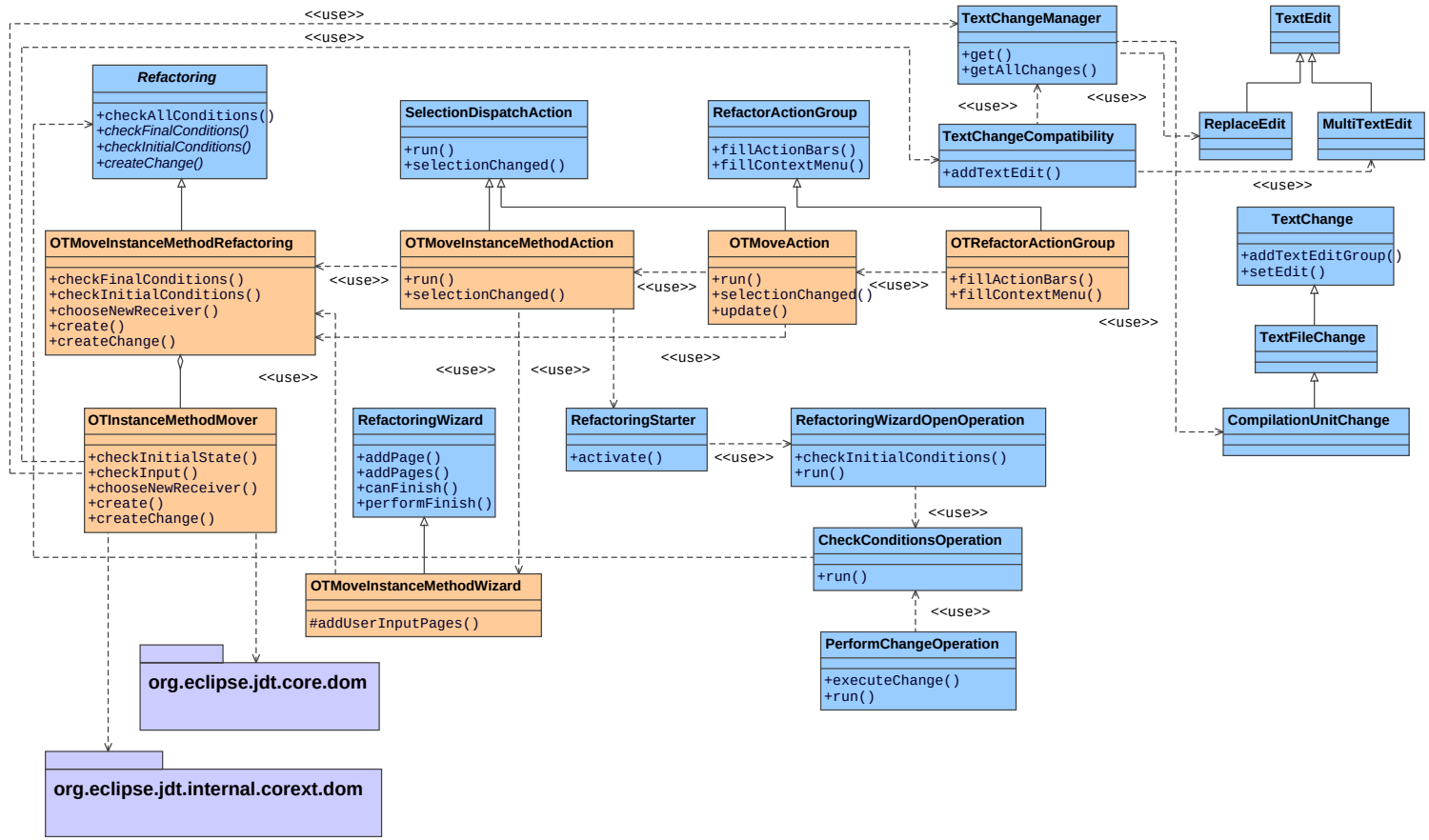

Abbildung A.4: Erweiterung des Move Instance Method Refactorings

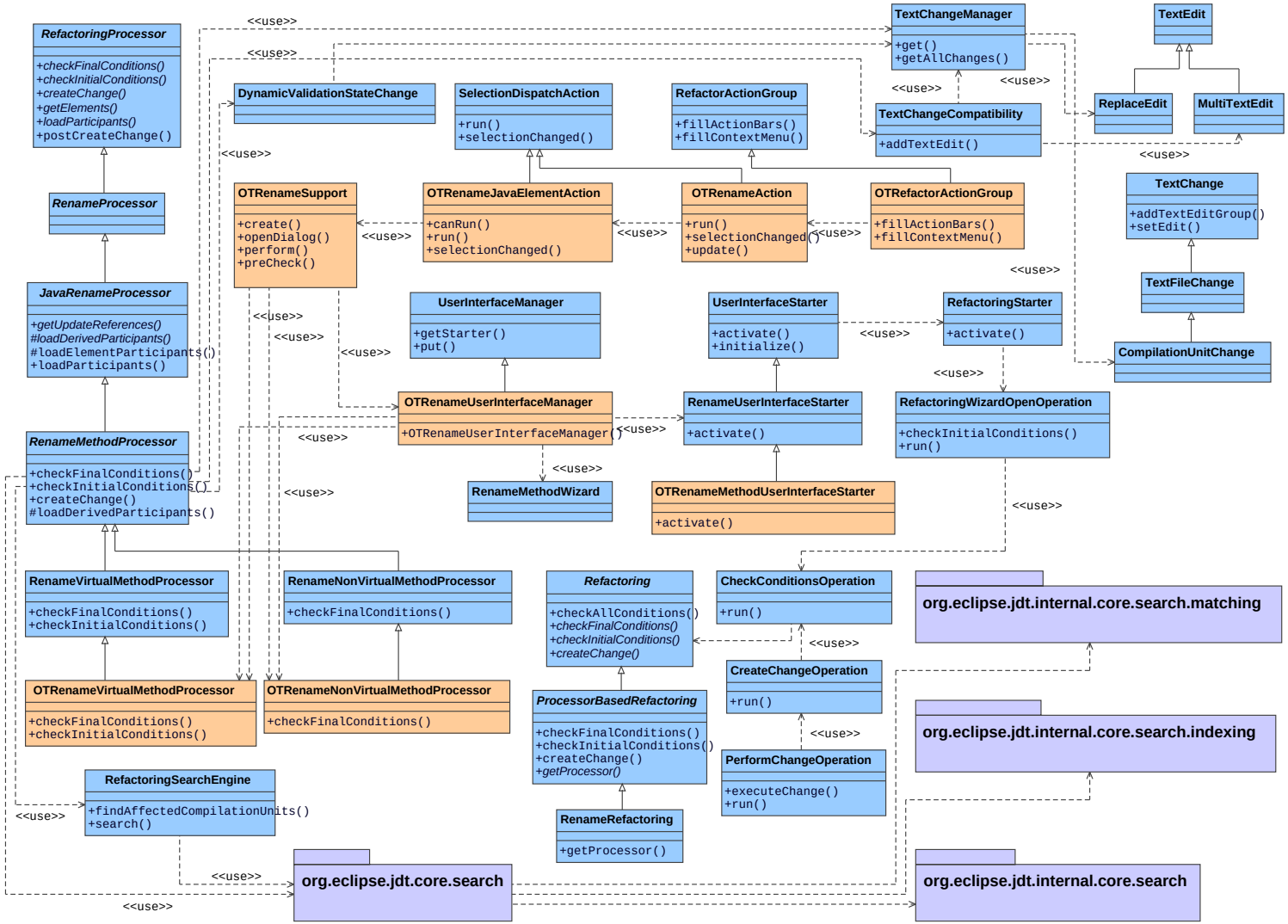

Abbildung A.5: Erweiterung des Rename Method Refactorings

Literaturverzeichnis

[1] S. Cook, H. Ji, and R. Harrison. Software evolution and software evolvability, 2000.

[2] ObjectTeams/Java Language Definition. http://www.objectteams.org/def/index.html.

[3] T. Dudziak and J. Wloka. Tool-supported discovery and refactoring of structural weaknesses in code. Master's thesis, Technische Universität Berlin, February 2002.

[4] The Eclipse project. http://www.eclipse.org.

[5] R. E. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-oriented software development.* Addison Wesley Professional, 2004.

[6] M. Fowler. Online catalogue of refactorings. http://www.refactoring.com/catalog/index.html.

[7] M. Fowler, K. Beck, J. Brant, and D. Roberts. *Refactoring: Improving the design of existing code.* Addison-Wesley Professional, first edition, 1999.

[8] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Net Object Days '02: Proceedings of Net Object Days*, 2002.

[9] AspectJ home page. http://www.aspectj.org.

[10] CaesarJ home page. http://www.caesarj.org.

[11] IntelliJ Idea. http://www.jetbrains.com/idea.

[12] B. Joy, G. Steele, J. Gosling, and G. Bracha. *The Java Language Specification.* Addison-Wesley, second edition, 2000.

[13] Ramnivas Laddad. I want my AOP!, Part 1. *Java World*, January 2002.

[14] Ramnivas Laddad. I want my AOP!, Part 2. *Java World*, March 2002.

[15] Ramnivas Laddad. Aspect-oriented refactoring series. Part 1 - Overview and process. *TheServerSide.com*, December 2003.

[16] M. Lehman. Programs, lifecycles and laws of software evolution. In *IEEE '80: Proceedings of the IEEE*, 1980.

[17] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components, 1999.

[18] O. L. Madsen and B. Moller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 397–406, New York, NY, USA, 1989. ACM Press.

[19] Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139, 2004.

[20] M. Mezini, L. Seiter, and K. Lieberherr. Component integration with pluggable composite adapters, 2000.

[21] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 97–116, New York, NY, USA, 1998. ACM Press.

[22] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[23] AspectJ programming guide. http://eclipse.org/aspectj/doc/.

[24] Topprax Projekt. http://www.topprax.de.

[25] S. Rura. Refactoring aspect-oriented software. Master's thesis, Williams College, Williamstown, Massachusetts, 2003.

[26] R. Unland S. Hanenberg, C. Oberschulte. Refactoring of aspect-oriented software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35, Erfurt, Germany, September 2003.

[27] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Corporation, http://www.research.ibm.com/hyperspace, 2000.

[28] Object Teams home page. http://www.objectteams.org.

[29] Frank Tip, Adam Kiezun, and Dirk Bäumer. Refactoring for generalization using type constraints. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 13–26, New York, NY, USA, 2003. ACM Press.

[30] M. Witte. Portierung, Erweiterung und Integration des Objectteams/Java Compilers für die Entwicklungsumgebung Eclipse. Master's thesis, Technische Universität Berlin, Dezember 2003.

[31] J. Wloka. Refactoring in the presence of aspects. In *ECOOP '03: Proceedings of the 17th European Conference on Object-Oriented Programming, PhD workshop*, 2003.

[32] J. Wloka. Aspect-aware refactoring tool-support, 2005. LATE '05: Position paper.