

FERNUNIVERSITÄT IN HAGEN
FAKULTÄT FÜR MATHEMATIK UND INFORMATIK
LEHRGEBIET PROGRAMMIERSYSTEME

Prof. Dr. Friedrich Steimann

Introduce Role Object Refactoring

Abschlussarbeit

im Studiengang Master of Science im Fach Informatik

vorgelegt von

Fabian Urs Stolz

Traddeweg 3
44269 Dortmund
Matrikelnummer 7337523
fabian.stolz@googlemail.com

betreut durch
Prof. Dr. Friedrich Steimann

30.01.2011

Abstract

Verbreitete klassenbasierte objektorientierte Programmiersprachen verfügen in der Regel nicht über native Sprachmittel, die es ermöglichen, Objekten kontextabhängig unterschiedliche Eigenschaften und Verhalten zuzuweisen. Das Role Object Pattern bietet sich als Lösung für Fälle an, in denen dies dennoch erforderlich ist. Diese Arbeit beschreibt die Konzeption und Implementierung eines automatisierten Refactoring-Tools zur Einführung des genannten Pattern in bestehende Java-Programme. Im Rahmen der Entwicklung hat sich gezeigt, dass die dabei erreichbare Lösung aufgrund ihrer umfangreichen Vorbedingungen sowie der hohen Komplexität und dementsprechend suboptimalen Les- und Wartbarkeit des refaktorierten Codes nicht überzeugen kann. Daher wird dem zunächst verfolgten Ansatz eine zweite, schlankere Lösung – „Introduce Lightweight Role Objects“ genannt – gegenüber gestellt, welche zwar die gleichen Anforderungen erfüllt, sich aber durch eine einfachere Umsetzung und bessere Anwendbarkeit auszeichnet.

Inhaltsverzeichnis

1 Einleitung	1
1.1 Motivation.....	1
1.2 Aufbau.....	1
1.3 Beitrag dieser Arbeit.....	2
2 Problemstellung	3
2.1 Begriffe.....	3
2.2 Umsetzung von Rollenkonzepten.....	4
2.3 Das Role Object Pattern.....	8
2.4 Refactoring to Role Objects.....	10
3 Lösungsansatz Role Object Pattern	14
3.1 Grundsätzliches Vorgehen.....	14
3.2 Logische Identität des Subjekts.....	17
3.3 Gemeinsamer Zustand – Self Encapsulate Field.....	19
3.4 Rollenmanagement.....	20
3.5 Offene Rekursion – Reverse Forwarding.....	22
3.6 Instanziierung.....	28
3.7 Geerbtes Verhalten und Schnittstellen.....	33
3.8 Zusammenfassung.....	34
4 Lösungsansatz Lightweight Role Objects	38
5 Implementierung	43
5.1 Refactorings mit Eclipse JDT.....	43
5.2 Umsetzung der Refactorings.....	47
5.2.1 Gemeinsam genutzte Elemente.....	47
5.2.2 Elemente des Role Object Pattern Refactorings.....	51
5.2.3 Elemente des Lightweight Role Objects Refactorings.....	55
6 Evaluation der Anwendbarkeit	57
6.1 Anwendbarkeit, Korrektheit und Komplexität.....	57
6.2 Rollenverhalten.....	61
7 Diskussion	63
7.1 Interpretation der Ergebnisse.....	63
7.2 Bewertung der Ergebnisse.....	63
7.3 Verwandte Arbeiten.....	64
8 Schlussbetrachtungen	66

8.1 Zusammenfassung.....	66
8.2 Ausblick.....	66
8.3 Fazit.....	67
9 Abbildungen und Tabellen	68
10 Literaturverzeichnis	69
11 Anhang A: Beiliegende CD	72
12 Erklärung	73

1 Einleitung

1.1 Motivation

Wie in [Steimann2010] beschrieben, sind Vererbungsbeziehungen von Klassen ein zentrales Mittel weit verbreiteter objektorientierter Programmiersprachen (z.B. Java, C++, C#) zur

Wiederverwendung von Eigenschaften und Verhalten. Subklassen einer gemeinsamen Superklasse teilen sich dabei das von dieser geerbte Verhalten ebenso wie die geerbten Eigenschaften. Dadurch ergibt sich eine starke Kopplung der Subklassen an ihre jeweiligen Superklassen dergestalt, dass Änderungen an einer Superklasse sich auf jede ihrer jeweiligen Subklassen auswirken (können).

Objekte verschiedener Subklassen einer gemeinsamen Superklasse sind jedoch, ebenso wie Objekte von Sub- und Superklasse, weitgehend voneinander unabhängig. Im Gegensatz zu prototypen-basierten Programmiersprachen (z.B. Self, vgl. [Ungar1987]) teilen sie sich den geerbten Zustand nicht, so dass Änderungen des Zustands eines solchen Objektes keine direkten Auswirkungen auf den Zustand der übrigen haben.

Dies ist dann von Nachteil, wenn sich herausstellt, dass Objekte, die wie oben beschrieben miteinander verwandt sind, nicht unterschiedliche Individuen mit jeweils individuellem, voneinander unabhängigem Zustand repräsentieren, sondern stattdessen unterschiedliche, kontextabhängige Sichten auf ein einzelnes Individuum darstellen. Im letztgenannten Fall ist es erforderlich, dass Änderungen des gemeinsamen Teils des Zustands eines Objekts, das eine bestimmte Sicht auf ein Individuum darstellt, sich direkt auf alle anderen Objekte auswirkt, welche Sichten auf dasselbe Individuum sind.

Da die oben genannten klassenbasierten objektorientierten Programmiersprachen jedoch nicht über native Sprachmittel verfügen, die das oben beschriebene Verhalten direkt unterstützen, sind andere Lösungen erforderlich, um dies zu ermöglichen. [Fowler1997] nennt das Role Object Pattern (ROP) nach [Bäumer1997] als eine solche Lösung. Die manuelle Anwendung dieses nicht-trivialen Pattern auf bestehende Programme ist komplex und deswegen sowohl aufwändig als auch potentiell fehleranfällig, so dass ein automatisiertes Werkzeug zur Durchführung dieser Aufgabe wünschenswert ist. Die Konzeption und Implementierung eines solchen Werkzeugs für die Sprache Java ist das Ziel dieser Arbeit.

1.2 Aufbau

Dieses erste Kapitel beschreibt die dieser Arbeit zu Grunde liegende Motivation, den Aufbau und Inhalt der folgenden Kapitel sowie den wissenschaftlichen Beitrag dieser Arbeit.

Kapitel 2 beschreibt die zu lösende Problemstellung. Zunächst werden die verwendeten Begriffe definiert. Es folgen eine kurze Erläuterung des Konzepts „Rolle“ sowie eine ausführliche Vorstellung des Role Object Pattern. Abschließend werden die grundsätzlichen Probleme der Einführung dieses Entwurfsmusters in bereits bestehende Programme mittels Refactoring umrissen.

Kapitel 3 und 4 enthalten zwei verschiedene Lösungsansätze für das in Kapitel 2 definierte Problem. Kapitel 3 beschreibt die notwendigen Schritte, eine bestehende Superklasse und deren Subklassen in die Struktur, welche das ROP vorgibt, zu überführen und definiert die dafür erforderlichen Vorbedingungen. Die notwendigen Anpassungen des Pattern, welche sich aus den besonderen Problemen von dessen Anwendung im Kontext eines Refactorings und aus den speziellen Eigenschaften der Sprache Java ergeben, werden detailliert erläutert.

Der zweite, in Kapitel 4 beschriebene, Ansatz stellt eine alternative Lösung dar, die im Folgenden „Introduce Lightweight Role Objects“ (ILRO) genannt wird. Diese lässt sich aus dem ersten Lösungsansatz ableiten, erfüllt dieselben maßgeblichen Anforderungen, ist aber weniger kompliziert umzusetzen. Dieser Ansatz wird ebenfalls inklusive der für ihn notwendigen (und im Vergleich zum ersten Ansatz deutlich reduzierten) Vorbedingungen beschrieben.

Kapitel 5 widmet sich der Implementierung der beiden Lösungsansätze aus den Kapiteln 3 und 4 in Form eines Plug-ins für die Java Development Tools der Entwicklungsumgebung Eclipse. Dabei werden zunächst die Möglichkeiten, die diese Umgebung zur Implementierung von Refactoring-Werkzeugen bietet, kurz umrissen und anschließend Architektur und Vorgehensweise des Werkzeugs beschrieben. Das Plug-in und der dazugehörige Quellcode befinden sich auf dem beiliegenden Datenträger.

Kapitel 6 enthält die Ergebnisse der Untersuchung von Anwendbarkeit und Korrektheit der in den vorangegangenen Kapiteln beschriebenen Lösung. Diese Ergebnisse werden in Kapitel 7 interpretiert und bewertet. Darüber hinaus enthält dieses Kapitel einen Überblick über verwandte Arbeiten.

Das abschließende Kapitel 8 enthält neben der Zusammenfassung einen Ausblick auf noch offene Fragen, die im Rahmen dieser Arbeit nicht betrachtet wurden, und ein abschließendes Résumé.

1.3 Beitrag dieser Arbeit

Der Beitrag dieser Arbeit besteht darin, ein Werkzeug für die automatisierte Einführung des Role Object Pattern in bestehende Java Programme zu beschreiben. Dabei werden die dafür notwendigen Vorbedingungen sowie die ebenfalls notwendigen Anpassungen dieses Entwurfsmusters für dessen Einsatz im Rahmen eines solchen Refactorings definiert. In Anbetracht der erzielten Ergebnisse wird eine alternative, schlanke Lösung für die Einführung eines Rollenkonzepts, ähnlich dem des ROP, in bestehende Programme gesucht und deren Umsetzung, ebenfalls in Form eines Refactorings, beschrieben. Durch eine Gegenüberstellung beider Ansätze wird dabei geklärt, welcher von beiden in Bezug auf Anwendbarkeit, Komplexität und Korrektheit für den praktischen Einsatz besser geeignet ist.

2 Problemstellung

In diesem Kapitel wird die dieser Arbeit zu Grunde liegende Problemstellung vorgestellt. Zunächst werden die hierfür verwendeten Begriffe definiert. Im zweiten Abschnitt werden dann verschiedene Möglichkeiten einander gegenübergestellt, Rollenkonzepte in objektorientierten Programmiersprachen umzusetzen. Dabei stellt sich das Role Object Pattern als vielversprechender Ansatz heraus, welcher in Abschnitt 2.3 näher erläutert wird. Im abschließenden Abschnitt 2.4 wird das aus den vorangegangenen Abschnitten abgeleitete Ziel dieser Arbeit beschrieben: Die Erstellung eines automatisierten Refactoring-Werkzeugs zur Transformation von Vererbung(shierarchien) in das ROP.

2.1 Begriffe

„In der objektorientierten Software-Entwicklung ist ein Objekt ein individuelles Exemplar von Dingen (z.B. Roboter, Auto), Personen (z.B. Kunde, Mitarbeiter) oder Begriffen der realen Welt (z.B. Bestellung) oder Vorstellungswelt (z.B. juristische und natürliche Personen). Ein Objekt besitzt einen bestimmten Zustand und reagiert mit einem definierten Verhalten auf seine Umgebung. Außerdem besitzt jedes Objekt eine Objekt-Identität, die es von allen anderen Objekten unterscheidet.“ [Balzert1999].

An den, in der oben zitierten Definition, genannten Beispielen lässt sich das in Abschnitt 1.1 angesprochene Manko klassenbasierter objektorientierter Programmiersprachen erkennen, keine kontextabhängigen Sichten auf Objekte zu unterstützen: Ein Objekt ist entweder Kunde oder Mitarbeiter, kann jedoch nicht in unterschiedlichen Kontexten verschiedene solcher Rollen (z.B. Unternehmen A gegenüber als Kunde und Unternehmen B gegenüber als Mitarbeiter) annehmen. Die Betrachtungsweise, dass ein und dieselbe Person in einem Kontext die Rolle Mitarbeiter und in einem anderen Kontext die Rolle Kunde annimmt bzw. – anders ausgedrückt – in den jeweiligen Kontexten über die entsprechenden Sichten betrachtet wird, ist gemäß der o.g. Definition nicht vorgesehen, obwohl solche Konstellationen in der realen Welt ständig vorkommen.

Zur besseren Unterscheidung und Abgrenzung von Objekten im Sinne der obigen Definition, wird im Rahmen dieser Arbeit der Begriff „Subjekt“ verwendet. Ein Subjekt ist ein Objekt, welches über alle der oben genannten Eigenschaften verfügt und darüber hinaus in der Lage ist, Rollen dynamisch anzunehmen und abzulegen. Dabei kann es gleichzeitig mehrere unterschiedliche Rollen spielen und auch mehrere Instanzen desselben Rollentyps annehmen. Der Teil eines Subjekts, der dessen Identität trägt sowie den rollenunabhängigen Zustand verwaltet und das rollenübergreifende Verhalten des Subjekts definiert, wird im Folgenden „Subjektkern“ genannt. Zustand und Verhalten, die dort definiert sind, werden dementsprechend als „Kernzustand“ und „Kernverhalten“ bezeichnet.

Eine Rolle ist stets Bestandteil genau eines Subjekts. Sie stellt eine kontextspezifische Sicht auf das Subjekt dar und kann dieses um zusätzlichen rollenspezifischen Zustand und ebensolches Verhalten erweitern. Im Gegensatz zu Subjekten und Objekten verfügen Rollen über keine eigene Identität, sondern sind an die Identität des Subjekts gebunden, von dem sie Bestandteil sind. Alle Rollen eines Subjekts teilen sich neben dessen Identität auch dessen rollenunabhängigen Zustand und rollenunabhängiges Verhalten.

Da ein Subjekt zwar eine logische Einheit darstellt, im Rahmen objektorientierter Programmiersprachen jedoch nur in Form eines Konglomerats mehrerer Objekte umgesetzt werden kann, wird der Begriff „logische Identität“ oder alternativ auch „Subjektidentität“ für die Identität der logischen Einheit Subjekt verwendet, um diese gegenüber der Identitäten der Objekte abzugrenzen, aus denen das jeweilige Subjekt besteht.

2.2 Umsetzung von Rollenkonzepten

Das Konzept der Rolle als Bestandteil und Mittel der Datenmodellierung findet bereits im Rahmen der unterschiedlichen Datenmodellierungskonzepte von [Codd1970], [Falkenberg1976] und [Bachman1977] Erwähnung. Einen ausführlichen Überblick über die Herkunft des Begriffs Rolle und die Verwendung von Rollen in der formalen Modellierung bietet [Steimann2000]. Grundlage für die hier vorliegende Arbeit ist jedoch vor allem die Frage nach Möglichkeiten der Umsetzung eines Rollenkonzepts mit den Mitteln existierender objektorientierter Sprachen (insbesondere Java).

Eine ausführliche Gegenüberstellung möglicher Lösungsansätze für dieses Problem liefert [Fowler1997]. Insgesamt werden dort fünf verschiedene Vorgehensweisen beschrieben, Rollenkonzepte in objektorientierten Programmiersprachen umzusetzen, wobei diese sich bezüglich ihrer Komplexität sowie ihrer jeweiligen Vorteile und Einschränkungen deutlich voneinander unterscheiden.

Die ersten beiden Möglichkeiten – Single Role Type und Separate Role Types – stellen dabei gar keine Rollenkonzepte im eigentlichen Sinne dar. Denn sie unterscheiden nicht zwischen Subjekten, die Rollen spielen könnten, und Rollen, die von Subjekten angenommen werden könnten.

Single Role Type kann in Situationen verwendet werden, in denen sich die angedachten Rollen in ihrem Zustand und Verhalten nicht – oder zumindest nur sehr geringfügig – unterscheiden (z.B. wenn in einem Programm Personen mit unterschiedlichen Berufen verwaltet werden, zwischen denen über die einfache Angabe einer Berufsbezeichnung hinaus jedoch keine weitere Differenzierung in Bezug auf Zustand und Verhalten erforderlich ist). Bei dieser Vorgehensweise wird nur ein einziger Typ benötigt, der Zustand und Verhalten des Subjekts inklusive aller seiner Rollen implementiert und die Information darüber verwaltet, welche Rolle(n) er gerade spielt.

Separate Role Types kann eingesetzt werden, wenn die einzelnen Rollen keine – oder zumindest kaum – Gemeinsamkeiten in Zustand und Verhalten aufweisen. In diesem Fall werden die einzelnen Rollen als voneinander unabhängige Typen definiert, die über keinerlei Beziehung zu irgendeiner Art von gemeinsamen Subjekt verfügen.

Deutlich interessanter ist die Betrachtung der verbleibenden drei Ansätze: Role Subtype, Role Object und Role Relationship. Da die beiden letztgenannten sich lediglich darin unterscheiden, dass Role Relationship auch den Kontext einbezieht, innerhalb dessen Subjekte Rollen spielen können, wird im Folgenden auf eine separate Betrachtung dieser Variante verzichtet. Die Aussagen, welche in Bezug auf die Variante Role Object getroffen werden, gelten gleichermaßen für Role Relationship. Es verbleiben also die Ansätze Role Subtype und Role Object.

Role Subtype bezeichnet die Vorgehensweise, Gemeinsamkeiten der einzelnen Rollen in einem Supertyp zu sammeln, von dem die einzelnen Rollen jeweils als Subtypen abgeleitet sind. Soll ein Subjekt nach diesem Ansatz mehrere Rollen gleichzeitig spielen, so müsste es gleichzeitig Instanz aller betroffenen Rollensubtypen sein und darüber hinaus noch die entsprechenden Typen dynamisch annehmen und ablegen können. Da diese beiden Erfordernisse von den gängigen objektorientierten Programmiersprachen nicht direkt erfüllt werden können, skizziert [Fowler1997] verschiedene Möglichkeiten, diesen Ansatz trickreich so zu implementieren, dass er dennoch das gewünschte Verhalten zeigt.

Der Ansatz Role Object bezieht sich auf das Role Object Pattern nach [Bäumer1997] und wird in Abschnitt 2.3 ausführlich vorgestellt.

Beide Ansätze werden nun in Hinblick auf die im vorangegangenen Abschnitt bereits erwähnten Eigenschaften von Rollenkonzepten nach [Steimann1999] verglichen:

- 1. Rollen verfügen über einen eigenen Zustand und eigenes Verhalten*
Ist in beiden Fällen gegeben, da die Rollen jeweils als eigene Typen vorliegen.
- 2. Rollen bestehen nur im Rahmen von Beziehungen zwischen Subjekt und Kontext*
Gilt in beiden Fällen: Technisch betrachtet, hören Rolleninstanzen zwar nicht auf zu existieren, wenn sie gerade in keinem Kontext Verwendung finden, die Beziehung von Kontext zu Subjekt erfolgt jedoch stets über eine Instanz der Rolle(n), die das Subjekt in diesem Kontext spielt.
- 3. Ein Objekt (Subjekt) kann unterschiedliche Rollen gleichzeitig spielen*
Ist in beiden Fällen möglich: Bei Role Object ist die Möglichkeit bereits inhärent im Konzept verankert, bei Role Subtype lässt es sich je nach konkreter Umsetzung einrichten.
- 4. Ein Objekt (Subjekt) kann die gleiche Rolle mehrfach zur selben Zeit spielen*
Role Subtype: Nicht möglich, da Clients nicht zwischen mehreren Instanzen der gleichen Rolle differenzieren können.
Role Object: In Abhängigkeit von der konkreten Implementierung grundsätzlich möglich.
- 5. Ein Objekt (Subjekt) kann Rollen dynamisch annehmen und ablegen*
Bei beiden Ansätzen möglich.
- 6. Die Reihenfolge, in der Rollen angenommen und abgelegt werden können, kann Einschränkungen unterliegen*
Role Subtype: In Abhängigkeit von der konkreten Implementierung grundsätzlich möglich.

Role Object: Abhängigkeiten und Constraints zwischen Rollen sind zwar grundsätzlich möglich, erfordern unter Umständen aber hohen Aufwand bei der konkreten Umsetzung und reduzieren die Lesbarkeit und Wartbarkeit des resultierenden Programms.

7. *Objekte (Subjekte) voneinander unabhängiger Typen können die gleichen Rollen spielen*

In beiden Fällen nicht möglich, da die Rollen in diesem Fall Subtypen beider Subjekttypen sein müssten.

8. *Rollen können Rollen spielen*

Role Subtype: Grundsätzlich nicht vorgesehen, je nach Umsetzung aber machbar.

Role Object: Bei hierarchisch geschachtelter Anwendung des Role Object Pattern möglich (s. Abbildung 2 in Abschnitt 2.3).

9. *Rollen können von einem Objekt (Subjekt) an ein anderes übertragen werden*

Role Subtype: Nicht möglich.

Role Object: Ursprünglich nicht vorgesehen, je nach konkreter Umsetzung aber möglich (s. Kapitel 3 und 4)

10. *Der Zustand eines Objekts (Subjekts) kann rollenspezifisch sein*

In beiden Fällen grundsätzlich nicht möglich, da der Zustand des Subjekts zentral für alle Rollen verwaltet wird. Allerdings bieten beide Ansätze jeweils die Möglichkeit, dass der Zustand des Subjekts dem jeweiligen Client gegenüber in Abhängigkeit von der Rolle, über die er darauf zugreift, unterschiedlich dargestellt wird.

11. *Eigenschaften (Verhalten und Zustand) eines Objekts (Subjekts) können rollenspezifisch sein*

In beiden Fällen möglich.

12. *Rollen können den Zugriff auf das Subjekt beschränken*

Role Subtype: Nicht möglich, da stets die gesamte öffentliche Schnittstelle des Subjekts zur Verfügung steht.

Role Object: Auch hier steht prinzipiell immer die Schnittstelle des Subjektkerns und der vom Client verwendeten Rolle zur Verfügung, allerdings können Rollen den Zugriff auf Zustand und Verhalten des Subjektkerns selektiv unterbinden.

13. *Verschiedene Rollen können Struktur (inkl. Zustand) und Verhalten teilen*

In beiden Fällen möglich.

14. *Ein Objekt (Subjekt) und seine Rollen teilen sich eine einzige gemeinsame Identität*

Role Subtype: Ist gegeben, da der Client unabhängig von der verwendeten Rolle immer auf dasselbe Objekt zugreift.

Role Object: Eine gemeinsame Identität existiert zunächst nicht. Es lässt sich jedoch eine gemeinsame logische Identität implementieren, welche an den Kern des Subjekts gebunden ist und von Clients geprüft werden kann.

15. Ein Objekt (Subjekt) und seine Rollen haben verschiedene Identitäten

Role Subtype: Aufgrund der zu Punkt 14 genannten Umstände ist dies nicht der Fall.

Role Object: Da es sich bei Subjekt(-Kern) und Rollen um unterschiedliche Objekte handelt, verfügen diese auch jeweils über eine eigene Identität.

Tabelle 1 enthält eine kompakte Übersicht aller in [Fowler1997] beschriebenen Vorgehensweisen zur Implementierung von Rollenkonzepten in objektorientierten Sprachen in Hinblick auf deren Erfüllung der o.g. Eigenschaften. Die Ziffern in der linken Spalte entsprechen dabei der Nummerierung der Eigenschaften in diesem Abschnitt. Ein „ja“ in der Tabelle bedeutet dabei, dass der entsprechende Ansatz die Eigenschaft aufweist. „Nein“ besagt, dass die entsprechende Eigenschaft nicht unterstützt wird. „Möglich“ heißt, dass die Eigenschaft in der Grundform des Ansatzes nicht erfüllt wird, durch moderate Anpassungen jedoch umgesetzt werden kann, während „eingeschränkt“ bedeutet, dass erhebliche Anpassungen erforderlich wären, um die entsprechende Eigenschaft bieten zu können.

	Single Role Type	Separate Role Types	Role Subtype	Role Object	Role Relationship
1	nein	ja	ja	ja	ja
2	nein	nein	möglich	ja	ja
3	ja	nein	möglich	ja	ja
4	nein	nein	nein	ja	ja
5	ja	nein	ja	ja	ja
6	ja	nein	möglich	eingeschränkt	eingeschränkt
7	nein	ja	nein	nein	nein
8	nein	nein	eingeschränkt	ja	ja
9	nein	nein	nein	möglich	möglich
10	nein	nein	nein	nein	nein
11	nein	nein	ja	ja	ja
12	nein	nein	nein	eingeschränkt	eingeschränkt
13	ja	nein	ja	ja	ja
14	ja	unzutreffend	ja	möglich	möglich
15	nein	unzutreffend	nein	ja	ja

Tabelle 1: Eigenschaften von Rollenkonzepten für objektorientierte Sprachen

2.3 Das Role Object Pattern

Wie in Abschnitt 2.2 gezeigt, bietet das Role Object Pattern nach [Bäumer1997] im Vergleich zu den übrigen in [Fowler1997] genannten Möglichkeiten zum Umgang mit Rollen in objektorientierten Sprachen die größte Abdeckung der in [Steimann1999] dargestellten Eigenschaften von Rollenkonzepten. Da das Pattern in [Bäumer1997] ausführlich beschrieben ist, beschränkt sich dieser Abschnitt auf eine kurze Zusammenfassung.

Ziel des Pattern ist es, unterschiedlichen Clients (gemeint sind damit Softwaresysteme oder -komponenten) eines objektorientierten Softwaresystems kontextabhängige Sichten (Rollen), die jeweils über einen eigenen Zustand und eigenes Verhalten verfügen, auf Kernelemente der Anwendungsdomäne dieses Systems (Subjekte) zu bieten, ohne dabei die Identität dieser Elemente zu spalten. Das heißt, dass unabhängig von der jeweiligen Sicht über die ein Client auf eine bestimmte Instanz eines solchen Elements zugreift, deren logische Identität (Subjektidentität) stets dieselbe sein muss.

Das Ziel des Pattern wird erreicht, indem jedes Subjekt nicht durch ein einziges, sondern durch mehrere Objekte verkörpert wird: Einen Subjektkern (core object, im Diagramm `SubjectCore`) und beliebig viele Rollen (role objects, daher der Name des Pattern). Kern und Rollen implementieren dabei dieselbe Schnittstelle, die Zugriff auf das Verhalten des Kerns erlaubt, welches so unabhängig von der konkreten Rolle, über die ein Client das Subjekt betrachtet, zur Verfügung steht. Error: Reference source not found stellt die grundlegende Struktur des ROP in Form eines UML Klassendiagramms dar.

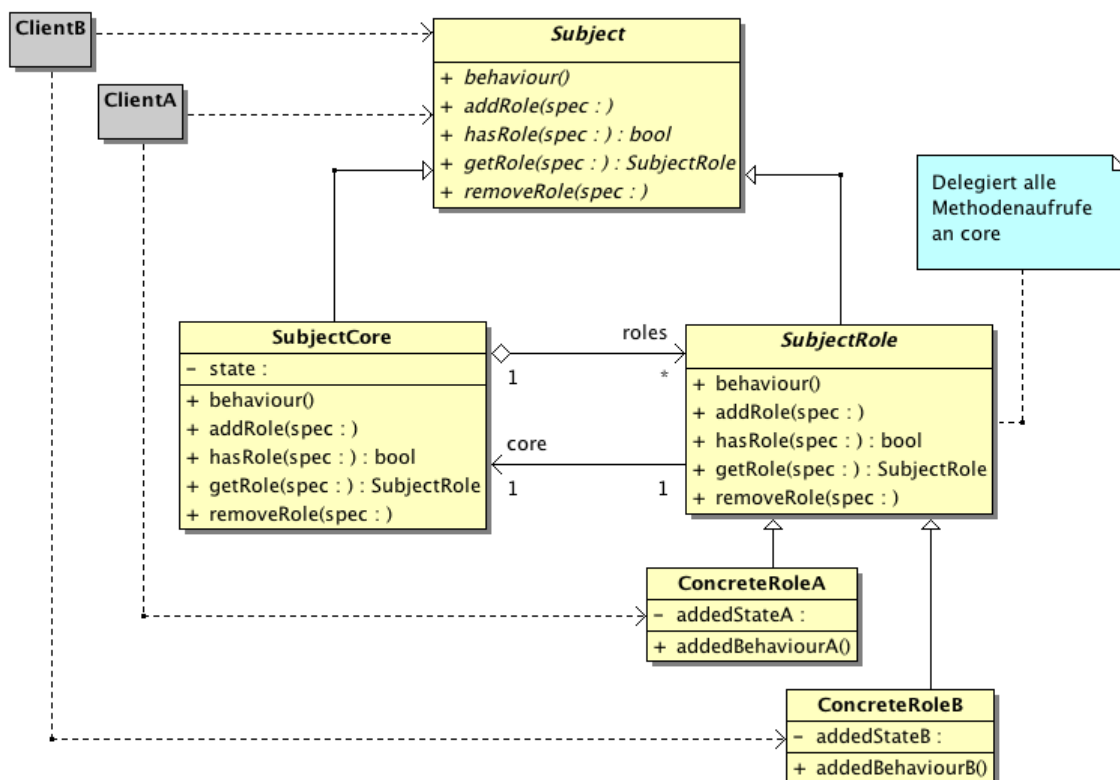


Abbildung 1: Struktur des Role Object Pattern nach [Bäumer1997]

Der abstrakte Typ `Subject` definiert die gemeinsame Schnittstelle der Objekte, die in Summe ein Subjekt mit einer individuellen logischen Identität bilden. Dazu gehören auch die Operationen zur Verwaltung, also dem Hinzufügen, Entfernen und Abfragen der Rollen, die das Subjekt annehmen kann. Diese gemeinsame Schnittstelle wird sowohl von `SubjectCore` als auch von `SubjectRole` implementiert.

Jede Instanz von `SubjectCore` bildet den Kern eines entsprechenden Subjekts. Daher wird hier der Kernzustand des Subjekts verwaltet, der unabhängig von den zu einem gegebenen Zeitpunkt angenommenen Rollen ist. Auch der rollenunabhängige Teil des Verhaltens des Subjekts (Kernverhalten) ist in `SubjectCore` implementiert. Darüber hinaus findet hier die Verwaltung und – auf Anforderung eines Clients auch – die Erzeugung der Rollen des Subjekts statt.

Der abstrakte Typ `SubjectRole` fungiert als Supertyp aller konkreten Rollen, die das Subjekt annehmen kann. Hier wird die Verbindung zum Kern des Subjekts gehalten. Zustände, in denen eine Rolle keine Verbindung zum Kern des Subjekts hat, sind ungültig. Die Hauptaufgabe von `SubjectRole` ist die Delegation aller Methodenaufrufe, die entweder das gemeinsame Kernverhalten oder die Verwaltung der Rollen des Subjekts betreffen, an den Kern des Subjekts, wo die entsprechenden Implementierungen vorliegen. Durch diese Delegation ist sichergestellt, dass alle Rollen eines Subjekts stets auf denselben gemeinsamen Kernzustand und dasselbe gemeinsame Kernverhalten zugreifen.

Die Subtypen von `SubjectRole` sind die konkreten kontextabhängigen Rollen über die verschiedene Clients auf das Subjekt zugreifen. Clients, die keine spezifische Sicht auf das Subjekt benötigen, können unter Verwendung der in `Subject` definierten Schnittstelle auch direkt auf den Kern des Subjekts zugreifen. Jede konkrete Rolle kann das über `SubjectRole` geerbte Verhalten nach ihren jeweiligen kontextabhängigen Bedürfnissen anpassen (durch Überschreiben) und/oder ergänzen. Des Weiteren können Rollen auch einen eigenen spezifischen Zustand in Ergänzung zum Kernzustand des Subjekts verwalten.

Neben der Beschreibung der Vor- und Nachteile, die sich aus dieser Struktur ergeben, gibt [Bäumer1997] eine Reihe von Hinweisen zur optimalen Implementierung des Pattern. Diese Hinweise werden größtenteils in Kapitel 3 aufgegriffen, unter Berücksichtigung der speziellen Anforderungen, die sich aus den Überlegungen in den Abschnitten 2.4 und 2.5 sowie der für die Implementierung gewählten Umgebung ergeben.

An dieser Stelle sei darauf hingewiesen, dass das Role Object Pattern auch hierarchisch angewendet werden kann, indem eine konkrete Rolle als eigenständiges Subjekt fungiert. [Bäumer1997] nennt als Beispiel für eine solche Konstellation die Kunden (Customers) einer Bank, welche sowohl die Rolle des Investors als auch die des Schuldners (Borrower) annehmen können. Kunde ist dabei wiederum eine Rolle, welche Personen im Geschäftsverkehr mit der Bank annehmen können. Eine andere Rolle, die Personen, welche mit der Bank interagieren, annehmen können, ist die des Bürgen (Guarantor), wobei eine Person weder Kunde sein muss, um als Bürge auftreten zu können, noch umgekehrt. Dieses Beispiel ist in Abbildung 2 dargestellt:

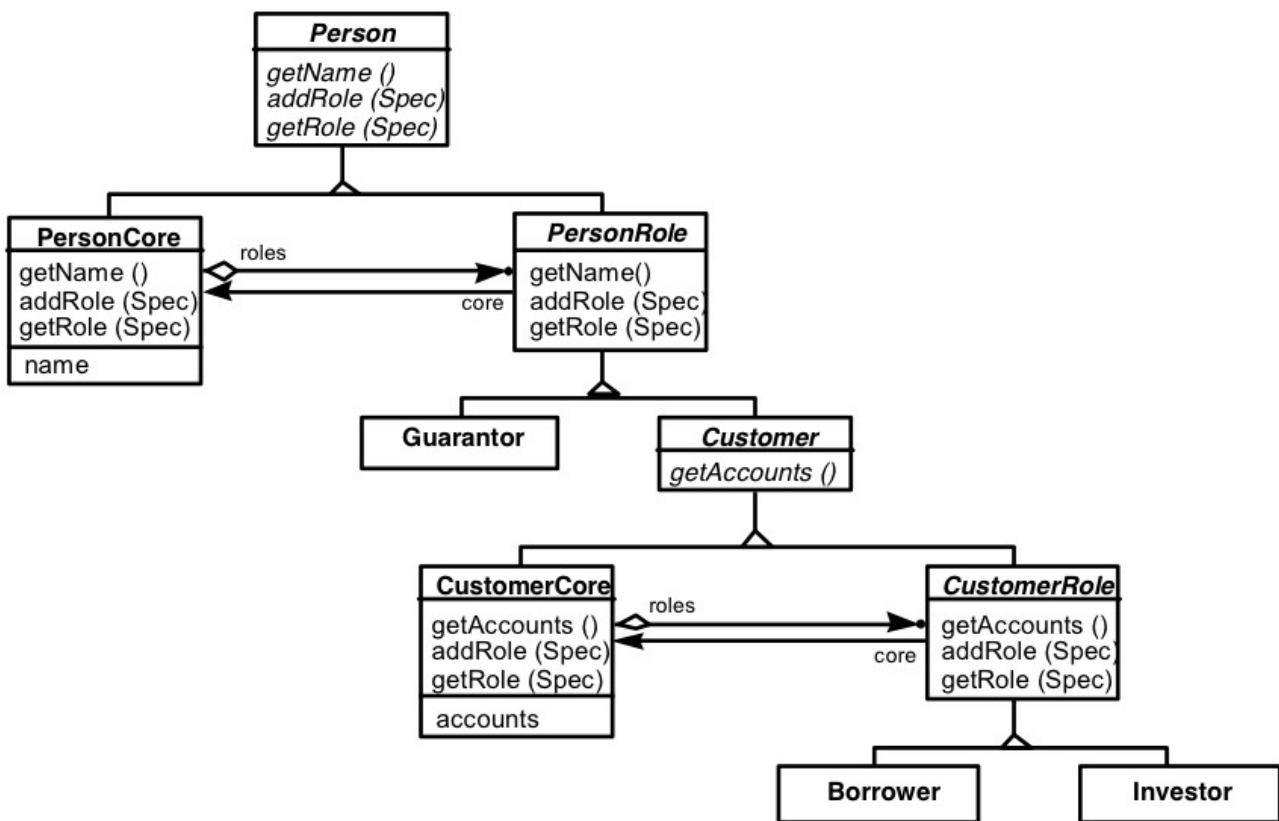


Abbildung 2: Anwendungsbeispiel für das Role Object Pattern (Quelle: [Bäumer1997])

2.4 Refactoring to Role Objects

Wie in Abschnitt 2.2 gezeigt, sind die in den meisten der verbreiteten objektorientierten Programmiersprachen zur Verfügung stehenden Mittel, Rollenkonzepte zumindest ansatzweise umzusetzen (namentlich Single Role Type, Separate Role Types und Role Subtype), in Hinblick auf die – im selben Abschnitt beschriebenen – wünschenswerten Eigenschaften solcher Konzepte, nicht übermäßig befriedigend. Dennoch mag man in den frühen Phasen eines Entwicklungsprojekts geneigt sein, Ansätze wie Role Subtype – was letztendlich nicht mehr ist, als das Ausnutzen von Subtyping mit Vererbung, welches ein zentrales Konzept der hier angesprochenen Sprachen darstellt – zu verfolgen, da diese leicht zu verstehen, zu modellieren und zu implementieren sind.

Stellt sich im weiteren Verlauf der Entwicklung dann heraus, dass dieser zunächst gewählte Ansatz den Anforderungen an das zu entwickelnde System nicht gerecht werden kann, ist eine Anpassung der Struktur des zu entwickelnden Systems zu mächtigeren Konzepten wie dem Role Object Pattern hin nötig. Dies kann zum Beispiel auf eine Änderung der ursprünglichen Anforderungen während der Entwicklung zurückzuführen sein, was, den Befürwortern agiler Entwicklungsmethoden nach, eher die Regel als die Ausnahme darstellt (s. [Fowler2000] und [Ambler2002]).

Bei dieser Änderung der Struktur des zu entwickelnden Programms sollte das bereits implementierte Verhalten möglichst unverändert bleiben. Änderungen der Struktur von Programmen, bei denen deren beobachtetes Verhalten unverändert bleibt, werden „Refactoring“

genannt und in [Opdyke1993] sowie [Fowler1999] ausführlich beschrieben. Da die manuelle Anwendung komplexer Refactorings, wie des hier angedachten, sowohl aufwändig als auch fehleranfällig ist, wäre eine Möglichkeit von Nutzen, diese Aufgabe automatisiert durchzuführen.

Refactorings, die Entwurfsmuster (Patterns), wie die aus [Gamma1995] bekannten, zum Ziel haben, sind keine neue Idee. Zusammenhänge zwischen Patterns und Refactorings werden schon von [Roberts1999] beschrieben und [Kerievsky2004] widmet sich vollständig dem Thema Entwurfsmuster als Zielzustand von Refactorings. Das Role Object Pattern ist aus [Bäumer1997] bekannt und findet unter anderem in ObjectTeams/Java (s. [Herrmann2007]) Verwendung. Ein automatisiertes Refactoring, welches Vererbungshierarchien im Sinne des Role Subtype Ansatzes in das ROP überführt, existiert allerdings noch nicht. Dieses Refactoring wird im Folgenden als „Introduce Role Object Pattern“ (IROP) bezeichnet.

Beschrieben wird das Refactoring für die Sprache Java in der Version 1.5, da diese weit verbreitet ist und Entwicklungswerkzeuge für sie frei verfügbar sind. Eine Adaptation auf andere Sprachen unter Berücksichtigung von deren spezifischen Eigenschaften ist möglich, wird im Rahmen dieser Arbeit aber nicht betrachtet. Die Implementierung des Refactorings erfolgt auf Basis der Entwicklungsumgebung Eclipse in der Version 3.4.2 und der dazugehörigen Java Development Tools (JDT), da diese umfangreiche Möglichkeiten zur Integration von Plug-ins – insbesondere auch zum Refactoring bestehenden Java Quellcodes – beinhaltet und ebenfalls frei verfügbar ist.

Die Ausgangssituation für die Anwendung des IROP Refactorings ist eine Vererbungshierarchie aus einer Superklasse und mindestens einer direkten Subklasse dieser Superklasse. Im Sinne des Role Subtype Ansatzes aus [Fowler1997] ist die Superklasse dabei als Subjekt und ihre direkten Subklassen als Rollen zu betrachten.

Ziel des Refactorings ist, diese Ausgangssituation in das Role Object Pattern zu überführen, wobei die konkrete Umsetzung des Pattern so zu wählen ist, dass die in Abschnitt 2.3 beschriebenen Eigenschaften von Rollenkonzepten möglichst vollständig unterstützt werden. Die öffentliche Schnittstelle der Superklasse wird zur neuen gemeinsamen Schnittstelle des Subjektkerns und der Rollen des Subjekts. Eigenschaften und Verhalten der Subklassen werden 1:1 von den neuen Rollen umgesetzt.

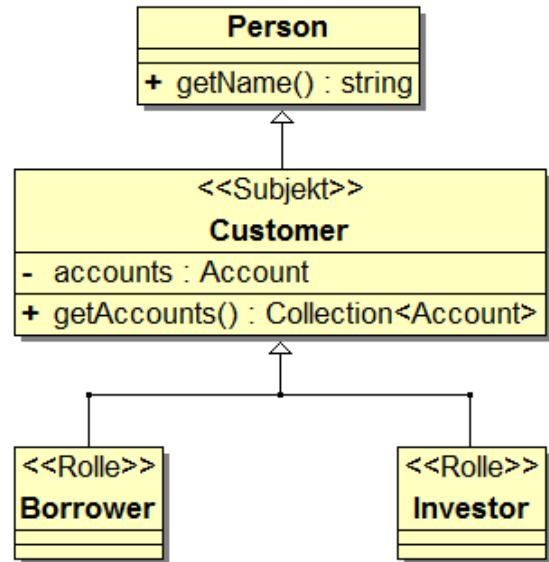


Abbildung 3: Beispiel: Ausgangssituation

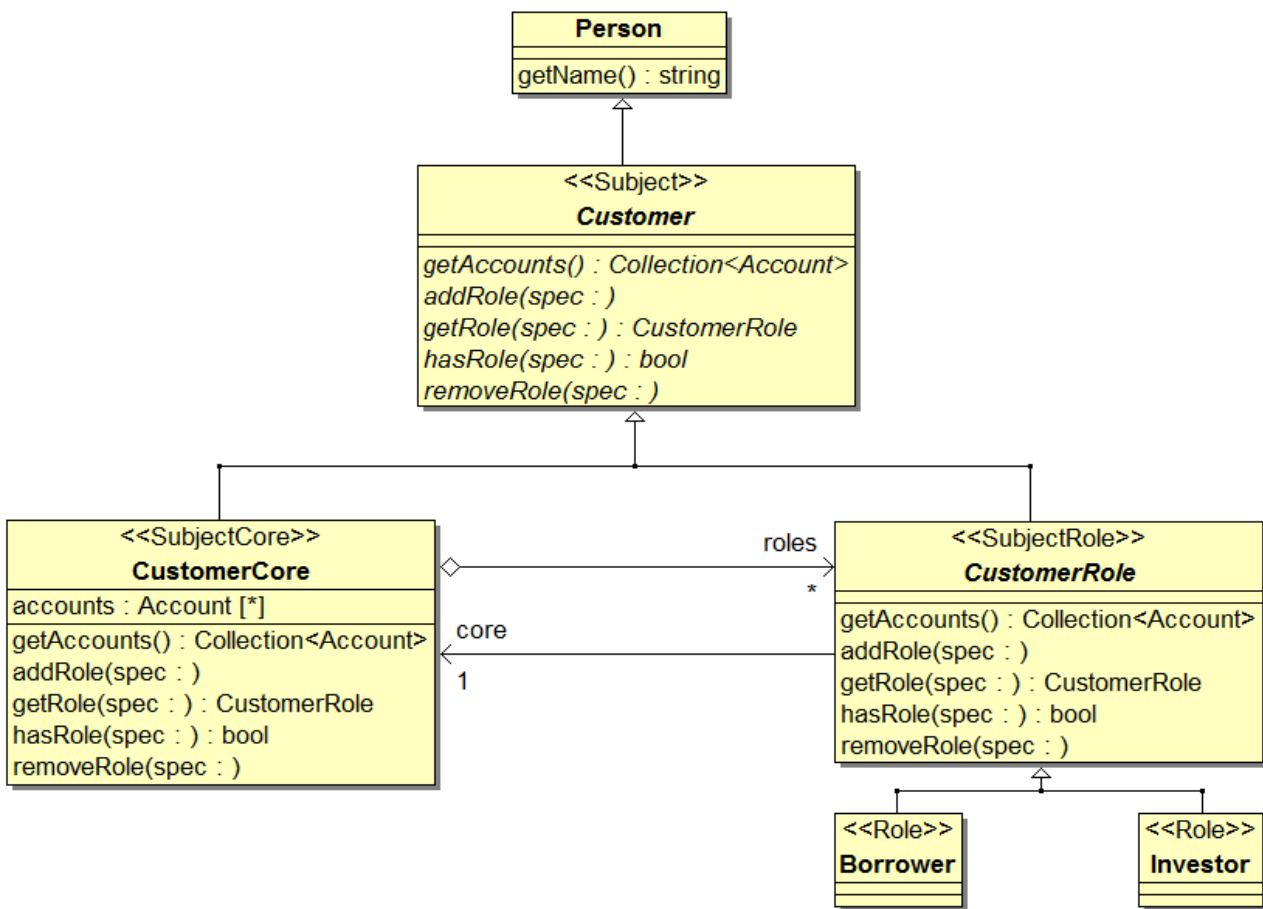


Abbildung 4: Beispiel: Zielzustand des Refactorings

Die Ausgangssituation birgt dabei das Problem, dass alle Instanzen des Subjekttyps und der Rollentypen grundsätzlich voneinander unabhängig sind, selbst wenn sie dasselbe Individuum der Anwendungsdomäne des zu modifizierenden Programms repräsentieren. Eine Überführung dieses Zustands in das Role Object Pattern ist im Rahmen eines Refactorings nicht möglich. Erstens ist dabei im Allgemeinen nicht erkennbar, ob zwei oder mehr dieser Instanzen dasselbe Individuum repräsentieren und zweitens stellt die Zusammenführung voneinander unabhängiger Objekte in ein einzelnes Subjekt in der Regel eine (für ein Refactoring) unzulässige Änderung der Bedeutung des zu modifizierenden Programms dar. Streng genommen bedeutet diese Unmöglichkeit, dass die Einführung des ROP in bestehenden Code durch ein Refactoring nicht vollständig erbracht werden kann. Denn entweder weist das Ergebnis das o.g. Problem auf, dann entspricht es zwar von der Struktur her dem ROP, setzt jedoch dessen Semantik nicht um oder die Transformation muss die Bedeutung des zu modifizierenden Programms entsprechend anpassen, darf dann aber nicht mehr Refactoring genannt werden.

[Steimann2010] weißt allerdings darauf hin, dass Refactorings häufig vorbereitend eingesetzt werden, um zunächst die Struktur von Programmen so zu modifizieren, dass die gewünschten Änderungen der Bedeutung dieser Programme anschließend in wenigen, möglichst einfachen Schritten erfolgen können. Diesem Gedanken folgend, handelt es sich bei dem in Kapitel 3

beschriebenen IROP Refactoring um eine Transformation, welche lediglich die Struktur der betroffenen Programmelemente von einer Ausgangssituation entsprechend dem Role Subtype Ansatz aus [Fowler1997] (s. Abbildung 3) in das ROP nach [Bäumer1997] (s. Abbildung 4) überführt. Um eine vollständige Umsetzung des Role Object Patterns zu erreichen, muss der Anwender des Refactorings nach dessen Durchführung selbst dafür Sorge tragen, voneinander unabhängige Objekte, welche dasselbe Individuum der Anwendungsdomäne repräsentieren, zu einem einzelnen Subjekt mit gemeinsamem Kernzustand und -verhalten sowie einer gemeinsamen logischen Identität zusammenzuführen.

3 Lösungsansatz Role Object Pattern

Dieses Kapitel beschreibt den Ansatz, die im vorangegangenen Kapitel beschriebene Aufgabe durch Verkettung einer Reihe bereits bekannter Refactorings zu lösen. Anschließend wird auf die besonderen Probleme eingegangen, die sich zum Einen daraus ergeben, dass das Role Object Pattern im Rahmen eines Refactorings eingeführt werden soll, wobei das beobachtbare Verhalten des zu modifizierenden Programms nicht verändert werden darf und zum Anderen den spezifischen Eigenschaften der Programmiersprache Java geschuldet sind. Für jedes dieser Probleme wird zwar eine Lösung vorgestellt, in Summe führen sie jedoch dazu, dass das erzielbare Ergebnis des Refactorings in Bezug auf die Kriterien Einfachheit, Lesbarkeit, Verständlichkeit und Wartbarkeit nicht unbedingt für befriedigend befunden werden kann.

Aus dieser eher unbefriedigenden Lösung lässt sich jedoch ein alternativer Ansatz ableiten, welcher den oben genannten Anforderungen besser gerecht wird, ohne dass bezüglich der Kriterien, welche in Abschnitt 2.2 genannt wurden, gegenüber dem ersten Ansatz Abstriche zu verzeichnen sind. Dieser alternative Ansatz wird unter dem Namen „Introduce Lightweight Role Objects“ (ILRO) in Kapitel 4 vorgestellt. Die mit diesen beiden Ansätzen erzielbaren Ergebnisse werden einander in den Kapiteln 6 und 7 gegenübergestellt.

3.1 Grundsätzliches Vorgehen

Auch wenn die automatisierte Transformation einer Vererbungshierarchie, die dem Gedanken des Role Subtype Ansatzes aus [Fowler1997] entspricht, in eine Struktur gemäß dem Role Object Pattern nach [Bäumer1997] auf den ersten Blick nicht übermäßig kompliziert erscheint, so gilt es dabei doch, eine ganze Reihe an Fallstricken zu umgehen. Diese rühren hauptsächlich daher, dass das Pattern als Entwurfsmuster darauf ausgelegt ist, bereits während des Entwurfs eines Softwaresystems eingesetzt zu werden, so dass das gesamte Design des Systems darauf ausgerichtet werden kann, die Schwachstellen dieses Musters zu berücksichtigen oder, wo nötig, zu umgehen. Da bei der Anwendung des IROP Refactorings jedoch nicht davon ausgegangen werden kann, dass beim Entwurf des zu modifizierenden Programms Rücksicht auf die Belange des anvisierten Pattern genommen wurde, müssen dabei einige Anpassungen im Vergleich zu dessen ursprünglicher Form vorgenommen werden.

Die beiden Hauptziele dieser notwendigen Anpassung sind:

1. Das angepasste ROP muss robust genug sein, seinen Zweck auch in einer Umgebung zu erfüllen, die nicht von vornherein darauf eingerichtet ist.
2. Das angepasste Pattern muss, gemäß der Grundanforderung an jedes Refactoring, nur die Struktur, nicht jedoch die Bedeutung eines Programms zu verändern, den Clients des transformierten Subjekts gegenüber dasselbe Verhalten aufweisen, wie es vor der Transformation der Fall war.

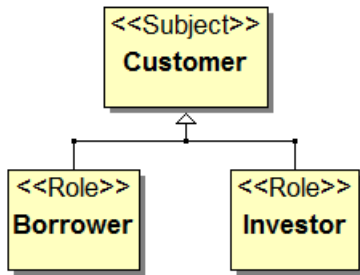


Abbildung 5:
Grundsätzliches Vorgehen:
Ausgangssituation

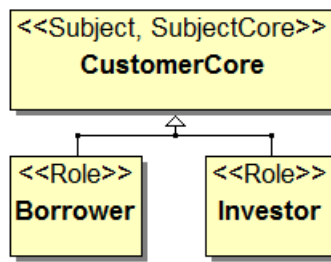


Abbildung 6:
Grundsätzliches Vorgehen:
Schritt 1

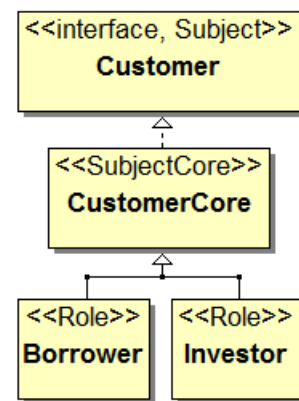


Abbildung 7: Grundsätzliches
Vorgehen: Schritt 2

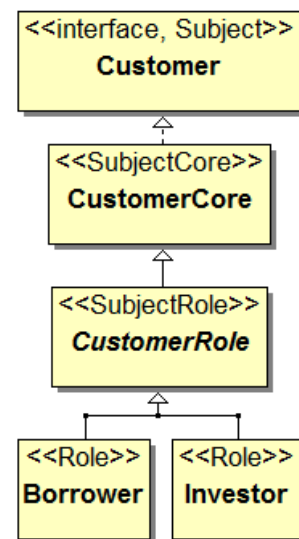


Abbildung 8: Grundsätzliches
Vorgehen: Schritt 3

Die Abbildungen 5 bis 10 zeigen, wie sich die gewünschte Transformation einer Vererbungshierarchie in das Role Object Pattern durch Anwendung mehrerer Teilrefactorings nacheinander realisieren lässt. Die Darstellung erfolgt schematisch vereinfacht anhand des Beispiels aus Abschnitt 2.4. Die im Einzelnen durchzuführenden Teilschritte sind:

1. Umbenennen von `Subject` in `SubjectCore` mittels des Refactorings „Rename Class“ (s. [Fowler1999]).
2. Anwenden des Refactorings „Extract Interface“ (s. [Fowler1999] und [Tip2003]) auf `SubjectCore`. Dabei erhält das extrahierte Interface den ursprünglichen Namen von `Subject` und wird in allen Typdeklarationen an Stelle des Subjektkerns verwendet.
3. Einfügen einer neuen abstrakten Klasse `SubjectRole` als gemeinsame Oberklasse aller Rollen und Subklasse von `SubjectCore`. Da diese Klasse zunächst leer bleibt, hat sie keinen Einfluss auf das Verhalten des Programms.
4. Anwenden des Refactorings „Replace Inheritance with Delegation“ (s. [Fowler1999] und [Kegel2008]) auf `SubjectRole`.

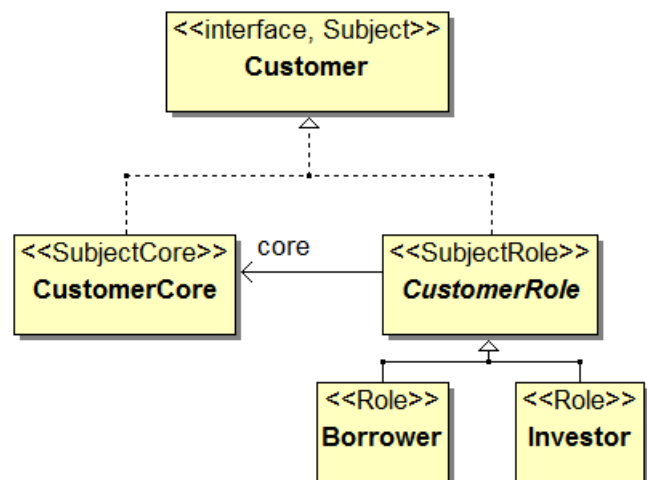


Abbildung 9: Grundsätzliches Vorgehen: Schritt 4

5. Hinzufügen der Rollenmanagementmethoden des ROP zu Subject. Die Implementierung dieser Methoden erfolgt in der Klasse SubjectCore, welche zusätzlich eine Collection „roles“ zur Verwaltung der Rollen erhält. SubjectRole implementiert die Rollenmanagementmethoden ebenfalls, allerdings lediglich durch Delegation an SubjectCore. Da die hinzugefügten Methoden im bestehenden Code (noch) nicht verwendet werden, beeinflussen sie das Verhalten des Programms nicht.

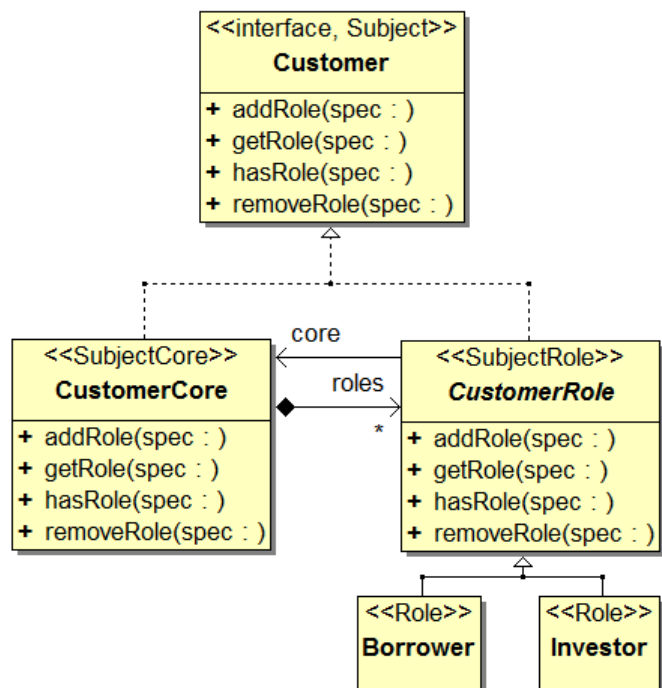


Abbildung 10: Grundsätzliches Vorgehen: Schritt 5

Der hier beschriebene Ansatz weist eine Abweichung in der Umsetzung des ROP gegenüber [Bäumer1997] auf: Die gemeinsame Schnittstelle des Subjekts ist dort als abstrakte Klasse beschrieben, während hier stattdessen ein Interface zum Einsatz kommt. Dies geschieht, um zu unterstreichen, dass dieses Element lediglich als gemeinsame Schnittstelle des Subjektkerns und der Rollen des Subjekts dient. Kernzustand und -verhalten des Subjekts werden ohnehin durch den Subjektkern zur Verfügung gestellt.

Aus den einzelnen Teilschritten ergeben sich einige Vorbedingungen, die das zu modifizierende Programm einhalten muss, damit das Refactoring angewendet werden kann. Die Vorbedingungen sind benannt, um ihre eindeutige Identifizierung zu ermöglichen. Eine vollständige Auflistung – inklusive jeweils einer Kurzbeschreibung mit Begründung der Notwendigkeit – befindet sich in Abschnitt 3.8. Dennoch sollen die Vorbedingungen, welche sich direkt aus den o.g. Teilschritten ableiten lassen, im Folgenden kurz vorgestellt werden.

Während Schritt 1 keine besonderen Voraussetzungen erfordert, verlangt Schritt 2 nach folgenden Vorbedingungen: Subject darf weder innere Typen (Vorbedingung „PC_SUBJECT_CONTAINS_INNER_TYPE“), noch statische Member, also Attribute oder Methoden, (Vorbedingungen „PC_SUBJECT_STATIC_FIELD“ und „PC_SUBJECT_STATIC_METHOD“) mit Ausnahme von öffentlichen Konstanten („public final“) enthalten, da diese Elemente nicht in Interfaces enthalten sein dürfen. Für die korrekte Durchführung des Refactorings ist jedoch erforderlich, dass das in Schritt 2 extrahierte Interface den Clients Zugriff auf alle Elemente von Subject ermöglicht.

Schritt 3 erfordert, dass `Subject` erweiterbar (also nicht `final`) sein muss (Vorbedingung „`PC_SUBJECT_IS_FINAL`“). Gegenteiliges wäre ohnehin nicht zielführend, da davon ausgegangen wird, dass `Subject` vor Anwendung des Refactorings bereits über Subklassen verfügt, welche als Rollen fungieren sollen.

Die Vorbedingungen von Schritt 4 sind in [Kegel2008] detailliert beschrieben. Besonders hervorzuheben sind im Kontext des IROP Refactorings folgende Vorbedingungen: `Subject` darf keine Subklasse von „`java.lang.Throwable`“ sein („`PC_SUBJECT_EXTENDS_THROWABLE`“), mindestens der Default-Konstruktor von `Subject` (bzw. `SubjectCore`) muss auch nach Entfernen der Vererbungsbeziehung noch für die ursprünglichen Subklassen sichtbar sein („`PC_SUBJECT_ONLY_PRIVATE_CONSTRUCTORS`“), Konstruktoren von Superklassen von `Subject` dürfen keine offene Rekursion enthalten („`PC_OPEN_RECURSION_IN_SUPERCLASS_CONSTRUCTOR`“, s. auch Abschnitte 3.5 und 3.6) und `Subject` darf nicht abstrakt sein („`PC_SUBJECT_IS_ABSTRACT`“).

Schritt 5 macht keine weiteren Vorbedingungen notwendig. In den folgenden Abschnitten wird auf die Probleme eingegangen, die sich aus dem in diesem Abschnitt beschriebenen Vorgehen ergeben. Deren Lösungen machen teilweise weitere Vorbedingungen erforderlich, welche im Rahmen der jeweiligen Lösungsbeschreibung angegeben werden.

3.2 Logische Identität des Subjekts

Einer der Nachteile des Role Object Pattern besteht darin, dass eine Instanz eines Subjekts in der Regel aus mehreren Objekten besteht: Einem Kern und mehreren Rollen. In Abhängigkeit davon, über welche Rolle ein Client auf ein Subjekt zugreift, arbeitet er mit verschiedenen Objekten, die jeweils über eine eigene Identität verfügen. Dieser Zusammenhang wird auch als „Objektschizophrenie“ (s. [Sekharaiah2002]) bezeichnet. Da das Konglomerat von Objekten, welches ein Subjekt ausmacht, aber in seiner Gesamtheit für eine einzige Entität der durch das zu modifizierende Programm abgebildeten Domäne steht, muss es eine Möglichkeit geben, festzustellen, ob zwei Objekte (jeweils Kern oder Rolle) zu demselben Subjekt gehören. Daraus ergibt sich die logische Identität des Subjekts.

Die Sprache Java bietet von Haus aus zwei Möglichkeiten, Objekte auf Identität und Gleichheit zu überprüfen: Der Vergleich zweier Objekte mit Hilfe des Infix-Operators „`==`“ prüft die technische Identität der Objekte. Wie oben aber bereits erläutert, besteht aber das Problem ja gerade darin, dass die Objekte, aus denen ein Subjekt besteht, technisch nicht identisch sind. Dieses Mittel ist also für Vergleiche bezogen auf die logische Identität und Zusammengehörigkeit von Subjektbestandteilen unbrauchbar.

Zur Überprüfung, ob zwei Objekte, die nicht identisch sein müssen, gleich sind, bietet sich die Methode `equals` an, die alle Java-Klassen von `java.lang.Object` erben. Für diese Prüfung wird nicht die technische Identität der Objekte herangezogen, sondern die Frage der Gleichheit wird in der Regel durch Prüfung der Übereinstimmung von Typ und Zustand der Objekte beantwortet.

Dabei ist es jeder Klasse freigestellt, zu definieren, wie weit zwei Objekte voneinander abweichen dürfen, um noch als gleich zu gelten, sofern die dadurch definierte Relation dabei die unter [Java1.5.0, `java.lang.Object#equals(java.lang.Object)`] genannten Eigenschaften erfüllt. Da der konkret stattfindende Vergleich dabei von jeder Klasse selbst definiert werden kann, ist es prinzipiell denkbar, die Beantwortung der Frage nach der logischen Identität über diesen Weg zu lösen. Allerdings ist es dann nicht mehr möglich, diese Methode zu nutzen, um festzustellen, ob zwei Rollen, die nicht über dieselbe logische Identität verfügen, trotzdem als gleich (zum Beispiel in Bezug auf ihren Zustand) betrachtet werden können.

Letzteres ist aber genau die Frage, die in der Regel mit der `equals`-Methode verbunden wird – für Identitätsvergleiche gibt es ja schließlich „`==`“. Daher empfiehlt es sich, zum Vergleich auf logische Identität einen dritten Weg zu beschreiten. Dieser besteht darin, eine eigene Methode „`isIdentical`“ in `Subject` zu definieren, die die Prüfung auf logische Identität übernimmt. Da jedes Subjekt stets über genau einen Kern und beliebig viele Rollen verfügt, kann sich die Prüfung der logischen Identität darauf beschränken, die technische Objektidentität der jeweiligen Kerne zweier Subjektbestandteile festzustellen. Dieser Ansatz entspricht dem von [Bäumer1997] favorisierten. Die Implementierung in `SubjectCore` sieht also wie folgt aus:

```
public boolean isIdentical(Subject other) {
    if (other instanceof SubjectRole) {
        return (((SubjectRole) other).getCore() == this);
    } else {
        return false;
    }
}
```

Dies setzt allerdings zwei Dinge voraus: Zum Einen muss `SubjectRole` eine Methode „`getCore`“ anbieten, die Zugriff auf den Kern des Subjekts bietet, zu welchem die Rolle gehört. Zum Anderen wird bei diesem Code-Fragment davon ausgegangen, dass stets Rollen, niemals aber Subjektkerne, auf eine gemeinsame logische Identität geprüft werden. Letzteres ergibt sich daraus, dass der Subjektkern ohnehin als Implementierungsdetail so weit wie möglich verborgen wird, so dass Clients ausschließlich über konkrete Rollen auf ein Subjekt zugreifen können. Das heißt, dass Clients nie direkten Zugriff auf Instanzen von `SubjectCore` erhalten und daher auch keinen Bedarf haben, deren logische Identität zu vergleichen. Der Parameter `other` ist mit `Subject` typisiert, um auch Clients gerecht zu werden, die die Rolleninstanzen, mit denen sie interagieren, ausschließlich über dieses Interface ansprechen und „kennen“ (statische Typisierung). Die Instantiierung von Subjekten, die zunächst ohne konkrete Kontext- oder Client-spezifische Rollen auskommen, erfolgt dabei mit Hilfe einer leeren Dummy-Rolle (s. Abschnitt 3.6).

Die Implementierung der Methode zum Identitätsvergleich seitens der Rollen erfolgt in `SubjectRole` und ist denkbar einfach, da sich `SubjectRole` an dieser Stelle darauf beschränken kann, das zu tun, wofür es ohnehin zuständig ist: Delegation an den dazugehörigen Subjektkern:

```
public boolean isIdentical(Subject other) {
    return core.isIdentical(other);
}
```

Das Zusammenspiel der beiden Implementierungen zeigt Abbildung 11:

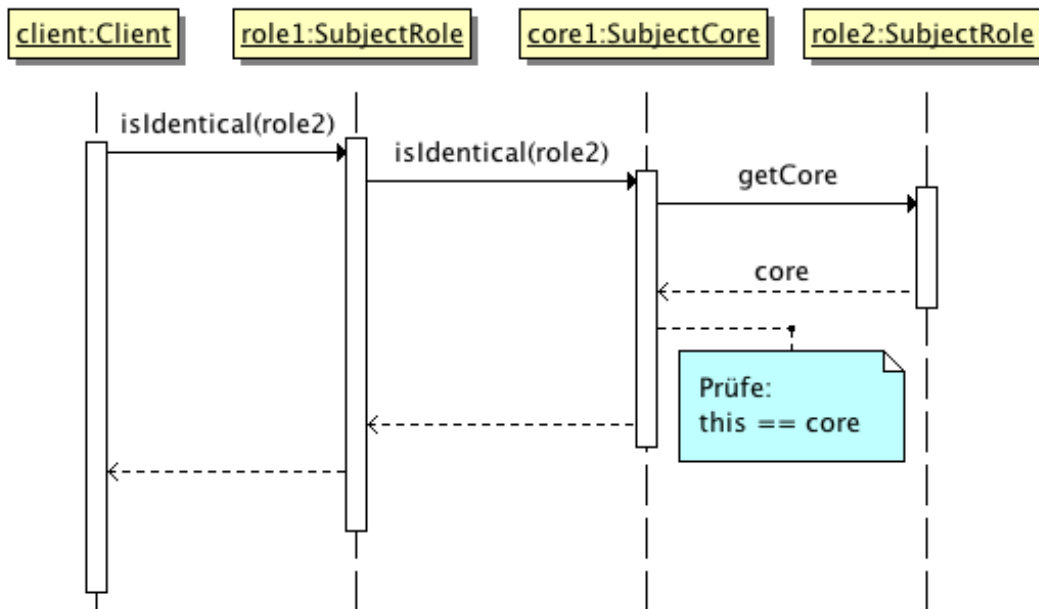


Abbildung 11: Ablauf der Prüfung auf logische Identität

3.3 Gemeinsamer Zustand – Self Encapsulate Field

Weitere Probleme ergeben sich beim direkten Zugriff von Clients und Rollen auf Attribute aus `Subject` oder dessen Superklassen: Einerseits haben Rollen nach erfolgter Transformation keine Möglichkeit mehr, direkt auf die Attribute von `Subject` mit Sichtbarkeitsmodifikator `protected` zuzugreifen, da diese nun in `SubjectCore` liegen (Schritt 2, s. Abschnitt 3.1), welcher nach Schritt 4 (s. Abschnitt 3.1) nicht mehr Superklasse der Rollen ist. Andererseits kann bei direkten Zugriffen auf Attribute (sowohl von `Subject` als auch von dessen Superklassen) aus anderen Klassen heraus nicht garantiert werden, dass alle Rollen eines Subjekts stets auf denselben gemeinsamen Zustand zugreifen.

Letzteres ist insbesondere bei Attributen gefährlich, die zu Superklassen von `Subject` gehören: Da die Rollen weiterhin (indirekte) Subklassen dieser Superklassen bleiben, erbt jede Rolle auch deren Attribute. Da diese Attribute aber Teil des gemeinsamen Subjektzustands und nicht des individuellen Rollenzustands sind, muss sichergestellt werden, dass von allen Rollen aus stets nur auf die vom Subjektkern verwalteten geerbten Eigenschaften zugegriffen wird. Das heißt, jede Rolle erbt zwar weiterhin alle Attribute der Superklassen von `Subject`, darf diese aber nicht mehr direkt verwenden, sondern muss stattdessen alle Zugriffe auf diese Attribute an `SubjectCore` delegieren, welcher dieselben Attribute erbt und als gemeinsamen Kernzustand verwaltet. Superinterfaces sind davon nicht betroffen, da sie keine Instanzvariablen deklarieren können.

Glücklicherweise lassen sich beide Probleme durch eine gemeinsame Lösung beheben. Diese besteht in der Anwendung des Self Encapsulate Field Refactorings (SEF) aus [Fowler1999] auf jedes der betroffenen Attribute. Dadurch wird der direkte Zugriff auf die entsprechenden Instanzvariablen unterbunden und durch Aufrufe von Zugriffsmethoden (getter und setter) ersetzt. Diese Aufrufe lassen sich in der Klasse `SubjectRole` auf die jeweils zugehörige Instanz von `SubjectCore` umleiten. Dadurch ist sowohl der Zugriff auf alle benötigten Attribute sichergestellt, als auch die Anforderung erfüllt, dass alle Rollen eines Subjekts immer auf denselben gemeinsamen Kernzustand zugreifen müssen.

Die Anwendung des SEF Refactorings auf alle Attribute des zu transformierenden Subjekts und seiner Superklassen ist zwingende Voraussetzung des IROP Refactorings. Die dafür zur Verfügung stehenden Tools lassen sich aber weder komfortabel auf mehrere Klassen gleichzeitig anwenden, noch führen sie in allen Fällen zu den gewünschten Ergebnissen (vgl. [Eclipse-Bugzilla], Einträge 273190 und 273192). Daher wird SEF als Teilschritt des hier beschriebenen IROP Refactorings auf die o.g. Attribute angewendet. Daraus ergibt sich, dass die Vorbedingungen des SEF auch für das IROP gelten müssen. Konkret bedeutet dies:

Es darf kein direkter Zugriff auf Attribute von `Subject` oder einer von dessen Superklassen innerhalb eines Postfix-Ausdrucks (z.B.: `a--` oder `subject.x++`) erfolgen. Dies ist der Tatsache geschuldet, dass die Eigenschaft von Postfix-Operatoren, erst nach Auswertung des Ausdrucks, auf den sie sich beziehen, angewendet zu werden, nicht durch Zugriffsmethoden abgebildet werden kann und somit in diesen Fällen das Verhalten des modifizierten Programms verändert würde. Diese Vorbedingung wird im Folgenden „PC_POSTFIX_FIELD_ACCESS“ genannt.

Um Konflikte mit bereits existierenden Zugriffsmethoden (getter/setter) zu vermeiden, empfiehlt es sich, bei der Erzeugung dieser Methoden von den üblichen Namenskonventionen (`getX` bzw. `setX`) abzuweichen. Ob und wie diese Abweichung erfolgen soll, kann durch den Anwender des hier beschriebenen Refactorings vorgegeben werden.

3.4 Rollenmanagement

Ein zentrales Element des Role Object Pattern ist die Verwaltung der Rollen. Ein Subjekt soll Rollen dynamisch annehmen und ablegen können. Neue Rolleninstanzen sollen erzeugt werden können, Clients sollen in der Lage sein, Subjekte auf das Vorhandensein bestimmter Rollen zu prüfen und natürlich sollen Clients auch Zugriff auf bereits vorhandene Rollen erhalten. Diese Funktionen lassen sich unter dem Begriff Rollenmanagement zusammenfassen.

Zuständig für die Umsetzung des Rollenmanagements ist der Subjektkern, welcher durch die Klasse `SubjectCore` realisiert wird. Dort befinden sich also die Implementierungen der einzelnen Rollenmanagementmethoden. Das ursprüngliche ROP sieht insgesamt vier Methoden jeweils zum Erzeugen (`addRole`), Entfernen (`removeRole`), Abfragen (`hasRole`) und Zugreifen (`getRole`) von bzw. auf Rollen vor. Um Eigenschaft 9 (Rollen können von einem Subjekt zu einem anderen übertragen werden) aus [Steimann1999] (s. Abschnitt 2.2) zu erfüllen, kommt in diesem

Lösungsansatz noch eine Methode zum Übertragen von Rollen von einem Subjekt an ein anderes hinzu. Durch Überladen heißt diese Methode im erzeugten Code ebenfalls „addRole“, erwartet als Argument aber eine bereits existierende Rolleninstanz anstelle der zur Erzeugung einer neuen Rolleninstanz notwendigen Parameter.

Das ursprüngliche Pattern sieht vor, dass jedes Subjekt maximal eine Rollen eines bestimmten Rollentyps (also pro Subklasse von `Subject` in der Ausgangssituation) gleichzeitig spielt. Eine Anpassung des Pattern um Eigenschaft 4 gemäß [Steimann1999] (mehrere Instanzen des gleichen Rollentyps pro Subjekt; s. Abschnitt 2.2) erfüllen zu können, ist einfach zu realisieren. Da dieser Fall jedoch zusätzliche Logik zur Auswahl der korrekten Rolleninstanz beim Zugriff auf eine Rolle seitens des Clients voraussetzt, sollte es dem Anwender des Refactorings überlassen bleiben, auszuwählen, ob diese Funktionalität im konkreten Einzelfall tatsächlich benötigt wird.

Alle Rollenmanagementmethoden erfordern die Angabe einer Rollenspezifikation (in den Diagrammen als Parameter mit Namen „spec“ aber ohne Typ dargestellt), die vorgibt, welcher Rollentyp hinzugefügt, entfernt, abgefragt oder zugegriffen werden soll. [Bäumer1997] empfiehlt (vor dem Hintergrund der Sprache C++), in einfachen Fällen Zeichenketten zu verwenden, die den Namen des gewünschten Rollentyps enthalten und bei weiterreichenden Anforderungen einen eigenen Rollenspezifikationstyp zu entwerfen.

Im Kontext der Sprache Java bietet sich jedoch eine elegantere Lösung an: Wird das `.class`-Literal des gewünschten Rollentyps als Rollenspezifikation verwendet, so ist es nicht notwendig einen oder gar mehrere Spezifikationstypen extra zu entwickeln. Dadurch, dass die Literale vom Compiler auf korrekte Schreibweise überprüfbar sind, entfällt die Gefahr, dass Tippfehler – wie bei der Zeichenkettenlösung denkbar – erst zur Laufzeit auffallen. Darüber hinaus kann mit Hilfe der Literale die typsichere Verwendung der Rollenmanagementmethoden sichergestellt werden, da so schon zur Compilezeit feststellbar ist, ob das angegebene Literal zu einer zu dem jeweiligen `SubjectCore` passenden Rollenklasse gehört. Die Signaturen der Rollenmanagementmethoden sehen dabei wie folgt aus:

```
public <R extends SubjectRole> R addRole(Class<R> spec,
    Object... arguments)
    throws CreationException, RoleAlreadyPresentException {...}

public <R extends SubjectRole> R getRole(Class<R> spec)
    throws RoleNotPresentException {...}

public <R extends SubjectRole> boolean hasRole(Class<R> spec) {...}

public <R extends SubjectRole> void removeRole(Class<R> spec) {...}
```

Durch die Einschränkung des generischen Typparameters `R` auf `SubjectRole` und deren Subklassen sind Aufrufe wie z.B. `subject1.hasRole(RoleA.class)`; nur dann fehlerfrei kompilierbar, wenn `RoleA` tatsächlich eine Subklasse von `SubjectRole` ist und somit eine zu `Subject` passende Rolle verkörpert. Die einzig verbleibende Möglichkeit, einen ungültigen

Rollentyp vorzugeben, die erst zur Laufzeit auffiele, wäre die Angabe des `.class`-Literal des abstrakten Typs `SubjectRole` als Rollenspezifikation. Da dies jedoch als offensichtlich unsinnig erkennbar sein sollte und `SubjectRole` darüber hinaus analog zu `SubjectCore` soweit wie möglich als Implementierungsdetail vor den Clients verborgen wird, ist die Wahrscheinlichkeit, dass dieser Fehlerfall eintritt, vermutlich eher gering.

Nicht zuletzt erleichtert die Verwendung dieser Art von Rollenspezifikation auch das Erzeugen neuer Rolleninstanzen durch `SubjectCore` mit Hilfe der Java Reflection API. So können auch Instanzen von Rollenklassen problemlos erzeugt werden, die zur Compilezeit des Subjektkerns noch gar nicht bekannt sind (z.B.: Weil sie erst zu einem späteren Zeitpunkt als Teil einer anderen Softwarekomponente, die das Subjekt um neue Rollen ergänzt, entwickelt werden).

3.5 Offene Rekursion – Reverse Forwarding

Ein Problem, das überhaupt erst durch den Einsatz des Role Object Pattern im Rahmen der Refaktorisierung bereits bestehenden Codes entsteht, ist folgendes: Aufgrund der nicht vorhandenen Vererbungsbeziehung zwischen `SubjectCore` und den Rollen liefert das dynamische Binden von Methodenaufrufen nicht mehr dieselben Ergebnisse wie vor Anwendung des Refactorings. Die Ursache dafür ist die sogenannte „offene Rekursion“, welche von [Pierce2002] wie folgt definiert wird:

„Another handy feature offered by most languages with objects and classes is the ability for one method body to invoke another method of the same object via a special variable called `self` or, in some languages, `this`. The special behavior of `self` is that it is late-bound, allowing a method defined in one class to invoke another method that is defined later, in some subclass of the first.“

Das sich daraus ergebende Problem lässt sich anhand des in Abbildung 12 dargestellten Beispiels nachvollziehen:

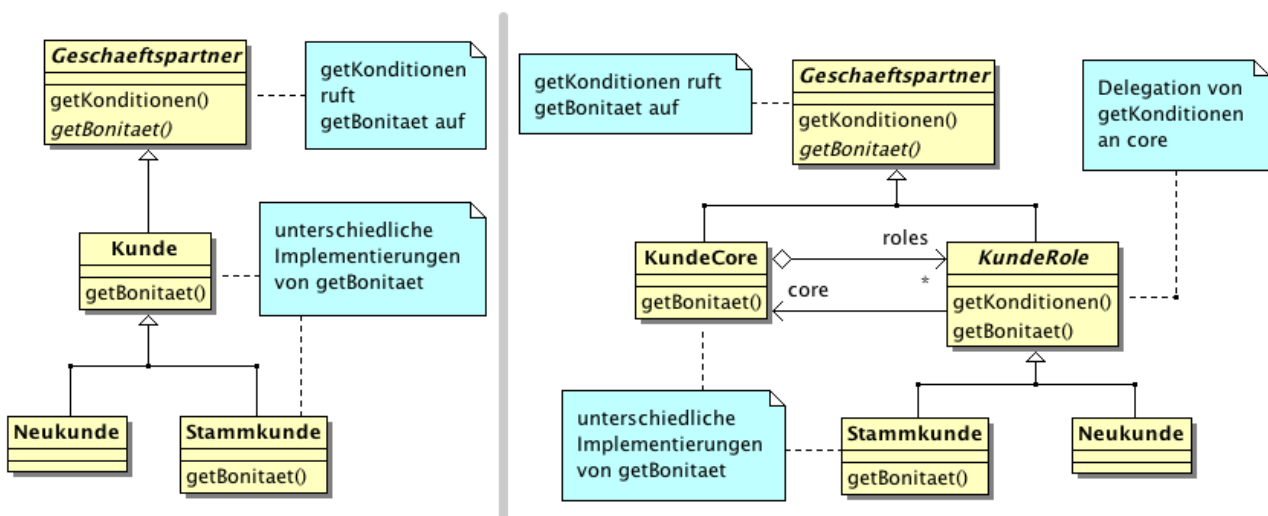


Abbildung 12: Beispiel für Problem des dynamischen Bindens

Die linke Seite des Beispieldiagramms zeigt die Ausgangssituation. Auf der rechten Seite ist eine, auf den zur Erläuterung des Problems wesentlichen Teil des Ergebnisses reduzierte, Abbildung dargestellt. Der Normalfall links ist leicht nachvollziehbar: Wird die von „Geschaeftspartner“ geerbte Methode „getKonditionen“ auf einem Objekt vom Typ „Stammkunde“ aufgerufen, so wird die Implementierung dieser Methode aus der Klasse „Geschaeftspartner“ ausgeführt. Da diese wiederum die Methode „getBonitaet“ auf dem Objekt selbst (implizites `this`) aufruft, kommt der Mechanismus des dynamischen Bindens zum Zuge, wodurch die Implementierung von „getBonitaet“ aus der Klasse „Stammkunde“ verwendet wird. Wird „getKonditionen“ stattdessen auf einem Objekt vom Typ „Neukunde“ aufgerufen, so kommt dadurch auf dem gleichen Weg die Implementierung von „getBonitaet“ aus der Klasse „Kunde“ zur Ausführung.

Letzterer Fall funktioniert auch auf der rechten Seite des Beispieldiagramms: „getKonditionen“ wird auf „Neukunde“ aufgerufen, die Implementierung dieser Methode in „KundeRole“ delegiert den Aufruf an „KundeCore“, wo „getKonditionen“ nicht überschrieben wird, so dass die Implementierung in „Geschaeftspartner“ zum Zuge kommt. Von dort wird „getBonitaet“ aufgerufen, was dazu führt, dass die Implementierung dieser Methode aus „KundeCore“ ausgeführt wird. Das Ergebnis ist also dasselbe, wie im ersten Teil des Beispiels.

Der Aufruf von „getKonditionen“ auf einem Objekt vom Typ „Stammkunde“ aus dem rechten Teil des Diagramms führt allerdings zu einer Abweichung im Verhalten: auch hier wird der Aufruf zunächst korrekt über „KundeRole“ und „KundeCore“ weitergereicht, so dass die Implementierung der Methode aus „Geschaeftspartner“ ausgeführt wird. Der dort initiierte Aufruf von „getBonitaet“ wird jedoch nicht per dynamischem Binden an „Stammkunde“ zurückdelegiert, da das Objekt von dem der Aufruf von „getKonditionen“ an die Klasse „Geschaeftspartner“ weitergegeben wurde, vom Typ „KundeCore“ ist. Dadurch kommt auch in diesem Fall die Implementierung der Methode „getBonitaet“ aus der Klasse „KundeCore“ zur Anwendung anstelle derjenigen aus „Stammkunde“.

Anhand des Sequenzdiagramms in Abbildung 13 wird deutlich, dass im ersten Fall (linke Seite) nur das Objekt „kunde“ vom Typ „Stammkunde“ in die Abwicklung des Methodenaufrufs involviert ist, während es im zweiten Fall (rechte Seite) zu einem Bruch bei der Bearbeitung kommt, und zwar genau zu dem Zeitpunkt, an dem „getKonditionen“ gemäß der Implementierung aus „KundeRole“ an das Objekt „core“ vom Typ „KundeCore“ delegiert wird. Dieser Bruch zwischen den unterschiedlichen Objekten kann mit dem Mittel des dynamischen Bindens allein nicht überbrückt werden.

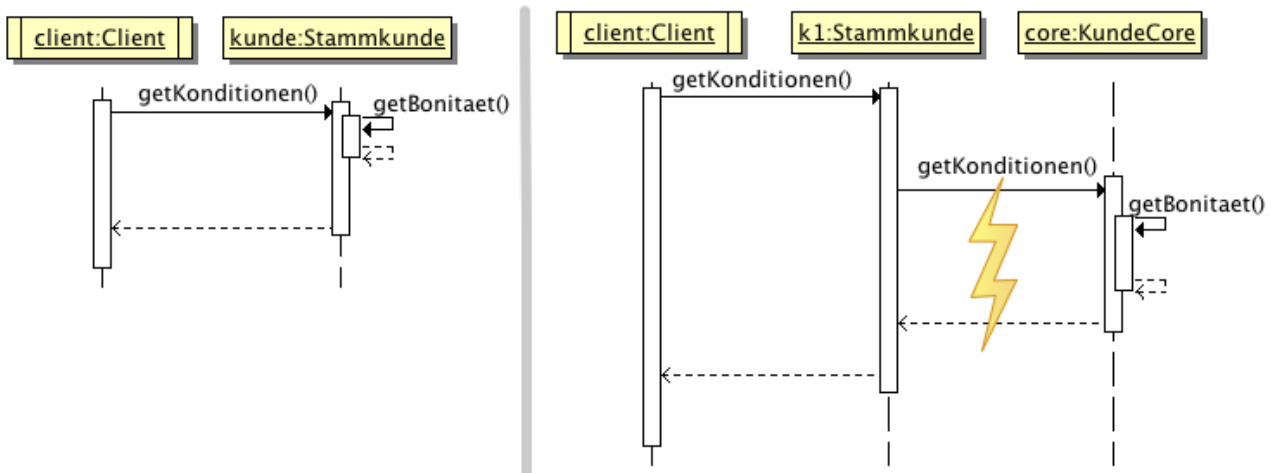


Abbildung 13: Beispiel für Problem des dynamischen Bindens - Sequenz

Die Ursache dieses Problems besteht darin, dass durch Teilschritt 4 (s. Abschnitt 3.1) des IROP Refactorings die Vererbungsbeziehung zwischen `SubjectCore` und `SubjectRole` durch Forwarding ersetzt wird. [Kegel2008] liefert im Rahmen der Beschreibung des in diesem Teilschritt eingesetzten „Replace Inheritance with Delegation“ Refactorings glücklicherweise auch gleich eine Lösung für das oben beschriebene Problem, welche sich die Eigenschaften innerer Klassen in der Sprache Java zu Nutze macht. Der dort vorgestellte Ansatz ergänzt die Delegation um ein „Reverse Forwarding“ genanntes Vorgehen:

„The trick is to have an inner class of *Delegator* subclass *Superclass*, and to add reversely forwarding methods for all methods formerly overridden in *Class* or any of its subclasses to this new class [...]. Since in JAVA an instance of a non-static inner class is always tied to an instance of its outer class (the so-called *enclosing instance* [...]), and because this instance can be accessed from the enclosed instance via `<OuterClass>.this`, reverse forwarding requires no extra fields or initialization. (Note that the reversely forwarding calls are dynamically bound, so that methods overridden in subclasses of *Class* are also reached.)“ [Kegel2008].

Durch diesen Trick lässt sich für die klassenbasierte objektorientierte Programmiersprache Java ein Verhalten umsetzen, welches dem Konzept der Delegation in prototypenbasierten objektorientierten Sprachen wie Self (vgl. [Ungar1987]) entspricht, obwohl in klassenbasierten Sprache üblicherweise nur Forwarding nativ unterstützt wird. Der Unterschied zwischen diesen beiden Konzepten besteht darin, dass beim Forwarding die `this`-Referenz des Objekts, an das „geforwarded“ wird, auf eben dieses Objekt selbst verweist, während sich die `this`-Referenz des Objekts, an das bei der Delegation delegiert wird, auf das Objekt bezieht, von dem die Delegation ausgeht. Abbildung 14 enthält eine schematische Darstellung dieses Unterschieds:

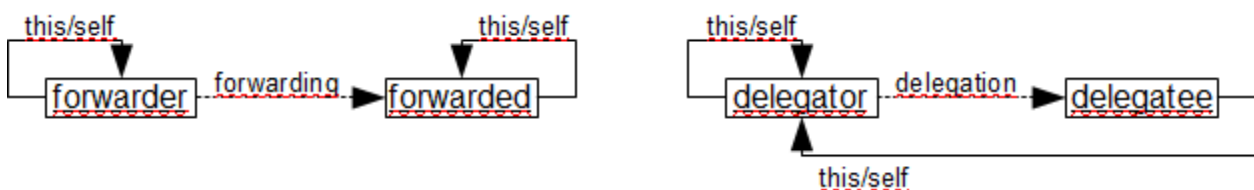


Abbildung 14: Unterschied zwischen Forwarding und Delegation (Quelle: [Kegel2008])

Angewendet auf die für die Zwecke des Introduce Role Object Refactoring modifizierte Variante des Role Object Pattern ergibt sich die in Abbildung 15 dargestellte Konstellation für das Beispiel aus Abschnitt 3.1.

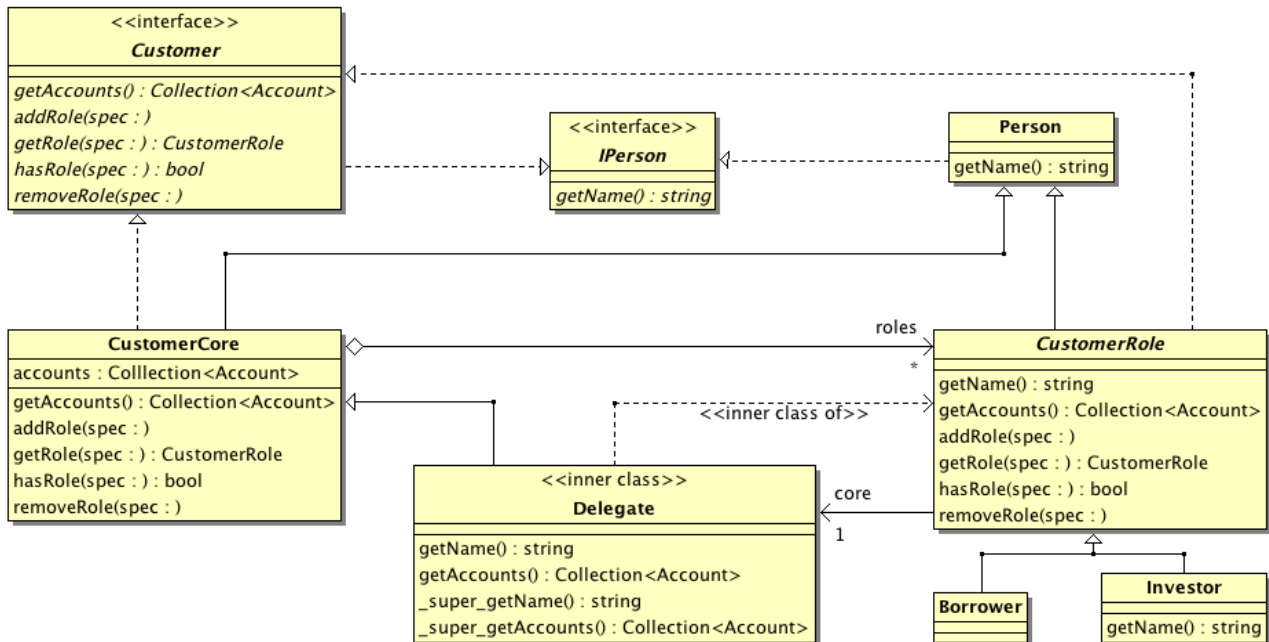


Abbildung 15: Beispiel - Zielzustand mit Reverse Forwarding

Im Vergleich zu Abbildung 5 ist hier die Klasse `Delegate` als innere Klasse von `SubjectRole` (`CustomerRole`) hinzugekommen, die von `SubjectRole` aus als `core` referenziert wird – anstelle von `SubjectCore` (`CustomerCore`), wie es zuvor der Fall war. `Delegate` ist eine Subklasse von `SubjectCore` und enthält für jede Methode aus `Subject` (`Customer`) oder dessen Superklassen, die potentiell durch Rollen überschrieben werden könnten, zwei Methoden:

1. Für die Delegation enthält `Delegate` jeweils eine Methode „`_super_<Methodenname>`“, an die alle Aufrufe der entsprechenden Methode von `SubjectRole` delegiert werden. Diese Methode macht nichts weiter, als den Aufruf an ihre Superklassen weiterzureichen. Für sich allein genommen ist der Effekt derselbe, als würde `Delegate` die geerbte Methode nicht überschreiben und `SubjectRole` einfach ohne Umleitung über die `_super_`-Methode an `Delegate` delegieren. Da `Delegate` aber für den zweiten Schritt, das Reverse Forwarding, die geerbten Methoden überschreiben muss, ist die `_super_`-Methode erforderlich, um die Delegation zu ermöglichen. Implementiert sehen diese Methoden wie folgt aus:

```

public String _super_getName() {
    return super.getName();
}

```

Der dazugehörige Aufruf aus SubjectRole (CustomerRole) ist wie folgt implementiert:

```
public String getName() {
    if (core == null)
        return null; //Alternativ: RuntimeException werfen
    return core._super_getName(); //Delegation an core (Delegate)
}
```

- Für die Rückdelegation (Reverse Forwarding) überschreibt Delegate alle betroffenen Methoden, so dass in Fällen offener Rekursion durch das dynamische Binden die Implementierungen der Methoden in Delegate aufgerufen werden. Diese wiederum delegieren den jeweiligen Aufruf an die umschließende Instanz von SubjectRole, von wo aus dann – ebenfalls durch dynamisches Binden – die passende Implementierung aus der ursprünglich aufgerufenen Rollenklasse ausgeführt wird. Sollte die entsprechende Rollenklasse die betroffene Methode nicht überschreiben, so erfolgt eine erneute Delegation von SubjectRole über Delegate an SubjectCore und es wird die Implementierung aus der nächstgelegenen Superklasse von Subject ausgeführt. Beide Fälle sind in den Abbildungen 16 bzw. 17 abgebildet.

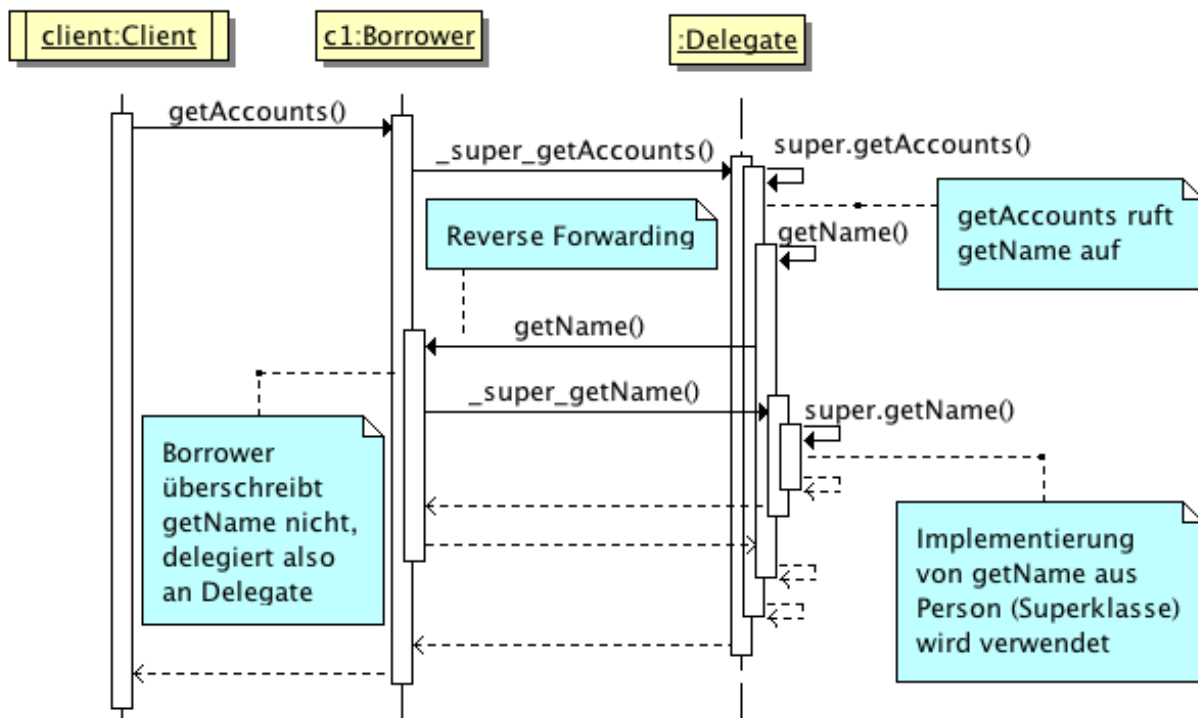


Abbildung 16: Beispiel Reverse Forwarding - Rolle überschreibt Methode nicht

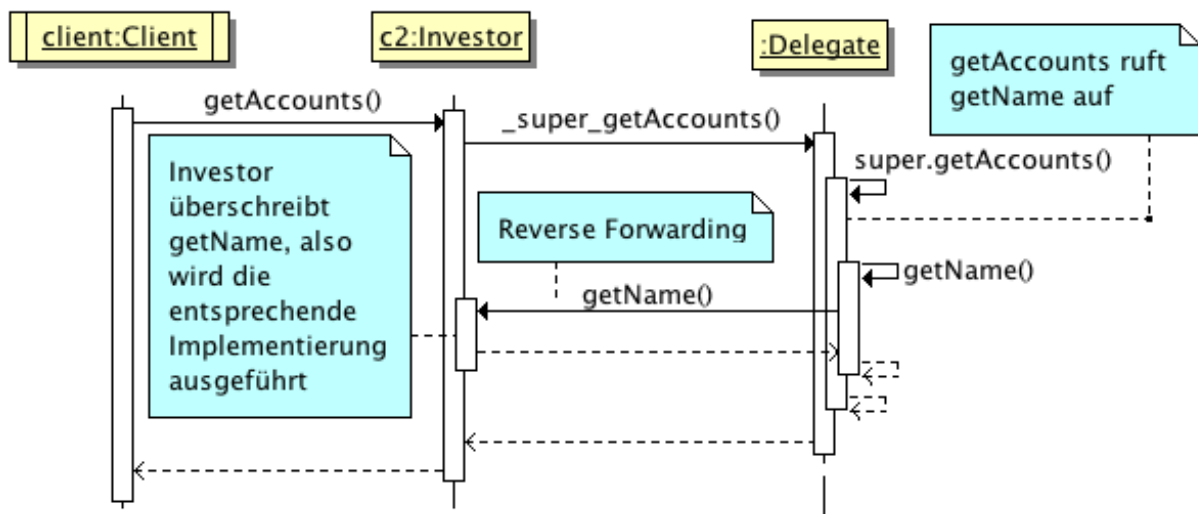


Abbildung 17: Beispiel Reverse Forwarding - Rolle überschreibt Methode

Die überschriebenen Methoden in `Delegate` sind folgendermaßen implementiert:

```
@Override
public String getName() {
    // "Lazy initialization", s. Abschnitt 3.6
    if (CustomerRole.this.core == null)
        CustomerRole.this.core = this;
    // Eigentliches Reverse Forwarding
    return CustomerRole.this.getName();
}
```

Leider verursacht diese Lösung ein neues Problem, das im Zusammenhang mit dem Role Object Pattern zum Tragen kommt: Dadurch, dass `Delegate` eine Subklasse von `SubjectCore` ist, erbt diese Klasse auch alle Attribute von `SubjectCore`, welche den gemeinsamen Zustand des Subjekts und die gespielten Rollen enthalten. Da `Delegate` aber gleichzeitig eine innere Klasse von `SubjectRole` ist und es für die oben beschriebene Funktionsweise des Reverse Forwarding erforderlich ist, dass jede Rolleninstanz über eine eigene an sie gebundene Instanz von `Delegate` verfügt, ergibt sich daraus, dass jede Rolle über eine eigene Version des Kernzustands des Subjekts verfügt. Dies widerspricht dem Gedanken des ROP.

Der Ausweg aus diesem Dilemma besteht darin, Rollenverwaltung und Zustand aus `SubjectCore` heraus in eine zusätzliche Klasse `SubjectState` zu verlagern, von der dann ebenfalls genau eine Instanz pro Subjekt existiert. `Delegate` referenziert diese Instanz von `SubjectState` und delegiert alle Aufrufe von Attributzugriffs- und Rollenmanagementmethoden daran. Die Implementierung des ursprünglichen Verhaltens von `Subject` verbleibt in `SubjectCore`. Damit `SubjectState` dem in Abschnitt 3.3 angesprochenen Umstand Rechnung tragen kann, dass auch von `Subjects` Superklassen geerbter Zustand Teil des gemeinsamen Subjektzustands ist, muss es als Subklasse von `SubjectCore` in die Vererbungshierarchie der Subjektbestandteile eingebunden werden. Entsprechend dieser Überlegungen angepasst sieht das Klassendiagramm des in diesem Abschnitt verwendeten Beispiels dann aus wie in Abbildung 18 dargestellt.

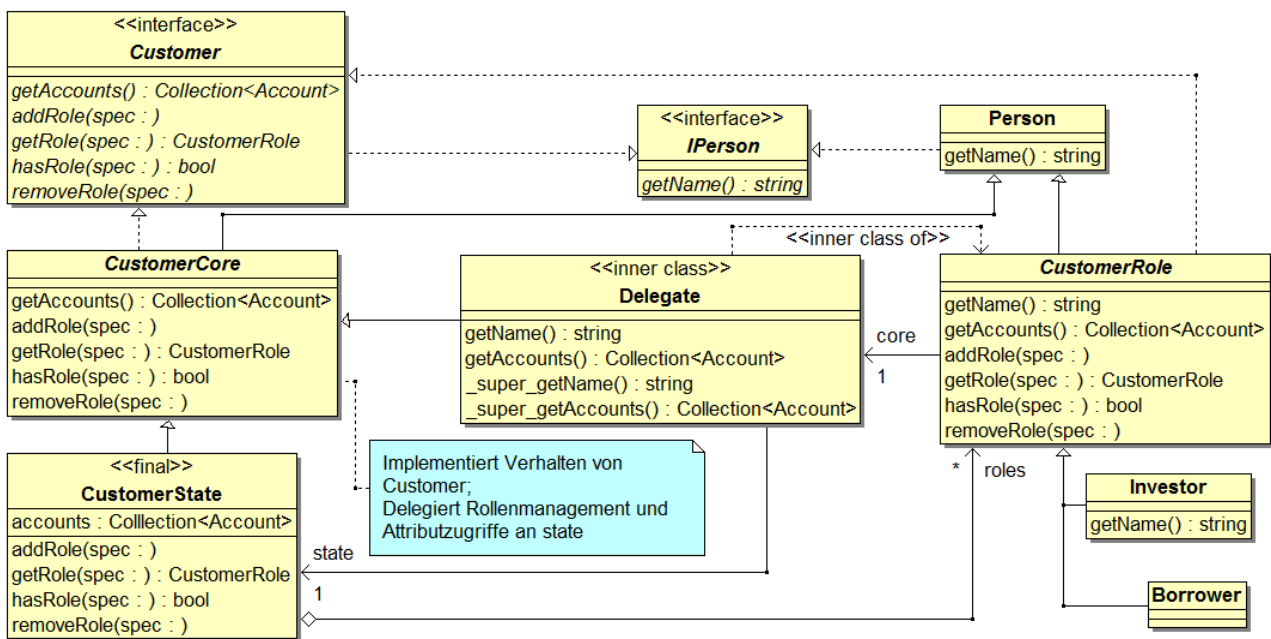


Abbildung 18: Beispiel mit Reverse Forwarding und SubjectState

Die Aufteilung von Kernzustand und -verhalten auf zwei verschiedene Klassen (SubjectCore und SubjectState) macht zwei weitere Vorbedingungen in Ergänzung zu den in Abschnitt 3.1 beschriebenen erforderlich:

1. In Methoden von Subject darf nicht auf private Member (Attribute und Methoden) von Objekten desselben Typs zugegriffen werden (Vorbedingung „PC_SUBJECT_PRIVATE_MEMBER_ACCESS“).
2. Anweisungen zur Initialisierung von Attributen von Subject innerhalb von deren Deklaration dürfen nicht auf private Methoden zugreifen („PC_FIELD_INITIALIZER_CALLS_PRIVATE_METHOD“).

3.6 Instanziierung

[Bäumer1997] sieht vor, dass für jedes Subjekt zunächst eine Instanz von SubjectCore erzeugt wird, die dann dafür zuständig ist, bei Bedarf Instanzen der von den Clients benötigten Rollen anzulegen und anschließend zu verwalten. Rolleninstanzen, die keine Verbindung zu einem Kern haben, sind dort nicht vorgesehen, ebenso wenig die Erzeugung von Rolleninstanzen durch Clients und die Übertragung von Rolleninstanzen von einem Subjekt zu einem anderen.

Bei Anwendung des angestrebten Introduce Role Object Refactorings ist davon auszugehen, dass Instanzen sowohl der Subjektklasse, welche im Rahmen des Refactorings in Subject, SubjectCore, SubjectRole, SubjectState und Delegate zerlegt wird, als auch der Subklassen dieser Subjektklasse (also der späteren Rollen) von ihren jeweiligen Clients bei Bedarf selbst erzeugt werden. Da zu diesem Zeitpunkt noch kein Rollenkonzept implementiert ist, ist jede dieser Instanzen – sowohl der Subjektklasse als auch der Rollenklassen – als individuelles Subjekt

zu betrachten. Dabei entsprechen Instanzen der Subjektklasse einem Subjekt ohne gespielte Rollen und Instanzen der Rollenklasse Subjekten, die jeweils genau eine Instanz der entsprechenden Rolle spielen. Die flexiblen dynamischen Funktionen der Rollenverwaltung, die das Role Object Pattern bietet, kann das modifizierte Programm erst nach weiteren, manuell umzusetzenden Änderungen durch den Anwender des Refactorings tatsächlich nutzen.

Da das Subjekt ohne Rollen durch die Schnittstelle `Subject` vollständig nutzbar ist und die einzelnen Rollen den Clients, die sie nutzen, ohnehin bekannt sind, ist es wünschenswert, die Klassen `SubjectCore`, `SubjectState`, `Delegate` und `SubjectRole` als

Implementierungsdetails soweit wie möglich zu verbergen. Clients sollten am besten nur mit der Schnittstelle `Subject` und den von ihnen jeweils benötigten Rollen in Kontakt kommen.

Abbildung 19 zeigt anhand eines Beispiels, welche Elemente der Implementierung des Subjekts für Clients sichtbar sind und welche verborgen werden.

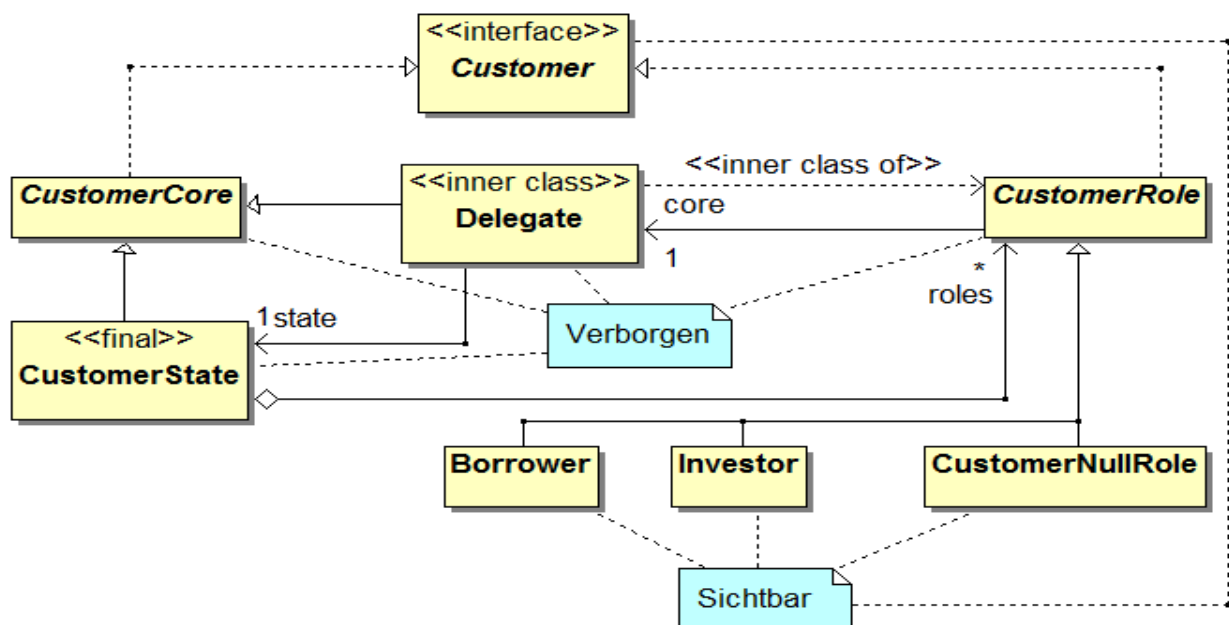


Abbildung 19: Beispiel mit NullRole - sichtbare und verborgene Elemente

Die Überlegung, `SubjectCore` als Implementierungsdetail zu verbergen, steht einer direkten Instantiierung dieser Klasse an Stellen, an denen vor Anwendung des Refactorings die Subjektklasse instantiiert wurde, entgegen. An Stellen, an denen zuvor Rollenklassen mit Hilfe eines einfachen Konstruktoren-Aufrufs instantiiert wurden, müssten sogar zwei Anweisungen eingesetzt werden: Zuerst die – wie gesagt, wenig erstrebenswerte – direkte Instantiierung von `SubjectCore` und anschließend die Nutzung des so erzeugten Subjekts zum Anlegen einer Instanz der gewünschten Rolle. Insbesondere in Bezug auf die Auswahl des im konkreten Einzelfall zu verwendenden Konstruktors von `SubjectCore` und die korrekte Bestückung von dessen Parametern ist die letztgenannte Transformation schwierig umzusetzen.

Um die letztgenannte Problematik zu vermeiden, bietet es sich an, das oben beschriebene Vorgehen umzukehren: Da jede Erzeugung einer Instanz einer Rollenklasse vor Anwendung des Refactorings der Erzeugung eines Subjekts entspricht, welches ebenjene Rolle genau einmal verkörpert, spricht nichts dagegen, diese Instanzen wie gehabt zu erzeugen und dabei im Hintergrund eine Instanz von `SubjectCore` anzulegen, der die erzeugte Rolleninstanz zugeordnet wird. Da die Rollen ohnehin als Decorator (vgl. Pattern „Wrapper“ bzw. „Decorator“ aus [Gamma1995]) des Subjekts fungieren, ist es aus Sicht des Clients unerheblich, dass er das Subjekt nur über die gerade verwendete Rolle „sieht“. Diese implementiert schließlich die Schnittstelle `Subject` ebenso wie `SubjectCore`.

Es verbleibt das Problem, dass nicht auszuschließen ist, dass die Subjektklasse vor Anwendung des Refactorings direkt instantiiert wird. Eine direkte Instantiierung von `SubjectCore` an den entsprechenden Stellen soll vermieden werden, da `SubjectCore` andernfalls nicht – wie oben beschrieben – als Implementierungsdetail vollständig vor den Clients von `Subject` verborgen werden kann. Die Lösung hierfür besteht darin, eine leere Dummy-Rolle „`SubjectNullRole`“ ohne eigenen Zustand oder Verhalten anzulegen, welche alle vormals in der Subjektklasse vorhandenen Konstruktoren durch einfaches Weiterreichen der übergebenen Parameter implementiert. `SubjectNullRole` kann dann überall dort instantiiert werden, wo ein Subjekt zunächst ohne Rollen erzeugt werden soll. Dadurch wird die Erzeugung von Subjekten einheitlich umgesetzt – unabhängig davon, ob vor Anwendung des Refactorings die Subjekt- oder eine der Rollenklassen instantiiert wurde.

Strukturell ist die Instantiierung von Subjekten damit gelöst, allerdings ergeben sich in Bezug auf deren dynamischen Ablauf einige weitere Schwierigkeiten, da sichergestellt werden muss, dass alle zu erzeugenden Objekte korrekt initialisiert und rechtzeitig an den erforderlichen Stellen zur Verfügung gestellt werden. Insbesondere Fälle von offener Rekursion in den beteiligten Konstruktoren erweisen sich dabei als problematisch.

Grundsätzlich sind zunächst folgende Teilschritte zur Instantiierung und Initialisierung eines Subjekts über eine Rolle (ggf. `SubjectNullRole`) notwendig:

1. Ein Konstruktor der Rollenklasse wird mit passenden Parametern aufgerufen.
Diese Aufrufe können im Falle von gewöhnlichen Rollen unverändert aus dem Programm vor Anwendung des Refactorings übernommen werden. Soll ein Subjekt ohne Rollen angelegt werden, so ist aus den o.g. Gründen der entsprechende Konstruktor von `SubjectNullRole` anstelle desjenigen von `SubjectCore` aufzurufen. Das setzt voraus, dass `SubjectNullRole` alle Konstruktoren anbietet, die vor dem Refactoring in der Subjektklasse vorhanden waren.
2. In Java unvermeidbar ruft der Konstruktor der Rollenklasse zunächst einen Konstruktor von deren direkter Superklasse `SubjectRole` auf. Da diese `Subject` als direkte Superklasse aller Rollen ablöst, muss es alle für diese Klassen sichtbaren Konstruktoren von `Subject`

implementieren. Erst nachdem die unten beschriebene Initialisierung von `SubjectRole` abgeschlossen ist, erfolgt die lokale Initialisierung der Rolleninstanz, welche weiterhin genau wie vor Anwendung des Refactorings durchgeführt wird.

3. Die Konstruktoren von `SubjectRole`, welche von den Konstruktoren der Rollen aufgerufen werden, haben zwei Aufgaben: Erstens den Aufruf eines passenden Konstruktors der Superklasse des Subjekts und zweitens die Initialisierung des Subjektkerns in Gestalt von `Delegate`.

Optimal wäre es, `Delegate` zu instantiieren, bevor die geerbten Eigenschaften der Rollenklasse initialisiert werden. Da Java aber den Aufruf eines Konstruktors der Superklasse (ob explizit oder nur implizit vorhanden), wie bereits unter Punkt 2 erwähnt, vor jeglicher lokaler Initialisierung durch den aufgerufenen Konstruktor erzwingt, steht nur die umgekehrte Reihenfolge zur Verfügung. Dies verursacht Probleme, wenn aus Konstruktoren der Superklassen heraus Methoden aufgerufen werden, die `SubjectRole` überschreibt (offene Rekursion, s. Definition nach [Pierce2002] in Abschnitt 3.5) und an `Delegate` delegiert, da zu diesem Zeitpunkt ja noch keine Instanz von `Delegate` existiert.

Ein nahe liegender Gedanke zur Lösung dieser Probleme besteht darin, ein dem Konzept der „lazy initialization“ (s. [Warren1998]) ähnliches Vorgehen zu verfolgen – obwohl es sich in diesem Falle eher um eine vorgezogene anstelle einer verzögerten Initialisierung handeln würde. Dafür müsste bei jeder Delegation von `SubjectRole` an `Delegate` geprüft werden, ob bereits eine Instanz von `Delegate` zur Verfügung steht. Sollte dies nicht der Fall sein, wird zunächst eine entsprechende Instanz erzeugt, bevor die Delegation erfolgt.

Leider lässt sich dieser Ansatz nicht zufriedenstellend realisieren, da zum Zeitpunkt dieser vorgezogenen Instantiierung, die für die korrekte Initialisierung von `Delegate` benötigten Parameter nicht bekannt sind. Es bleibt also nur, das Vorhandensein offener Rekursion in Konstruktoren von Superklassen von `Subject` per Vorbedingung zu verbieten.

Über diese unschöne Einschränkung hinaus sind unerwünschte Seiteneffekte durch den unvermeidbaren mehrfachen Aufruf der Superklassenkonstruktoren möglich. Dieser erfolgt insgesamt drei Mal: Zuerst durch `SubjectRole`, dann durch `Delegate` und zuletzt durch `SubjectState`, da alle drei Subklassen der ursprünglich direkten Superklasse von `Subject` sind. Dabei werden nur die durch den letzten Aufruf initialisierten Attribute der Oberklassen als Teil des gemeinsamen Zustands des Subjekts im weiteren Programmverlauf verwendet. Die im Rahmen der ersten beiden Aufrufe angelegten Instanzen der Attribute der Superklassen sind zwar technisch vorhanden, müssen aber logisch ignoriert werden.

4. Der aus `SubjectRole` aufgerufene Konstruktor von `Delegate` sorgt durch den unumgänglichen Aufruf eines Konstruktors von `SubjectCore` gezwungenermaßen für den zweiten Aufruf eines Konstruktors der ehemaligen Superklasse von `Subject`. Seine eigentliche Aufgabe besteht jedoch darin, eine Instanz von `SubjectState` als Subjektkern anzulegen und die umschließende Rolleninstanz bei diesem als Rolle zu registrieren.

Dabei ist zu beachten, dass die Instantiierung der umschließenden Rolleninstanz selbst zu diesem Zeitpunkt noch gar nicht vollständig abgeschlossen ist, da der Aufruf des Konstruktors von `Delegate` ja aus dem entsprechenden Konstruktor von `SubjectRole` heraus erfolgt ist. Insbesondere ist die Zuweisung einer Referenz auf die soeben erzeugte Instanz von `Delegate` an die entsprechende Variable „`core`“ in `SubjectRole` noch nicht erfolgt.

Das führt dazu, dass im Falle offener Rekursion in den Konstruktoren von `SubjectCore` (welche im Gegensatz zu offener Rekursion in Konstruktoren der Superklassen nicht durch eine entsprechende Vorbedingung ausgeschlossen ist) die entsprechenden Aufrufe zwar von `Delegate` an die umschließende Rolle „reverse forwarded“ werden, die Delegation zurück an `Delegate` – falls die Rolle eine oder mehrere der betroffenen Methoden nicht überschreiben sollte – jedoch fehlschlägt, da die dafür benötigte Referenz auf `Delegate` in `SubjectRole` zu diesem Zeitpunkt ja noch nicht vorliegt. Die Lösung hierfür besteht darin, dass `Delegate` vor der „Rück“-Delegation an die umschließende Rolleninstanz zunächst prüft, ob dort die Variable `core` bereits belegt ist und dieser andernfalls eine Referenz auf sich selbst zuweist.

5. Das Erlauben offener Rekursion in den Konstruktoren von `Subject` bzw. `SubjectCore` birgt eine weitere Gefahr in sich: Wird von dort aus auf den gemeinsamen Subjektzustand zugegriffen, so führt dies zu einem Fehler, da die entsprechende Instanz von `SubjectState` zu diesem Zeitpunkt noch nicht durch den Konstruktor von `Delegate` erzeugt werden konnte. Diese Fehlerquelle lässt sich vermeiden, indem `SubjectCore` für jeden seiner Konstruktoren eine Methode „`initState`“ mit identischer Parameterliste anlegt, welche von dem jeweils entsprechenden Konstruktor als erstes aufgerufen und durch `Delegate` so überschrieben wird, dass dort die benötigte Referenz auf `SubjectState` initialisiert wird. Als Folge dessen müssen die Konstruktoren von `Delegate` anschließend lediglich prüfen, ob die Initialisierung dieser Referenz bereits im Rahmen des Aufrufs eines Konstruktors von `SubjectCore` erfolgt ist oder noch aussteht.
6. Die Erzeugung einer Instanz von `SubjectState` aus `Delegate` heraus führt zum dritten, letzten und vor allem entscheidenden Aufruf der Konstruktoren von `SubjectCore` und dessen Superklassen, da nur die hierbei initialisierten Eigenschaften als Teil des gemeinsamen Subjektzustands betrachtet werden.

Das oben beschriebene Vorgehen wirft die Frage nach dem Ablauf der Erzeugung neuer Rollen im Rahmen eines bereits bestehenden Subjekts mittels `addRole` auf. Das Vorgehen dabei ist dasselbe wie hier geschildert. Das heißt, dass auch in diesem Fall beim Anlegen der Rolleninstanz zunächst ein eigener Subjektkern erzeugt wird. Anschließend wird die Rolleninstanz aber sofort an den mit ihrer Erzeugung beauftragten Subjektkern übertragen, was vom Ablauf her identisch mit der in Abschnitt 3.3 beschriebenen Übertragung bestehender Rollen von einem Subjekt an ein anderes ist. Diese Verfahrensweise ist sicherlich nicht die Effizienteste, erlaubt aber ein einheitliches Vorgehen unabhängig davon, ob ein komplettes Subjekt oder eine Rolleninstanz erzeugt werden soll und nutzt dabei geschickt die ohnehin vorhandene Migrationsmöglichkeit von Rollen zwischen verschiedenen, passenden Subjekten.

3.7 Geerbtes Verhalten und Schnittstellen

Durch die Anwendung des Extract Interface Refactorings (Schritt 2, s. Abschnitt 3.1) fällt `Subject` aus der bisher bestehenden Vererbungshierarchie insofern heraus, als dass Interfaces nicht von Klassen abgeleitet werden können. Da die öffentliche Schnittstelle von `Subject` aber weiterhin das von den Superklassen geerbte Verhalten anbieten muss, um unverändert verwendet werden zu können, gilt es, eine Lösung zu finden, die es dem Interface `Subject` ermöglicht auch weiterhin das geerbte Verhalten zugreifbar zu machen.

Hierzu bietet es sich an, für jede Superklasse von `Subject` ein Interface anzulegen, welches alle nicht privaten Methoden der jeweiligen Superklasse enthält und von dieser implementiert wird. Diese Interfaces werden in einer zum `Subject` vor der Transformation und seinen Superklassen parallelen Vererbungshierarchie angeordnet, so dass das Interface `Subject` vom Interface seiner ehemaligen Superklasse abgeleitet werden und somit die benötigten Methoden erben kann.

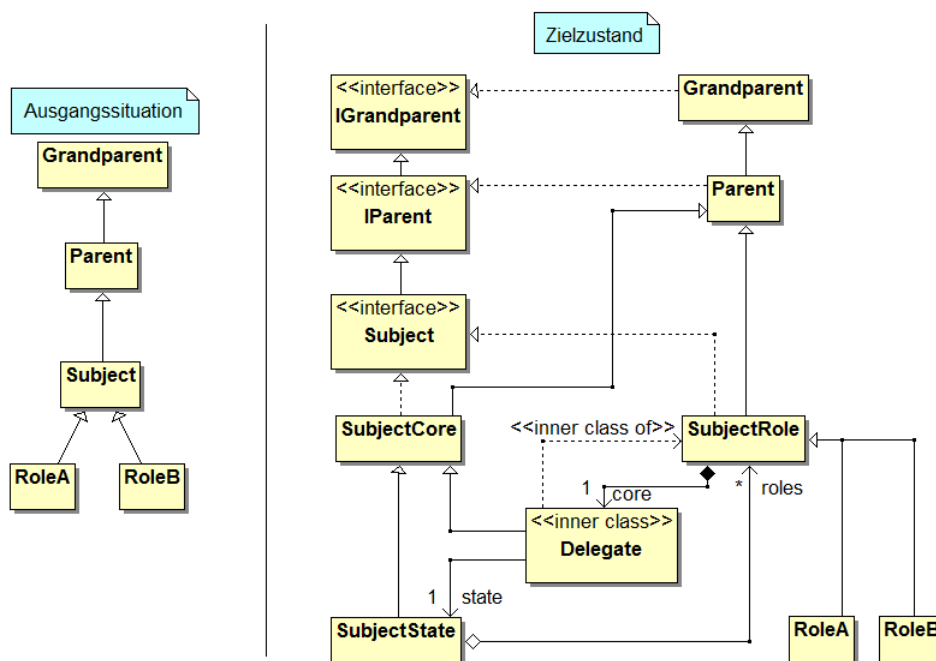


Abbildung 20: Struktur bei mehreren Superklassen

Einzigster Nachteil dieser Lösung ist, dass Interfaces nur öffentlich sichtbare (`public`) Methoden enthalten dürfen, was dazu führt, dass die Sichtbarkeit der betroffenen Methoden gegebenenfalls entsprechend angepasst werden muss. Dies zieht Änderungen an allen Subklassen der Superklassen von `Subject` (im Folgenden „Siblings“ (engl. Geschwister) von `Subject` genannt) nach sich. Falls diese geerbte Methoden überschreiben, deren Sichtbarkeit erhöht werden musste, muss auch die Sichtbarkeit der überschreibenden Methode des betroffenen Siblings entsprechend angepasst werden, da überschreibende Methoden die Sichtbarkeit der überschriebenen Methode nicht einschränken dürfen.

Um die Anzahl der von einer Erhöhung ihrer Sichtbarkeit betroffenen Methoden zu reduzieren, ist es möglich, anstelle idiosynkratischer Interfaces (vgl. [FernUniversität01853]), welche die komplette öffentliche Schnittstelle der Klassen enthalten, von denen sie jeweils implementiert werden, Interfaces anzulegen, die nur diejenigen Methoden der korrespondierenden Superklasse enthalten, welche tatsächlich von den Clients von `Subject` verwendet werden. Um diese minimalen Interfaces zu definieren, muss die maximal verallgemeinerte Schnittstelle von `Subject` bestimmt werden. Wie dieses Ziel mittels Constraint-basierter Typinferenz erreicht werden kann, beschreibt [Kegel2007]. Bei der Umsetzung des Refactorings werden die dort beschriebenen Möglichkeiten jedoch nicht wahrgenommen, da dadurch einerseits die Flexibilität der Verwendung von `Subject` im Rahmen der Weiterentwicklung des zu modifizierenden Programms eingeschränkt wird und andererseits Fehler im Zusammenhang mit Clients auftreten können, deren Quellcode bei der Anwendung des Refactorings nicht zur Verfügung steht und daher bei der Ermittlung der maximal verallgemeinerten Schnittstelle von `Subject` nicht berücksichtigt werden kann.

3.8 Zusammenfassung

Die in den vorangegangenen Abschnitten beschriebenen Probleme ergeben sich einerseits aus den spezifischen Eigenschaften der Sprache Java und andererseits aus der Anforderung, das beobachtbare Verhalten des zu modifizierenden Programms auch nach Einführung des Role Object Patterns zunächst unverändert beibehalten zu müssen. Ihre jeweiligen Lösungen führen in ihrer Gesamtheit zu einem Ergebnis, welches von dem in Abschnitt 3.1 skizzierten grundsätzlichen Vorgehen und auch dem o.g. Pattern in seiner ursprünglichen Form deutlich abweicht. Insgesamt ergibt sich nun ausgehend von einer Situation, wie sie in Abbildung 21 dargestellt ist, ein Zielzustand, wie Abbildung 22 ihn zeigt.

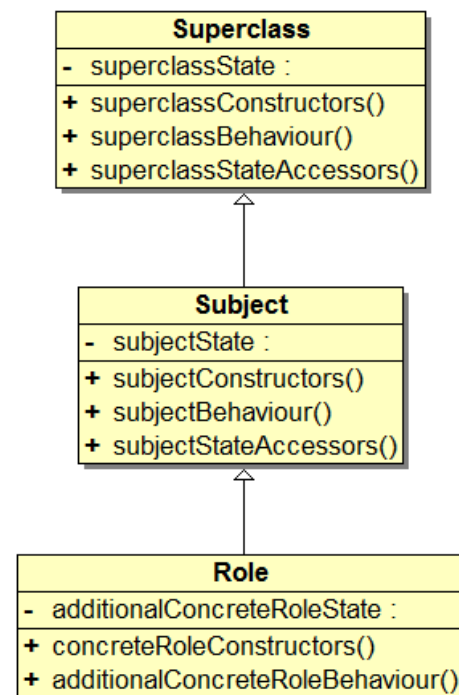


Abbildung 21: IROP Ausgangssituation - Struktur schematisch

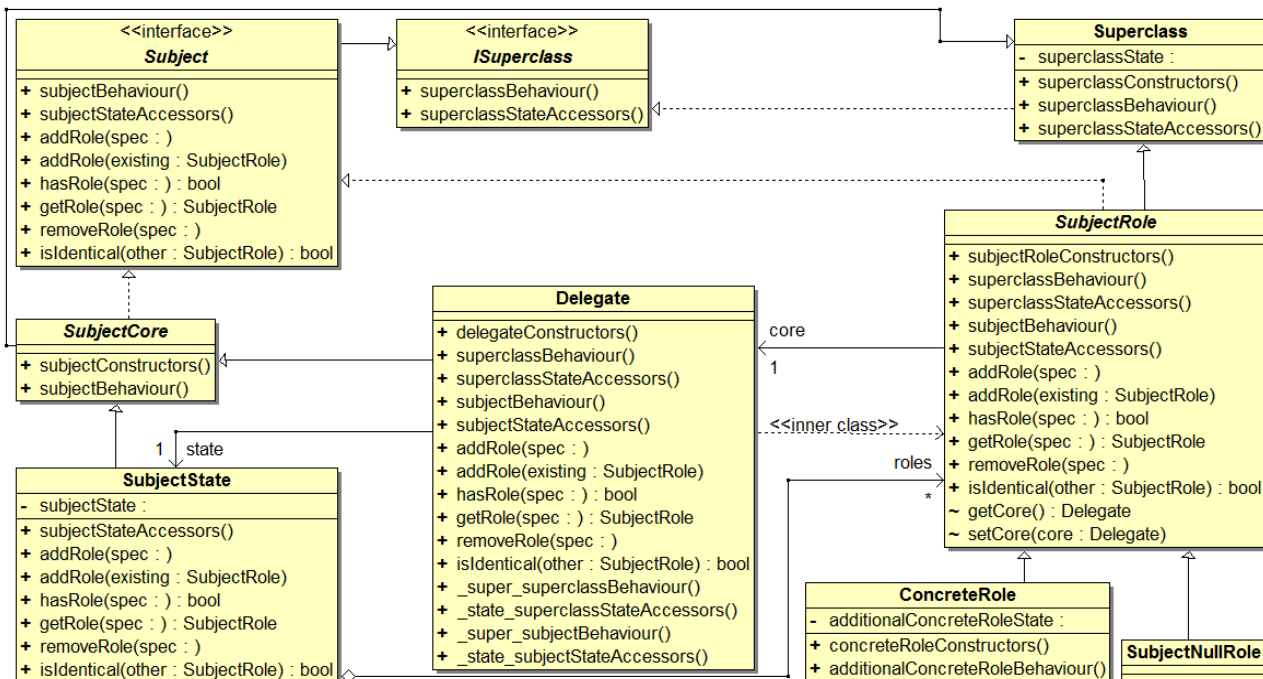


Abbildung 22: IROP Zielzustand - Struktur schematisch

Damit dieses Refactoring durchgeführt werden kann, muss das Programm bzw. die konkrete Java-Klasse, auf die es angewendet werden soll, folgende Vorbedingungen erfüllen:

1. PC_POSTFIX_FIELD_ACCESS

Field access in postfix expression.

Der Zugriff auf Felder innerhalb von Postfix-Ausdrücken verhindert das Self Encapsulate Fields Refactoring, welches eine notwendige Voraussetzung für die weiteren Transformationsschritte des Refactorings ist.

2. PC_SUBJECT_EXTENDS_THROWABLE

Subject must not extend „java.lang.Throwable“.

Errors und Exceptions dienen primär dazu, die technische Behandlung von Fehlern von der Fachlogik eines Programms zu trennen. Da die Motivation zur Einführung von Rollenobjekten jedoch aus der Anwendungsdomäne des Programms hervorgehen sollte, werden die Besonderheiten von Throwable im Rahmen des Refactorings nicht berücksichtigt.

3. PC_SUBJECT_IS_FINAL

Subject must not be final.

Bei einer finalen Klasse ist davon auszugehen, dass sie bewusst keine Subklassen haben soll. Diese sind aber – sowohl in der Ausgangssituation als Vorlage für die zu erstellenden Rollen als auch im Zielzustand als technische Grundlage der Rollen – integraler Bestandteil

der dem Refactoring zu Grunde liegenden Konzepte. Daher wird davon abgesehen, finale Klassen im Rahmen des Refactorings in nicht-finale umzuwandeln. Falls dennoch gewünscht, müsste dies als vorbereitender Schritt durch den Anwender manuell erfolgen.

4. PC_SUBJECT_ONLY_PRIVATE_CONSTRUCTORS

Subject must contain at least one non-private constructor (including the default constructor).

Da aus `Subject` der gemeinsame Kern aller Rollen eines Subjekts abgeleitet wird und dieser sinnvoll instantiierbar und initialisierbar sein muss, muss in der Ausgangssituation mindestens ein sichtbarer Konstruktor zur Verfügung stehen.

5. PC_OPEN_RECURSION_IN_SUPERCLASS_CONSTRUCTOR

Superclass constructor must not contain open recursion.

Da bei Java-Programmen stets die Konstruktoren der Superklassen einer Klasse vor deren eigenen Konstruktoren ausgeführt werden, führt offene Rekursion in den Konstruktoren von Superklassen zu Fehlern, da zu diesem Zeitpunkt die Referenzen auf den gemeinsamen Kern des Subjekts noch nicht initialisiert werden konnten (vgl. Abschnitt 3.6).

6. PC_SUBJECT_IS_ABSTRACT

Subject must not be abstract.

Der aus `Subject` abzuleitende Typ `SubjectCore` muss instantiierbar sein und sollte sinnvolle Implementierungen für alle Methoden von `Subject` und dessen Superklassen enthalten. Beides ist bei abstrakten Klassen nicht garantiert. Falls dennoch gewünscht, müsste dies als vorbereitender Schritt durch den Anwender sichergestellt und anschließend durch Entfernen des Schlüsselworts `abstract` kenntlich gemacht werden.

7. PC_SUBJECT_CONTAINS_INNER_TYPE

Subject must not contain any inner types.

Da `Subject` in ein Interface transformiert wird und diese keine inneren Typen enthalten können, darf `Subject` in der Ausgangssituation ebenfalls keine inneren Typen enthalten.

8. PC_SUBJECT_STATIC_METHOD

Subject must not contain static methods.

Da `Subject` in ein Interface transformiert wird und diese keine abstrakten Methoden enthalten können, darf `Subject` in der Ausgangssituation ebenfalls keine abstrakten Methoden enthalten.

9. PC_SUBJECT_STATIC_FIELD

Subject must not contain static fields (unless they are „public final“).

Da `Subject` in ein Interface transformiert wird und diese keine statischen Attribute enthalten können, darf `Subject` in der Ausgangssituation ebenfalls keine statischen Attribute enthalten. Hiervon ausgenommen sind nur öffentliche Konstanten (`public static final`).

10. PC_SUBJECT_PRIVATE_MEMBER_ACCESS

Subject must not access private members (field or method) of another object of its own class.

Java erlaubt Objekten den Zugriff auf `private` Member (Attribute und Methoden) anderer Objekte desselben Typs. Da im Rahmen des Refactorings die Methoden und Attribute aus `Subject` auf unterschiedliche Typen verteilt werden (Methoden → `SubjectCore`, Attribute → `SubjectState`), führen solche Zugriffe anschließend zu Fehlern und sind daher unzulässig.

11. PC_FIELD_INITIALIZER_CALLS_PRIVATE_METHOD

Field initializer invokes a `private` method.

Bei Attributen, die bei ihrer Deklaration bereits initialisiert werden, werden die entsprechenden Anweisungen zusammen mit den Attributen in den neu angelegten Typ `SubjectState` verschoben. Werden dabei `private` Methoden von `Subject` aufgerufen, welche im Rahmen des Refactorings in den ebenfalls neu angelegten Typ `SubjectCore` verschoben werden, so sind diese von `SubjectState` aus nicht sichtbar.

4 Lösungsansatz **Lightweight Role Objects**

Das in Kapitel 3 beschriebene Refactoring erfordert eine Reihe harter Vorbedingungen, die sich zu großen Teilen aus den Anforderungen der für die einzelnen Teilschritte verwendeten Refactorings `Self Encapsulate Fields`, `Extract Interface` und `Replace Inheritance with Delegation` ergeben. Inwiefern diese Vorbedingungen die Anwendbarkeit des Refactorings einschränken, wird in den Kapiteln 6 und 7 anhand einer Evaluation ermittelt.

Davon abgesehen ergibt sich aus den Modifikationen des Role Object Pattern, die für dessen Einführung im Rahmen eines Refactorings erforderlich sind, ein relativ kompliziertes Konstrukt, welches dem von Refactorings im Allgemeinen verfolgten Ziel entgegensteht, Lesbarkeit und Wartbarkeit eines Programms durch Änderungen an dessen Struktur zu verbessern. Tatsächlich erschweren insbesondere die Vielzahl der für das Reverse Forwarding benötigten Methoden und deren wechselseitige Abhängigkeiten das Verständnis des erzeugten Codes. Die Ergänzung zusätzlichen Verhaltens des Subjekts gestaltet sich dadurch ebenfalls aufwändig und fehleranfällig, da in diesem Falle manuell sicherzustellen ist, dass alle notwendigen delegierenden und zurückdelegierenden Methoden korrekt angelegt und implementiert werden. In Anbetracht dieser Probleme kommt [Steimann2010] zu dem Schluss, dass das Ergebnis des oben beschriebenen Refactorings eher einen Kandidaten für die Anwendung weiterer Refactorings als einen erstrebenswerten Zielzustand darstellt. Vor diesem Hintergrund liegt es nahe, über alternative Lösungen nachzudenken. Der im Folgenden beschriebenen Ansatz nennt sich „Introduce Lightweight Role Objects“ (ILRO) Refactoring.

Die Grundidee des Role Object Pattern besteht darin, dass ein Subjekt aus einem Kern und beliebig vielen Rollen besteht, die sich einen gemeinsamen Kernzustand, gemeinsames Kernverhalten und eine gemeinsame logische Identität teilen. Gemeinsamer Zustand und logische Identität dürfen dabei genau einmal pro Subjekt, nicht aber pro Rolleninstanz existieren. Wo das gemeinsame Verhalten implementiert wird, ist dabei unerheblich, sofern es nur dem Subjekt, inklusive aller seiner Rollen, zur Verfügung steht. Das Kernverhalten muss also nicht zwingend zusammen mit dem Kernzustand aus der Subjektklasse, wie sie vor Anwendung des angestrebten Refactorings besteht, ausgelagert werden. Wird das Verhalten nicht ausgelagert, so verbleibt es innerhalb der bestehenden Vererbungshierarchie, also dort, wo nach dem bisherigen Ansatz `SubjectRole` angesiedelt ist. Das Ergebnis dieser Überlegungen entspricht der Anwendung eines `Move Method Refactorings` (s. [Fowler1999]) auf die Methoden in `SubjectCore`, welche das gemeinsame Verhalten des Subjekts implementieren. Die Notwendigkeit von `Delegation` und `Reverse Forwarding` entfällt dadurch für diese Methoden.

Entfällt die Notwendigkeit der Delegation, so kann auch auf `Delegate` verzichtet werden, wodurch die Notwendigkeit der Separation von `SubjectCore` und `SubjectState` ebenfalls entfällt. Diese beiden Klassen können also wieder unter dem Namen `SubjectCore` zusammengefasst werden. Da dieser neue `SubjectCore` das Kernverhalten des Subjekts nicht mehr implementiert, implementiert er auch das Interface `Subject` nicht mehr.

Da `SubjectRole` die Methoden aus dem Interface `Subject` nun direkt selbst implementiert, kann dieses Interface vollständig entfallen und `SubjectRole` kann dessen Namen annehmen. Durch das Wegfallen des Interfaces entfällt darüber hinaus auch die Notwendigkeit, ein idiosynkratisches Interface für jede Superklasse des Subjekts anzulegen.

Die Instantiierung von Subjekten ohne Rollen kann bei diesem Ansatz direkt durch Erzeugen einer Instanz von `Subject` erfolgen, sofern `Subject` nicht abstrakt ist. Abweichend zur Umsetzung von `SubjectRole` beim Ansatz IROP, sollte letzteres hier nur der Fall sein, wenn `Subject` schon vor Anwendung des Refactorings abstrakt war. Die Vorbedingung `PC_SUBJECT_IS_ABSTRACT` (s. Abschnitt 3.8) entfällt beim Ansatz ILRO, da hier keine Gefahr besteht, abstrakte Methoden in `SubjectCore` aufnehmen zu müssen. Aufgrund dieser Überlegungen kann die Klasse `SubjectNullRole` ersatzlos entfallen.

Zusätzlich ist die erneute Einführung einer abstrakten Klasse `SubjectRole` sinnvoll, welche von `Subject` abgeleitet ist und als Superklasse aller Rollen fungiert. Der Grund hierfür ist, dass den in `SubjectCore` verbliebenen Rollenmanagementmethoden so ermöglicht wird, zwischen Instanzen von `Subject` selbst und Instanzen von Rollen bereits zur Compilezeit unterscheiden.

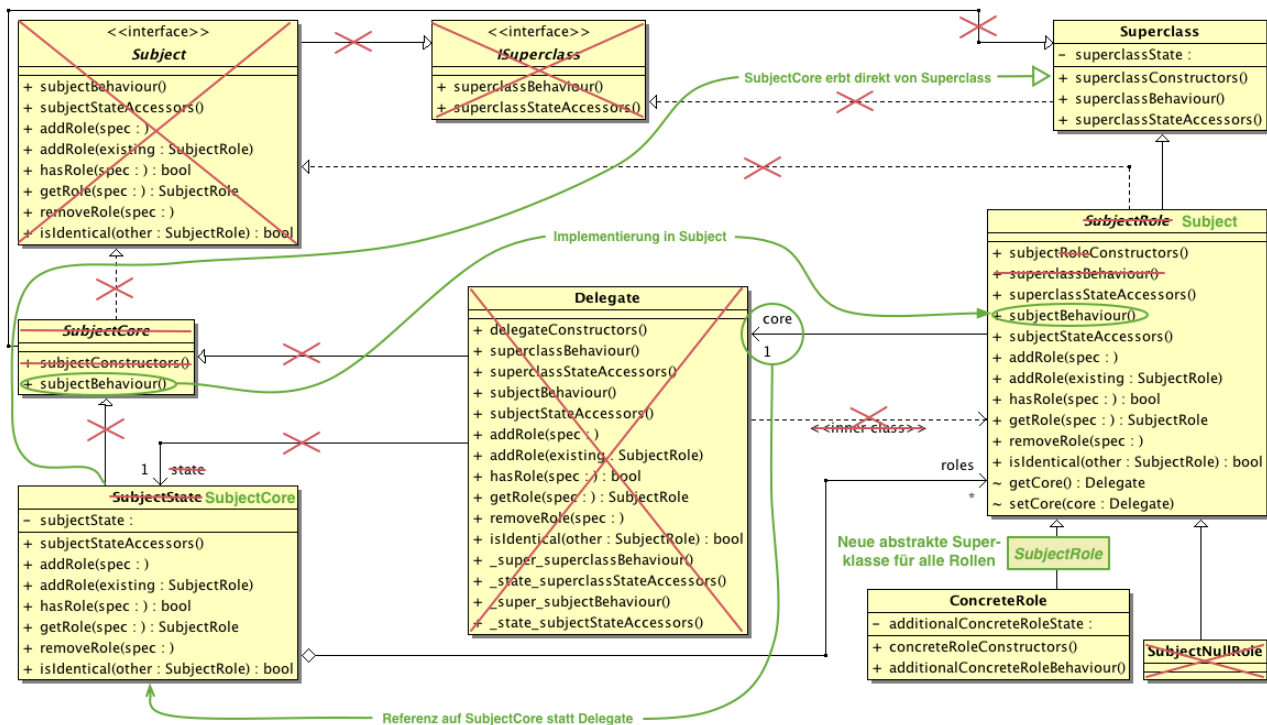


Abbildung 23: Darstellung der Unterschiede zwischen den Lösungsansätzen

Aus Sicht der Ausgangssituation vor Anwendung des Refactorings stellt sich die Transformation gemäß den obigen Überlegungen sogar deutlich einfacher dar, als aus Sicht eines erfolgten Refactorings gemäß der Beschreibung in Kapitel 3. Denn hierfür sind nur folgende Schritte notwendig:

1. Anwenden von Self Encapsulate Field auf die Attribute von `Subject` und dessen Superklassen.
2. Verschieben aller Instanzvariablen von `Subject` in eine neu anzulegende finale Klasse `SubjectCore` mittels des Extract Class Refactorings (s. [Fowler1999]). Diese implementiert sowohl die Zugriffsmethoden auf den Zustand aus `Subject` als auch die bekannten Rollenmanagementmethoden. Als Superklasse von `SubjectCore` dient dabei die direkte Superklasse von `Subject`.
3. Implementierung der Zugriffsmethoden des Subjektzustands (inklusive des von `Subjects` Superklassen geerbten Zustands) in `Subject` durch Delegation an `SubjectCore`.
4. Implementierung der Rollenmanagementmethoden in `Subject` ebenfalls durch Delegation an `SubjectCore`.
5. Anlegen einer abstrakten Klasse `SubjectRole`, welche von `Subject` abgeleitet ist und als Superklasse aller zuvor direkt von `Subject` abgeleiteten Rollenklassen fungiert. Diese Klasse ist bis auf die Implementierung von Konstruktoren, welche denen von `Subject` entsprechen, und lediglich die erhaltenen Parameter an diese weiterreichen, leer, da ihre eigentliche Aufgabe allein darin besteht, zur Typsicherheit des Rollenmanagements beizutragen.

Der Zielzustand dieses Refactorings – ausgehend von der in Abbildung 21 (s. Abschnitt 3.8) dargestellten Situation – sieht damit wie folgt aus:

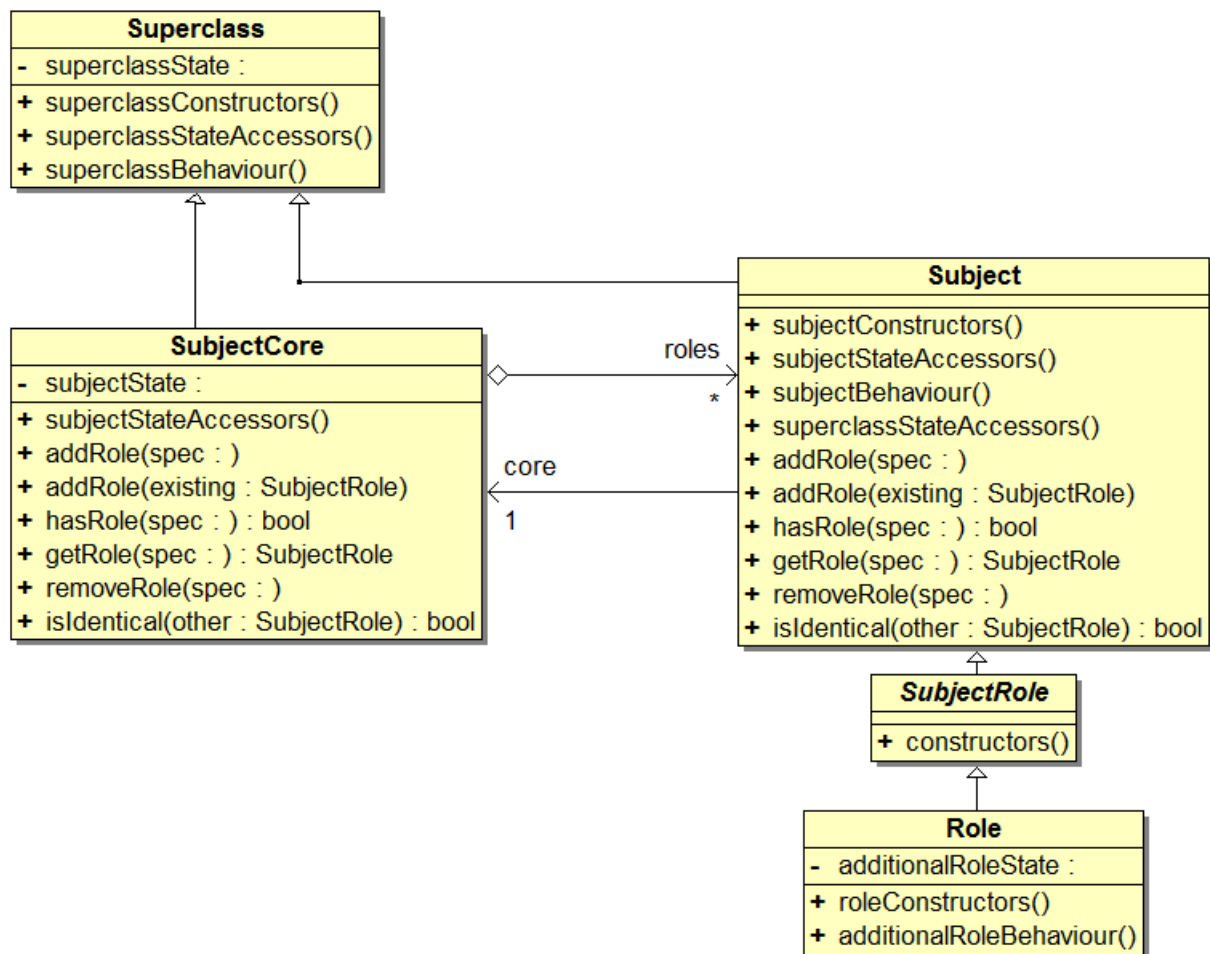


Abbildung 24: ILRO Zielzustand - Struktur schematisch

Die Menge der Vorbedingungen, die diese vereinfachte Variante des ursprünglich angestrebten Refactorings voraussetzt, ist deutlich kleiner, als die des in Kapitel 3 beschriebenen Ansatzes. Die ersten fünf der dort genannten Vorbedingungen gelten auch für die Anwendung des ILRO Refactorings – die übrigen nicht. Dafür kommen zwei neue Vorbedingungen hinzu:

1. PC_POSTFIX_FIELD_ACCESS
Field access in postfix expression.
2. PC_SUBJECT_EXTENDS_THROWABLE
Subject must not extend „java.lang.Throwable“.
3. PC_SUBJECT_IS_FINAL
Subject must not be final.
4. PC_SUBJECT_ONLY_PRIVATE_CONSTRUCTORS
Subject must contain at least one non-private constructor (including the default constructor).

5. PC_OPEN_RECURSION_IN_SUPERCLASS_CONSTRUCTOR

Superclass constructor must not contain open recursion.

6. PC_FIELD_INITIALIZER_CALLS_ON_THIS

Field initializer invokes a non-inherited method on „this“.

Bei Attributen, die bei ihrer Deklaration bereits initialisiert werden, werden die entsprechenden Anweisungen zusammen mit den Attributen in den neu angelegten Typ `SubjectCore` verschoben. Von dort aus sind die Methoden von `Subject` nicht sichtbar, so dass entsprechende Aufrufe zu Fehlern führen.

7. PC_SUBJECT_PRIVATE_INNER_TYPE

Subject must not contain private inner types.

Als `private` markierte innere Typen von `Subject` sind in `SubjectCore` nicht sichtbar. Dies führt zu Fehlern, falls sie im Rahmen von Methodensignaturen oder als Typen von Attributen verwendet werden.

Eine Gegenüberstellung beider Ansätze inklusive Bewertung befindet sich in den Kapiteln 6 und 7.

5 Implementierung

Die Entwicklungsumgebung Eclipse wurde aus mehreren Gründen für die Implementierung der in den Kapiteln 3 und 4 beschriebenen Refactorings ausgewählt: Zum Einen ist sie quelloffen, kostenlos verfügbar und insbesondere im Java-Umfeld weit verbreitet, zum Anderen bietet sie mit ihrer modularen Architektur aber auch eine umfassende Unterstützung für die Entwicklung und Einbindung von Plug-ins. Das folgende Kapitel bietet eine kurze Übersicht über die Komponenten von Eclipse, welche für die Implementierung von Refactoring-Werkzeugen für die Sprache Java nützlich sind. Anschließend werden Struktur und Vorgehensweise der Implementierung der im vorangegangenen Kapitel beschriebenen Lösungsansätze erläutert.

5.1 Refactorings mit Eclipse JDT

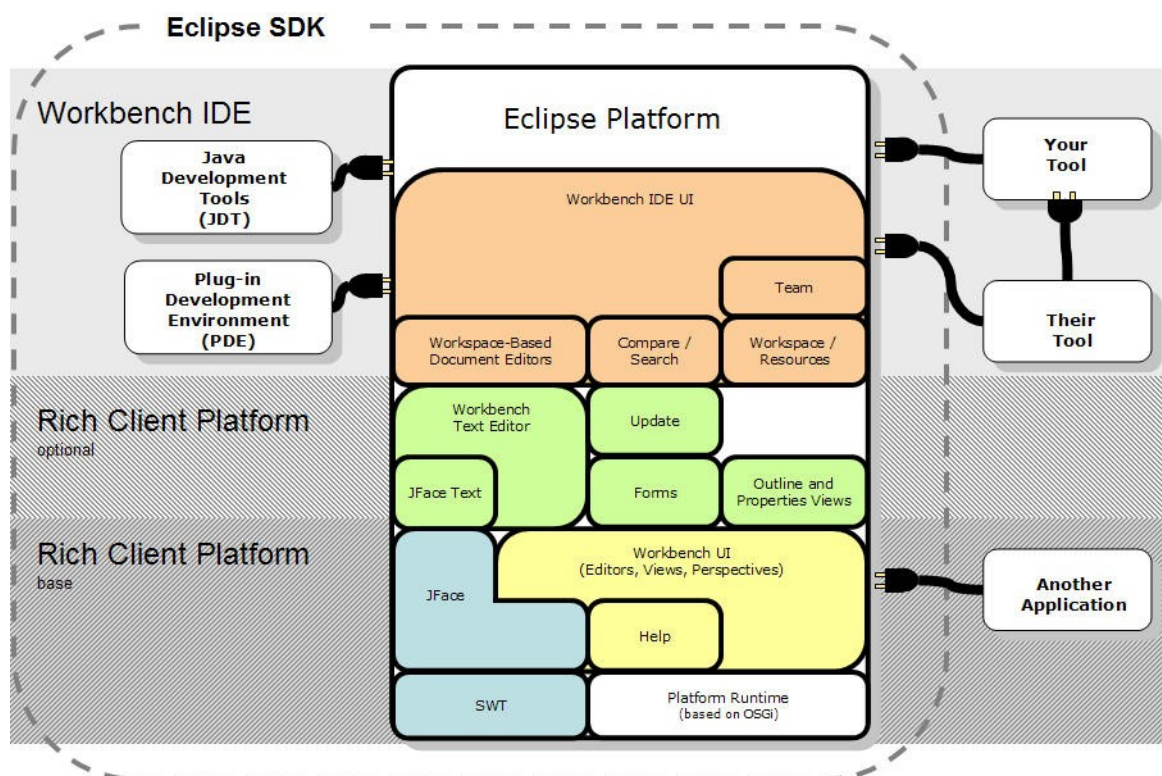


Abbildung 25: Architektur von Eclipse (Quelle: [D'Anjou2004])

Wie in Abbildung 25 dargestellt, ist Eclipse hochgradig modular in Form einzelner Komponenten aufgebaut, die aufeinander und letztendlich auf der relativ schlank gehaltenen „Platform Runtime“ aufsetzen. Sogar die für die Entwicklung von Eclipse selbst verwendeten Umgebungen Java Development Tools (JDT) zur Entwicklung von Java-Anwendungen und Plug-in Development Environment (PDE) zur Entwicklung von Plug-ins für die Eclipse Platform sind in Form von Plug-ins realisiert. Diese Plug-in-orientierte Architektur erlaubt die einfache und komfortable Einbindung des in den folgenden Abschnitten beschriebenen Refactoring Plug-ins in die

Entwicklungsumgebung. Im weiteren Verlauf dieses Abschnitts steht jedoch nicht die Architektur von Eclipse im Allgemeinen im Vordergrund, sondern diejenigen Komponenten, welche für die Entwicklung und Verwendung von Refactoring Tools für die Sprache Java relevant sind.

Zu nennen sind dabei vor allem das Language Toolkit (LTK), welches sprachunabhängige Werkzeuge für die Bearbeitung von Programmen zur Verfügung stellt. Ein wichtiger Bestandteil des LTK ist das Refactoring Framework, welches eine sprachunabhängige und bei Bedarf auch -übergreifende Unterstützung für die Durchführung von Refactorings bietet. Refactorings, die mit Hilfe des LTK realisiert werden, profitieren von vielen Features der Eclipse Platform (Grafische Oberflächen mit Vor- und Zurückbuttons, Vorschau- und Undo-Funktionalität etc.). [Frenzel2006] zeigt, wie sich mit Hilfe des LTK einfache Textdateien (z.B. Properties-Dateien) transformieren lassen.

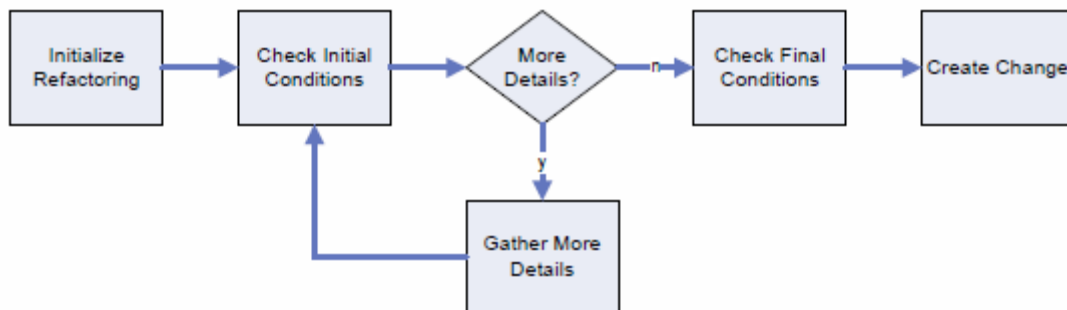


Abbildung 26: Ablauf LTK-basierter Refactorings (Quelle [Petito2007])

Abbildung 26 zeigt den grundsätzlichen Ablauf eines LTK-basierten Refactorings: Nach der Initialisierung wird ein Satz Vorbedingungen geprüft, der Aufschluss darüber gibt, ob das Refactoring überhaupt durchführbar ist und ob weitere Informationen für eine genauere Prüfung oder die Durchführung des Refactorings benötigt werden. Falls diese erste Prüfung die Anwendung des Refactorings nicht von vornherein ausschließt und alle benötigten Informationen vorliegen (z.B. nach Eingabe weiterer Daten oder Auswahl von Optionen durch den Anwender), kann die ausführliche Prüfung aller Vorbedingungen des Refactorings erfolgen. Erzielt auch diese ein positives Ergebnis, so kann die Beschreibung der umzusetzenden Veränderungen (Changes) erzeugt werden. In der Abbildung nicht dargestellt – aus Anwendersicht aber ebenfalls wichtig – kann anschließend optional eine Vorschau der geplanten Änderungen angezeigt werden, was dem Anwender die Möglichkeit bietet, diese zu prüfen und gegebenenfalls abubrechen oder zwei Schritte zurückzugehen und die von ihm beeinflussbaren Optionen anzupassen. Abschließend können die erzeugten Changes auf die zu modifizierenden Elemente angewandt und – bei entsprechender Auslegung der Changes – falls nötig auch wieder rückgängig gemacht werden.

Die Struktur der an dem oben dargestellten Ablauf beteiligten Elemente ist aus Abbildung 26 ersichtlich: Das Refactoring Werkzeug (in der Abbildung „Implementer“) wird auf ein zuvor ausgewähltes Element (Selected Element, z.B. eine Klasse oder ein Ausdruck) oder mehrere angewendet. Die Kommunikation mit dem Anwender erfolgt bidirektional über eine graphische Oberfläche – meist in Form eines mehrseitigen Dialogs. Durch Untersuchung des ausgewählten

Elements, Auswertung der vom Anwender vorgegebenen Optionen und unter Zugriff auf die im Arbeitsbereich verfügbaren und damit potentiell zu modifizierenden Ressourcen erzeugt das Refactoring-Werkzeug eine Beschreibung der durchzuführenden Änderungen (Changes). Diese können aus mehreren Teilschritten bestehen und wirken sich auf die Ressourcen im Arbeitsbereich aus.

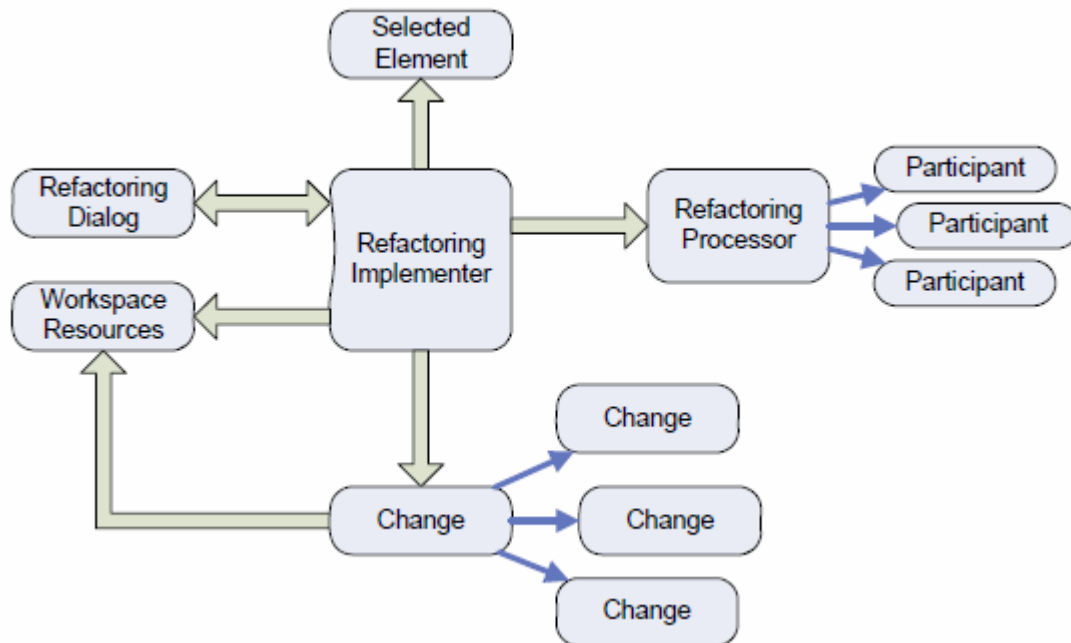


Abbildung 27: Struktur LTK-basierter Refactorings (Quelle: [Petito2007])

Das Refactoring Werkzeug kann einen so genannten „Refactoring Processor“ verwenden um anderen Plug-ins die Teilnahme am Refactoring Prozess zu ermöglichen. Diese müssen dafür passende „Participants“ zur Verfügung stellen, welche eigene Vorbedingungen und Changes beisteuern können. Ein Beispiel für eine solche Beteiligung sind spezielle Werkzeuge (z.B. für Webanwendungen oder Datenzugriffs-Frameworks), die bei der Umbenennung von Java-Klassen ihre Konfigurationsdateien entsprechend anpassen.

Komplizierter als im Beispiel von [Frenzel2006] wird es, wenn – anstelle einfacher Textdateien – Quellcode modifiziert werden soll. Dreh- und Angelpunkt für die Bearbeitung von Java-Programmen sind die JDT. [Widmer2006] beschreibt anhand eines relativ überschaubaren „Introduce Indirection“ Refactorings, wie JDT und LTK zusammen verwendet werden können, um Refactoring Werkzeuge für die Sprache Java auf Basis der Entwicklungsumgebung Eclipse zu realisieren. Dabei kommt den JDT vor allem in Hinblick auf die Analyse des zu transformierenden Programms und die Beschreibung der erforderlichen Veränderungen Bedeutung zu.

Leider ist die Menge der Quellen, die sich detailliert mit der Verwendung der JDT zur programmgesteuerten Manipulation von Quellcode auseinandersetzen, überschaubar. Eine wertvolle Übersicht über die zu diesem Zweck verwendbaren Komponenten liefert [IBM2005]. Insbesondere sind dies: Java Model, Search Engine und Abstract Syntax Tree (AST).

Das Java Model stellt eine schlanke Sicht auf die wichtigsten strukturellen Bausteine von Java Programmen (Typen, Attribute und Methoden) bereit, unabhängig davon, ob sie als Quellcode oder in kompilierter Form vorliegen. Mit Hilfe des Java Models lässt sich über die genannten Elemente navigieren, Typhierarchien sind analysierbar und einfache Operationen wie das Anlegen, Umbenennen und Löschen von Elementen (sofern sie im Quellcode vorliegen) sind möglich. Die genannten Möglichkeiten lassen sich schnell und Ressourcen-schonend nutzen, reichen aber zur vollständigen Analyse von Java-Programmen nicht aus.

Eine deutlich detailliertere, aber auch zeit- und speicherintensivere Analyse lässt sich mit Hilfe von Abstract Syntax Trees (ASTs) durchführen. Ein AST stellt den kompletten Syntaxbaum eines Programmelements (maximal einer kompletten „Compilation Unit“, was im Falle der Sprache Java dem Inhalt einer .java-Datei entspricht) dar und lässt sich durch das Parsen von Java-Quellcode erzeugen. Nach dem Parsen, das auch für einzelne Statements oder Code-Fragmente durchgeführt werden kann, enthält der AST einen Knoten für jedes einzelne syntaktische Element des geparsen Codes. ASTs lassen sich komfortabel mit Hilfe des Visitor Pattern (s. [Gamma1995]) analysieren und modifizieren. Dabei sind sie sogar in der Lage, vorgenommene Änderungen zu protokollieren, so dass diese relativ einfach in ein Change-Objekt zum Einsatz im Rahmen eines LTK-Refactorings integriert werden können.

Zentraler Nachteil von ASTs ist, dass sie – abgesehen von ihrer baumförmigen Struktur – zunächst keine Informationen über die Bedeutung einzelner Knoten und deren Zusammenhänge enthalten. So kann die Erkenntnis, ob es sich bei einem Knoten, der zum Beispiel einen Namen darstellt, um den Namen eines Typs, einer lokalen Variable oder einer Methode handelt, nur durch mehr oder weniger aufwändige Analyse der übrigen Knoten des ASTs gewonnen werden – wenn überhaupt. Die einzige Möglichkeit, die Bedeutung solcher Knoten ohne eigene, aufwändige Analyse korrekt zu bestimmen, ist die Verwendung der so genannten „Compiler Bindings“, welche den Bezug sowohl zum jeweiligen AST-Knoten als auch zum korrespondierenden Element des Java Models herstellen. Diese können für bestimmte Knoten (Typen, Variablen, Methoden und Ausdrücke) beim Parsen erstellt werden. Dies funktioniert allerdings nur beim Parsen kompletter, bereits vorhandener, fehlerfrei kompilierbarer Compilation Units, sofern diese als Quellcode verfügbar sind. Einmal erzeugt passen sich die Bindings im Gegensatz zu den Elementen des Java Models nicht mehr an Änderungen der Elemente an, auf die sie sich beziehen. Bindings können ausschließlich für die Analyse, nicht jedoch für die Modifikation von Programmelementen verwendet werden.

Die dritte wichtige Säule der JDT ist die Search Engine. Diese erlaubt die schnelle und in ihrer Genauigkeit feingranular anpassbare Suche nach Java-Elementen oder Text-Fragmenten in einem vorgebbaren Bereich (z.B. kompletter Arbeitsbereich, einzelnes Projekt oder konkretes Package). Interessant ist dabei insbesondere die Möglichkeit, alle Referenzen auf ein bestimmtes Element (z.B. einen Typ oder eine Methode) innerhalb des vorgegebenen Bereichs zu finden – falls nötig sogar kontextabhängig (z.B. alle Referenzen auf einen bestimmten Typ, bei denen dieser als obere Schranke einer Typvariablen dient). Diese Möglichkeit ist für Refactoring-Werkzeuge

ausgesprochen nützlich, da sie es ermöglicht, alle Vorkommen eines von den Veränderungen betroffenen Elements aufzuspüren, ohne dafür das komplette Programm selbst analysieren zu müssen.

5.2 Umsetzung der Refactorings

Die Implementierung der beiden in den Kapiteln 3 und 4 vorgestellten Lösungsansätze lässt sich grob in vier Bereiche gliedern:

1. Eclipse Plug-in Spezifika
Hierbei handelt es sich um Java-Klassen und Konfigurationsdateien, die das Eclipse Plug-in als solches kennzeichnen und beschreiben. Sie gewährleisten die Einbindung an die Benutzeroberfläche und die Anbindung an JDT. Die Plug-in Spezifika sind im Package `org.intoj.irop.plugin` implementiert.
2. Gemeinsame Elemente
Da der grundsätzliche Verarbeitungsablauf für beide Lösungsansätze gleich ist und viele der zu prüfenden Bedingungen und der durchzuführenden Transformationsschritte Gemeinsamkeiten aufweisen, bilden diese Elemente ein Framework, welches für beide Lösungsansätze gleichermaßen genutzt wird. Die gemeinsam genutzten Elemente sind im Package `org.intoj.irop.common` implementiert.
3. Elemente des Introduce Role Object Pattern Refactorings
Aufbauend auf dem o.g. gemeinsamen Framework implementieren diese Elemente die spezifischen Prüfungen und Transformationsschritte für den in Kapitel 3 beschriebenen Lösungsansatz „Role Object Pattern“. Die Elemente des IROP Refactorings sind im Package `org.intoj.irop.rop` implementiert.
4. Elemente des Introduce Lightweight Role Objects Refactorings
Analog zu Punkt 3 implementieren diese Elemente die Spezifika des in Kapitel 4 vorgestellten Lösungsansatzes „Lightweight Role Objects“. Die Elemente dieses ILRO Refactorings sind im Package `org.intoj.irop.lro` implementiert.

Da es sich bei den unter Punkt 1 zusammengefassten Elementen um reinen „Boilerplate“-Code handelt, werden in den folgenden Unterabschnitten ausschließlich die unter den Punkten 2 bis 4 genannten Elemente detailliert behandelt.

5.2.1 Gemeinsam genutzte Elemente

Die gemeinsam genutzten Elemente bilden ein Framework (zuzüglich einiger Utility-Klassen), auf welches sowohl das IROP Refactoring als auch das ILRO Refactoring aufsetzen. Es ist im Package `org.intoj.irop.common` implementiert (sofern nicht anders angegeben befinden sich alle in diesem Abschnitt beschriebenen Klassen und Packages dort) und setzt seinerseits auf die in Abschnitt 5.1 beschriebenen LTK und JDT Frameworks auf.

Das Infer Type Refactoring (bzw. dessen Implementierung) aus [Kegel2007] dient nicht als Grundlage der hier beschriebenen Refactorings. Grundsätzlich wäre die Verwendung von Typinferenz zur Bestimmung des minimalen Interfaces von `Subject` im Rahmen des Ansatzes IROP zwar möglich, sie kommt hier jedoch aus den in Abschnitt 3.7 genannten Gründen nicht zum Einsatz. Auch erfolgen die Prüfung der Vorbedingungen und die Bestimmung der notwendigen Änderungen des zu modifizierenden Programms im Gegensatz zu [Kegel2007] nicht Constraint-basiert, so dass die entsprechenden Elemente der Implementierung des Inter Type Refactorings hier ebenfalls keine Verwendung finden.

Das zentrale Element des Frameworks ist die Klasse `IROPRefactoring`, welche ein LTK Refactoring gemäß Abbildung 26 implementiert. Die Anwendung des Refactorings erfolgt in fünf Schritten:

Der erste Schritt – `checkInitialConditions` – dient dabei lediglich der Prüfung, ob das im Eclipse Workspace selektierte Element überhaupt für die Anwendung des Refactorings in Frage kommt, d.h. ob es sich um eine benannte top-level Java-Klasse handelt, deren Quellcode vorliegt und modifizierbar ist.

Im zweiten Schritt erhält der Anwender die Gelegenheit, die Benennung der vom Refactoring zu modifizierenden oder zu erzeugenden Elemente anzupassen und zu entscheiden, ob ein Subjekt mehrere Instanzen eines Rollentyps verwalten können soll oder nicht. Eine Entscheidung für diese Option führt zu etwas komplexerem Code. Insbesondere muss der Anwender sich darum kümmern, jeweils die richtige Instanz einer Rolle zu verwenden, wenn das Subjekt über mehrere Rollen desselben Typs verfügt. Im Gegenzug ist diese Lösung jedoch flexibler einsetzbar und erfüllt Eigenschaft 4 der in Abschnitt 2.2 beschriebenen Eigenschaften von Rollenkonzepten.

Der dritte Schritt – `checkFinalConditions` – beinhaltet mehrere Aktionen:

1. Finden aller für die Umsetzung des Refactorings benötigten Elemente. Dazu gehören: Alle Sub-, Super- und Geschwisterklassen des zukünftigen Subjekts (`Subject`), alle Attribute und Methoden (jeweils eigene und geerbte) von `Subject`, alle Clients von `Subject` (i.e. alle Klassen, die `Subject` oder dessen Subklassen instantiiieren, direkt auf Attribute von `Subject` zugreifen oder eine der Superklassen von `Subject` als Schranke eines generischen Typparameters verwenden).
2. Prüfen, ob die zuvor gefundenen, vom Refactoring betroffenen Elemente die erforderlichen Vorbedingungen erfüllen. Um die Auswertung der Anwendbarkeit der Refactorings zu erleichtern, werden hierbei stets alle Vorbedingungen beider Refactorings überprüft. Ob die Verletzung einer Vorbedingung für das jeweilige Refactoring relevant ist, wird von dessen konkreter Implementierung der Framework-Klasse `IROPRefactoring` bestimmt.

3. Bestimmen der notwendigen Modifikationen bereits bestehenden Codes zur Umsetzung des jeweiligen Refactorings. Da dieser Schritt naturgemäß stark von dem konkreten Refactoring bestimmt wird, findet seine Implementierung auch erst auf dessen Ebene statt (s. Abschnitte 5.2.2 und 5.2.3).
4. Bestimmen der notwendigen Ergänzungen des bereits bestehenden Codes. Beschränkt sich Punkt drei auf die Modifikation von Klassen und Interfaces, die bereits vorhanden sind, so bezieht sich der Begriff Ergänzung in diesem Fall auf das Hinzufügen neuer Klassen und Interfaces. Diese Differenzierung ergibt sich aus der unterschiedlichen Herangehensweise, welche der Umgang mit zuvor nicht vorhandenen Elementen durch die JDT im Vergleich zu bereits bestehendem Code erfordert. Analog zum vorangegangenen Punkt ist die Umsetzung dieser Aktion vom konkreten Refactoring abhängig und dementsprechend dort implementiert.

Die Punkte drei und vier der o.g. Aufzählung wären nach Betrachtung von Abbildung 26 eigentlich erst im nächsten Schritt zu erwarten. [Widmer2006] argumentiert jedoch, dass zur Überprüfung der Vorbedingungen ohnehin ein Großteil des Aufwands zur Bestimmung der erforderlichen Änderungen erfolgen muss und es daher aus Performancegründen sinnvoll ist, beides miteinander zu verbinden.

In Schritt vier wird dem Anwender eine Vorschau der zur Durchführung des Refactorings erforderlichen Änderungen des zu modifizierenden Programms präsentiert. Diese Funktionalität wird vom LTK bereitgestellt, welches dafür auf die Methode `createChanges` der Klasse `IROPRefactoring` zurückgreift. Da diese Änderungen bereits in Schritt drei ermittelt werden, beschränkt sich die Implementierung dieser Methode auf die Rückgabe der zuvor bestimmten Changes.

Schritt fünf beinhaltet schließlich die Anwendung der o.g. Änderungen auf das zu modifizierende Programm, sofern der Anwender diesen nach Sichtung der Vorschau zugestimmt hat. Die Implementierung dieses Schritts erfolgt vollständig durch das LTK.

Neben der Klasse `IROPRefactoring`, die das oben beschriebene Vorgehen implementiert, sind in den gemeinsamen Elementen zahlreiche weitere Klassen enthalten, die die Implementierung der konkreten Refactorings unterstützen:

1. Die Packages `ui` und `configuration` stellen einen Rahmen für die Darstellung (gui) und Beschreibung der vom Anwender in Schritt zwei vorgebbaren Optionen sowie die Speicherung der jeweils tatsächlich erfolgten Vorgaben dar. Die konkreten Refactorings müssen nur die o.g. Beschreibung bestücken, um den Anwender ihre jeweiligen Optionen zu präsentieren und Zugriff auf die entsprechenden Vorgaben zu erhalten.
2. Das Package `preconditions` enthält die Klassen `Precondition`, welche eine einzelne Verletzung einer Vorbedingung beschreibt und `Preconditions`, welche als Container aller verletzten Vorbedingungen im Zuge der Anwendung eines Refactorings dient. Das Sammeln

verletzter Vorbedingungen in einem, von `IROPRefactoring` verwalteten, Container erlaubt die möglichst vollständige Prüfung aller Vorbedingungen ohne einen Abbruch des Refactorings zu provozieren. So können dem Anwender nicht nur die erste, sondern alle durch das zu modifizierende Programm verletzten Vorbedingungen angezeigt werden.

3. Die Analyse des zu modifizierenden Programms (Finden der benötigten Elemente und Prüfung der Vorbedingungen) kann leider nicht in allen Fällen über das leichtgewichtige und performante Java Model der JDT erfolgen. So sind über das Java Model zwar die Sub- und Superklassen von `Subject` bestimmbar, nicht jedoch seine Clients. Diese spürt die Klasse `ClientFinder` (Package `search`) mit Hilfe der Search API von Eclipse auf. Auch bestimmte Vorbedingungen lassen sich nur durch Analyse des Abstract Syntax Tree (AST) der betroffenen Elemente überprüfen. Hierfür steht im Package `visitors.check` die Klasse `CheckVisitor` zur Verfügung, die `ASTVisitor` (aus JDT) erweitert und deren Subklasseninstanzen jeweils für die Überprüfung einer bestimmten Vorbedingung auf potentiell betroffene Elemente angewendet werden können. Insgesamt werden für nur vier der in den Kapiteln 3 und 4 definierten Vorbedingungen solche Visitors benötigt:
 - 3.1. `DetectOpenRecursionVisitor`: wird zur Überprüfung der Vorbedingung `PC_OPEN_RECURSION_IN_SUPERCLASS_CONSTRUCTOR` (s. Abschnitt 3.8) auf die Konstruktoren der Superklassen von `Subject` angewendet und prüft, ob dort Methodenaufrufe auf `this` (sowohl explizit als auch implizit) erfolgen.
 - 3.2. `DetectPrivateMemberAccessThroughVariableVisitor` wird zur Überprüfung der Vorbedingung `PC_SUBJECT_PRIVATE_MEMBER_ACCESS` (s. Abschnitt 3.8) auf die Klasse `Subject` angewendet und prüft, ob darin auf private Attribute anderer Objekte desselben Typs zugegriffen wird.
 - 3.3. `InitializerVisitor` wird zur Überprüfung der beiden Vorbedingungen `PC_FIELD_INITIALIZER_CALLS_PRIVATE_METHOD` (s. Abschnitt 3.8) und `PC_FIELD_INITIALIZER_CALLS_ON_THIS` (s. Kapitel 4) auf die Initialisierungsstatements der Attribute von `Subject` angewendet und prüft, ob diese Aufrufe von privaten Methoden bzw. Methodenaufrufe auf `this` (explizit oder implizit) enthalten.
 - 3.4. `PostfixFieldAccessVisitor` wird zur Überprüfung der Vorbedingung `PC_POSTFIX_FIELD_ACCESS` (s. Abschnitt 3.8) auf alle Anweisungen angewendet, welche auf Attribute zugreifen, auf die das Self Encapsulate Field Refactoring angewendet werden soll. Dabei wird festgestellt, ob die betroffenen Attribute innerhalb von Postfix-Ausdrücken verwendet werden.
4. Neben `check` enthält `visitors` noch die Packages `collect` und `modify`. Diese sind analog zu `visitors.check` aufgebaut. Die dort enthaltenen `ASTVisitors` erfüllen jedoch andere Aufgaben: Diejenigen in `collect` dienen der Sammlung der für die Durchführung

des Refactorings benötigten Informationen über Attribute und Methoden von `Subject`, welche nicht aus dem Java Model ersichtlich sind. Die so gewonnenen Informationen werden dann in den entsprechenden Instanzen der Klassen aus dem Package `util.info` abgelegt. Die Visitors in `modify` bilden die Grundlage für die Modifikation bereits vorhandener Programmelemente im Rahmen der dritten Aktion aus Schritt drei (s.o.). Insbesondere die Anwendung des in Abschnitt 3.3 genannten Self Encapsulate Fields Refactorings erfolgt durch einen solchen Visitor (Klasse `SelfEncapsulateFieldAccessVisitor`).

5. Im Gegensatz zur Modifikation bereits bestehender Elemente kann das Hinzufügen zusätzlicher Klassen und Interfaces in Form neuer Compilation Units nicht durch `ASTVisitors` erfolgen, da diese nur auf bereits vorhandenen Baumstrukturen operieren können. Die Grundlage für die Ergänzung neuer Programmelemente bilden die Klassen im Package `creators`. Deren konkrete Erweiterungen im Rahmen der einzelnen Refactorings müssen ausschließlich Name, Art (Klasse oder Interface), Supertypen und natürlich den Inhalt des zu erzeugenden Elements definieren. Die eigentliche Erzeugung verläuft davon unabhängig immer gleich und ist dementsprechend als Teil der gemeinsamen Elemente in den o.g. Klassen implementiert.
6. Abschließend sind noch die Packages `util.exceptions` und `util.manipulation` zu erwähnen. Ersteres enthält eine Exception, die ausgelöst wird, wenn eine Typvariable im Rahmen der Analyse nicht aufgelöst werden kann, während letzteres zwei Utility-Klassen enthält, die eine Reihe nützlicher Methoden zur Manipulation von ASTs bieten. Dazu gehören: Das Erzeugen von ASTs, das Finden bestimmter Knoten innerhalb eines ASTs, das Austauschen von Knoten, das Ermitteln voll qualifizierter Typnamen und Auflösen von Typvariablen, usw.

Die in den beiden folgenden Abschnitten erläuterten Implementierungen der beiden in den Kapiteln 3 und 4 beschriebenen Lösungsansätze können sich aufgrund der umfangreichen Funktionalität, die durch die gemeinsamen Elemente zur Verfügung gestellt wird, darauf beschränken `IROPRefactoring` durch die Definition der relevanten Vorbedingungen und der zur Verfügung stehenden Konfigurationselemente sowie die Einbindung eigener Visitors und Creators zur Bestimmung der erforderlichen Veränderungen des zu modifizierenden Programms zu erweitern.

5.2.2 Elemente des Introduce Role Object Pattern Refactorings

Da die Implementierung des IROP Refactorings auf das im vorangegangenen Abschnitt beschriebene Framework aufsetzt, ist sie von ihrer Struktur her recht einfach aufgebaut: Die Klasse `ROPRefactoringWizard` stellt den Einsprungpunkt in dieses konkrete Refactoring dar. `ROPRefactoring` erweitert `IROPRefactoring` (aus `common`, s.o.) um die Angabe der für diese Variante relevanten Vorbedingungen (s. Abschnitt 3.8), die Definition der vom Anwender vorgebbaren Optionen sowie eine Reihe von Visitors und Creators, die der Bestimmung der für die

Umsetzung des Refactorings erforderlichen Änderungen dienen. Da durch die Bestimmung der Änderungen die eigentliche Umsetzung des Refactorings erfolgt, wird die Funktion der daran beteiligten Elemente im Folgenden genauer erläutert:

`ROPCreator` dient als gemeinsame Oberklasse aller `Creators` dieses Refactorings und stellt eine Reihe von Hilfsmethoden zur Verfügung, die von den `Creators` dieses Refactorings, nicht jedoch von denen des ILRO Refactorings benötigt werden.

`CoreCreator` erzeugt die abstrakte Klasse `SubjectCore`, welche die Methoden von `Subject` implementiert. Abweichend von der Beschreibung in Kapitel 3 und [Steimann2010] wird bei dieser Implementierung nicht `Subject` in `SubjectCore` transformiert und ein zusätzliches Interface mit dem Namen von `Subject` angelegt, sondern umgekehrt: `Subject` wird in ein Interface gleichen Namens transformiert und `SubjectCore` neu erzeugt. Der Grund für diese umgekehrte Vorgehensweise ist, dass es von der Implementierung her komplizierter ist, eine `Compilation Unit` unter einem Namen anzulegen, der bereits existiert. Das Ergebnis ist unabhängig von dieser Abweichung bei der Vorgehensweise aber in beiden Fällen identisch.

`CoreCreator` legt `SubjectCore` als abstrakte Klasse mit der ehemaligen Superklasse von `Subject` als Superklasse und dem in ein Interface transformierten `Subject` als Superinterface an. Generische Typparameter werden von `Subject` übernommen, sofern vorhanden. Für die (eigenen und geerbten) Attribute von `Subject` erhält `SubjectCore` passende Zugriffsmethoden (`getter/setter`). Die Methoden und Konstruktoren von `Subject` (nur eigene) werden inklusive Implementierung übernommen, wobei alle nicht-privaten Methoden `public` werden (was aufgrund der Transformation von `Subject` in ein Interface notwendig ist) und alle privaten Konstruktoren `protected` (was notwendig ist, damit `Delegate` darauf Zugriff erhält). Darüber hinaus wird für jeden Konstruktor eine Methode zur Initialisierung von `SubjectState` (s. Abschnitt 3.6) mit denselben Parametern angelegt.

`StateCreator` erzeugt die Klasse `SubjectState` als Subklasse von `SubjectCore`. Ihre Aufgabe ist die Verwaltung des Zustands und der Rollen des Subjekts. Sie erhält dieselben Typparameter wie `Subject` und implementiert das Interface `java.io.Serializable`, falls dies auch bei `Subject` der Fall ist. Neben den Attributen von `Subject` und den dazugehörigen Zugriffsmethoden erhält `SubjectState` seinen Aufgaben gemäß eine Implementierung der Rollenmanagementmethoden, wie sie das Role Object Pattern vorsieht. Darüber hinaus übernimmt `SubjectState` die Konstruktoren von `Subject`, wobei `private` Konstruktoren `protected` werden, damit es nicht zu Konflikten mit der Superklasse `SubjectCore` kommt, wo dies ebenfalls geschieht.

`RoleCreator` erzeugt die abstrakte Klasse `SubjectRole` als Subklasse der ehemaligen Superklasse von `Subject` und mit dem zum Interface transformierten `Subject` als Superinterface. Diese Klasse dient als abstrakte Oberklasse aller Rollen des Subjekts, also der ehemaligen Subklassen von `Subject`. Die Typparameter sind auch hier dieselben wie bei `Subject`.

`SubjectRole` enthält `Delegate` als innere Klasse (um den Trick des Reverse Forwarding einsetzen zu können, s. Abschnitt 3.5), diese wird jedoch durch einen separaten Creator erzeugt (s.u.). `SubjectRole` erhält alle Konstruktoren und Methoden von `Subject` und implementiert erstere durch Anlegen einer entsprechenden Instanz von `Delegate` und letztere durch Delegation an genau die zuvor durch einen der Konstruktoren erzeugte Instanz von `Delegate`. Auch die Zugriffsmethoden auf die Attribute von `Subject` werden durch Delegation an `Delegate` implementiert.

`DelegateCreator` erzeugt `Delegate` als innere Klasse von `SubjectRole` und Subklasse von `SubjectCore`. Diese innere Klasse dient ausschließlich der Umsetzung des in Abschnitt 3.5 beschriebenen Reverse Forwardings. Sie hält eine Referenz auf die Instanz von `SubjectState`, welche den Kern des Subjekts darstellt, zu dem die sie umschließende Rolleninstanz gehört. `Delegate` implementiert zum Einen Zugriffs- und Rollenmanagementmethoden durch Delegation an `SubjectState`, zum Anderen die Konstruktoren von `Subject` durch Initialisieren einer passenden Instanz von `SubjectState` und darüber hinaus die Methoden von `Subject` jeweils durch ein Methodenduo, welches das gewünschte Reverse Forwarding umsetzt.

`NullRoleCreator` wird benötigt, da `Subject` nach erfolgter Transformation nicht mehr instantiierbar ist, es jedoch nicht ausgeschlossen werden kann, dass vor Anwendung des Refactorings Instanzen von `Subject` direkt erzeugt wurden. Da `SubjectCore`, `-State` und `-Role` sowie `Delegate` als Implementierungsdetails verborgen bleiben sollen, wird eine leere Rolle benötigt, die überall dort einspringt, wo bisher `Subject` direkt instantiiert wurde. Diese Rolle erzeugt `NullRoleCreator` als Subklasse von `SubjectRole`, die nichts weiter enthält als die Konstruktoren von `Subject`, deren Argumente sie an die entsprechenden Konstruktoren von `SubjectRole` weiterreicht.

`SuperclassInterfaceCreator` erzeugt zu jeder der ehemaligen Superklassen von `Subject` ein idiosynkratisches Interface, welches aufgrund der in Abschnitt 3.7 geschilderten Problematik benötigt wird. Die mit Hilfe dieses Creators erzeugten Interfaces bilden eine parallele Vererbungshierarchie zu derjenigen der ehemaligen Superklassen von `Subject`.

`ROPVISITOR` fungiert als gemeinsame abstrakte Oberklasse aller Visitors dieses Refactorings. Die Subklassen dieses Visitors erben eine Reihe von Modifikationen, die für alle zu modifizierenden Elemente gleichermaßen gelten. Dazu gehören: Das Kapseln von Feldzugriffen (Self Encapsulate Fields auf `Subject` angewendet, s. Abschnitt 3.3), das Ersetzen von Instantiierungen von `Subject` durch Instantiierungen von `SubjectNullRole` und das Ersetzen einfacher Typnamen durch ihre vollqualifizierten Pendanten. Letzteres ist streng genommen für das Refactoring selbst nicht direkt notwendig, dadurch lassen sich jedoch Fehler im Zusammenhang mit `import`-Anweisungen zuverlässig vermeiden.

`SubjectVisitor` ist für die Transformation der Klasse `Subject` in ein gleichnamiges Interface zuständig. Neben den geerbten Modifikationen setzt er folgende Änderungen um: `Subject` wird in ein Interface umgewandelt, die Superklasse wird, falls vorhanden, durch das entsprechende idiosynkratische Interface als Superinterface ersetzt, alle nicht-statischen Attribute werden entfernt, stattdessen werden die entsprechenden Zugriffsmethoden hinzugefügt, alle `private` Methoden werden entfernt, die übrigen Methoden werden `public` und die Signaturen der Rollenmanagementmethoden sowie die `import`-Anweisungen der dazugehörigen Exceptions werden hinzugefügt.

`SuperclassVisitor` wird auf alle Superklassen von `Subject` angewendet. Er sorgt dafür, dass das idiosynkratische Interface der Superklasse von dieser implementiert wird, erhöht die Sichtbarkeit aller nicht-statischen, nicht-privaten Methoden auf `public` und fügt Zugriffsmethoden für alle an `Subject` vererbten Attribute hinzu.

`SiblingVisitor` wird auf alle Geschwisterklassen von `Subject` angewendet. Das sind solche Klassen, die zwar weder Sub- noch Superklassen von `Subject` sind, aber mit diesem gemeinsame Superklassen haben (ausgenommen natürlich `java.lang.Object`). Diese sind nur insofern von dem Refactoring betroffen, als dass die Sichtbarkeit geerbter Methoden entsprechend der durch `SuperclassVisitor` vorgenommenen Veränderungen angepasst werden muss.

`RolesVisitor` ist für die direkten benannten Subklassen von `Subject` zuständig. Deren Superklasse ist von `Subject` in `SubjectRole` zu ändern und gegebenenfalls muss auch hier die Sichtbarkeit geerbter Methoden angepasst werden.

`SubclassOfRoleVisitor` wird auf alle indirekten benannten Subklassen von `Subject` angewendet. Die Änderungen erfolgen analog zu `RolesVisitor` mit der Ausnahme, dass eine Anpassung der Superklasse nicht nötig ist.

`AnonymousRolesVisitor` ist für alle direkten, aber anonymen Subklassen von `Subject` zuständig. Deren Superklasse ist analog zu den benannten Rollen in `SubjectRole` zu ändern, allerdings ist die Syntax bei anonymen Subklassen eine andere, so dass der Einfachheit halber ein separater Visitor zum Einsatz kommt.

`AnonymousSubclassOfRoleVisitor` wird auf die indirekten anonymen Subklassen von `Subject` angewendet, also auf anonyme Subklassen von Subklassen (Subklassen anonymer Klassen gibt es nicht). Über das von den übergeordneten Visitors Geerbte hinaus sind keine weiteren Änderungen erforderlich.

`ClientVisitor` gilt für alle Clients, die in keiner Vererbungsbeziehung zu `Subject` stehen, aber dieses instantiiert, auf seine Attribute zugreifen oder `Subject` bzw. eine seiner Superklassen als Schranke generischer Typparameter verwenden. Die notwendigen Modifikationen dieser Elemente stimmen mit den im Rahmen der Beschreibung von `ROPVisitor` (s.o.) genannten überein, werden also von `ClientVisitor` geerbt, womit dort keine weiteren Änderungen erforderlich sind.

`TransformThisExpressionVisitor` sei nur der Vollständigkeit halber erwähnt. Dieser Visitor modifiziert keine komplette Klasse, sondern stellt lediglich eine Hilfe dar, eines der vielen kleinen Java-spezifischen Probleme bei der Anwendung dieses Refactorings zu behandeln. Klassennamen von Zugriffen per `this`-Ausdruck aus anonymen Klassen innerhalb von Methoden des Subjekts auf die umschließende Instanz ändert er in den Klassennamen von `SubjectCore`. So behält die betroffene Methode nach dem Verschieben von `Subject` nach `SubjectCore` ihre Gültigkeit. Am Beispiel der Methode `runTest` der Klasse `ActiveTestSuite` (aus JUnit 3.8.1) lässt sich dies nachvollziehen:

```
public void runTest(final Test test, final TestResult result) {
    Thread t= new Thread() {
        public void run() {
            try {
                test.run(result);
            } finally {
                ActiveTestSuite.this.runFinished(test);
            }
        }
    };
    t.start();
}
```

Wird diese Methode in eine andere Klasse verschoben (z.B. `ActiveTestSuiteCore`), so muss auch der `this`-Ausdruck in obigem Beispiel entsprechend angepasst werden.

5.2.3 Elemente des Introduce Lightweight Role Objects Refactorings

Die Struktur des ILRO Refactorings ist analog zu der im vorangegangenen Abschnitt beschriebenen gestaltet. Aufgrund der beträchtlichen Vereinfachung, die dieser Ansatz gegenüber dem IROP Refactoring darstellt, ist die Menge der Creators und Visitors jedoch deutlich kleiner. Selbst auf gemeinsame Oberklassen (analog zu `ROPVisitor` bzw. `-Creator`) konnte verzichtet werden, da alle Gemeinsamkeiten der hier verwendeten modifizierenden Elemente auch im IROP-Ansatz Verwendung finden und somit bereits im Bereich der gemeinsam genutzten Elemente enthalten sind. Im Einzelnen umfasst die Implementierung des ILRO-Ansatzes folgende Creators und Visitors:

`CoreCreator` erzeugt die Klasse `SubjectCore`, welche sowohl den Zustand als auch die Rollen des Subjekts verwaltet. Diese Klasse ist `public` und `final`, implementiert gegebenenfalls `java.io.Serializable` (falls dies auch für `Subject` gilt) und verhält sich in Bezug auf Typparameter und Superklasse genau wie `Subject`. Neben Zugriffsmethoden für alle Attribute von `Subject` enthält `SubjectCore` leere Stubs für alle geerbten abstrakten Methoden sowie eine Implementierung der Rollenmanagementmethoden gemäß dem Role Object Pattern. In Sachen Konstruktoren verfügt die Klasse über einen Konstruktor für jeden für sie sichtbaren Konstruktor ihrer direkten Superklasse. Diese Konstruktoren haben einzig die Aufgabe, die erhaltenen

Argumente an den korrespondierenden Konstruktor der Superklasse weiterzuleiten. Die Initialisierung der Attribute von `SubjectCore` (also des Subjektzustands) erfolgt über deren Zugriffsmethoden aus `Subject` selbst heraus.

`RoleCreator` erzeugt die Klasse `SubjectRole` als Superklasse aller Rollen (also der bisherigen Subklassen von `Subject`). Diese ist abstrakt und eine direkte Subklasse von `Subject`. Auch etwaige Typparameter stimmen mit denen von `Subject` überein. Darüber hinaus enthält `SubjectRole` für jeden Konstruktor von `Subject` ein entsprechendes Gegenstück mit der Aufgabe, empfangene Argumente einfach durchzureichen. Der einzige Zweck dieser relativ leeren Klasse besteht darin, eine typsichere Differenzierung zwischen `Subject` und seinen Rollen zu erreichen.

`SubjectVisitor` transformiert `Subject`, was sich deutlich weniger komplex gestaltet als beim oben beschriebenen IROP-Ansatz: Eine Referenz auf `SubjectCore` wird angelegt, alle nicht-statischen Attribute werden entfernt, in allen Constructoren, die keinen Konstruktor derselben Klasse aufrufen (`this(...)`), wird die Referenz auf `SubjectCore` initialisiert, Self Encapsulate Fields wird durchgeführt, dabei delegieren die Zugriffsmethoden an ihre jeweiligen Pendants in `SubjectCore` ebenso wie die neu hinzugefügten Rollenmanagementmethoden. Sollte `Subject` nicht über einen expliziten Konstruktor verfügen, so wird ein Default-Konstruktor angelegt, dessen einzige Aufgabe die Initialisierung der Referenz auf `SubjectCore` ist.

`SuperclassVisitor` führt Self Encapsulate Fields für alle Superklassen von `Subject` mit Ausnahme von deren Constructoren durch und ergänzt die betroffenen Klassen um die dafür benötigten Zugriffsmethoden.

`RolesVisitor` wird ausschließlich auf die direkten benannten Subklassen von `Subject` angewendet und ändert deren Superklasse in `SubjectRole`.

`AnonymousRolesVisitor` ändert die Superklasse aller direkten anonymen Subklassen von `Subject` in `SubjectRole`. Analog zum IROP-Ansatz besteht die Trennung von `RolesVisitor` und `AnonymousRolesVisitor` allein aufgrund der syntaktischen Unterschiede bei der Angabe der Superklasse.

6 Evaluation der Anwendbarkeit

Dieses Kapitel beschreibt die Kriterien zur Überprüfung der Qualität der beiden in den vorangegangenen Kapiteln vorgestellten Refactoring-Werkzeuge und liefert die zur Durchführung dieser Untersuchung benötigten Rohdaten. Eine Bewertung der im Folgenden vorgestellten Ergebnisse findet in Kapitel 7 statt.

Ein automatisiert anwendbares Refactoring muss folgende Anforderungen erfüllen:

1. Korrektheit:
Dies bedeutet, dass das durch das Refactoring modifizierte Programm alle Nachbedingungen des Refactorings erfüllt. Dazu gehört insbesondere, dass das beobachtbare Verhalten des Programms durch die Anwendung des Refactorings nicht geändert werden darf.
2. Komplexität:
Ziel eines Refactorings ist die Verbesserung eines Programmcodes hinsichtlich der nicht-funktionalen Kriterien Lesbarkeit, Wartbarkeit und Erweiterbarkeit, das heißt, der Vereinfachung der Weiterentwicklung des Programms, ohne dass dabei das Verhalten des Programms eine Änderung erfährt. Die genannten Kriterien sind allerdings nur schwer messbar, daher wird im Folgenden die reine Änderung des Umfangs des modifizierten Programms, gemessen in Anzahl der Codezeilen (Lines of Code, LoC), als Indiz für die Komplexität des Programms herangezogen.
3. Anwendbarkeit:
Die Anwendung eines Refactorings setzt die Einhaltung bestimmter Vorbedingungen voraus. Je restriktiver diese sind, auf desto weniger Programme bzw. Programmelemente lässt sich das Refactoring anwenden. Ein Refactoring-Werkzeug, das nicht anwendbar ist, kann jedoch seinen Nutzen nicht entfalten, so dass eine möglichst hohe Anwendbarkeitsrate wünschenswert ist.

Die drei in diesem Abschnitt genannten Kriterien gelten grundsätzlich für jedes Refactoring und werden in Abschnitt 6.1 untersucht. Darüber hinaus ist für die Bewertung der Qualität der in dieser Arbeit vorgestellten Refactorings relevant, inwiefern der resultierende Programmcode die in Abschnitt 2.2 vorgestellten Eigenschaften von Rollenkonzepten implementiert. Dieser Frage wird in Abschnitt 6.2 nachgegangen.

6.1 Anwendbarkeit, Korrektheit und Komplexität

Wie schon in [Kegel2008] und [Steimann2010] beschrieben, ist der Nachweis der Korrektheit selbst einfachster Refactorings in Hinblick auf die unveränderte Beibehaltung des beobachtbaren Verhaltens des modifizierten Programms nur sehr schwer zu führen. Ist dieser Beweis formal nicht oder nur mit unvertretbarem Aufwand zu leisten, so bleibt nur das empirische Vorgehen: Durch

Überprüfen der Korrektheit des Refactorings in einer Anzahl von Testfällen, die möglichst viele der denkbaren Anwendungsfälle des Refactorings abdecken, kann auf die Korrektheit des Refactorings in den entsprechenden Anwendungsfällen geschlossen werden.

Die im Folgenden angewandte Methode zur Prüfung der Korrektheit des Refactorings im Einzelfall besteht darin, das Refactoring auf möglichst viele Klassen eines Programms, zu dem eine möglichst große Anzahl von Testfällen vorliegt, anzuwenden und anschließend festzustellen, ob die entsprechenden Test weiterhin fehlerfrei durchgeführt werden können. Als Korrektheitsnachweis weist diese Methode zwei Probleme auf:

1. Wird ein Anwendungsfall durch die überprüften Einzelfälle nicht abgedeckt, so bleibt die Korrektheit des Refactorings für diesen Anwendungsfall ungeprüft.
2. Wird ein Fehler bei der Anwendung des Refactorings im Einzelfall durch die vorliegenden Testfälle nicht entdeckt, so bleibt er unbemerkt.

Der Vorteil der Methode ist allerdings, dass sich durch die massenhafte Anwendung des Refactorings zwecks Korrektheitsnachweis gleichzeitig auch Erhebungen zur Anwendbarkeit des Refactorings und Beobachtungen über die Veränderung des Programmumfangs anstellen lassen. Für die folgende Auswertung wurden vier bekannte Java-Projekte ausgewählt, deren Quellcode öffentlich zugänglich ist und die zum Einen umfangreichen Gebrauch von Java-Sprachmitteln machen, die im Rahmen von Refactorings mit besonderer Sorgfalt zu behandeln sind (Generics, innere und anonyme Klassen, komplexe Vererbungshierarchien, Einsatz von Annotations und Marker-Interfaces, Verwendung statischer Methoden und Variablen, etc.) und zum Anderen über eine große Anzahl definierter Testfälle verfügen. Diese ausgewählten Projekte sind:

1. Commons Collections with Generics 4.01
erhältlich unter Apache License 2.0 von <http://sourceforge.net/projects/collections/>
2. JPaul 2.5.1
erhältlich unter einer modifizierten Version der BSD License von <http://jpaul.sourceforge.net/>
3. JUnit 3.8.1
erhältlich unter IBM's Common Public License 0.5 von <http://junit.sourceforge.net/>
4. JUnit 4.7
erhältlich unter Common Public License 1.0 von <http://junit.sourceforge.net/>

Die folgende Tabelle enthält eine Zusammenfassung der Ergebnisse, welche dadurch erzielt wurden, beide Refactorings auf alle Klassen, welche die jeweiligen Vorbedingungen erfüllen – Testfälle von vornherein ausgenommen – des jeweiligen Programms anzuwenden und anschließend die dazugehörigen Testfälle auszuführen:

Beispielprojekte	Commons-Collections 4.01		JPaul 2.5.1		JUnit 3.8.1		JUnit 4.7	
	IROP	ILRO	IROP	ILRO	IROP	ILRO	IROP	ILRO
Top-Level Klassen gesamt	202		71		61		219	
Lines of Code (LoC) gesamt	42 594		6 916		4 859		13 982	
<i>Anzahl Klassen, die eine Vorbedingung verletzen (mehrere pro Klasse möglich)</i>								
PC_POSTFIX_FIELD_ACCESS	5		3		1		2	
PC_SUBJECT_EXTENDS_THROWABLE	0		0		0		0	
PC_SUBJECT_IS_FINAL	50		9		0		2	
PC_SUBJECT_ONLY_PRIVATE_CONSTRUCTORS	35		8		1		7	
PC_OPEN_RECURSION_IN_SUPERCLASS_CONSTRUCTOR	13		0		1		13	
PC_SUBJECT_IS_ABSTRACT	24		22		3		17	
PC_SUBJECT_CONTAINS_INNER_TYPE	23		20		12		76	
PC_SUBJECT_STATIC_METHOD	130		22		14		43	
PC_SUBJECT_STATIC_FIELD	117		16		4		35	
PC_SUBJECT_PRIVATE_MEMBER_ACCESS	8		13		0		2	
PC_FIELD_INITIALIZER_CALLS_PRIVATE_METHOD	0		2		0		1	
PC_FIELD_INITIALIZER_CALLS_ON_THIS		0		2		0		1
PC_SUBJECT_PRIVATE_INNER_TYPE		2		15		0		9
Refaktorisierbare Klassen	23	129	16	47	33	58	80	189
Anwendbarkeit in %	11	64	23	66	54	95	37	86
Ø LoC pro refaktorisierbarer Klasse	49	83	31	35	30	64	35	55
Ø Hinzugefügte LoC pro Refactoring	545	287	445	235	484	275	461	245
Relativer Zuwachs LoC pro Refactoring	11,1	3,5	14,4	6,7	16,1	4,3	13,2	4,5

Tabelle 2: Auswertung der massenhaften Anwendung der Refactorings

Hinweise zur Tabelle und der ihr zugrundeliegenden Auswertung:

1. Der Sockel der LoC für die Rollenmanagementmethoden beträgt für den Ansatz ILRO konstant ca. 90 (bzw. 100 wenn mehrere Instanzen eines Rollentyps pro Subjekt erlaubt unterstützt werden sollen) und für den Ansatz IROP konstant ca. 150 (bzw. 160). Die Differenz zwischen beiden Ansätzen ergibt sich daraus, dass im Falle IROP die Deklaration

der Rollenmanagementmethoden im Interface `Subject` sowie deren Implementierungen in `SubjectRole` und `Delegate` zusätzlich anfallen. Die Angaben in der Tabelle sind jeweils nicht um diesen Sockel bereinigt.

2. LoC wurden sowohl für den Ausgangszustand jedes Programms ermittelt als auch nach Anwendung des jeweiligen Refactorings auf alle Klassen des Programms, bei denen dies nicht aufgrund verletzter Vorbedingungen unmöglich war. Die Kennzahl „Ø Hinzugefügte LoC pro Refactoring“ wurde berechnet als Differenz der LoC vor und nach Anwendung des Refactorings geteilt durch die Anzahl der Klassen auf die das Refactoring angewendet werden konnte. Sie zeigt also die durchschnittliche Zunahme von LoC des Programms pro erfolgreicher Anwendung des Refactorings.
3. Nach Anwendung der Refactorings auf diese Programme laufen deren Testfälle mit nur zwei Ausnahmen weiterhin fehlerfrei durch:
 - 3.1. JUnit 3.8.1 (IROP & ILRO): `junit.tests.runner.TestCaseClassLoaderTest.testJarClassLoading` scheitert beim Versuch eine Klasse aus einem externen Jar-File zu laden.
 - 3.2. JUnit 4.7: Bei den Tests der experimentellen Features in den Packages `org.junit.*` treten Fehler im Zusammenhang mit Annotations auf:
 - 3.2.1. ILRO: `org.junit.tests.experimental.rules.NameRulesTest` scheitert daran, dass ein mit einer Annotation versehenes Attribut in `NamesRulesTestCore` ausgelagert wird, die Annotation jedoch nur innerhalb von `NameRulesTest` gesucht wird und somit unberücksichtigt bleibt.
 - 3.2.2. IROP: Durch die Transformation von `Subject` in ein Interface bleiben sämtliche Annotationen an den transformierten `Subject`-Klassen und deren (nach Core verschobenen) Attributen unberücksichtigt, was zu 121 fehlschlagenden Tests (von 335 Tests insgesamt) führt.

Generell sind Fehlerquellen, die auf Ursachen außerhalb des verfügbaren Java-Quellcodes des Programms zurückgehen, mit den hier verwendeten Mitteln (Analyse und Transformation von Java-Quellcode) nicht zu beherrschen. Das gilt zum Beispiel für den Einsatz der Reflection-API (inkl. Verarbeitung von Annotations), externer Konfigurationsdateien sowie Komponenten, die in anderen Programmiersprachen implementiert sind oder nicht in Form von Quellcode vorliegen. Daher muss der Anwender sich stets darüber im Klaren sein, dass es an solchen Stellen zu Problemen kommen kann, ohne dass dies dem Refactoring als Fehler anzulasten ist.

4. Die verletzten Vorbedingungen wurden wie folgt ermittelt: Für jede Klasse der untersuchten Programme, die die Grundvoraussetzungen für die Anwendung des Refactorings erfüllt (benannte Top-Level Klasse, deren Quellcode vorliegt und modifizierbar ist; vgl. Abschnitt 5.2.1), wurden die in Tabelle 2 genannten Vorbedingungen geprüft. Klassen, die die

Vorbedingung `PC_SUBJECT_EXTENDS_THROWABLE` verletzen, wurden dabei nicht berücksichtigt. Die in der Tabelle angegebene Zahl stellt die Anzahl der Top-Level Klassen des jeweiligen Programms dar, welche die entsprechende Vorbedingung verletzen. Die Verletzung mehrerer unterschiedlicher Vorbedingungen durch eine einzelne Klasse wurde dabei jeweils voneinander unabhängig berücksichtigt, die mehrfache Verletzung derselben Vorbedingung durch eine einzelne Klasse wurde jedoch nur einfach gewertet.

- Eclipse-Versionen älter als 3.6 enthalten einen Bug ([Eclipse Bugzilla], Nr. 308754), der bei Anwendung des IROP-Refactorings auf Klassen, die mit einer Annotation versehen sind, welche ein `.class-Literal` als Argument erhält (z.B. `org.junit.tests.experimental.ExperimentalTests` aus JUnit 4.7), zu syntaktisch ungültigem Code führt.

6.2 Rollenverhalten

Die folgende Tabelle enthält eine Gegenüberstellung der in Abschnitt 2.2 beschriebenen Eigenschaften von Rollenkonzepten, welche sich mittels Role Object Pattern theoretisch umsetzen lassen, und der durch die in den Kapiteln 3 und 4 beschriebenen Ansätze tatsächlich realisierten. Die Testfälle, welche zur Überprüfung der Eigenschaften herangezogen wurden, befinden sich auf der beiliegenden CD (s. Anhang A).

Eigenschaft	Umsetzbar	IROP		ILRO	
		single	multi	single	multi
1	ja	ja	ja	ja	ja
2	ja	ja	ja	ja	ja
3	ja	ja	ja	ja	ja
4	ja	nein	ja	nein	ja
5	ja	ja	ja	ja	ja
6	eingeschränkt	nein	nein	nein	nein
7	nein	nein	nein	nein	nein
8	ja	nein	nein	ja	ja
9	möglich	ja	ja	ja	ja
10	nein	nein	nein	nein	nein
11	ja	ja	ja	ja	ja
12	eingeschränkt	ja	ja	ja	ja
13	ja	ja	ja	ja	ja
14	möglich	ja	ja	ja	ja
15	ja	ja	ja	ja	ja

Tabelle 3: Übersicht Umsetzungsgrad Rollen-Eigenschaften

Hinweise zur Auswertung:

1. Eigenschaft 4 (Mehrere Rolleninstanzen desselben Typs pro Subjekt) kann durch den Anwender aktiviert (multi) oder deaktiviert (single) werden. Diese Option steht zur Verfügung, da die Verwaltung und insbesondere die Abfrage (getRole) der Rollen bei Aktivierung dieser Eigenschaft zusätzlichen Aufwand erfordern, der durch die Deaktivierung vermieden wird, sofern diese Eigenschaft im konkreten Anwendungsfall nicht benötigt wird.
2. Eigenschaft 6 (Einschränkungen bezüglich der Reihenfolge in der Rollen angenommen bzw. abgelegt werden können) kann durch die Refactorings nicht umgesetzt werden, da keine Informationen darüber vorliegen, welche Abhängigkeiten zwischen verschiedenen Rollen im konkreten Anwendungsfall bestehen. Wird diese Eigenschaft benötigt, so ist deren Umsetzung nachträglich durch den Anwender erforderlich.
3. Eigenschaft 8 (Rollen können Rollen spielen) ist für die Variante IROP in den der Auswertung zu Grunde liegenden Fällen nicht umgesetzt, da durch das Self Encapsulate Fields während der Anwendung dieses Refactorings auf den Beispielfall offen rekursive Aufrufe von Zugriffsmethoden in den Konstruktoren der Superklassen erzeugt werden (PC_OPEN_RECURSION_IN_SUPERCLASS_CONSTRUCTOR, s. Abschnitt 3.6), welche eine manuelle Behandlung vor erneuter Anwendung des gleichen Refactorings auf eine andere Klasse aus derselben Vererbungshierarchie erfordern.
Dies ist nicht grundsätzlich der Fall, kommt aber insbesondere dann vor, wenn Konstruktoren von `Subject` zur Initialisierung von Attributen verwendet werden. Denn dabei wird die entsprechende Zuweisung durch einen Zugriffsmethoden-Aufruf ersetzt, welcher beim anschließenden Versuch der Transformation einer Rollenklasse in ein Sub-Subjekt als unzulässige offene Rekursion in einem Konstruktor einer Superklasse beanstandet wird.

7 Diskussion

In diesem Kapitel werden die in Kapitel 6 präsentierten Ergebnisse interpretiert, bewertet und mit den Ergebnissen verwandter Arbeiten verglichen.

7.1 Interpretation der Ergebnisse

Erwartungsgemäß geht aus Tabelle 2 hervor, dass die Anwendbarkeit des IROP Refactorings aufgrund ihrer zahlreichen und restriktiven Vorbedingungen stark eingeschränkt ist. Im Durchschnitt kommt nur gut jede viertetop-level Klasse (27%) für die Anwendung des Refactorings in Frage. Das ILRO Refactoring unterliegt in dieser Hinsicht weniger Einschränkungen und kommt auf eine Anwendbarkeitsrate zwischen 64 und 95 Prozent.

Ebenfalls wenig überraschend ist auch festzustellen, dass die durchschnittliche Anzahl der durch das Refactoring hinzugefügten LoC pro erfolgreicher Anwendung im Falle IROP in allen betrachteten Fällen im Vergleich zur Variante ILRO mehr als doppelt so hoch ist. Hinzu kommt, dass das ILRO Refactoring pro Anwendung exakt zwei zusätzliche Klassen erzeugt – `SubjectCore` und `SubjectRole`, die beide einigermaßen überschaubar bleiben: `SubjectCore` verwaltet lediglich Rollen und gemeinsamen Zustand des Subjekts, während `SubjectRole` bis auf das Durchreichen der Argumente von Konstruktoraufrufen an `Subject` praktisch leer ist. Das IROP Refactoring hingegen erzeugt bei jeder erfolgreichen Anwendung mindestens fünf Klassen (davon eine innere) und ein Interface zuzüglich eines weiteren Interfaces für jede Superklasse von `Subject`. Das heißt, dass auch bei der Verwendung der neu erzeugten Typen (Klassen und Interfaces) pro erfolgreicher Anwendung als Metrik die entsprechende Kennzahl der IROP Variante ein vielfaches jener der ILRO Variante beträgt.

In Bezug auf die mit beiden Ansätzen jeweils umsetzbaren Eigenschaften von Rollenkonzepten (s. Abschnitt 2.2) zeigt Abschnitt 6.2 trotz der deutlich schlankeren Struktur dieses Ansatzes keine Defizite des ILRO Refactorings gegenüber dem IROP Ansatz.

7.2 Bewertung der Ergebnisse

Die Bewertung der Ergebnisse orientiert sich an den in Kapitel 6 vorgestellten Qualitätskriterien. Wie dort schon angemerkt wurde, kann bei dem Korrektheitsnachweis mittels massenhafter Anwendung und anschließender Überprüfung mit Hilfe vorhandener Testfälle von einem tatsächlichen Beweis der Korrektheit des jeweiligen Refactorings keine Rede sein. Es ist daher nicht auszuschließen, dass beim zukünftigen Einsatz beider Refactorings trotz der Vielfalt der verwendeten Testfälle Fehler auftreten können. Während der Testphase im Rahmen dieser Arbeit, sind beim IROP Ansatz aufgrund der höheren Komplexität seiner Implementierung mehr unerwartete Fehler aufgetreten als bei der Variante ILRO, was aber nicht bedeutet, dass letztere sich völlig frei von solchen gezeigt hätte.

In Hinblick auf die Betrachtungen zur Komplexität des erzeugten Codes ist zunächst festzustellen, dass beide Ansätze diese im Vergleich zur Ausgangssituation eher erhöhen als senken. Zumindest wenn ausschließlich von der stark vereinfachenden Annahme ausgegangen wird, dass weniger umfangreiche Programme den in Kapitel 6 genannten nicht-funktionalen Kriterien eher gerecht werden als Programme, die dasselbe Verhalten mit einer höheren Anzahl von Codezeilen umsetzen.

Die Zielsetzung beider Refactorings besteht jedoch vor allem darin, Programmen eine semantisch korrekte Repräsentation der dort abzubildenden Wirklichkeit zu erlauben, auch wenn deren Strukturen zunächst gar nicht darauf ausgerichtet waren, die tatsächlich vorhandene Komplexität dieser Wirklichkeit in Bezug auf die Notwendigkeit der Verwendung eines Rollenkonzepts für bestimmte Elemente der Fachlogik zu erfassen. Vor diesem Hintergrund ist eine überschaubare Erhöhung der Komplexität des zu modifizierenden Programms verschmerzbar, da zum Einen diese Komplexität auf eine entsprechend komplexe abzubildende Wirklichkeit zurückzuführen ist und zum Anderen eine deutlich größere Erhöhung der Programmkomplexität zu befürchten ist, falls der notwendige Einsatz eines Rollenkonzepts durch wie auch immer geartete Workarounds umgangen wird. Positiv fällt in dieser Hinsicht der aus Tabelle 3 hervorgehende hohe Abdeckungsgrad der in Abschnitt 2.2 vorgestellten Eigenschaften von Rollenkonzepten ins Gewicht.

Schwerer wiegt die eingeschränkte Anwendbarkeit beider Lösungen. Auch wenn der ILRO Ansatz hier Vorteile gegenüber der IROP Variante bietet, stellen die in Kapitel 6 für die Anwendbarkeit ermittelten Werte unter Umständen ein potentiell Hemmnis für Anwender dar, sich überhaupt mit dem Refactoring auseinanderzusetzen.

Zusammenfassend lässt sich feststellen, dass der ILRO Ansatz in allen Belangen der IROP Variante überlegen ist. Allerdings gilt für beide Fälle die in Abschnitt 2.4 erläuterte Erschwernis, dass es sich bei diesen Refactorings jeweils nur um eine vorbereitende Transformation der Struktur des zu modifizierenden Programms handelt, auf die im Anschluss noch eine entsprechende Anpassung von dessen Semantik durch den Anwender des Refactorings erfolgen muss, um eine vollständige Umsetzung des Role Object Pattern zu erzielen.

7.3 Verwandte Arbeiten

Der in [Steimann1999] mit „roles as adjunct instances“ bezeichnete Ansatz, Rollen in objektorientierten Sprachen als separate Instanzen eigener Typen abzubilden, ist weit verbreitet. Er kommt in unterschiedlichen Formen in [Albano1993], [Gottlob1996], [Kristensen1995], [Pernici1990] und [Sciore1989] zum Einsatz, ist aber stets mit den Problemen der Delegation (s. Abschnitt 3.5) und Objektschizophrenie (s. Abschnitt 3.2) verbunden.

Auch die Variante, Rollen als Subtypen zu modellieren, was der in Abschnitt 2.4 dargestellten Ausgangssituation der hier vorgestellten Refactorings entspricht, wird z.B. in [Fowler1997] sowie in [Fowler1999] genannt (in letzterem als Beispiel für das Replace Delegation with Inheritance Refactoring). Diese Vorgehensweise weist jedoch genau die Nachteile auf, welche die Motivation für die in den vorangegangenen Kapiteln beschriebenen Refactorings bilden. Darüber hinaus

verhindert dieser Ansatz, dass unterschiedliche Typen von Subjekten dieselben Rollen annehmen können, da diese sonst Subtypen beider Subjekttypen sein müssten. Da die Subtypenbeziehung zwischen Rolle und Subjekt durch die Einführung des Role Object Pattern bzw. dessen Alternative Lightweight Role Objects nicht aufgelöst wird, besteht dieses konzeptionelle Problem allerdings auch für die beiden in dieser Arbeit vorgestellten Lösungsansätze.

Eine dritte Möglichkeit, welche das zuletzt genannte Problem nicht aufweist, ist die Umsetzung von Rollen in Form partieller Spezifikationen von Objekten, also Supertypen (z.B. in Form von Java Interfaces), wie in [Bachman1977] und [Steimann2001] beschrieben. Da ein Subjekttyp in diesem Fall Subtyp jeder Rolle sein muss, die das Subjekt annehmen können soll, führt dieser Ansatz jedoch einerseits schnell zu übermäßig aufgeblähten Subjekttypen und andererseits erfordert dadurch das nachträgliche Hinzufügen weiterer Rollen stets eine Anpassung des Subjekttyps. In der Praxis kommt diese Lösung daher meist – wenn überhaupt – nur nachträglich zum Einsatz, z.B. durch Anwendung der Refactorings Infer Type (s. [Kegel2007]) oder Use Supertype Where Possible (s. [Tip2003]).

Da alle drei der in diesem Abschnitt genannten Ansätze, Rollen in objektorientierten Sprachen mit den dort zur Verfügung stehenden Sprachmitteln umzusetzen, je nach Einsatzzweck unterschiedlich stark ins Gewicht fallende Makel aufweisen, gibt es Bestrebungen, die Umsetzung von Rollenkonzepten durch Erweiterung der Sprachen zu vereinfachen. Der in [Hermann2007] beschriebene Ansatz „Objekt Teams/Java“ stellt mit den Mitteln der Aspekt-orientierten Programmierung (vgl. [Kiczales1997]) quasi eine native Implementierung des Role Object Pattern nach [Bäumer1997] dar. Das Problem der Delegation wird dabei durch die Kombination von Forwarding (von der Rolle an das Subjekt) und Abfangen (Interception) von Methodenaufrufen (Rückdelegation vom Subjekt an die Rolle) gelöst. Das Problem der logischen Identität, welche sich über mehrere Objekte verteilt, wird durch eine automatische, kontextabhängige Übersetzung des Subjekts angegangen. Dabei wird das Subjekt innerhalb eines bestimmten Kontextes über die Rolle, welche es innerhalb dieses Kontextes spielt, betrachtet. Die automatische Umwandlung des Subjekts in eine Instanz der passende Rolle nennt [Hermann2007] „lifting“. Beim Verlassen des Kontextes wird die Rolleninstanz wieder auf die ihr zu Grunde liegende Subjektkerninstanz zurückgeführt („lowering“).

8 Schlussbetrachtungen

8.1 Zusammenfassung

Verbreitete klassenbasierte objektorientierte Programmiersprachen (z.B. C# oder Java) kennen in der Regel das Konzept von Rollen nicht in der Form, dass es sich dabei um kontextabhängige Sichten (mit jeweils eigenem Zustand und Verhalten) auf ein Individuum (mit einem allen Rollen gemeinsamen Kernzustand und -verhalten) handelt, welches solche Rollen dynamisch annehmen und ablegen kann. Das Role Object Pattern nach [Bäumer1997] beschreibt eine Möglichkeit, ein solches Rollenkonzept mit den Mitteln dieser Sprachen umzusetzen.

Das Ziel dieser Arbeit ist, Softwareentwicklern ein Refactoring-Werkzeug an die Hand zu geben, welches es ermöglicht das o.g. ROP in Programmen einzusetzen, die stattdessen ursprünglich den verbreiteteren Role Subtype Ansatz (s. [Fowler1997]) zur Abbildung von Rollen im weiteren Sinne verwendeten. Da ein solches Refactoring das Verhalten des umzustrukturierenden Programms nicht ändern darf, kann es nur einen Teilschritt auf dem Weg zur vollständigen Umsetzung des ROP darstellen, dem entsprechende Anpassungen der Semantik des Programms durch den Entwickler folgen müssen.

Die notwendigen Anpassungen des o.g. Pattern an die Eigenheiten der Sprache Java (eine Anpassung des Prinzips an ähnliche Sprachen, wie z.B. C#, ist denkbar) sowie die speziellen Anforderungen, die der Einsatz als Zielzustand eines Refactorings mit sich bringt, führen dazu, dass das resultierende Refactoring restriktive Vorbedingungen erfordert und recht komplexen, wenig übersichtlichen Code erzeugt. Glücklicherweise ist es im Rahmen dieser Arbeit gelungen, eine Vereinfachung des ROP zu finden, welche praktisch dieselben Möglichkeiten in Bezug auf die Verwendung des Rollenkonzepts bietet, aber als Refactoring leichter umzusetzen ist, weniger komplexen Code produziert und weniger restriktive Vorbedingungen erfordert. Dieser Ansatz wird „Lightweight Role Objects“ (LRO) und das daraus abgeleitete Refactoring „Introduce Lightweight Role Objects“ (ILRO) genannt.

Die Gegenüberstellung beider in dieser Arbeit vorgestellten Ansätze ergibt, dass die ILRO Variante zwar in allen untersuchten Belangen besser abschneidet, als zuverlässiges Werkzeug für den Entwickleralltag jedoch auch nicht vollständig zu überzeugen vermag.

8.2 Ausblick

Diese Arbeit hat gezeigt, dass sich das Role Object Pattern erfolgreich auf die Sprache Java übertragen lässt und selbst die Einführung in bereits bestehenden Code mit Hilfe eines automatisierten Refactoring-Werkzeugs möglich ist. Allerdings zeigt die Auswertung auch, dass dabei weiterhin Probleme bestehen, die letztlich damit zusammenhängen, dass es in Java keine nativen Sprachmittel zur Unterstützung umfassender Rollenkonzepte gibt.

Die Entwicklung eines solchen nativen Sprachmittels und dessen Einführung im Rahmen einer verbreiteten klassenbasierten objektorientierten Programmiersprache wäre an dieser Stelle wünschenswert und hätte bei guter Umsetzung voraussichtlich beste Chancen, Rollenkonzepte, wie sie in Kapitel 2 beschrieben sind, als Ausdrucksmittel in der objektorientierten Softwareentwicklung zu verankern.

Die Erweiterung der Sprache Java mit den Mitteln der aspektorientierten Programmierung, wie sie in [Hermann2007] beschrieben wird, ist sicherlich ein begrüßenswerter Schritt in diese Richtung. Ein natives Sprachkonstrukt hätte jedoch eine bessere Aussicht, die notwendige Verbreitung zu erlangen, um sich im Entwickleralltag zu etablieren.

8.3 Fazit

Abschließend bleibt festzustellen, dass beide der in dieser Arbeit vorgestellten Refactoring-Werkzeuge ihren Zweck erfüllen, wobei es mit dem Introduce Lightweight Role Objects Refactoring sogar gelungen ist, eine deutlich verbesserte Lösung gegenüber dem ursprünglich verfolgten Ansatz zu finden. Vollständig befriedigend sind jedoch beide Lösungen nicht, da es sich in beiden Fällen letztendlich um den unvollkommenen Versuch handelt, ein in der Sprache ursprünglich nicht vorgesehenes Konzept mit den vorhandenen Sprachmitteln zu simulieren, was naturgemäß mit Einschränkungen verbunden ist.

9 Abbildungen und Tabellen

Abbildungsverzeichnis

Abbildung 1: Struktur des Role Object Pattern nach [Bäumer1997].....	8
Abbildung 2: Anwendungsbeispiel für das Role Object Pattern (Quelle: [Bäumer1997]).....	10
Abbildung 3: Beispiel: Ausgangssituation.....	11
Abbildung 4: Beispiel: Zielzustand des Refactorings.....	12
Abbildung 5: Grundsätzliches Vorgehen: Ausgangssituation.....	15
Abbildung 6: Grundsätzliches Vorgehen: Schritt 1.....	15
Abbildung 7: Grundsätzliches Vorgehen: Schritt 2.....	15
Abbildung 8: Grundsätzliches Vorgehen: Schritt 3.....	15
Abbildung 9: Grundsätzliches Vorgehen: Schritt 4.....	15
Abbildung 10: Grundsätzliches Vorgehen: Schritt 5.....	16
Abbildung 11: Ablauf der Prüfung auf logische Identität.....	19
Abbildung 12: Beispiel für Problem des dynamischen Bindens.....	22
Abbildung 13: Beispiel für Problem des dynamischen Bindens - Sequenz.....	24
Abbildung 14: Unterschied zwischen Forwarding und Delegation (Quelle: [Kegel2008]).....	24
Abbildung 15: Beispiel - Zielzustand mit Reverse Forwarding.....	25
Abbildung 16: Beispiel Reverse Forwarding - Rolle überschreibt Methode nicht.....	26
Abbildung 17: Beispiel Reverse Forwarding - Rolle überschreibt Methode.....	27
Abbildung 18: Beispiel mit Reverse Forwarding und SubjectState.....	28
Abbildung 19: Beispiel mit NullRole - sichtbare und verborgene Elemente.....	29
Abbildung 20: Struktur bei mehreren Superklassen.....	33
Abbildung 21: IROP Ausgangssituation - Struktur schematisch.....	34
Abbildung 22: IROP Zielzustand - Struktur schematisch.....	35
Abbildung 23: Darstellung der Unterschiede zwischen den Lösungsansätzen.....	39
Abbildung 24: ILRO Zielzustand - Struktur schematisch.....	41
Abbildung 25: Architektur von Eclipse (Quelle: [D'Anjou2004]).....	43
Abbildung 26: Ablauf LTK-basierter Refactorings (Quelle [Petito2007]).....	44
Abbildung 27: Struktur LTK-basierter Refactorings (Quelle: [Petito2007]).....	45

Tabellenverzeichnis

Tabelle 1: Eigenschaften von Rollenkonzepten für objektorientierte Sprachen.....	7
Tabelle 2: Auswertung der massenhaften Anwendung der Refactorings.....	59
Tabelle 3: Übersicht Umsetzungsgrad Rollen-Eigenschaften.....	61

10 Literaturverzeichnis

- [**Albano1993**] Albano, A., Bergamini, R., Ghelli, G., and Orsini, R. 1993. An Object Data Model with Roles. In *Proceedings of the 19th international Conference on Very Large Data Bases* (August 24 - 27, 1993). R. Agrawal, S. Baker, and D. A. Bell, Eds. Very Large Data Bases. Morgan Kaufmann Publishers, San Francisco, CA, 39-51.
- [**Ambler2002**] Ambler, S.W. 2002. Agile Requirements Change Management. (Online) available at <http://www.agilemodeling.com/essays/changeManagement.htm#WhyRequirementsChange>, 05.09.2010.
- [**Bachman1977**] Bachman, C. W. and Daya, M. 1977. The role concept in data models. In *Proceedings of the Third international Conference on Very Large Data Bases - Volume 3* (Tokyo, Japan, October 06 - 08, 1977). B. C. Housel and A. G. Merten, Eds. Very Large Data Bases. VLDB Endowment, 464-476.
- [**Balzert1999**] Balzert, H. 1999. Lehrbuch Grundlagen der Informatik. Spektrum Akademischer Verlag.
- [**Bäumer1997**] Bäumer, D., Riehle, D., Siberski, W. and Wulf, M. 1997. The Role Object Pattern. In *Proceedings of the 4th Pattern Languages of Programming Conference*.
- [**Codd1970**] Codd, E. F. 1983. A relational model of data for large shared data banks. *Commun. ACM* 13, 6 (1970), 377-387.
- [**D'Anjou2004**] D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J. and McCarthy, P. 2004. The Java Developer's Guide to Eclipse, 2nd Edition. Addison-Wesley Professional.
- [**Eclipse-Bugzilla**] Eclipse Bugs. (Online) available at <https://bugs.eclipse.org/bugs/>, 05.09.2010.
- [**Falkenberg1976**] Falkenberg, E. D. 1976. Concepts for Modelling Information. *IFIP Working Conference on Modelling in Data Base Management Systems*. 95-109.
- [**FernUniversität01853**] Steimann, F. Keller, D., Forster, F. and Grebe, I. 2007. Moderne Programmier-techniken und -methoden. Fakultät für Mathematik und Informatik, FernUniversität in Hagen.
- [**Fowler1997**] Fowler, M. 1997. Dealing with Roles. (Online) available at <http://www.martinfowler.com/apsupp/roles.pdf>, 05.09.2010.
- [**Fowler1999**] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. 1999. Refactoring: Improving the Design of existing Code. Addison-Wesley Professional.
- [**Fowler2000**] Fowler, M. 2000. The New Methodology. (Online) available at <http://martinfowler.com/articles/newMethodology.html#TheUnpredictabilityOfRequirements>, 05.09.2010.

- [**Frenzel2006**] Frenzel, L. 2006. The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs. (Online) available at <http://www.eclipse.org/articles/Article-LTK/ltk.html>, 05.09.2010
- [**Gamma1995**] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Systems. Addison-Wesley International.
- [**Gottlob1996**] Gottlob, G., Schrefl, M. and Röck, B. 1996. Extending Object-Oriented Systems with Roles, *ACM Trans. Inf. Syst.* 14, 3 (1996). 286-296.
- [**Herrmann2007**] Herrmann, S. 2007. A precise model for contextual roles: The programming language ObjectTeams/Java. *Appl. Ontol.* 2, 2 (Apr. 2007), 181-207.
- [**IBM2005**] Aeschlimann, M., Bäumer, D. and Lanneluc, J. 2005. Java Tool Smithing – Extending the Java Development Tools. (Online) available at http://eclipsecon.org/2005/presentations/EclipseCON2005_Tutorial29.pdf, 05.09.2010.
- [**Java1.5.0**] J2SE 1.5.0 API Documentation. (Online) available at <http://download.oracle.com/javase/1.5.0/docs/api/index.html>, 05.09.2010.
- [**Kegel2007**] Kegel, H. 2005. Constraint-basierte Typinferenz für Java 5. Diplomarbeit. Fakultät für Mathematik und Informatik. FernUniversität in Hagen.
- [**Kegel2008**] Kegel, H., Steimann, F. 2008. Systematically refactoring inheritance to delegation in Java. In *Proceedings of the 30th ICSE* (May 2008). 431-440.
- [**Kerievsky2004**] Kerievsky, J. 2004. Refactoring to Patterns. Addison-Wesley, München (2005).
- [**Kiczales1997**] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. 1997. Aspect-oriented Programming. In *Proceedings of ECOOP 1997*. Finland. Springer-Verlag.
- [**Kristensen1995**] Kristensen, B. B. 1995. Object-Oriented Modelling with Roles. In *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*. Dublin. Springer, 1996. 57-71.
- [**Opdyke1993**] Opdyke, W. F. 1993. Refactoring Object-Oriented Frameworks. PhD Thesis. University of Illinois at Urbana-Champaign.
- [**Pernici1990**] Pernici, B. 1990. Objects with Roles. In *Proceedings of the Conference on Office Information*. SIGOIS Bulletin 11, 2/3. ACM Press, New York, 1990. 205–215.
- [**Petito2007**] Petit, M. 2007. Eclipse Refactoring. (Online) available at <http://people.clarkson.edu/~dhou/courses/EE564-s07/Eclipse-Refactoring.pdf>, 05.09.2010.
- [**Pierce2002**] Pierce, B. C. 2002. Types and Programming Languages. Cambridge, Massachusetts. The MIT Press, 2002.
- [**Roberts1999**] Roberts, D. B. 1999. Practical Analysis for Refactoring. PhD Thesis. University of Illinois at Urbana-Champaign.

- [Sciore1989]** Sciore, E. 1989. Object Specialization. *ACM Trans. Inf. Syst.* 7, 2 (1989). 103-122.
- [Sekharaiah2002]** Sekharaiah, K. C. and Ram, D. J. 2002. Object Schizophrenia Problem in Modelling Is-Role-Of Inheritance. In *Proceedings of the Inheritance Workshop at the 16th European Conference on Object Oriented Programming, ECOOP 2002*, June 2002, University of Malaga, Spain.
- [Steimann1999]** Steimann, F. 1999. On the representation of roles in object-oriented conceptual modelling. In *Data Knowl. Eng.* 35, 1 (Oct. 2000). 83-106.
- [Steimann2000]** Steimann, F. 2000. Formale Modellierung mit Rollen. Habilitationsschrift. Universität Hannover.
- [Steimann2001]** Steimann, F. 2001. Role = Interface: A Merger of Concepts. In *Journal of Object-Oriented Programming* 14, 4 (2001). 23–32.
- [Steimann2010]** Steimann, F. and Stolz, F. U. 2010. Refactoring to Role Objects. Unpublished. Fakultät für Mathematik und Informatik. FernUniversität in Hagen.
- [Tip2003]** Tip, F., Kiezun A. and Bäumer, D. 2003. Refactoring for generalization using type constraints. *SIGPLAN Not.* 38, 11 (Nov. 2003), 13-26.
- [Ungar1987]** Ungar, D. and Smith, R. B. 1987. Self: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Orlando, FL, (October 1987). 227-241.
- [Warren1998]** Warren, N. and Bishop, P. 1998. *Java in Practice: Design Styles & Idioms for Effective Java*. Addison-Wesley.
- [Widmer2006]** Widmer, T. 2006. Unleashing the Power of Refactoring. (Online) available at <http://www.eclipse.org/articles/article.php?file=Article-Unleashing-the-Power-of-Refactoring/index.html>, 05.09.2010.

11 Anhang A: Beiliegende CD

Der gedruckten Form dieser Arbeit liegt eine CD mit folgendem Inhalt bei:

1. **Introduce Role Object Refactoring.pdf**
Dieser Text in Form eines PDF-Dokuments.
2. **Javadoc**
Dokumentation des Quellcodes des Eclipse Plug-ins, welches sowohl das IROP als auch das ILRO Refactoring implementiert, im HTML-Format.
3. **Plug-ins**
Enthält die Eclipse Plug-ins `org.intoj.irop` (IROP und ILRO Refactoring) und `org.intoj.irop.multirefac` (Werkzeug zur massenhaften Anwendung der o.g. Refactorings) in Form von deploybaren Jar-Dateien. Die Mindestvoraussetzung für den Einsatz dieser Plug-ins ist Eclipse 3.4.2, es empfiehlt sich allerdings die Verwendung von Eclipse 3.6.
4. **Quellcode**
Enthält den Quellcode der beiden o.g. Plug-ins in Form von Eclipse Projekten, welche einzeln als Zip-Archive gepackt sind und sich direkt in Eclipse importieren lassen. Zusätzlich ist der Quellcode der Testfälle für das Rollenmanagement (s. Abschnitt 6.2) ebenfalls in Form eines Zip-komprimierten Eclipse Projekts enthalten.

12 Erklärung

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

Dortmund, den 30. Januar 2011

Fabian Urs Stolz