Modularisierung und Wiederverwendung durch generische Klassen in Object Teams

vorgelegt von

Dipl.-Inform.
Andreas Mertgen
geboren in Neuwied am Rhein

von der Fakultät IV - Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.-Ing. -

genehmigte Dissertation

Promotionsausschuss:

Vorsitzender: Prof. Dr. P. Pepper
Gutachter: Prof. Dr. S. Jähnichen
Gutachter: Prof. Dr. U. Aßmann

Tag der wissenschaftlichen Aussprache: 13. Dezember 2013

Berlin 2013
D 83

Zusammenfassung

Modularisierung und Wiederverwendung sind von wesentlicher Bedeutung für die Produktivität in der Softwareentwicklung. Die rollenorientierte Programmierung ergänzt die Konzepte der objektorientierten Programmierung mit dem Ziel, die Modularisierung zu fördern. Die Programmiersprache *Object Teams*, eine rollenorientierte Erweiterung der Sprache Java, bietet eine gezielte Unterstützung der Modularisierung von Kollaborationen zwischen Objekten. Eine Klasse bzw. ein Objekt kann unterschiedliche Rollen in separaten Kontexten annehmen. Die Abbildung von Rollen und Kontext wird in Object Teams durch spezielle Klassen unterstützt. Eine Rollenklasse definiert dabei eine der Vererbung ähnelnde Beziehung zu einer sogenannten *Basis*klasse, welche die Rolle spielt. Der Rollenmechanismus erlaubt einerseits eine feinere und flexiblere Modularisierung kontextspezifischer Daten und Funktionen, führt aber andererseits auch zu einer starken Kopplung der beteiligten Klassen und wirkt sich damit negativ auf die Wiederverwendbarkeit aus. Ursache dafür ist die fehlende Möglichkeit zur Abstraktion der Bindung zwischen Rolle und Basis.

Das Ziel dieser Arbeit ist es, die starke Kopplung zwischen den beteiligten Klassen zu reduzieren und damit deren Flexibilität und deren Wiederverwendbarkeit zu fördern. Als Lösungsansatz werden Mechanismen zur Abstraktion von Klassen eingeführt, mit denen vor allem strukturelle Abhängigkeiten zwischen Rollen- und Basisklassen in generischer Form ausgedrückt werden können. Solche *generischen* Klassen repräsentieren Templates, die je nach Anwendung, in der sie zum Einsatz kommen, unterschiedlich ausgeprägt werden. Im Quellcode der Klasse können dazu an ausgewählten Punkten Metavariablen anstelle von statischen Definitionen verwendet werden. Die Metavariablen dienen dabei als Platzhalter für spezifische Programmelemente wie Klassen und Methoden und stellen eine Form der Metaprogrammierung dar. Zur konkreten Definition der Metavariablen wird eine deklarative Beschreibungssprache nach Art der logischen Programmierung in Prolog eingeführt. Diese erlaubt es, spezifische Programmelemente anhand ihrer strukturellen Eigenschaften auszuwählen – unabhängig von ihren Bezeichnern. Die Funktionalität einer Rolle kann so in generischer Form definiert werden. Die spezifischen Basisklassen und deren Methoden, die mit der Rolle für eine konkrete Anwendung verknüpft werden sollen, werden automatisch anhand der deklarativen Beschreibung identifiziert. Die konkrete Ausprägung einer generischen Klasse wird für jede Anwendung neu bestimmt, indem Metavariablen in Abhängigkeit vom jeweiligen Kontext durch spezifische Programmelemente ersetzt werden. Im Rahmen der Verarbeitung des Programms werden die generischen Anteile des Codes zu ausführbarem Code transformiert.

In dieser Arbeit werden zunächst die Probleme von Object Teams hinsichtlich Modularisierung und Wiederverwendbarkeit analysiert. Darauf aufbauend wird dann die neu entwickelte Spracherweiterung *Generic Object Teams* mit Syntax und Semantik vorgestellt, die Object Teams um die beschriebenen Möglichkeiten zur Metaprogrammierung erweitert. Bestandteil dieser Erweiterung ist die Abbildung und Auswertung des Programms in Prolog. Der Effekt der neuen Sprachmittel auf die Modularisierung und die Wiederverwendbarkeit wird anhand einer Studie zu Entwurfsmuster-Anwendungen evaluiert.

Die Arbeit zeigt, dass sich die Metaprogrammierung mit Prolog erfolgreich in Object Teams integrieren lässt. Der entwickelte Ansatz wirkt sich einerseits positiv auf die Modularisierung und die Wiederverwendbarkeit aus, führt andererseits aber auch zu einer gesteigerten Komplexität der Programme. Mit der Komplexität steigen gleichermaßen die Anforderungen an den Programmierer, was die Anwendbarkeit des Ansatzes einschränkt. Insgesamt sehen wir in dem Ansatz einen nützlichen Beitrag für die rollenorientierte Programmierung in Object Teams, der auch als Anregung und Grundlage für weitere Entwicklungen in diesem Bereich dienen kann.

Abstract

Modularization and reuse are essential for the productivity of software development. Role-oriented programming complements the concepts of object-oriented programming with the goal to improve modularization. The programming language *Object Teams*, a role-oriented extension of the Java language, provides specific support for modularization of collaborations between objects. A class or object can play different roles in separate contexts. The realization of roles and context is supported in Object Teams through special classes. A role class defines a relationship, which is similar to inheritance, to a so called *base* class that plays the role. Roleplaying on the one hand allows a finer and more flexible modularization of context-specific data and functions, but on the other hand also leads to a strong coupling of the classes involved and thus has a negative impact on the reusability. This is due to the inability to abstract the binding between role and base.

The goal of this work is to reduce the strong coupling between the classes involved and to improve both their flexibility and their reusability. Our approach introduces mechanisms for the abstraction of classes, which primarily allow expressing structural dependencies between role and base classes in a generic form. Such *generic* classes represent templates that are transformed depending on the specific application in which they are used. Metavariables can be used instead of static definitions in selected positions of the source code of the class. Metavariables represent placeholders for specific program elements such as classes and methods and provide a form of metaprogramming. A declarative description language, which is similar to logic programming in Prolog, is introduced for the specific definition of the metavariables. This allows the programmer to select specific program elements based on their structural properties – regardless of their identifiers. This way the function of a role may be defined in a generic form. The specific base classes and their methods that are to be bound to the role for a specific application are automatically identified based on the declarative description. The concrete form of a generic class is created for each new application by replacing the metavariables with specific program elements, depending on the respective context. As part of the compilation process of the program, the generic elements will be transformed into executable code.

In this work we analyze the problems of Object Teams regarding modularization and reusability. Based on the analysis, the newly developed language extension *Generic Object Teams*, which extends Object Teams to include the options described for metaprogramming, is presented with syntax and semantics. The extension includes the mapping and evaluation of the program in Prolog. The effect on the modularization and reusability of the new language features will be evaluated with a study of design pattern applications.

This work shows that metaprogramming in Prolog can be successfully integrated into Object Teams. The developed approach improves the modularization and reusability, but also leads to an increased complexity of programs. The complexity raises the demands on the programmer and therefore limits the applicability of the approach. In summary, we consider the approach a useful contribution for role-oriented programming in Object Teams, which can serve as an inspiration and a basis for further developments in this area.

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Fachgebiet Softwaretechnik unter Leitung von Prof. Dr. Stefan Jähnichen. Bei ihm möchte ich mich sehr herzlich bedanken, für gute Ratschläge, fortwährende Motivation und dass er mir überhaupt die Möglichkeit geboten hat, in seinem großartigen Team zu arbeiten und zu promovieren. Ich habe in meiner Zeit dort viel von ihm lernen können. Ebenfalls bedanken möchte ich mich bei meinem Zweitgutachter Prof. Dr. Uwe Aßmann für seine freundliche Unterstützung und seine aufschlussreichen Anmerkungen.

Ferner möchte ich mich besonders bei Dr. Stephan Herrmann und Marco Mosconi bedanken, die mir als Mentoren für Object Teams zur Seite gestanden haben und damit überhaupt erst den Grundstein für meine Arbeit gelegt haben. Darüber hinaus gilt mein Dank all den Kollegen, die mich über die Jahre begleitet und unterstützt haben, im Besonderen Prof. Dr. Steffen Helke, Alexandra Mehlhase, Rodger Burmeister, Kristian Duske, Marcus Mews, Alexander Rein-Jury, Doris Fähndrich und Gabi Ambach.

Auch meine Diplomanden Thomas Brendtner, Daniél Gómez Esperón und Jan Marc Hoffmann möchte ich an dieser Stelle erwähnen. Sie haben mit ihren Arbeiten viel zur erfolgreichen Umsetzung meiner Arbeit beigetragen. Ebenfalls gilt mein Dank Dr. Günter Kniesel und Dr. Tobias Rho für ihre hilfreiche Unterstützung bei der Entwicklung meines Ansatzes.

Zuletzt möchte ich meiner Familie und vor allem meiner Liebsten Julika für ihre ausdauernde Unterstützung danken. Ihr Rückhalt hat mir sehr geholfen, diese Arbeit erfolgreich zum Abschluss zu bringen.

Inhaltsverzeichnis

1 Einleitung **11**
- 1.1 Motivation . . . 11
- 1.2 Zielsetzung . . . 13
- 1.3 Aufbau dieser Dissertation . . . 13

2 Grundlagen **15**
- 2.1 Rollenorientierte Programmierung . . . 15
  - 2.1.1 Modulkonzept . . . 16
  - 2.1.2 Rolle-Basis-Beziehung . . . 17
  - 2.1.3 Rollen im Kontext . . . 20
  - 2.1.4 Vererbung . . . 20
- 2.2 Aspektorientierte Programmierung . . . 23
  - 2.2.1 Modulkonzept . . . 24
  - 2.2.2 Aspekt-Basis-Beziehung . . . 26
  - 2.2.3 Abgrenzung zur Rollenorientierung . . . 27
- 2.3 Logische Programmierung . . . 29
  - 2.3.1 Fakten und Abfragen . . . 29
  - 2.3.2 Variablen und Regeln . . . 30
  - 2.3.3 Listen . . . 31
- 2.4 Wiederverwendung . . . 32
  - 2.4.1 Prozess . . . 32
  - 2.4.2 Kosten und Nutzen . . . 34
  - 2.4.3 Techniken . . . 35
  - 2.4.4 Skalierbarkeit . . . 37

3 Modularisierung und Wiederverwendung in Object Teams **39**
- 3.1 Problemdiskussion . . . 39
  - 3.1.1 Motivierendes Beispiel . . . 39
  - 3.1.2 Problemstellung . . . 41
  - 3.1.3 Problemanalyse . . . 43
- 3.2 Lösungsansatz . . . 44
  - 3.2.1 Introspektion . . . 45
  - 3.2.2 Quantifizierung . . . 46
  - 3.2.3 Generizität . . . 48
  - 3.2.4 Integration . . . 48

4 Die Spracherweiterung Generic Object Teams **51**
- 4.1 Faktendarstellung . . . 52
- 4.2 Queries . . . 55
  - 4.2.1 Query-Deklaration . . . 55
  - 4.2.2 Metavariablen . . . 56
  - 4.2.3 Query-Definition . . . 57

4.2.4 Query-Klassen . . . 61
4.2.5 Auswertung der Queries . . . 62
4.3 Grundlegende Transformation . . . 63
4.3.1 Teamquery . . . 63
4.3.2 Generizität . . . 67
4.3.3 Rollout . . . 70
4.3.4 Gültigkeit generischer Anweisungen . . . 78
4.3.5 Gültigkeit und Korrektheit des Einsatzes von Metavariablen . 80
4.4 Vererbung und erweiterte Transformation . . . 82
4.4.1 Anforderungen zur Wiederverwendung generischer Teams . . 82
4.4.2 Wiederverwendung generischer Teams . . . 84
4.4.3 Vollständiges Beispiel . . . 88
4.4.4 Rekursive Strukturen . . . 92
4.5 Korrektheit der Transformation . . . 94

5 Technische Umsetzung **97**
5.1 Realisierung . . . 97
5.1.1 Compiler und Anbindung an Eclipse . . . 98
5.1.2 Anbindung an Prolog . . . 99
5.1.3 Transformation . . . 99
5.2 Fazit und gewonnene Erkenntnisse . . . 100

6 Evaluation **103**
6.1 Aufbau und Kriterien . . . 103
6.2 Verbesserung durch generische Teams . . . 106
6.2.1 Visitor . . . 106
6.2.2 Memento . . . 108
6.2.3 Abstract Factory . . . 114
6.2.4 Observer und Mediator . . . 116
6.3 Verbesserung durch nicht-generische Teams . . . 116
6.3.1 Chain of Responsibility . . . 116
6.3.2 Composite . . . 118
6.3.3 Modularisierung mit gebundenen Rollen . . . 120
6.3.4 Modularisierung mit ungebundenen Rollen . . . 122
6.4 Keine Verbesserung durch Teams . . . 124
6.5 Andere Anwendungen . . . 126
6.6 Diskussion vergleichbarer Studien . . . 127
6.6.1 *Hannemann und Kiczales 2002* . . . 127
6.6.2 *Monteiro und Gomes 2010 und 2012* . . . 128
6.7 Fazit . . . 134

7 Verwandte Arbeiten **137**
7.1 LMP-Ansätze in der AOP . . . 137
7.2 Generizität durch strukturierte Metaprogrammierung . . . 138
7.3 Rollen und Kontext . . . 139
7.4 Modularisierung von Design Patterns . . . 140

8 Zusammenfassung und Ausblick **143**
8.1 Beiträge und Ergebnisse . . . 143
8.2 Diskussion . . . 144
8.3 Ausblick . . . 145

A Grammatik **147**

Abbildungsverzeichnis **153**

Listingverzeichnis **154**

Abkürzungsverzeichnis **155**

Literaturverzeichnis **156**

Kapitel 1

Einleitung

1.1 Motivation

Die Motivation für diese Dissertation entstammt der einfachen Frage: „Warum muss ich das noch einmal machen?“. Vor einiger Zeit implementierten wir im Rahmen einer einfachen Java-Anwendung eine Variante des Observer Patterns – es war nicht unsere erste – und während wir abermals unsere `notify`-Aufrufe im Code verteilten, stellte sich eben diese Frage. Die grundlegende Idee des Musters ist altbekannt; ähnliche Anwendungsfälle haben wir bereits mehrfach auf diese Weise gelöst und die konkrete Ausprägung für unseren aktuellen Anwendungsfall lässt sich in einem simplen Satz beschreiben – und dennoch fehlt uns eine effiziente Möglichkeit, bestehende Implementierungen für unsere Zwecke wiederzuverwenden. Auch der Einsatz einer Programmiersprache wie *Object Teams*, die eine bessere Unterstützung zur Umsetzung des Observer Patterns bietet und die wir später im Detail vorstellen werden, hilft uns an dieser Stelle nicht – das Problem der Wiederverwendbarkeit ist universell und begegnet uns in jeder objektorientierten Programmiersprache. Stattdessen schreiben wir Code, den wir bis auf wenige Nuancen in dieser Form schon mehrfach geschrieben haben.

Das DRY-Prinzip (*„don't repeat yourself“*) zählt längst zum Standard der Softwareentwicklung [65]. Die Erkenntnis, dass Wiederholungen im Code mit Aufwand für Erstellung und Pflege verbunden sind, kann früher oder später jeder Programmierer auch aus eigener Erfahrung nachvollziehen. Redundanz ist eben nicht effizient. Die Lösung ist so alt wie die Softwarekrise: Wiederverwendung. Die Idee der Wiederverwendung besteht in der Generalisierung und Spezialisierung von Lösungen. Wir möchten allgemeingültige Anteile einer Lösung wiederverwenden und spezifische Teile ergänzen oder redefinieren. Grundlage hierfür ist das Prinzip *Separation of Concerns* [27], das die Aufteilung von Software in kleine Module mit klar abgegrenzter Funktionalität verlangt [90]. Man spricht von *Modularisierung*. Die klare Trennung der Funktionalitäten erlaubt es, interne Details des Moduls, die für dessen beobachtbares Verhalten nicht relevant sind, vor der Außenwelt zu verbergen (*Information Hiding* [89]). Diese Abstraktion erleichtert das Verständnis und die Wartbarkeit von Modulen, da der relevante Code zur Umsetzung einer Funktionalität in einer überschaubaren Einheit gebündelt ist, die keine oder nur wenige Abhängigkeiten zu anderen Modulen aufweist (das Prinzip *Lokalität*[1]). Je eher ein Modul diesen Prinzipien gerecht wird, desto einfacher wird es sich mit anderen Modulen kombinieren lassen und taugt somit zur Wiederverwendung in verschiedenen Anwendungen.

[1] Liskov: „Locality allows a program to be implemented, understood, or modified one module at a time“ [79]

Demgegenüber steht eine andere Entwicklung der Softwareindustrie: das Streben nach größeren Modulen. Die Evolution der Programmiersprachen wird begleitet vom Wachstum der Einheiten, mit denen wir unseren Code organisieren: Anweisungen wurden in Funktionen gebündelt, Funktionen in Bibliotheken; die gemeinsame Bündelung von Daten und Funktionen wurde durch Datenstrukturen realisiert, die wiederum in Paketen o.ä. gebündelt wurden. Das Konzept solcher abstrakter Datentypen bildet mittlerweile einen weitverbreiteten Standard – manifestiert in Klassen und Objekten der objektorientierten Programmierung. Ein Ende des Modul-Wachstums ist nicht in Sicht. In den vergangenen Jahren sprechen wir verstärkt von Komponenten – ein Begriff, der allerdings mit sehr unterschiedlichen Vorstellungen verbunden ist. Zudem sind zahlreiche Frameworks entstanden, die als vorgefertigtes Gerüst für ganze Applikationen dienen. Dahinter steht der Wunsch, größere Code-Einheiten zu bilden, die sich einfacher kombinieren bzw. integrieren lassen und somit neue Dimensionen der Wiederverwendung ermöglichen. Der darin liegende Widerspruch ist offensichtlich: Wiederverwendbarkeit erfordert von Modulen, dass sie eigenständig und abgeschlossen sind, während Kombinierbarkeit verlangt, dass Module flexibel und anpassbar sind. Auf der Klassen-Ebene kennen wir dieses Dilemma als das Offen-Geschlossen-Prinzip ([85], Kapitel 3), das zumindest ausreichend durch Vererbung gelöst wird: die Schnittstelle ist geschlossen, die Implementierung dagegen – dank dynamischen Bindens – offen für Änderungen. Dennoch ist auch diese Lösung keineswegs perfekt. Jüngere Entwicklungen legen nahe, dass ein einzelner Mechanismus zur Komposition und Dekomposition von Modulen allein nicht ausreicht – es existiert kein universeller Mechanismus, der allen Belangen gerecht werden kann. Aus dieser Überlegung sind zahlreiche ergänzende Konzepte zur Modularisierung entstanden, wie die aspektorientierte und die rollenorientierte Programmierung. Diese Ansätze erlauben eine mehr-dimensionale Zerlegung als Mittel, um der „Tyrannei der dominanten Dekomposition“ [102] durch Vererbung zu begegnen. Das Dilemma der Wiederverwendbarkeit ist damit jedoch nicht gelöst, es zeigt sich nur in anderen Facetten.

Modulkonzepte sind eng mit der statischen Struktur eines Programms verbunden. Auch die Komposition von Modulen, etwa durch Vererbung, erfolgt üblicherweise statisch. Diese Statik stellt eine bedeutende Grenze für die Flexibilität von Modulen dar – und damit auch für deren universelle Einsetzbarkeit. Die Überwindung eben dieser Grenze ist ein wesentliches Mittel, um Wiederverwendbarkeit zu fördern. Erforderlich dazu ist eine Aufweichung oder besser formuliert eine „Dynamisierung“ der statischen Strukturen. Bei Klassen wird dies durch virtuelle Methoden und dynamisches Binden erreicht. Methodenaufrufe an einem Objekt werden dynamisch delegiert, erst zur Laufzeit entscheidet sich, welche Implementierung zur Ausführung kommt. Die damit realisierte Dynamik führt zu einer anderen bzw. vielfältigeren Programmausführung, als es die statische Struktur des Programms zunächst vermuten lässt. Weiterführende Modulkonzepte wie Aspekte oder Rollen nutzen das gleiche Prinzip, um die rigide statische Kopplung und Ablaufsteuerung zwischen Modulen flexibler zu gestalten. Auch hier entstehen dynamische Kontrollflüsse, die nicht mehr allein den statischen Strukturen folgen.

Ob Vererbung und dynamisches Binden, Aspekte oder Rollen, stets geht es darum, dass ein Modul ein anderes Modul beeinflusst, ohne dass dieses selbst an der Definition beteiligt ist (oder überhaupt Kenntnis davon hat). Ein Modul definiert Effekte, die in anderen Modulen zum Tragen kommen. Auf diese Weise gelingt es, Module sehr viel flexibler zu gestalten, denn die Grenzen der dynamischen Adaption werden über die Grenzen des statischen Moduls hinaus verschoben – eine konsequente Unterstützung des Separation-of-Concerns-Prinzips und damit natürlich auch im Interesse der Wiederverwendung.

Die Kehrseite zeigt sich im Detail. Die erbende Klasse, der Aspekt und die Rolle, sie alle sind eng an ihre Partnermodule gekoppelt. Die Dynamik ist somit fest in

der statischen Struktur verankert und verliert dabei einen Teil ihrer Flexibilität. Zwar teilen wir augenscheinlich die Funktionalitäten in mehrere Module auf, deren Zusammenspiel ist aber stellenweise so eng gekoppelt, dass eine Dekomposition nur noch eingeschränkt möglich ist. Im Endeffekt formieren wir einen festen Verbund von Modulen und tragen somit weiter zum Wachstum der Code-Einheiten bei. Die Möglichkeiten der Wiederverwendung wachsen dabei leider nicht mit, sie folgen weitgehend den altbekannten Mechanismen der Vererbung.

1.2 Zielsetzung

Diese Dissertation soll einen Beitrag leisten, die Problematik der Wiederverwendung besser zu verstehen und die Wiederverwendbarkeit von Software-Modulen zu fördern. Dazu sollen Lösungsansätze entwickelt werden, um die angesprochene, teils feste Kopplung von vermeintlich separierten Modulen zu reduzieren. Ziel ist die Dynamisierung statischer Strukturen, um damit Modularisierung zu fördern und neue Möglichkeiten der Wiederverwendung zu schaffen.

Als Ausgangsbasis dient das Paradigma der rollenorientierten Programmierung, die das Modulkonzept klassischer objektorientierter Programmiersprachen dahingehend verfeinert, dass ein dynamisches Zusammenspiel von Klassen in Kollaborationen möglich ist. Dieses Modulkonzept soll darüber hinaus erweitert werden, dass es Klassen mit flexibler Struktur erlaubt, deren Eigenschaften und Verhalten deklarativ beschrieben werden. Die deklarative Definition soll die Struktur unabhängig von einem bestimmten Kontext machen, so dass sie ohne zusätzliche Verfeinerungen in verschiedenen Kontexten anwendbar ist. Dies erfordert einen zusätzlichen Grad an Generizität, damit die spezifische statische Struktur je nach Kontext unterschiedliche Ausprägungen annehmen kann. Konkret wird die Erweiterung für die rollenbasierte Programmiersprache *Object Teams* umgesetzt.

Diese Dissertation baut auf fortgeschrittenen Techniken der Modularisierung und Programmierung auf, wohlwissend, dass einiges davon noch nicht zum etablierten Repertoire der vorherrschenden Programmiersprachen gehört. Der Ansatz kommt entsprechend der gewachsenen Komplexität der Problematik nicht ohne eine wachsende Komplexität der Technik aus, deren Beherrschung man sicher nicht ad hoc von „Joe the Programmer“[2] verlangen darf. Nichtsdestotrotz hoffen wir, einen nützlichen Gedankenanstoß zu bieten, der Möglichkeiten für die zukünftige Weiterentwicklung der Modularisierungstechniken aufzeigt. Evolution bedarf bekanntlich längerer Entwicklungszeiträume. Schließlich gab es Zeiten, in denen man auch dynamisches Binden als zu exotisch und kompliziert für „Joe“ angesehen hat – mittlerweile gehört diese Technik längst zum Standard.

1.3 Aufbau dieser Dissertation

Diese Dissertation besteht aus acht Kapiteln.

Kapitel 2 gibt eine Einführung in die rollenorientierte, die aspektorientierte sowie die logische Programmierung. Jedes der Programmierparadigmen wird anhand einer repräsentativen Programmiersprache erläutert. Die vorgestellten Sprachen werden in den folgenden Kapiteln als Referenz für konkrete Fragestellungen zu Syntax und Semantik herangezogen. Darüber hinaus werden in Abschnitt 2.4 die grundlegenden Prinzipien und Techniken der Wiederverwendung vorgestellt.

In Kapitel 3 werden die Defizite gängiger Modularisierungskonzepte der objekt- bzw. rollenorientierten Programmierung mit Blick auf die Wiederverwendbarkeit

[2] der Ausdruck basiert auf *Joe the Plumber*, der im US-Wahlkampf 2008 im Rahmen einer Diskussion um die Steuerpolitik als plakatives Beispiel herangezogen wurde. *Joe* wird seither gern als durchschnittlicher Repräsentant einer Gruppe von pragmatisch-orientierten Individuen genannt.

analysiert. In diesem Zusammenhang formulieren wir die grundlegende Problemstellung dieser Dissertation. Davon ausgehend motivieren wir einen weiterführenden Ansatz zur Wiederverwendung, den wir mit unserer Arbeit verfolgen.

Die Kapitel 4 bis 6 bilden den Hauptbeitrag dieser Dissertation. In Kapitel 4 stellen wir *Generic Object Teams*, eine Spracherweiterung zu Object Teams, als Realisierung des in Kapitel 3 vorgestellten Ansatzes vor. Wir diskutieren dabei sowohl theoretische Fragen des Sprachentwurfs als auch praktische Details der Realisierung. Kapitel 5 geht in Kürze auf die Umsetzung eines Prototyps für die Eclipse-Plattform ein.

In Kapitel 6 wird die Anwendbarkeit und der Nutzen von Object Teams anhand von Design Patterns diskutiert. Es werden konkrete Beispiele vorgestellt, mit denen der Mehrwert der Spracherweiterung in Bezug auf die Wiederverwendbarkeit von Entwurfsmuster-Implementierungen demonstriert wird.

In Kapitel 7 werden vergleichbare Ansätze diskutiert. In Kapitel 8 wird die Arbeit mit einer Zusammenfassung und einem Ausblick abgeschlossen.

Kapitel 2

Grundlagen

Zum besseren Verständnis dieser Arbeit werden in diesem Kapitel grundlegende Paradigmen und Konzepte der Programmierung erläutert. Diese Dissertation beschäftigt sich vornehmlich mit Konzepten des Paradigmas der objektorientierten Programmierung (OOP), in jener Ausprägung, wie sie in den vorherrschenden objektorientierten Programmiersprachen Java, C++ und C# (siehe Tiobe-Index [103]) zu finden ist. Im Kern steht ein Modulkonzept, in dem Klassen die wesentlichen Typen und Programmiereinheiten sind. Der wesentliche Mechanismus zur Verfeinerung von Klassen ist Vererbung. Information Hiding wird durch Kapselung realisiert. Wir gehen davon aus, dass die Konzepte der OOP hinlänglich bekannt sind. Sollte der Leser damit nicht vertraut sein, raten wir zur Lektüre von Meyer [85], worin die Motive und Mechanismen der OOP detailliert erläutert werden. Wenn es um konkrete programmiersprachliche Konstrukte und Anwendungen geht, so beziehen wir uns auf die Sprache Java und verwandte Sprachen. Ein umfassende Einführung in Java gibt [104].

Im Fokus dieser Arbeit steht die rollenorientierte Programmierung (ROP), welche Ähnlichkeiten zur aspektorientierten Programmierung (AOP) aufweist und meist in deren Kontext diskutiert wird. Beide Ansätze erweitern die klassischen Modulkonzepte der OOP und werden in den Abschnitten 2.1 und 2.2 näher vorgestellt. Darüber hinaus wird in Abschnitt 2.3 mit der logischen Programmierung (LP) ein weiteres Paradigma vorgestellt, das in dieser Arbeit im Rahmen der Metaprogrammierung zum Einsatz kommt. Alle genannten Arten und Konzepte der Programmierung werden konkret anhand einer repräsentativen Programmiersprache der jeweiligen Richtung erläutert, beschränkt auf die für das Verständnis dieser Arbeit wesentlichen Teile; eine umfassendere Erläuterung ist in der zitierten Literatur gegeben.

Im Vordergrund der weiteren Betrachtungen dieser Arbeit steht die Modularisierung von Software, ein besonderer Fokus liegt dabei auf der Wiederverwendung. Die Motivation und der Prozess der Wiederverwendung werden in Abschnitt 2.4 einführend erläutert.

2.1 Rollenorientierte Programmierung

Die rollenorientierte oder, weiter gefasst, rollenbasierte Programmierung hat zum Ziel, das gedankliche Konzept der *Rolle* [74] direkt in der Programmiersprache zu unterstützen. Eine Rolle beschreibt die Kollaboration eines Objekts in einem Kontext. Zustand und Verhalten des Objekts können nicht mehr ausschließlich global, sondern auch in Abhängigkeit von einem Kontext definiert werden. Die ROP bietet allgemein verbesserte Möglichkeiten zur Modularisierung, indem kontextspezifische

Merkmale separat vom Objekt in einer Rolle verwaltet werden können. Zwischen einem Objekt und seinen Rollen besteht dabei typischerweise eine Sharing-Relation, d.h. Zustand und Funktionalität werden von den Beteiligten (zumindest partiell) geteilt. Steimann diskutiert das Konzept der Rolle in [97] ausführlich und definiert 15 Kriterien, die für eine erfolgreiche Realisierung des Konzepts zu erfüllen sind.

Obwohl Rollen in mehreren Ebenen des Softwaredesigns – von der allgemeinen sprachlichen Beschreibung bis hin zur Modellierung – breite Verwendung finden, so gibt es doch nur wenige Programmiersprachen, die eine direkte Unterstützung von Rollen anbieten [56]. Somit können viele objektorientierte Programmiersprachen der Regel der direkten Abbildung (*Direct Mapping*, nach Meyer aus [85]) von Modulstrukturen durch die verschiedenen Ebenen des Designs nicht gerecht werden. Ein Programmiermodell mit expliziter Unterstützung von Rollen ist Object Teams (OT) [53]. Das Modell wurde konkret in der Programmiersprache Object Teams/Java (OT/J)[3], als Erweiterung der Sprache Java, realisiert. OT/J erfüllt weitgehend die in [97] aufgeführten Kriterien [56]. OT steht im Fokus dieser Arbeit und ist Grundlage für die in Kapitel 4 vorgestellte Spracherweiterung *Generic Object Teams*.

2.1.1 Modulkonzept

OT ergänzt das klassenbasierte Modulkonzept um zwei neue Modultypen: *Team* und *Rolle*. Ein Team repräsentiert den Kontext einer Kollaboration; eine Rolle beschreibt die kontextspezifischen Merkmale eines Objekts, das an der Kollaboration beteiligt ist. Ein Beispiel ist der Kontext *Universität*, der eine Rolle *Student* für ein Objekt *Person* beschreibt; eine kontextspezifische Eigenschaft des Studenten ist seine Matrikelnummer.

Eine Rolle pflegt zwei spezielle Beziehungen: (1) Eine Rolle ist existentiell an einen Kontext gebunden, der ihre Bedeutung definiert. Ohne Kontext, z.B. eine Universität, kann es keine Rolle, wie einen Studenten, geben. (2) Eine Rolle kann an eine Basis gebunden sein; diese Basis ist das Objekt, das die Rolle im Kontext spielt. Hier wird zwischen gebundenen und ungebundenen Rollen unterschieden. Eine gebundene Rolle benötigt einen „Spieler“, der die Rolle einnimmt, als Basis. Die Basis ist dabei ein Objekt, das unabhängig vom Kontext existiert, beispielsweise eine Person als Basis für einen Studenten. Die Bindung zwischen Rolle und Basis definiert eine erweiterte Sharing-Beziehung, ähnlich der Vererbung: die gebundene Rolle teilt (zumindest einige) Merkmale mit der Basis, z.B. ist der Name des Studenten gleich dem Namen der Person. Darüber hinaus kann sie das Verhalten der Basis verfeinern. Eine ungebundene Rolle benötigt dagegen keine Basis, d.h. ihre Merkmale sind vollständig innerhalb des Kontexts definiert. Im Kontext einer Universität kann z.B. das *Prüfungsamt* eine wichtige Rolle spielen. Für das Prüfungsamt gibt es aber im Normalfall kein korrespondierendes Objekt außerhalb des Kontexts, mit dem Merkmale geteilt werden sollen, insofern bleibt die Rolle ungebunden. Die direkte Verwendung ungebundener Rollen ist allerdings eher selten. In der Regel dient eine ungebundene Rolle als Generalisierung für weitere Rollen, die spezialisiert und gebunden werden. In den folgenden Kapiteln ist, sofern nicht anders angegeben, von gebundenen Rollen die Rede.

Sowohl Kontext als auch Rollen sind in OT als spezialisierte Ausprägungen normaler Klassen realisiert. Ein Kontext wird von einer *Teamklasse* repräsentiert. Sie wird durch das zusätzliche Schlüsselwort `team` definiert. Rollen werden von *Rollenklassen* repräsentiert. Sie sind innere Klassen einer Teamklasse. Eine beispielhafte

[3]Die Unterscheidung zwischen dem Modell OT und der Sprache OT/J ist im Rahmen dieser Arbeit irrelevant, da OT/J bis dato die einzige Umsetzung von OT ist. Die Begriffe können daher im praktischen Sinne als synonym aufgefasst werden. Fortan werden wir uns allgemein auf OT beziehen.

Implementierung des Universitäts-Szenarios ist in folgendem Code dargestellt:

```
public class Person {
  ...
}

public team class University {
  public class Student playedBy Person {
    ...
  }
}
```

Als Basis dient eine einfache Klasse `Person`. Den Kontext bildet die Teamklasse `University`. Innerhalb des Kontexts ist die Rollenklasse `Student` definiert, die mit dem Schlüsselwort `playedBy` an die Basisklasse gebunden ist. Potentiell kann jede Klasse, selbst wenn es sich um eine Rollen- oder Teamklasse handelt, als Basisklasse in einer Rollenbeziehung eingesetzt werden[4].

Entsprechend der Semantik für innere Klassen sind Instanzen der Rollenklassen, die *Rollenobjekte*, an die Instanz der äußeren Teamklasse, das *Teamobjekt*, gebunden. Für eine gebundene Rolle gilt dabei, dass sie auch eine Referenz auf eine Instanz der angegeben Basisklasse, das *Basisobjekt*, besitzt. Diese Referenz ist in OT unveränderlich über die gesamte Lebensdauer des Rollenobjekts. Das Basisobjekt ist ein fester Bestandteil des Rollenobjekts. Für die Instanziierung einer gebundenen Rollenklasse ist somit zwingend ein Basisobjekt erforderlich. Ein Basisobjekt kann normalerweise im Kontext eines bestimmten Teamobjekts jede Rolle nur einmal annehmen. Der Vorgang wird *Lifting* genannt. Dabei wird nur dann ein neues Rollenobjekt im umschließenden Team erstellt, sofern dort noch keine Instanz der jeweiligen Rollenklasse existiert, die demselben Basisobjekt zugeordnet ist. Ansonsten wird das bereits vorhandene Rollenobjekt verwendet. Beispielsweise kann für das Objekt `paul:Person` nur eine Studenten-Rolle innerhalb des Teams `tu:University` existieren. Weitere Studenten-Rollen könnte das Objekt nur im Kontext anderer Teaminstanzen annehmen. Lifting wird durch mehrere implizite und explizite Mechanismen gesteuert, die im Abschnitt 2.1.3 genauer erläutert werden.

Durch den Rollenmechanismus in OT bekommen Objekte eine höhere Bedeutung gegenüber Klassen. Ein Basisobjekt wird nicht mehr allein durch sich selbst repräsentiert, sondern kann mit einer beliebigen Menge von Rollenobjekten verbunden sein, die nach außen hin als ein gemeinsamer Verbund agieren. Je nach Kontext ist nur ein Teil dieses Verbunds jeweils aktiv bzw. sichtbar. Diese Art der Komposition bietet eine flexible und dynamische Alternative zur Vererbung, gemäß dem Grundsatz *„favor object composition over class inheritance“* (das zweite Prinzip objektorientierten Designs aus [39]).

2.1.2 Rolle-Basis-Beziehung

Die Beziehung zwischen Rollen- und Basisklasse teilt die wesentlichen Eigenschaften einer Vererbungsbeziehung zwischen Unter- und Oberklasse: (1) *Sharing*: die Rolle kann Merkmale der Basis importieren. (2) *Overriding*: die Rolle kann das Verhalten der Basis verfeinern. (3) *Polymorphie*: es existiert eine Substitutionsrelation zwischen Rolle und Basis. Nachfolgend erläutern wir diese Mechanismen im Detail.

[4] Aus technischen Gründen können in OT bis dato allerdings einige Klassen der Java Standard API nicht durch Rollen adaptiert werden. Interfaces können nur eingeschränkt adaptiert werden, dies wird in Abschnitt 6.5 thematisiert.

Sharing

Eine Rolle kann auf Eigenschaften und Methoden der Basis zugreifen. Im Unterschied zur Vererbung erbt eine Rolle allerdings nicht implizit alle Merkmale der Basis. Stattdessen deklariert eine Rolle ihr Interface explizit und unabhängig vom Interface der Basis; es besteht keine Konformität zwischen Rolle und Basis wie etwa in einer Subtypbeziehung. Zur Definition ihrer Merkmale kann die Rolle selektiv auf Merkmale der Basis verweisen. Eine solche Importanweisung wird in OT als *Callout* bezeichnet. Dabei sind der Rolle auch teilweise solche Zugriffe an der Basis möglich, die aufgrund von Kapselung normalerweise nicht erlaubt wären. Dieses Prinzip einer flexibel abgestuften Form der Kapselung wird *Gradual Decapsulation* [57] genannt.

Ein Callout wird mit dem Operator „`->`“ definiert. Auf der linken Seite steht dabei eine von der Rolle deklarierte abstrakte Methode. Auf der rechten Seite muss eine Methode der Basis aufgeführt werden, die als Implementierung der abstrakten Methode verwendet werden soll. Der Callout definiert also die Methode aus dem Interface der Rolle mit einem Verweis auf eine Methode der Basis. Effekt ist, dass ein Aufruf der Rollenmethode an die Basismethode weitergeleitet wird. Es handelt sich hier um einfaches Forwarding vom Rollenobjekt zum Basisobjekt. Alternativ zur Angabe einer Methode der Basis kann ein Callout auch über die Modifikatoren `get` und `set` direkt den Verweis auf ein Feld der Basis definieren. Beide Varianten des Callouts sind beispielhaft in folgendem Codefragment dargestellt:

```
public class Person {
  private String name;
  public String getName() {...}
}

public team class University {
  public class Student playedBy Person {
    abstract String getStudentName1();
    getStudentName1 -> getName;
    abstract String getStudentName2();
    getStudentName2 -> get name;
  }
}
```

Die deklarierte Methode `getStudentName1` der Rolle wird durch einen Callout auf die Methode `getName` der Basis definiert. Die Methode `getStudentName2` wird analog dazu über einen Callout auf das Feld `name` der Basis definiert.

Sollten die Signaturen von Rollen- und Basismethode in einem Callout identisch sein, so muss der Callout nicht explizit definiert werden, sondern kann implizit inferiert werden. Sollten die Signaturen nicht konform sein oder die Argumente einer Anpassung bedürfen, so kann die Bindung durch sogenanntes *Parameter Mapping* verfeinert werden. Dabei sind Anzahl und Typ der Argumente mit einem Ausdruck der Form `with {...}` so anzupassen, dass eine Weiterleitung zwischen den Methoden möglich ist, beispielsweise

```
void m(Integer n) -> void f(int i) with {
  n.intValue() -> i
};
```

Overriding

Eine Rolle kann das Verhalten der Basis verfeinern, analog zum Overriding bei einer Vererbungsbeziehung. Diese Form der Bindung wird in OT *Callin* genannt

und mit dem Operator „<-“ definiert. Auf der linken Seite des Operators steht eine (nicht-abstrakte) Rollenmethode, auf der rechten Seite ein Modifikator zur Spezifikation des Callins und eine Basismethode. Eine Callin-Bindung hat den Effekt, dass der Befehl zur Ausführung der entsprechenden Basismethode abgefangen (*Method Call Interception* (MCI)) und an die gebundene Rollenmethode weitergeleitet wird. Man spricht hier auch von *Implicit Invocation*, da hier kein direkter Aufruf der Rollenmethode stattfindet, sondern die Rolle die Ausführung der Basismethode als implizites Ereignis nutzt, um sich selbst an dieser Stelle aufzurufen. Dabei können drei Varianten des Verhaltens durch verschiedene Callin-Modifikatoren spezifiziert werden: *before*, *after* und *replace*. Before- und After-Callins ergänzen die Basismethode um einen Vor- bzw. Nachspann, ein Replace-Callin ersetzt die Basismethode vollständig. Eine in einem Replace-Callin gebundene Rollenmethode muss mit dem Schlüsselwort `callin` deklariert werden; innerhalb dieser Methode kann über eine Referenz `base` auf die Original-Implementierung der Basismethode zugegriffen werden (vergleichbar mit *super*-Aufrufen bei Vererbung). Ein Beispiel für eine Callin-Bindung ist in folgendem Codefragment dargestellt:

```
public class Person {
  public boolean doParty() {return true;}
}

public team class University {
  public class Student playedBy Person {
    callin boolean dontParty() {return false;}
    dontParty <- replace doParty;
  }
}
```

Die Rollenmethode `dontParty` ersetzt hier die Basismethode `doParty` (ohne *base*-Aufruf). Die Ausführung der Methode `doParty` wird entsprechend an die Methode `dontParty` delegiert.

Analog zur Callout-Bindung kann auch bei einer Callin-Bindung über Parameter Mapping Konformität zwischen unterschiedlichen Methodensignaturen hergestellt werden.

Polymorphie

Die Rolle-Basis-Beziehung in OT definiert keine Subtyp-Beziehung, wie es etwa bei Vererbung der Fall ist. Die Interfaces von Rolle und Basis sind unabhängig voneinander deklariert, durch Callouts und Callins wird lediglich selektiv Konformität einzelner Merkmale hergestellt. Dennoch ermöglicht OT die Substitution von Werten zwischen den Typen von Rolle und Basis. Grundlage hierfür ist, dass eine Rolle nicht durch ein einzelnes Objekt, sondern durch einen Verbund aus einem Rollen- und einem Basisobjekt repräsentiert wird. Der Typ der Rolle kann daher ohne Weiteres als zuweisungskonform zum Typ der Basis interpretiert werden, da mit einem gebundenen Rollenobjekt zwingend ein Basisobjekt verbunden ist, dass an seiner Stelle die Anforderungen des Typs erfüllen kann. Entsprechend wird bei der Belegung einer Variablen deklariert mit dem Typ der Basis, wie z.B.

```
Person p = student; // student von Typ Student
```

das Rollenobjekt einfach durch das mit ihm verbundene Basisobjekt ersetzt. Der Mechanismus wird in OT *Lowering* genannt. Die entgegengesetzte Richtung wird durch den Lifting-Mechanismus behandelt: Ein Basisobjekt, das in einem gegebenen Kontext den Typ einer Rolle annehmen soll, wird durch ein korrespondierendes Rollenobjekt ersetzt. Die verschiedenen Arten des Liftings werden im Abschnitt 2.1.3

genauer erläutert. Diese Form der Polymorphie in OT wird *Translation Polymorphism* genannt.

2.1.3 Rollen im Kontext

Teamklassen bilden den Kontext für Kollaborationen von Rollen. Der gesamte Mechanismus des Rollenspiels wird in OT dynamisch gesteuert. Eine Rolle ist nur dann aktiv und nimmt Einfluss auf eine Basis, wenn der Kontext aktiv ist. Dies bedeutet konkret, dass eine Instanz der Teamklasse existieren muss. Das Teamobjekt kann dann dynamisch aktiviert und deaktiviert werden. Nur im aktiven Zustand werden Callin-Bindungen der Rollen ausgeführt.

Eine Callin-Bindung einer Rollenklasse ist zunächst universell für alle Instanzen der deklarierten Basisklasse gültig und führt bei Ausführung der Basismethode implizit zum Lifting des Basisobjekts. Sofern zum Zeitpunkt des Liftings noch kein Rollenobjekt für das Basisobjekt existiert, wird es an dieser Stelle neu erzeugt. Das Lifting kann aber durch sogenannte *Guard Predicates* so eingeschränkt werden, dass es (und damit der Rollenmechanismus) nur in bestimmten Situationen aktiv wird. Guard Predicates können auf der Ebene einzelner Methoden, Rollen- oder Teamklassen angegeben werden. Ein Guard Predicate wird durch das Schlüsselwort `when` eingeleitet, gefolgt von einer Bedingung in Form eines boolschen Ausdrucks, z.B.

```
public team class University {
  boolean isLecturePeriod() {...}

  public class Student playedBy Person
    base when (isLecturePeriod()) {...}
}
```

Nur bei positiver Auswertung der Bedingung kommt es zum Lifting und die Rolle wird aktiv. Basisobjekte können auch explizit zu Rollenobjekten geliftet werden, durch sogenanntes *Declared Lifting*. Hierzu dient der Operator `as`, der die Typen von Rolle und Basis bei der Typangabe eines Methodenparameters verbindet. Ein Beispiel ist mit Methode `matriculate` in der Teamklasse `University` dargestellt:

```
public team class University {
  void matriculate(Person as Student student) {...}
  ...
}
```

Erwartet wird hier ein Argument vom Typ `Person`. Das übergebene Objekt wird bei Ausführung der Methode automatisch durch ein Rollenobjekt vom Typ `Student` ersetzt. Innerhalb der Methode hat der Parameter `student` den Typ `Student`.

2.1.4 Vererbung

Die normale Vererbungsbeziehung zwischen Klassen mit `extends` ist auch auf Rollen- und Teamklassen anwendbar. Die Semantik wird dabei um einige Rollen- bzw. Team-spezifische Besonderheiten erweitert.

Vererbung zwischen Rollen

Die Vererbung zwischen Rollen entspricht der normalen Vererbung zwischen zwei Klassen, wobei die erweiterte Rollendefinition, wie z.B. eine Rolle-Basis-Beziehung, ebenfalls vererbt wird. Das Ergänzen und Redefinieren von Funktionalität ist nach wie vor möglich. Bei Fragen der Konformität muss nun allerdings auch die Rolle-Basis-Beziehung berücksichtigt werden. Eine Rolle, die von einer ungebundenen

Rolle erbt, darf eine Basisbeziehung ergänzen. Eine existierende Bindung zu einer Basis darf von einer erbenden Rolle nicht entfernt, durchaus aber kovariant verfeinert werden. Dadurch kann es möglich sein, dass beim Lifting für einen Basistyp mehrere Rollentypen derselben Vererbungshierarchie zur Auswahl stehen, indem eine Rolle für einen konkreten Basistyp und eine andere Rolle für dessen Subtyp existieren. Der Lifting-Prozess ist dabei so realisiert, dass er (analog zur üblichen Semantik) automatisch die spezifischste unter den passenden Rollen auswählt, wobei auch der dynamische Typ des Basisobjekts berücksichtigt wird — die intelligente Auswahl der Rolle wird *Smart Lifting* genannt.

Im folgenden Beispiel nehmen wir an, dass zwei Klassen `A` und `B` existieren, wobei `B` eine Subklasse von `A` ist. Das Team

```
public team class T0 {

  abstract public class R {}

  public class RA extends R playedBy A {}

  public class RB extends RA playedBy B {}

}
```

deklariert drei Rollen. Die abstrakte Rolle `R` dient als Oberklasse für die Rolle `RA`. `RA` ergänzt die Beziehung zur Basisklasse `A`, während die Oberklasse `R` ungebunden ist. Die Rolle `RB` wiederum spezialisiert `RA` und verfeinert dabei die Basisklasse zu `B`. Dies ist erlaubt, da `B` ein Subtyp von `A` ist. Die drei Rollenklassen bilden eine gemeinsame Hierarchie innerhalb des Teams, ein Basisobjekt darf daher nur von einer der Rollen gespielt werden. Existiert nun ein Objekt der Klasse `B`, so wären beide Rollen `RA` und `RB` für das Rollenspiel geeignet. Durch Smart Lifting ist nun sichergestellt, dass die jeweils speziellste Rolle, in diesem Fall `RB`, zum Einsatz kommt.

Vererbung zwischen Teams

Auch für die Teamklassen gilt die übliche Vererbungssemantik. Bedingt durch die Schachtelung von Rollenklassen in Teamklassen besteht nun aber auch die Notwendigkeit, Rollenklassen im Verbund mit dem umschließenden Team zu verfeinern. Daraus ergibt sich eine zusätzliche Form der Vererbung für Rollenklassen: die implizite Rollenvererbung. Voraussetzung hierfür ist eine *extends*-Beziehung zwischen zwei Teamklassen. Das erbende Team erbt dabei implizit alle Rollenklassen des Oberteams und kann diese nun über die Angabe von konformen Rollenklassen gleichen Namens überschreiben. Die geerbten Rollen bzw. Rollenmerkmale werden in das erbende Team importiert (*Copy Inheritance*), gleiche Merkmale werden über Namensgleichheit identifiziert. Analog zur normalen Vererbung darf auch bei der Verfeinerung durch Copy Inheritance eine ungebundene Rolle gebunden werden. Das folgende Team

```
public team class T1 extends T0 {

  public class R playedBy Object {}

}
```

definiert eine Vererbungsbeziehung zu `T0` und erbt implizit dessen Rollenklassen. Die Rollenklasse `R` wird dabei redefiniert und spezifiziert zusätzlich eine Basisbeziehung. Diese muss konform sein, d.h. kovariant zu den Basisbeziehungen erbender Rollen.

Rollen in OT sind somit eine Art von virtuellen Klassen [82]. Die Rollen eines Teams stellen eine Familie von Klassen dar, die eine kovariante Spezialisierung erlauben. Im Kontext eines Teams werden dabei stets konsistent die Rollenobjekte der zugehörigen Klassenfamilie erzeugt. Dieses Konzept wird in der Literatur als *Familiy Polymorphism* bezeichnet [33]. Ein Beispiel ist in Abbildung 2.1 skizziert. Das Team `BoardGame` definiert allgemein die Struktur eines Brettspiels. Dazu gehören Rollen für die Spieler, das Spielbrett und die Spielfiguren. Erbende Teams für konkrete Brettspiele wie Schach oder Go können nun diese Rollen überschreiben, um eine konkrete Realisierung für die spezifischen Rollen, also z.B. eine Schachfigur, zu geben. Die Instanziierung der Rollen erfolgt konsistent anhand des dynamischen Typs der Teaminstanz, es kann also niemals ein Gospieler an einem Schachbrett sitzen.

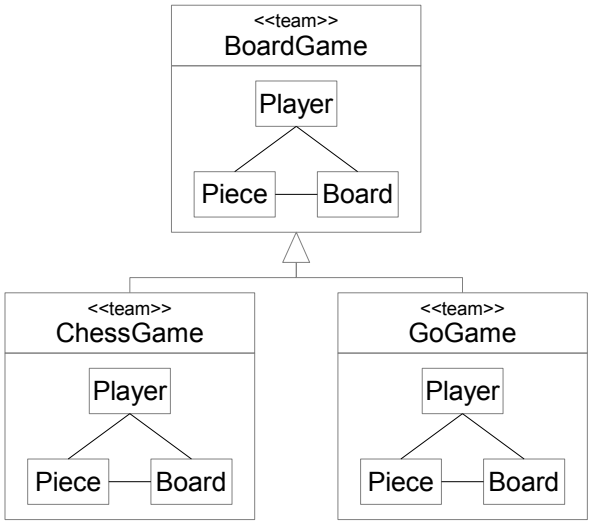

Abbildung 2.1: Beispiel für *Family Polymorphism* in OT

Die klassische Subtyp-Beziehung (im Sinne der Konformität, in Java `instanceof`) gilt nach wie vor nur für die Vererbung mit *extends*. Zwischen implizit überschriebenen Rollen besteht diese Beziehung nicht. Sie ist dort allerdings im Normalfall nicht erforderlich, da die Rollentypen durch *Smart Lifting* automatisch für den aktuellen Kontext aufgelöst werden. Konsequenz des Familiy Polymorphism ist auch, dass der spezifische Typ der Rolle erst in Abhängigkeit vom Kontext vollständig definiert ist. Außerhalb eines Teams darf der Typ der Rolle daher nur mit einer Teaminstanz, in der dieser Typ „verankert“ ist, verwendet werden. Man spricht von einer *externalisierten Rolle*; bei deren Typ handelt es sich aufgrund der Abhängigkeit zur Teaminstanz um einen *Dependent Type*. Die Syntax dafür sieht folgendermaßen aus:

```
final ChessGame chess = ...; // Team
Player<@chess> chessplayer = ...; // ext. Rolle
```

Auf diese Weise kann auch außerhalb des spezifischen Kontexts die typsichere Verwendung der Rollenobjekte gewährleistet und das Vermischen verschiedener Familien verhindert werden [54, 58].

Soll der Rollentyp außerhalb des Teams auch in generalisierter Form zu Verfügung stehen, dann kann dies über die Implementierung eines allgemeinen Interfaces realisiert werden. Diese Vorgehensweise ist in OT eine übliche Alternative zu externalisierten Rollen, Details der Rollenimplementierung bleiben so im Team gekapselt.

Teilen Rolle und Basis dasselbe Interface, so spricht man von einer *transparenten Rolle*. Rollen- und Basisobjekt sind so vollständig substituierbar. Ein Beispiel dafür ist in Listing 2.1 skizziert.

```
1  public interface I {...}
2
3  public class B implements I {...}
4
5  public team class T {
6    protected class R implements I playedBy B {...}
7
8    public I lift(B as R obj) {return obj;}
9  }
10
11 // client code
12 I obj = new B();
13 obj = new T().lift(obj);
```

Listing 2.1: Beispiel einer transparenten Rolle

Die Rollenklasse `R` und die Basisklasse `B` implementieren beide dasselbe Interface `I`. Sowohl auf das Rollen- wie auf das Basisobjekt kann allgemein über den Typ `I` zugegriffen werden. Eine Abhängigkeit zum Team existiert auch für das Rollenobjekt nicht, da die Rolle selbst vollständig im Team gekapselt bleibt.

Die Möglichkeit, Rollen zu überschreiben und dabei auch deren Bindung zu verfeinern, ist eine Grundvoraussetzung für die Wiederverwendbarkeit von Rollen. Allgemeine Anteile einer Rolle können in einem Oberteam definiert werden, dabei darf die Implementierung auch abstrakt bleiben. Spezifische Anteile können dann von einem erbenden Subteam durch Überschreiben der Rolle individuell ergänzt oder verfeinert werden.

Zusammengefasst unterstützt OT drei unterschiedliche Arten von Vererbung: die normale Vererbung mit `extends`, die implizite Rollenvererbung und die Rolle-Basis-Beziehung. Im Unterschied zur normalen Vererbung ist die Rolle-Basis-Beziehung in OT dynamischer und weniger restriktiv gestaltet. Die Konzepte Forwarding und Overriding existieren unabhängig voneinander. Beim Forwarding durch einen Callout ändert sich die Self-Referenz vom Rollen- zum Basisobjekt; bei der Method Execution Interception durch einen Callin in umgekehrter Richtung. Beide Konzepte im Verbund realisieren so echte Delegation zwischen Rolle und Basis, wie sie auch bei einer Vererbungsbeziehung existiert. Die Dynamik von Rollen und Teams erlaubt dabei, dass Basisklassen zur Laufzeit nur temporär oder nur unter bestimmten, dynamisch auswertbaren Bedingungen verfeinert werden.

Die rollenorientierte Programmierung in OT wird meist im Kontext der aspektorientierten Programmierung diskutiert, da sie verschiedene Mechanismen, vor allem die *Implicit Invocation*, gemeinsam haben.

2.2 Aspektorientierte Programmierung

Die aspektorientierte Programmierung (AOP) hat die Modularisierung von *Crosscutting Concerns* („querschneidenden Anforderungen“)[5] zum Ziel [69]. Eine Anforderung wird dann als Crosscutting Concern bezeichnet, wenn sie mit herkömmlichen

[5]Wir haben bei diesem Fachbegriff – wie bei vielen anderen – auf die Verwendung einer deutschen Übersetzung verzichtet, da der englische Ausdruck geläufig und verständlicher ist.

Modularisierungskonzepten nicht separat in einem Modul gekapselt werden kann, sondern über mehrere Module verteilt (*Code Scattering*) oder mit anderen Anforderungen in einem Modul vermischt ist (*Code Tangling*). Eine solche Anforderung wird auch *Aspekt* genannt. Oft handelt es sich hierbei um nichtfunktionale Anforderungen. Ein Beispiel eines Crosscutting Concerns ist der Einsatz von Logging, bei dem viele Logging-Anweisungen im Code verstreut sind. Die Idee der AOP ist, ein neuartiges Modul anzubieten, das einen Aspekt zentral erfassen kann. Die grundsätzliche Idee ist dabei unabhängig vom zugrundeliegenden Programmierparadigma, wird aber vor allem im Rahmen der OOP angewandt und entsprechend auch hier diskutiert. Die vorherrschende aspektorientierte Programmiersprache ist AspectJ [68], die Java um Aspekte erweitert.

2.2.1 Modulkonzept

Ein Aspekt ist ein eigenständiges Modul, das einen Crosscutting Concern implementiert. Da der Aspekt naturgemäß an mehreren Stellen im Programm Einfluss nimmt, diese Einflussnahme aber zentral implementiert werden soll, ist eine Umkehrung der Aufrufrichtung erforderlich. Der Aspekt muss selbst definieren, an welchen Stellen des Programms er ausgeführt werden soll; er ruft sich sozusagen selbst auf. Kern der AOP ist es, Programmanweisungen der folgenden Form zu ermöglichen: *„In einem Programm P, wenn immer Bedingung C erfüllt ist, führe Aktion A aus“* [35]. Eine aspektorientierte Programmiersprache muss daher drei Konzepte anbieten: (1) die Möglichkeit, bestimmte Punkte der Programmausführung deklarativ durch Bedingungen C zu definieren; (2) eine Schnittstelle für Anweisungen bereitstellen, die an diesen Punkten als Aktion A ausgeführt werden können; und (3) ein Mechanismus, der dafür sorgt, dass die Aktion A an den durch C beschriebenen Punkten im Programm tatsächlich ausgeführt wird.

Programmpunkte, an denen ein Aspekt aktiv werden kann, werden *Joinpoints* genannt. Eine Menge von Joinpoints wird als *Pointcut* bezeichnet. Pointcuts werden durch eine gesonderte Beschreibungssprache definiert. Die Aktion des Aspekts wird auch *Advice* genannt. Der Vorgang, Advice-Code an den durch Joinpoints definierten Stellen im Programm zu integrieren, wird als *Weben* bezeichnet und vom Compiler oder der Laufzeitumgebung umgesetzt.

Ein einfaches Beispiel für einen Aspekt in AspectJ ist in Listing 2.2 dargestellt. Der Aspekt `ShapeChangeLogger` soll nach jeder Ausführung einer Setter-Methode der Klasse `Shape` oder einer ihrer Subklassen eine Nachricht ausgeben. Er besteht aus einem Pointcut `allsetters` (Zz. 2–3) und einem Advice (Zz. 5–7).

```
1 aspect ShapeChangeLogger {
2   pointcut allsetters():
3     execution(void Shape+.set*(..));
4
5   after(): allsetters() {
6     System.out.println("Execute: " + thisJoinPoint);
7   }
8 }
```

Listing 2.2: Ein Aspekt in AspectJ

Joinpoints und Pointcuts

Die Möglichkeiten von Aspekten werden maßgeblich durch die Mächtigkeit des Joinpoint-Modells bestimmt. Das Modell definiert die Menge der Punkte, an denen Aspekt-Code eingewoben werden kann. AspectJ bietet ein umfangreiches Modell,

das es erlaubt, Joinpoints nach der Art, dem Geltungsbereich und dem Kontext der Anweisung zu spezifizieren. Dazu können sowohl statische als auch dynamische Informationen herangezogen werden. Zur Selektion der Joinpoints dient eine eigene Pointcutsprache. Wesentlicher Bestandteil dieser Sprache sind die Pointcut-Designatoren, welche die verschiedenen Eigenschaften der Joinpoints abbilden. Beschreiben lässt sich so z.B. der Aufruf oder die Ausführung einer Methode (Pointcut-Designatoren `call` bzw. `execution`), sowie der Code innerhalb einer Klasse (`within`) oder innerhalb eines Kontrollflusses (`cflow`).

Ein weiterer wichtiger Bestandteil der Pointcutsprache sind *Platzhalter* (*Wildcards*). Platzhalter ermöglichen die Beschreibung von Mustern anstelle konkreter Bezeichner, Parameter oder Typen. Für lexikalische Muster, z.B. bei Bezeichnern von Methoden oder Typen, wird der Platzhalter „`*`“ verwendet. Er repräsentiert beliebige Zeichenfolgen (unter Ausnahme des Punkts). Auf Typen kann der Platzhalter „`+`“ angewendet werden. Der jeweilige Typ repräsentiert damit sich selbst sowie alle seine Subtypen. Der Platzhalter „`..`“ repräsentiert eine beliebige Anzahl von Methodenparametern.

Der folgende Code (aus Listing 2.2 (Zz. 2–3)) zeigt das Beispiel eines Pointcuts unter Verwendung von Wildcards:

```
pointcut allsetters(): execution(void Shape+.set*(..))
```

Der Pointcut `allsetters` selektiert `execution`-Joinpoints von Methoden der Klasse `Shape` und ihrer Subklassen, deren Rückgabetyp `void` ist und deren Bezeichner mit der Buchstabenfolge *„set“* beginnen (Anzahl und Typen der Methodenparameter sind beliebig). Das Muster soll alle Setter-Methoden der Klassen beschreiben.

Advice

Die eigentliche Aspekt-Funktionalität wird durch einen oder mehrere Advices definiert. Ein Advice verbindet einen Pointcut mit einem Anweisungsblock. Der Advice ähnelt im Grunde einer Methode. Der Anweisungsblock entspricht dabei dem Methodenrumpf. Im Gegensatz zu einer Methode wird der Advice allerdings nicht explizit im Programm aufgerufen, stattdessen definiert er selbst durch Angabe eines Pointcuts, an welchen Joinpoints er ausgeführt werden soll. Der Advice benötigt daher keinen Bezeichner, stattdessen wird über die Angabe vorgegebener Schlüsselwörter spezifiziert, wo genau der Advice-Code eingewoben werden soll. Dies kann u.a. vor (Schlüsselwort `before`), nach (`after`) oder anstelle (`around`) des Joinpoints geschehen. Analog zu Methodenparametern kann der Advice eine Liste von Parametern definieren, die mittels spezieller Anweisungen des Pointcuts durch Werte des aktuellen Programmkontexts belegt werden, z.B. mit der aktuellen Belegung der Self-Referenz `this`. Diese Parameter stehen dann auch im Rumpf zur Verfügung. Darüber hinaus bietet der Advice zusätzliche Arten von Self-Referenzen, z.B. `thisJoinPoint`, die den jeweiligen Joinpoint repräsentieren.

Der folgende Code (aus Listing 2.2 (Zz. 5–7)) zeigt einen After-Advice:

```
after(): allsetters() {
  System.out.println("Execute: " + thisJoinPoint);
}
```

Der Advice definiert, dass nach allen, durch den Pointcut `allsetters` beschriebenen Joinpoints eine Nachricht, die eine Textrepräsentation des jeweils aktuellen Joinpoints enthält, auf den Output-Stream geschrieben wird.

Inter-type Declarations

Neben der Beeinflussung der Kontrollflüsse ermöglichen Aspekte auch, die Struktur von Klassen zu verändern, indem neue Felder, Methoden oder Konstruktoren hinzugefügt werden oder auch die Vererbungsbeziehung angepasst wird. Man spricht von *Inter-type Declarations*. Beispielhaft ist in Listing 2.3 ein Aspekt dargestellt, der die Klasse `Shape` um ein boolesches Feld `changed` ergänzt (Z. 2). Ein Advice sorgt dafür, dass der Wert des Feldes nach jeder Ausführung einer Setter-Methode auf `true` gesetzt wird (Zz. 4–6).

```
1 public aspect ShapeChangedAspect {
2   public boolean Shape.changed;
3
4   after(Shape s): execution(void Shape+.set*(..)) {
5     s.changed = true;
6   }
7 }
```

Listing 2.3: Inter-type Declarations in AspectJ

Durch Inter-type Declarations ergeben sich Möglichkeiten (und Probleme), wie sie in ähnlicher Form von Mixins[6] oder partiellen Klassen[7] geboten werden, mit der Besonderheit, dass sie mit Quantifizierung kombiniert werden können. Ein typischer Anwendungsfall ist z.B. die nachträgliche Ergänzung eines Interfaces zu einer Klasse. Inter-type Declarations unterscheiden sich in Bezug auf Variabilität und Wiederverwendbarkeit deutlich von Rollen. Inter-type Declarations eines abstrakten Aspekts können im Gegensatz zu Rollen nicht (gleichzeitig) von mehreren konkreten Aspekten mit alternativen Implementierungen wiederverwendet werden. Hier gilt auch nicht die Polymorphie virtueller Klassen. Zudem kann ein Objekt nicht mehrere kontextabhängige Varianten derselben Eigenschaft besitzen, wie es bei mehreren Rollenobjekten der Fall ist.

2.2.2 Aspekt-Basis-Beziehung

Die Beziehung zwischen Aspekt und Basisprogramm ist gekennzeichnet durch zwei Eigenschaften, die den Kern der AOP ausmachen [35]: *Quantifizierung* und *Obliviousness*.

Quantifizierung

Von Quantifizierung (oder Quantification) im Kontext der Logik spricht man, wenn eine freie Variable über Angabe eines Ausdrucks an mehrere Werte gebunden wird, die dann jeweils einzeln auf eine Formel angewendet werden. Der quantifizierende Ausdruck ist dabei ein Prädikat oder eine Funktion, welche die Menge der gültigen Belegungen für die Variable definiert. Im Bereich der AOP sind Joinpoints des Basisprogramms die Elemente, über die quantifiziert wird. Pointcut-Designatoren bilden die Quantoren, da sie jeweils ein Muster von Joinpoints beschreiben. Der Advice ist der vollständige, quantifizierende Ausdruck. Der Pointcut des Advices formuliert ein Prädikat. Auf jeden Joinpoint, der das Prädikat erfüllt, wird der Advice angewendet und somit in das Basisprogramm eingewoben.

[6] Ein *Mixin* ist ein Klassenfragment, das mit anderen Klassen kombiniert werden kann, um neue Subklassen zu erzeugen. Mixins können als spezielle Form der Mehrfachvererbung interpretiert werden [15].

[7] Von *partiellen Klassen* (u.a. realisiert in C# und Objective-C) spricht man, wenn es erlaubt ist, eine Klassendefinition in mehrere Teile aufzuteilen

Quantifizierung ist das grundlegende Mittel, um Crosscutting Concerns modularisieren zu können. Quantifizierung ermöglicht es, die Menge der Joinpoints, die von einem Crosscutting Concern betroffen sind, durch einen einzigen Ausdruck zu beschreiben. In Verbindung mit der Umkehrung der Aufrufrichtung ist so die zentrale Kapselung des Crosscutting Concerns in einem eigenständigen Modul möglich. Der entscheidende Vorteil der Quantifizierung ist die Möglichkeit, eine Menge von Joinpoints durch qualifizierende Eigenschaften deklarativ in einem Pointcut zu beschreiben. Der Pointcut definiert somit eine Vielzahl von Aufrufen durch eine einzelne Anweisung. Ohne Quantifizierung wäre eine explizite Aufzählung jedes einzelnen Joinpoints erforderlich. In diesem Fall bestünde abgesehen von der Frage, in welchem Quellcode-Modul die Advice-Anweisungen definiert werden, kein Unterschied mehr zur Programmierung ohne Aspekte.

Obliviousness

Obliviousness bezeichnet im Kontext der AOP die Eigenschaft, dass das Basisprogramm sich der Adaptierung durch Aspekte nicht „bewusst“ ist. Das bedeutet, es gibt keinerlei direkte Kopplung von einem Basismodul zu einem Aspekt. Der Aspekt definiert selbst, wo er auf das Basisprogramm Einfluss nehmen möchte. Folglich sollte das Basisprogramm ohne den Einsatz von Aspekten ein quasi-vollständiges Programm sein, indem alle Anforderungen mit Ausnahme der Crosscutting Concerns erfüllt sind.

Striktere Auslegungen von Obliviousness schließen auch den Programmierer des Basisprogramms mit ein und fordern, dass dieser keine konkrete Kenntnis von einer Adaptierung durch Aspekte hat. In der Anwendung ist das Konzept der Obliviousness, insbesondere bei strikter Auslegung, umstritten [19]. Zwar können Aspekte prinzipiell völlig unabhängig vom Basisprogramm entwickelt werden, i.A. eröffnet die gemeinsame Entwicklung von Basisprogramm und Aspekten aber bessere Möglichkeiten für ein erfolgreiches Zusammenspiel der Module. Tatsächlich kann es in der Praxis erforderlich sein, ein Basisprogramm gezielt anzupassen, damit Aspekte dort korrekt eingesetzt werden können [46]. Es existiert kein allgemeiner Konsens darüber, inwieweit Obliviousness eine grundlegende Voraussetzung für die AOP darstellt. Einige Ansätze beurteilen Obliviousness sogar als nachteilig und verlangen explizite Schnittstellen zwischen Aspekten und dem Basisprogramm [2, 45, 99].

Grundlegende Voraussetzung für die Unabhängigkeit eines Aspekts ist wie bei der Quantifizierung die Umkehrung der Aufrufrichtung. Das Weben von Aspekt-Code führt zu Kontrollflüssen, die bei Betrachtung des Basisprogramms nicht zu ersehen sind. Einerseits fördert dies die Entkopplung von Basisprogramm und Aspekt, andererseits kann es dadurch anspruchsvoller werden, das Programmverhalten nachzuvollziehen. Verdeckte Kontrollflüsse sind bereits durch das dynamische Binden von überschriebenen Methoden bekannt, werden aber durch Aspekte deutlich ausgeprägter, da sie nun an zahlreicheren (und weniger offensichtlichen) Stellen des Basisprogramms auftreten können.

2.2.3 Abgrenzung zur Rollenorientierung

Die vorgestellten Ansätze zur ROP und AOP besitzen als wesentliche Gemeinsamkeit, dass sie Modularisierung in Gestalt neuartiger Module unterstützen, die durch Implicit Invocation mit anderen Modulen kombiniert werden. Beide definieren Joinpoints als implizite Ereignisse bzw. *Hooks* (nach dem Prinzip der *Template-* und *Hook*-Methoden [39]), um Einfluss auf den Kontrollfluss zu nehmen. Rollen können mittels *Method Call Interception* die Methodenausführung an einem Basisobjekt abfangen und an ein Rollenobjekt delegieren, was dem Überschreiben einer Methode gleichkommt. Für Aspekte bieten Pointcuts feingranularere Möglichkeiten, um Join-

points im Basismodul zu definieren, an denen in der Folge Aspektcode eingewebt wird.

Trotz der ähnlichen Technik unterscheiden sich die Ansätze in mehreren Punkten, von denen wir zwei wesentliche hervorheben möchten.

Module zur direkten Abbildung der Wirklichkeit

Rollen repräsentieren ein natürliches Konzept, das mit einer konkreten Vorstellung in der Begriffswelt verbunden ist. Das Konzept der Rolle wurde daher natürlicherweise in den früheren Phasen der Softwareentwicklung, wie der Modellierung, verwendet und hat von dort aus Einzug in die technischeren Bereiche – die Programmierung – gefunden. Entsprechend existieren (vergleichsweise) klare Vorstellungen über die charakteristischen Eigenschaften von Rollen und deren Abbildung in den verschiedenen Ebenen der Software [97].

Anders verhält es sich dagegen mit Aspekten. Aspekte sind eine primär technisch motivierte Entwicklung, um die Modularisierung von Crosscutting Concerns zu ermöglichen. Die Natur eines Aspekts ist somit darüber definiert, *wozu* er dient, nicht *was* er ist. Obwohl Crosscutting Concerns unmittelbar unserer Vorstellungswelt entstammen, liefert diese indirekte Definition doch keine Antwort auf die Frage, was ein Aspekt oberhalb der technischen Ebene eigentlich repräsentiert[8]. Entsprechend schwierig gestalten sich die Bestrebungen, Aspekte in anderen Phasen der Softwareentwicklung, z.B. der Modellierung, geeignet abzubilden [109]. Der Mangel an intrinsischer Bedeutung von Aspekten wird grundsätzlich kritisiert [56, 98]. Die Praxis bestätigt dies: der überwiegende Teil der Crosscutting Concerns, die mit AspectJ modularisiert werden, besteht in Kollaborationen zwischen Rollen [5]. Aspekte sind zwar technisch vielseitiger, das Gros der Anwendungen wird durch Kollaborationen aber prägnanter beschrieben.

Flexibilität und Kapselungsbruch

Kapselung und Information Hiding gelten als essentielle Mechanismen der Modularisierung. In der OOP führt dies zu einer klaren Trennung von der Schnittstelle (dem *Was*) und der Implementierung (dem *Wie*). Trotzdem kann durch Vererbung ein Kapselungsbruch entstehen, da nicht alle relevanten Details über die Schnittstelle transportiert werden, sondern implizit durch die Implementierung gegeben sind (*Fragile Base Class Problem* [86]). Dies bietet Anlass zu berechtigter Kritik, da die Verträge eines Programms auf Ebene der Schnittstelle definiert werden und entsprechend vom Kapselungsbruch betroffen sein können. Dennoch existieren Regeln und Richtlinien, unter welchen Umständen ein solcher Kapselungsbruch möglich und sinnvoll ist, die in der Praxis ein ausgewogenes Maß zwischen Flexibilität und Verständlichkeit gewährleisten.

Die nicht antizipierte Adaption eines Moduls durch Implicit Invocation, wie sie Rollen und Aspekte durchführen, erfordert eine Flexibilität, für welche die üblichen Kapselungsformen oftmals zu strikt sind. Sowohl die Motivation als auch die technische Umsetzung von Rollen und Aspekten führen dabei zu einer deutlich unterscheidbaren Auslegung des erforderlichen Kapselungsbruchs.

Rollen sind im Vergleich zu Aspekten restriktiver in ihren Möglichkeiten zur Einflussnahme, da für sie eine geringere Menge an Joinpoints[9] zur Verfügung steht. Der Mechanismus des Rollenspiels, insbesondere des kontextabhängigen Überschreibens,

[8] Tatsächlich ist es uns außerordentlich schwer gefallen, die Natur der AOP in diesem Kapitel hinreichend zu charakterisieren. Wir haben uns an AspectJ als dem vermeintlich repräsentativsten Vertreter orientiert.

[9] Das Äquivalent zu *Joinpoints* in der ROP sind die Callin-Bindungen zwischen Rolle und Basis.

ähnelt dem Mechanismus der Vererbung. Der Zugriff auf das Basismodul kann dabei zwar allgemeine Zugriffsbeschränkungen ignorieren (z.B. `private`), richtet sich ansonsten aber nach der bestehenden Schnittstelle und nutzt nur die üblichen Zugriffsmöglichkeiten auf Klassen, Methoden und Attribute [57]. Es handelt sich daher lediglich um eine Flexibilisierung der Schnittstelle, nicht aber um eine Änderung derselben.

Aspekte besitzen aufgrund des umfangreichen Joinpoint-Modells breitere Möglichkeiten zur Adaption. Joinpoints sind dabei unabhängig von der eigentlichen Schnittstelle definiert und bieten potentiell Zugriffsmöglichkeiten, die nicht in der Schnittstelle vorgesehen sind (darüber hinaus ist evtl. sogar eine Änderung der Schnittstelle möglich). Eine Adaptierung auf dieser Ebene erfordert eine Kenntnis der Implementierungsdetails, was einen erheblichen Kapselungsbruch darstellt und grundsätzliche Kritik an der AOP hervorruft [21]. Das Ausmaß der Komplexität, welche die feingranulare Joinpoint-Selektion mit Pointcuts mit sich bringt, zeigt sich u.a. darin, dass die statische Typprüfung in AspectJ Bindungen zwischen Aspekten und Basisklassen zulässt, die Typfehler zur Laufzeit zur Folge haben [24, 108]. Hier kann zwar entgegnet werden, dass auch diese Form des Kapselungsbruchs beherrschbar ist, jedoch sind dazu unzweifelhaft ein umfangreicheres Verständnis und mehr Verantwortung vom Programmierer gefordert, als bei der Restriktion durch eine dedizierte Schnittstelle.

2.3 Logische Programmierung

Die logische Programmierung (LP) ist ein Programmierparadigma, das auf mathematischer Logik basiert. Die Programmierung ist deklarativ: im Unterschied zur imperativen Programmierung besteht ein Programm hier nicht aus einer Sequenz von Anweisungen sondern aus einer Sammlung von Fakten. Der Nutzer kann Anfragen zu den Fakten stellen, der Interpreter sucht nach gültigen Lösungen. Eine der bedeutendsten logischen Programmiersprachen ist Prolog [20]. Wir werden Prolog im Rahmen der Metaprogrammierung als Abfragesprache nutzen, um Informationen über die statische Struktur eines Programms zu gewinnen. An dieser Stelle wird zunächst die grundlegende Funktionsweise von Prolog erläutert, da sie sich fundamental von der OOP unterscheidet.

2.3.1 Fakten und Abfragen

In Prolog werden Aussagen über Objekte und Relationen zwischen Objekten gemacht. Ein Objekt ist dabei nicht vergleichbar mit dem Objekt-Begriff der OOP. Objekte in Prolog haben weder Attribute noch sind sie Instanzen einer Klasse; die einzige Gemeinsamkeit ist, dass ein Objekt eine eindeutige Identität besitzt.

Die universelle Datenstruktur in Prolog, mit der alle Programme konstruiert werden, ist der *Term*. Objekte werden durch konstante Terme repräsentiert. Ein Term kann weiterhin auch durch eine Variable oder eine Verknüpfung von Termen gebildet werden. Ein Prolog-Programm besteht aus einer Menge von Termen, die entweder Fakten oder Regeln repräsentieren, auch *Faktenbasis* genannt. Fakten werden in der folgenden Form notiert:

```
father(bob, john).
```

Der Fakt besteht aus einer Relation *father* zwischen zwei Objekten *bob* und *john*. Er wird durch einen Punkt abgeschlossen. Die Namen von Relationen und Objekten dienen als eindeutige Bezeichner; für sie gilt, dass sie mit einem Kleinbuchstaben beginnen müssen. Der Kopf der Relation wird *Prädikat* genannt, die Objekte sind die *Argumente*. Zum Prädikat gehört auch die Stelligkeit der Argumente, d.h. gleiche

Prädikatnamen mit unterschiedlicher Anzahl von Argumenten bilden unterschiedliche Prädikate. Der aufgeführte Fakt gehört also zum zweistelligen Prädikat `father`.

Die Anwendung eines Prolog-Programms durch einen Nutzer besteht in der Formulierung von Anfragen, die vom Prolog-Interpreter auf Basis der vorhandenen Terme ausgewertet und beantwortet werden. Im Ergebnis erhält der Nutzer Auskunft darüber, ob eine Anfrage wahr ist bzw. welche Fakten die Anfrage erfüllen. Dies sind entweder direkt vorhandene Fakten oder solche, die sich aus den Regeln inferieren lassen. Prolog ermöglicht die Abfrage der Fakten ähnlich wie bei einer Datenbank. Dazu werden Aussagen in Form von Termen notiert, für die der Prolog-Interpreter entscheidet, ob sie wahr oder falsch sind. Solche Abfragen werden mit dem Präfix „`?-`“ dargestellt. Die Abfrage

```
?- father(bob, john).
```

liefert das Ergebnis *„true“*, was bedeutet, dass die Aussage im Rahmen der Faktenbasis wahr ist. Können keine Fakten gefunden werden, die den Term bestätigen, so ist das Ergebnis *„false“*. Dies gilt implizit auch für unbekannte Fakten: solange nicht bewiesen werden kann, dass eine Aussage wahr ist, wird angenommen, dass sie falsch ist (dieser Grundsatz wird als *Closed World Assumption* bezeichnet).

2.3.2 Variablen und Regeln

Prolog ist nicht nur in der Lage zu entscheiden, ob eine Aussage wahr oder falsch ist, es kann auch jene Objekte identifizieren, für die ein Prädikat wahr ist. Anstelle von konkreten Objekten werden dazu Variablen als Argumente verwendet. Die Bezeichner von Variablen müssen mit einem Großbuchstaben beginnen. Die Abfrage

```
?- father(bob, X).
```

liefert als erstes Ergebnis *„X = john“*, was bedeutet, dass die Aussage mit der Belegung `john` für die Variable `X` wahr ist. Sollten weitere Belegungen die Aussage erfüllen, so können sie der Reihe nach ausgegeben werden, bis keine erfüllende Belegung mehr in der Faktenbasis gefunden wird. Prolog durchsucht die Fakten dabei in der Reihenfolge ihrer Definition.

Mit der Hilfe von Variablen ist es auch möglich Regeln zu formulieren. Regeln sind allgemeingültige Aussagen über Objekte und Relationen. Sie dienen dazu, ableitbare Fakten zu definieren, ohne diese redundant aufführen zu müssen. Eine Regel besteht aus einem Kopf und einem Körper, verbunden durch das Symbol „`:-`“ (genannt *if*), das eine umgekehrte Implikation darstellt. Der Kopf der Regel definiert einen Fakt, der dann wahr ist, wenn der Körper der Regel wahr ist und die Belegung der Variablen auf beiden Seiten gleich ist. Die Regel

```
child(X, Y) :- father(Y, X).
```

definiert, dass für jede *father*-Relation eine umkehrte *child*-Relation besteht. Damit definiert sie indirekt den Fakt

```
child(john, bob).
```

Regeln quantifizieren über freie Variablen und repräsentieren somit eine Menge von Fakten des jeweiligen Prädikats. Fakten und Regeln werden allgemein auch als Klauseln bezeichnet.

Aussagen können über die logischen Operatoren „`,`“ (UND) und „`;`“ (ODER) auch miteinander verknüpft werden. Die Regel

```
grandfather(X, Y) :- father(X, Z), parent(Z, Y).
```

definiert eine *grandfather*-Relation über die Verundung einer *father*- mit einer *parent*-Relation. Der direkte Zusammenhang entsteht durch die Verwendung der freien

Variable Z in beiden Relationen. Veroderte Regeln können auch separat aufgeschrieben werden, wie es üblicherweise getan wird. Die Regel

```
parent(X, Y) :- father(X, Y); mother(X, Y).
```

besteht aus zwei veroderten Prädikaten und kann mit zwei getrennten Regeln äquivalent abgebildet werden:

```
parent(X, Y) :- father(X, Y).
parent(X, Y) :- mother(X, Y).
```

Das Vorgehen zur Belegung der Variablen wird *Unifikation* genannt. Wird eine Anfrage an Prolog gestellt, so durchsucht Prolog die vorhandenen Klauseln, bis alle freien Variablen durch konstante Terme ersetzt sind, die an ihrer Stelle wahr sind. Eine Variable gilt als belegt, wenn ein Objekt gefunden wurde, dass durch die Variable repräsentiert wird. Variablen, deren Belegung nicht von Bedeutung ist, können der Einfachheit halber mit dem Platzhalter „_“ notiert werden. Damit eine Aussage wahr ist, müssen für alle Variablen (auch Platzhalter) Belegungen gefunden werden. Das Ergebnis einer Abfrage ist ein Tupel von Belegungen der Variablen, es wird als *Lösung* bezeichnet. Im Kontext dieser Arbeit interessiert uns dabei immer die Menge aller Lösungen, für die eine Aussage wahr ist. Als Ergebnis einer Abfrage bezeichnen wir daher, sofern nicht anders aufgeführt, nicht eine einzelne Lösung, sondern die Menge aller Lösungen.

2.3.3 Listen

Listen werden in Prolog durch normale Terme repräsentiert, deren Elemente ebenfalls normale Terme sind. Da Listen eine häufig gebrauchte Datenstruktur darstellen, gibt es für sie eine besondere Kurzschreibweise. Eine Liste wird durch eine Sequenz von Elementen (durch Kommata getrennt) innerhalb eckiger Klammern notiert, z.B.

```
[bob, john]
```

Die Schreibweise mit eckigen Klammern ist lediglich eine syntaktische Abkürzung für eine kompliziertere Struktur. Eine nicht-leere Liste besteht aus den zwei Teilen *Head* und *Tail*, wobei Tail selbst eine Liste ist, die direkt über das Symbol „|“ erreicht werden kann. Die Listennotation erlaubt damit auch Pattern Matching. Die Abfrage

```
?- [H | T] = [bob, john].
```

liefert die Lösung $H = bob,\ T = [john]$ als Ergebnis.

Die Arbeit mit Listen ist häufig mit Rekursionen verbunden. Eine Regel, die Auskunft darüber gibt, ob ein Element Bestanteil einer Liste ist, lässt sich rekursiv formulieren:

```
member(X, [H | T]) :- X=H; member(X, T).
```

Die Regel zum zweistelligen Prädikat `member` greift bei ihrer Definition auf das Prädikat selbst zurück, wobei die Liste mit jedem Rekursionsschritt um ein Element verkürzt wird. Der Fall der leeren Liste wird vom Pattern Matching der Argumente nicht abgedeckt und bildet somit implizit das Abbruchkriterium der Rekursion.

Programme einer logischen Programmiersprache wie Prolog bestehen letztlich aus einer Menge von dynamisch quantifizierenden Anweisungen. Insofern halten wir Prolog für besonders geeignet, um Quantifizierung von Programmelementen, wie es etwa der Zweck von Pointcutsprachen ist, zu realisieren. Wir werden dies in Abschnitt 3.2 detailliert diskutieren.

2.4 Wiederverwendung

Wiederverwendung von Software ist die Verwendung bestehender Software, um damit neue Software zu entwickeln. Die Idee der Wiederverwendung manifestierte sich erstmals 1968 in McIlroys Vision von Standard-Komponenten, die *„off-the-shelf“* verwendet werden können – als Antwort auf die Softwarekrise [83]. McIlroy formulierte das grundlegende Problem der Effizienz in der Softwareindustrie: bei der Entwicklung neuer Software werde die Frage „Welche Mechanismen sollen wir bauen?“ gestellt, anstatt der Frage „Welche Mechanismen sollen wir nutzen?“. Aus dieser Überlegung resultiert der Wunsch nach wiederverwendbaren Software-Einheiten, als Basis für die effiziente Entwicklung komplexer Systeme. Forscher und Entwickler haben sich in den vergangenen Jahrzehnten intensiv mit der Thematik auseinandergesetzt und es sind zahlreiche Ansätze zur Wiederverwendung entstanden. In diesem Abschnitt werden wir die Grundlagen der Wiederverwendung vorstellen und darauf aufbauend in Kapitel 3 die Problemstellung und Motivation für unseren konkreten Ansatz zur Wiederverwendung diskutieren.

2.4.1 Prozess

Der Prozess zur Wiederverwendung von Software kann in vier Tätigkeiten untergliedert werden: das Abstrahieren, Selektieren, Spezialisieren und Integrieren von Software-Artefakten [13, 75].

Abstraktion

Abstraktion ist die Konzentration auf das Allgemeingültige. Sie besteht in der Überführung einer komplexen Anschauung (z.B. der Realität) auf eine einfachere Darstellung (z.B. ein Modell), erreicht durch das Auslassen von Details. Sie ermöglicht es, eine Menge von in Einzelheiten unterschiedlich ausgeprägten Individuen über ihre Gemeinsamkeiten repräsentativ zu beschreiben.

Abstraktion bedeutet Reduktion und Generalisierung von Information. Letzteres ist der Schlüssel zur Wiederverwendung. Soll eine bestehende Software-Lösung an anderer Stelle wiederverwendet werden, so müssen die Anteile der Lösung identifiziert werden, die für beide Lösungen gleichermaßen anwendbar sind. Ist es möglich, diese Anteile allgemeingültig zu beschreiben, so existiert eine Abstraktion, die auf mehrere Anwendungsfälle übertragen werden kann.

Die Abstraktion einer Lösung besteht aus einem *invarianten* Teil, der allgemeingültig für mehrere gleichartige Lösungen ist, und einem *varianten* Teil, der individuell für jede Lösung ausgeprägt werden muss. Eine Abstraktion wird – in Folge des Prinzips *Separation of Concerns* – üblicherweise getrennt von den spezifischen Ausprägungen, z.B. in einem eigenständigen Modul, aufgeschrieben. Ein Beispiel aus der OOP ist eine *abstrakte Klasse*, die separat von ihren konkreten Subklassen verfasst wird. Eine modulare Struktur ist insbesondere für die effiziente Wiederverwendung von Code von Bedeutung: Code in Form eines separierten Bausteins lässt sich einfacher in eine Anwendung integrieren und darin warten als Code, der eng mit (vielen) anderen Code-Teilen verbunden ist. Die grundlegende Herausforderung der Wiederverwendung besteht darin, invariante und variante Anteile einer Lösung so voneinander zu trennen, dass die invarianten Teile einen nützlichen Beitrag zur Lösung vieler Probleme leisten können. Dies kann umso besser gelingen, je geringer die Abhängigkeit beider Teile voneinander ist.

Die allgemeingültige Beschreibung der Abstraktion wird als *Spezifikation* bezeichnet, während eine konkrete Ausprägung der Abstraktion mit spezifischen Details als deren *Realisierung* bezeichnet wird [75]. Spezifikation und Realisierung bilden zwei Ebenen der Abstraktion, wobei die Spezifikation die höhere und die

Realisierung die niedrigere Ebene repräsentiert. Die Zuordnung ist relativ, da Software üblicherweise aus mehreren Abstraktionsebenen aufgebaut ist. Der Quellcode eines Programms, das in einer Hochsprache geschrieben ist, wird meist auf eine Assembler-Sprache abgebildet, die wiederum auf Maschinen-Code abgebildet wird. Jede Zwischenebene stellt dabei sowohl die Realisierung der höheren als auch die Spezifikation der niedrigeren Ebene dar. So bildet z.B. ein Modell die Spezifikation für eine Realisierung in Form von Quellcode. Der Quellcode wiederum ist die Spezifikation für eine Realisierung in Gestalt ausführbaren Maschinencodes.

Selektion

Bei der Realisierung einer Anforderung steht der Entwickler stets vor der Frage, ob dazu eine zur Wiederverwendung geeignete Abstraktion existiert. Der Entwickler muss die Menge vorhandener Abstraktionen durchsuchen und in Hinblick auf ihre Anwendbarkeit evaluieren. Auf dem aktuellen Stand der Technik steht üblicherweise eine Fülle von Modulen, Mustern und Konzepten der Wiederverwendung zur Verfügung. Für einen effizienten Selektionsprozess ist daher die Kategorisierung von Abstraktionen, wie z.B. in Musterkatalogen, und deren Organisation in physischen Einheiten, wie z.B. Bibliotheken, erforderlich. Eine strukturierte Verwaltung ermöglicht zudem auch eine bessere Unterstützung des Entwicklers bei der Selektion durch Werkzeuge.

Spezialisierung

Viele Techniken der Wiederverwendung basieren auf der Generalisierung von gleichartigen Lösungen in einer *generischen* Struktur. Der Entwickler nutzt die generische Lösung als Ausgangsbasis für seine spezifische Lösung und gestaltet sie nach seinen Bedürfnissen individuell aus; i.A. spricht man dabei von *Spezialisierung* oder auch *Verfeinerung*. Je nach Technik stehen dem Entwickler unterschiedliche Variationspunkte zur Verfeinerung zur Verfügung, z.B. durch Parametrisierung, Transformation oder Redefinition. Das Interface `java.util.List<E>` beschreibt beispielsweise die Abstraktion von Listenstrukturen und bietet dabei zwei Varianten zur Verfeinerung: (1.) ein Typparameter erlaubt die Liste für beliebige Typen von Elementen anzuwenden und (2.) Vererbung erlaubt die individuelle Redefinition der Funktionalität, so dass die Liste z.B. sowohl auf Basis eines Arrays als auch in Form einer verketteten Liste realisiert werden kann.

Integration

Die grundsätzliche Problematik der Integration besteht in der Komposition der wiederverwendeten mit den spezialisierten Teilen einer Lösung. Techniken wie Dynamisches Binden, Code-Generation oder Transformation sorgen dafür, dass aus dem über mehrere Abstraktionsebenen kombinierten Code ein Programm entsteht, das im Sinne des Entwicklers ausgeführt wird. Die Integration wiederverwendeter Software-Module wird dabei meist durch spezielle Sprachkonstrukte unterstützt, die für die automatisierte Umsetzung in Compilern und Werkzeugen sorgen. Bestenfalls ist ein Import eines Moduls ausreichend, so dass es gleich vom Programmierer als Klient verwendet werden kann. Ist dagegen erst eine Verfeinerung des Moduls erforderlich, so müssen die betroffenen Variationspunkte ausgestaltet werden. Dies wird u.a. über Schlüsselwörter und Kompositionsregeln gesteuert. Beispielsweise legt der Entwickler einer Klasse über einen mit einem Schlüsselwort verbundenen Variationspunkt fest, ob eine Vererbungsbeziehung zu einer anderen Klasse existiert. Eine Kompositionsregel legt fest, dass zur Redefinition einer geerbten Methode eine Methode mit identischem Namen und kompatiblem Parametern verwendet wird.

Eine Wiederverwendungstechnik ist nur dann effizient, wenn der Aufwand zur Abstraktion, Selektion, Spezialisierung und Integration von Lösungen insgesamt geringer ausfällt, als die Realisierung gleichwertiger Lösungen ohne den Einsatz dieser Technik. Dieser Aufwand resultiert aus der *kognitiven Distanz* zwischen einer Abstraktion und einer konkreten Lösung [75]. Die kognitive Distanz beschreibt ein intuitives Maß für die intellektuelle Leistung des Entwicklers, die erforderlich ist, um eine Abstraktion in eine konkrete Lösung zu überführen. Wiederverwendungstechniken sollten den Entwickler im Sinne der Effizienz dahingehend unterstützen, dass die kognitive Distanz auf ein Minimum reduziert wird. Dies gelingt durch eine geringe Anzahl von Variationspunkten, die Kapselung invarianter Anteile und eine automatisierte Abbildung der Spezifikation auf die Realisierung.

2.4.2 Kosten und Nutzen

Eine gesteigerte Produktivität ist die wesentliche Motivation für die Wiederverwendung von Software. Von der Wiederverwendung erwartet man positive Auswirkungen auf Zeit, Kosten und Qualität – die kritischen Erfolgsfaktoren – von Software-Projekten. Die wesentlichen Gründe dafür sind naheliegend:

- **Zeit**. Ist bereits Software verfügbar, die für ein neues System verwendet werden kann, so muss insgesamt weniger Software entwickelt werden. Dies sollte im Normalfall auch bedeuten, dass die Software schneller entwickelt werden kann. Wenn mehrere Systeme gemeinsame Anteile von Software besitzen, dann ist insgesamt der Umfang der Software geringer, was auch eine schnellere Wartung ermöglicht.

- **Kosten**. Aus der reduzierten Zeit für Entwicklung und Wartung ergibt sich gleichermaßen eine Kostenersparnis, sofern es sich bei der wiederverwendeten Software um eine Eigenentwicklung handelt oder die Kosten zur Beschaffung geringer als die einer Eigenentwicklung sind.

- **Qualität**. Software, die zur Wiederverwendung dient, kann qualitativ hochwertiger entwickelt werden, da potentiell mehr Ressourcen als bei einer Einzelentwicklung zur Verfügung stehen – gerechtfertigt durch die Summe der mittels Wiederverwendung gesparten Ressourcen. Wiederverwendbare Software soll zuverlässiger sein, da mehr Aufwand in die Qualitätssicherung fließen kann und Fehler aufgrund des Einsatzes in mehreren Systemen potentiell früher entdeckt werden. Wiederverwendbare Software soll effizienter sein, da eine sorgfältigere Auswahl von Datenstrukturen und Algorithmen durch spezialisierte Entwickler erfolgen kann. Bestenfalls kann die hohe Qualität der wiederverwendeten Software auch Vorbildcharakter für die übrige Software haben, und sich somit auch über die eigenen Grenzen hinaus positiv auf die Qualität auswirken.

Diese Argumente mögen im ersten Moment überzeugend klingen, sie implizieren aber die fragwürdige Prämisse, dass der Aufwand, um wiederverwendbare Software zu entwickeln und diese wiederzuverwenden, vergleichbar ist mit dem Entwicklungsaufwand von Software ohne Wiederverwendung. Die *Wiederverwendbarkeit* von Software gilt als generisches Maß, wie geeignet Software für die Wiederverwendung ist. Dabei geht es einerseits um die Anzahl und Qualität der konkreten Lösungen, die damit realisiert werden können, und andererseits um den Aufwand, der mit der Realisierung einer konkreten Lösung verbunden ist – also letztlich den allgemeinen Einfluss auf die kritischen Erfolgsfaktoren. Tatsächlich existieren zahlreiche Gründe, die nahelegen, dass sowohl die Entwicklung wiederverwendbarer Software als auch die Wiederverwendung dieser Software zusätzliche Anforderungen und

einen Mehraufwand gegenüber einer Entwicklung ohne Wiederverwendung mit sich bringen:

- **Zeit**. Die initiale Erstellung wiederverwendbarer Software ist anspruchsvoller (aufgrund der Abstraktion als zusätzliche Anforderung) und entsprechend langwieriger als die Erstellung spezialisierter Software für einen konkreten Anwendungsfall. Wiederverwendbare Software kann zudem nicht ad-hoc eingesetzt werden. Die Selektion, Spezialisierung und Integration wiederverwendbarer Software benötigt zusätzliche Entwicklungszeit.

- **Kosten**. Die Herstellung oder Beschaffung wiederverwendbarer Software kann mit hohen initialen Kosten verbunden sein. Eine Ersparnis kann oft erst durch den Einsatz in mehreren Projekten realisiert werden (mit einer Anzahl, die eventuell aber nie erreicht wird).

- **Qualität**. Wiederverwendbare Software definiert notwendigerweise einen Standard, der eventuell sogar auf Kompromissen basiert. Eine Standardlösung leistet zwar das Wesentliche, ist aber in Bezug auf eine konkrete Anwendung selten optimal. Einzellösungen, die spezialisiert auf einen bestimmten Anwendungsfall sind, können den Anforderungen in der Regel besser gerecht werden.

Wiederverwendung an sich bringt also nicht zwingend einen Gewinn. Sie wird erst dann produktiv, wenn die Techniken der Wiederverwendung eine Reduktion des Aufwands oder eine Steigerung der Qualität ermöglichen. Tatsächlich zählt daher die Wiederverwendbarkeit mittlerweile zu den bedeutenden Qualitätsmerkmalen von Software [38].

Wie Meyer ausführt ([85], Kapitel 4), steht ein Entwickler grundsätzlich vor der Entscheidung, ob er die Rolle des Konsumenten oder des Produzenten von wiederverwendbarer Software einnehmen möchte. Beide Rollen gleichzeitig einzunehmen ist mit einem Zielkonflikt verbunden. Software, die gleichermaßen Wiederverwendung betreibt und Wiederverwendbarkeit unterstützt, muss zwei gegensätzlichen Zielsetzungen, Spezialisierung und Abstraktion, gerecht werden – was eine erhebliche Herausforderung für den Entwickler darstellt.

2.4.3 Techniken

Es existieren zahlreiche Techniken der Wiederverwendung. Im Kern meint Wiederverwendung dabei stets die Wiederverwendung der invarianten Teile einer Abstraktion. Die Techniken dazu lassen sich grob in zwei Kategorien einteilen, je nachdem ob Spezifikation und Realisierung einer Abstraktion (1) in gleichartiger oder (2) in unterschiedlicher Weise definiert werden. (1) ist der Fall, wenn ein und dieselbe Sprache bzw. Darstellungsform sowohl zum Verfassen der Spezifikation als auch der Realisierung zum Einsatz kommt; im Vordergrund steht hier die Kombination von bestehenden Software-Einheiten. (2) ist der Fall, wenn die Spezifikation in einer Darstellungsform vorliegt, die sich wesentlich von der für die Realisierung verwendeten Darstellungsform unterscheidet; hier steht die Generation von Code auf Basis von sich wiederholenden Mustern im Vordergrund. Es folgt ein (nicht notwendigerweise vollständiger) Überblick zur Einordnung verschiedener Techniken:

Abstraktion mit gleichartiger Spezifikation und Realisierung

- **Kopie**. Die vielleicht einfachste Form der Wiederverwendung von Software ist die Übernahme bestehender Software durch Kopie („copy-paste“). Die Abstraktion entsteht erst zu dem Zeitpunkt, wenn die Software-Einheit als Vorlage für neue Software verwendet wird, wobei potentiell alle Teile der Software

variiert werden können. Die Produktivität dieser Art der Wiederverwendung hängt davon ab, welche Gewichtung die einzelnen Tätigkeiten erfahren. Der Aufwand zur Integration ist offensichtlich denkbar gering. Der Aufwand zur Spezialisierung ist beliebig und variiert, je nachdem wie gut die bestehende Lösung zu den neuen Anforderungen passt. Ein wesentliches Problem besteht in der Selektion, da es auf dieser Ebene keine allgemeine Einheit für die Erfassung und Kategorisierung nutzbarer Software gibt. Ein Entwickler nutzt entweder seinen eigenen Erfahrungshorizont oder er sucht in fremdem Code oder Dokumentationen („Kochbücher“) nach geeigneten Vorlagen. Als eigentlicher Nachteil wird der Aufwand zur Abstraktion gesehen, da der Entwickler letztlich alle Teile der zu übernehmenden Software vollständig verstehen muss und daher keine Vorteile durch die Kapselung invarianter Anteile genießt. Das wiederholte Kopieren von Code wird i.A. als schlechte Praxis bewertet, da es zu Redundanzen führt, welche die Verständlichkeit und Wartbarkeit der Software beeinträchtigen (siehe auch „Rule of three“ in [36]). Insgesamt wird Kopieren als die ungeeignetste Technik der Wiederverwendung angesehen [23].

- **Komponenten**. Die komponentenbasierte Wiederverwendung meint die Wiederverwendung von Software in Form von kombinierbaren Bausteinen. McIllroy nennt in seiner ursprünglichen Vision Routinen als wesentliche Einheit: „The most important characteristic of a software components industry is that it will offer families of routines for any given job.“ [83] Auf dem heutigen Stand der Technik gibt es daneben noch eine Vielzahl weiterer Software-Bausteine, die dafür in Frage kommen. Der Begriff der *Komponente* wird in der Literatur unterschiedlich ausgelegt. Im wörtlichen Sinne kann man dahinter ein Stück Software verstehen, dass zur *Komposition* eines Systems verwendet werden kann [4]. In der OOP steht die komponentenbasierte Form der Wiederverwendung in Form von Modulen im Vordergrund; Meyer formuliert es so: „The appropriate unit of reuse is some form of abstracted module, providing an encapsulation of a certain functionality through a well-defined interface“ ([85], Kapitel 4).

  Abstraktion und Spezialisierung eines Moduls kann bis zu einem gewissen Grad durch Generizität und Parametrisierung erreicht werden. Im einfachen Fall kann die Wiederverwendung eines Moduls daher allein mit dem Aufruf eines Klienten realisiert werden (z.B. bei Funktions-Bibliotheken). In vielen Fällen, gerade bei komplexeren Modulen, ist aber eine individuelle Spezialisierung erforderlich, die wesentliche Änderungen oder Ergänzungen an der Struktur des Moduls mit sich bringt. Der vorherrschende Mechanismus hierzu ist Delegation auf Basis von Vererbung. Liskov diskutiert verschiedene Arten der Vererbung und stellt fest, dass Kapselung einen wesentlichen Mehrwert zur Wiederverwendbarkeit leistet [79]. Laut Coulange bietet OOP die insgesamt beste Technik zur Wiederverwendbarkeit [23].

- **Frameworks**. Unter einem Software-Framework versteht man eine generische Applikation, die über verschiedene Mechanismen erweitert und angepasst werden kann, so dass auf dessen Basis konkrete Applikationen realisiert werden können. Im Unterschied zu kleineren Software-Einheiten gibt das Framework bereits den Hauptteil einer vollständigen Anwendung vor. Das Framework entspricht somit einer maximal vertikal skalierten Komponente und ist spezifisch auf eine Domäne ausgerichtet.

Abstraktion mit verschiedenartiger Spezifikation und Realisierung

- **Compiler**. Die intuitivste Form der Wiederverwendung geschieht durch Programmiersprachen und Compiler. Sie ist so grundlegend, dass sie i.A. nicht als Wiederverwendung aufgefasst wird. Sie genügt jedoch den Prinzipien der Abstraktion mit Spezifikation und Realisierung. Die Sprachdefinition selbst bildet die Spezifikation, indem sie Syntax und Semantik der Sprachbausteine definiert. Die Sprachbausteine wiederum bieten diverse Möglichkeiten zur Variation durch Kombination und Parametrisierung. Ein in der Sprache verfasstes Programm bildet somit eine Realisierung der Spezifikation. Der Compiler generiert daraus ausführbaren Code.

- **Code-Generatoren und Transformatoren**. Das Prinzip der Compiler kann auch auf höhere Sprachebenen übertragen werden. So existieren beispielsweise zahlreiche Modellierungssprachen, aus denen Quellcode in einer Hochsprache generiert werden kann. Die Modellierungssprache bildet die Spezifikation. Die Realisierung wird in Form eines konkreten Modells gestaltet, aus dem dann Quellcode generiert werden kann. Im Unterschied zum Compiler einer normalen Programmiersprache liefert der Code-Generator einer höheren Sprache aber nicht in allen Fällen ein vollständiges Programm, da im Modell nicht unbedingt alle Details ausreichend spezifiziert werden können. In diesem Fall ist eine weitere Ausgestaltung notwendig. Ein klassisches Beispiel für Code-Generatoren sind Parser-Generatoren, die auf Basis einer vorzugebenden Grammatik ein vollständiges Programm generieren können, das in der Lage ist, Quellcode der entsprechenden Grammatik zu analysieren.

  Besonders zu erwähnen sind hier domänenspezifische Sprachen (*Domain-specific languages - DSLs*), die sich zunehmender Popularität erfreuen [37]. DSLs sind Programmiersprachen, die auf bestimmte Anwendungsdomänen spezialisiert sind. Die Sprachmittel sind einerseits optimal auf die Problemfelder einer Domäne zugeschnitten und andererseits begrenzt, so dass andere Problemfelder nicht unterstützt werden. Dies soll Vorteile für die Erlernbarkeit, Lesbarkeit und Prägnanz der Sprache bieten. DSLs bilden somit das Gegenstück zu den klassischen Programmiersprachen mit universellem Einsatzzweck. Ein in einer DSL verfasstes Programm wird üblicherweise von einem Codegenerator in die Darstellung in einer universellen Programmiersprache überführt, wobei das Programm bestenfalls ohne weitere Ausgestaltung vollständig ist und gleich weiterverarbeitet werden kann.

- **Entwurfsmuster**. Entwurfs- und Architekturmuster beschreiben Lösungsansätze für wiederholt auftretende Problemstellungen, deren Realisierungen sich in den strukturellen Details so gravierend unterscheiden, dass eine Wiederverwendung auf der Detailebene i.A. nicht sinnvoll umzusetzen ist [95]. Auf der Entwurfsebene ist aber durchaus eine Wiederverwendung der grundlegenden Ideen und Mechanismen möglich. Der Übertrag muss dabei vom Entwickler selbst ohne die Unterstützung automatisierter Prozesse geleistet werden. Die Beobachtung, dass auf der Detailebene keine sinnvolle Wiederverwendung von Code möglich ist, werden wir in Abschnitt 3.1 diskutieren und Alternativen aufzeigen.

2.4.4 Skalierbarkeit

In der OOP steht die komponentenbasierte Wiederverwendung im Vordergrund. Deren angestrebte Nutzenmaximierung führt in der Praxis zu einem Wachstum der wiederverwendbaren Software-Einheiten, wie bereits einleitend in Kapitel 1 erläutert. Der Gedanke dahinter ist offensichtlich: je größer die Einheit ist, die in einem

Arbeitsprozess wiederverwendet wird, umso günstiger kann das Verhältnis zwischen Aufwand und Ersparnis ausfallen. Der Aufwand wirft allerdings die Frage nach der Skalierbarkeit der Wiederverwendbarkeit von Software-Komponenten auf.

- **Vertikale Skalierung**. Unter der vertikalen Skalierung versteht man das Wachstum der Einheiten. Unter der Annahme, dass der Aufwand zur Abstraktion, Selektion und Integration nicht proportional mit dem Umfang der Einheit wächst, sondern bestenfalls sogar konstant dazu ist, bieten größere Komponenten eine besseres Kosten-Nutzen-Verhältnis bei der Wiederverwendung. Das Wachstum hat allerdings einen entscheidenden Nachteil: große Bausteine passen in weniger Lücken, als kleine Bausteine. Eine größere Komponente ist notwendigerweise spezialisierter für bestimmte Anwendungskontexte, aufgrund der umfangreicheren Funktionalität, die sie mitbringt. Die Zahl der Anwendungen, in denen genau diese Funktionalität benötigt wird, sinkt mit dem Umfang derselben.

- **Horizontale Skalierung**. Unter horizontaler Skalierbarkeit versteht man die Variabilität einer Komponente. Je spezialisierter eine Komponente ist, u.a. als Folge der vertikalen Skalierung, umso geringer ist die Zahl der Anwendungen, in denen die Komponente mit einem Mehrwert eingesetzt werden kann. Um diese Beschränkung abzumildern, müssen Möglichkeiten zur Variation bzw. Verfeinerung der Komponente angeboten werden. Je mehr Variationspunkte existieren, umso besser sollte die Komponente für verschiedene Anwendungsfälle angepasst werden können. Auch hier gibt es einen entscheidenden Nachteil, denn mit der Zahl der Variationspunkte wächst auch der Aufwand der Spezialisierung.

Beide Dimensionen der Skalierung wirken gegensätzlich: die vertikale Skalierung spart Aufwand zur Abstraktion, Selektion und Integration bei der Wiederverwendung, schränkt aber zugleich die Wiederverwendbarkeit durch Spezialisierung ein; die horizontale Skalierung verbessert die Wiederverwendbarkeit durch Adaptivität, verursacht aber wiederum zusätzlichen Aufwand zur Spezialisierung bei der Wiederverwendung. Idealerweise möchte man eine Komponente sowohl vertikal wie auch horizontal skalieren, um eine optimale Wiederverwendbarkeit zu erlangen. Tatsächlich bringt die Kombination beider Dimensionen aber zusätzliche negative Effekte. Mit jeder weiteren Wachstumsstufe kommt Funktionalität hinzu, die gleichermaßen auch neue Variationspunkte erfordert. Dies kann zu einer kombinatorischen Explosion führen. Wird eine Komponente, die N Varianten in einer Dimension erlaubt, um M neue Varianten einer anderen Dimension erweitert, so ergeben sich daraus insgesamt $N \times M$ denkbare Varianten. Dabei kann es durchaus sein, dass auch zusätzliche Variationspunkte in der bestehenden Funktionalität notwendig werden, wenn die Teile abhängig voneinander sind. Die vertikale Skalierung macht die horizontale Skalierung erforderlich, wenn die Zahl der Anwendungsmöglichkeiten beibehalten werden soll; die realisierte Ersparnis der größeren Einheit kann aber leicht vom zusätzlichen Aufwand zur Spezialisierung übertroffen werden. Biggerstaff nennt dies das Dilemma der vertikalen und horizontalen Skalierung [12]: die Skalierung in beiden Dimensionen ist grundsätzlich mit negativen Konsequenzen verbunden.

Biggerstaff führt aus, dass die klassischen Wiederverwendungstechniken, wie z.B. Vererbung, nur einen geringen Produktivitätsgewinn bieten. Eine signifikante Verbesserung sei vor allem durch die Erweiterung dieser Technologien um generative Ansätze zu erreichen, u.a. durch neue Abstraktionstechniken, mit denen sich flexibel anpassbare Software-Komponenten erzeugen lassen. Wir teilen diese Überzeugung und möchten auf dieser Basis unseren erweiterten Ansatz zur Wiederverwendung in OT motivieren.

Kapitel 3

Modularisierung und Wiederverwendung in Object Teams

In diesem Kapitel werden die konkreten Möglichkeiten zur Modularisierung und Wiederverwendung in OT untersucht. Modularisierung ist hier – wie allgemein in der OOP – ein Schlüssel, um wiederverwendbare Software zu schaffen. Der Wiederverwendbarkeit von Modulen sind aber mit den klassischen Techniken, wie in Abschnitt 2.4 ausgeführt, Grenzen gesetzt. Die Defizite werden nachfolgend diskutiert. Im Anschluss wird ein Lösungsansatz entworfen, der eine Erweiterung um generative Programmierung vorschlägt.

3.1 Problemdiskussion

3.1.1 Motivierendes Beispiel

Zum besseren Verständnis und zur Motivation unseres Lösungsansatzes zeigen wir im Folgenden ein konkretes Beispiel-Szenario, das durchgängig in diesem und folgenden Kapiteln zur Diskussion herangezogen wird. Das Szenario behandelt eine klassische Problemstellung aus *Model-View-Controller*-Architekturen: die Synchronisation von Objekten der Modellschicht mit darstellenden Komponenten der Präsentationsschicht. Kern der Problemstellung ist, Änderungen an den Eigenschaften der Objekte automatisch an die Präsentationsschicht zu kommunizieren, damit deren Repräsentation (z.B. die Darstellung in textueller oder grafischer Form) den aktuellen Datenstand korrekt wiedergibt. Diese Aufgabe wird üblicherweise durch Anwendung des Entwurfsmusters *Observer* realisiert [39]. Das Muster beschreibt die Rolle *Subject* (Subjekt) im Kontext eines *Observers* (Beobachter). Observer-Objekte können sich an Subject-Objekten als Beobachter registrieren; Subject-Objekte sind dann dafür verantwortlich, registrierte Beobachter über relevante Änderungen ihres Zustands zu informieren.

Als konkretes Beispiel beschreiben wir eine Anwendung zur Darstellung von geometrischen Formen, wie z.B. Kreisen und Rechtecken. Denkbare Darstellungsarten sind eine grafische Zeichnung oder eine textuelle Beschreibung eines Form-Objekts. Wann immer sich der Zustand eines solchen Objekts ändert, müssen zugehörige Repräsentationsobjekte über die Änderung informiert werden, damit sie daraufhin ihre Darstellung aktualisieren. Eine klassische Realisierung des Szenarios ist als Klassendiagramms in Abbildung 3.1 dargestellt. Die konkreten Klassen für Subjekte sind `Circle` und `Rectangle`. Sie erben von einer abstrakten Oberklasse `Shape`, die als

Basistyp für Subjekte in dieser Beispielanwendung dient. Die Klasse `Shape` erbt wiederum von der Oberklasse `Subject` die spezifische Funktionalität der Subjekt-Rolle, die zwei Aufgaben erfüllt: (1) die Verwaltung der registrierten Beobachter (Methoden `attach` und `detach`) und (2) die Benachrichtigung der Beobachter über Zustandsveränderungen (ausgelöst durch die Methode `notify`). Beobachter implementieren dazu das Interface `Observer`, das lediglich eine Methode `update` anbietet (in vielen Fällen auch mit einer Liste von Parametern, die wir aber in unserem Beispiel nicht benötigen). Die Methode `notify` des Subjekts ruft an allen registrierten Beobachtern die Methode `update` auf, um eine Zustandsveränderung zu signalisieren. Der konkrete Beobachter in unserem Beispiel ist die Klasse `ShapeDisplay` als Teil einer Benutzerschnittstelle, die eine Menge von Shape-Objekten darstellen soll.

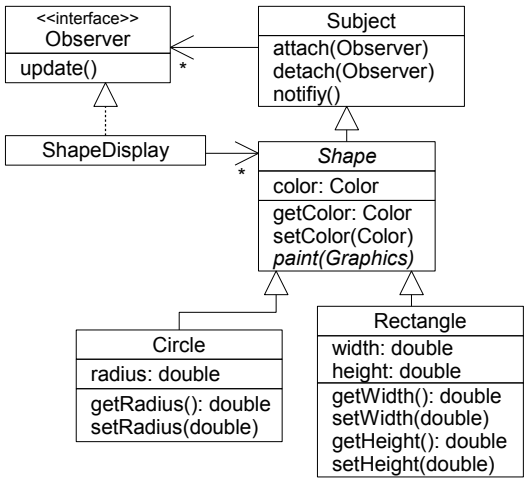

Abbildung 3.1: Eine Anwendung zur Darstellung von Formen, realisiert mit dem Entwurfsmuster *Observer*

Das Observer Pattern beschreibt das spezifische Verhalten von Objekten, welche die Rolle eines Subjekts im Kontext eines Beobachters annehmen. Die Mittel der klassischen Modularisierung mit Klassen und Interfaces bieten keine Möglichkeit, die besondere Art der Beziehung und Zusammenarbeit zwischen Rolle und Kontext in spezialisierter Form abzubilden. Das Modulkonzept von OT dagegen erlaubt eine direkte Abbildung zwischen dem Modell und der Implementierung. Eine Implementierung in OT ist in Listing 3.1 dargestellt. Die Klasse `ShapeDisplay` bildet den eigentlichen Beobachter-Kontext des Szenarios und ist somit als Team realisiert. Es enthält eine Liste von Shapes (Z. 2), deren aktueller Zustand durch Zeichnen auf ein Graphics-Objekt (in der Methode `update`, Zz. 4–7) visualisiert wird. Innerhalb dieses Kontexts spielen die konkreten Subklassen von `Shape`, das sind die Klassen `Circle` und `Rectangle`, die Rolle der Subjekte. Für beide ist eine entsprechende Rollenklasse definiert und an die jeweilige Basisklasse gebunden (Z. 9 bzw. 14). Ein Base Guard Predicate steuert dynamisch die Rollenfunktionalität (Z. 10 bzw. 15): nur jene Instanzen von `Circle` und `Rectangle` können die Rolle des Subjekts im Kontext von `ShapeDisplay` annehmen, die Element der zugehörigen Liste `shapes` sind. Die Benachrichtigung des Beobachters durch die Subjekte ist durch Callin-Bindungen realisiert (Z. 12 bzw. 17). Nach jeder Ausführung einer zustandsverändernden Methode am Subjekt wird die Methode `update` am Beobachter `ShapeDisplay` aufgerufen.

```
1  public team class ShapeDisplay {
2    private List<Shape> shapes;
3    ...
4    public void update() {
5      for (Shape s : shapes)
6        {s.paint(graphics);}
7    }
8
9    protected class SubjectCircle playedBy Circle
10     base when (shapes.contains(base))
11   {
12     update <- after setRadius, setColor;
13   }
14   protected class SubjectRectangle playedBy Rectangle
15     base when (shapes.contains(base))
16   {
17     update <- after setWidth, setHeight, setColor;
18   }
19 }
```

Listing 3.1: Eine Teamklasse, die das Observer Pattern einsetzt

Die OT-Lösung bietet mehrere Vorteile in Hinblick auf die Modularisierung gegenüber einer reinen Java-Lösung. In OT sind die Klasse `Subject` und das Interface `Observer` überflüssig. Der gesamte Code, der mit der Implementierung des Observer-Mechanismus zusammenhängt, ist in der Observer-Klasse `ShapeDisplay` realisiert. Die Basisklassen, welche die Rolle des Subjekts spielen, bleiben unverändert. Die Methoden `attach` und `detach` der Subjekte werden nicht mehr benötigt, da die Beziehung zwischen Subjekt und Beobachter über Callins und Guard Predicates kontrolliert wird. Auch die Methode `notify` wird nicht mehr benötigt, da die Rollen direkt die Methode `update` des umschließenden Teams aufrufen können. Die OT-Lösung zeigt insgesamt eine bessere Aufteilung der kontextabhängigen Funktionalität zwischen den beteiligten Modulen und vermeidet Code Tangling.

3.1.2 Problemstellung

Obwohl OT die Möglichkeiten der Modularisierung beträchtlich verbessert, so gilt dennoch, dass zwischen Rollen- und Basisklasse eine starke Kopplung besteht. An sämtlichen Bindungspunkten (dies sind vor allem Callin-, Callout- und Playedby-Beziehungen, aber auch Guard Predicates und Parameter Mappings) müssen in der Rolle die Basis selbst oder deren konkrete Elemente, wie z.B. Methoden, explizit genannt werden. Zwar ermöglicht OT auch hier die Trennung von spezifischen und allgemeinen Teilen der Implementierung durch Vererbung, da die Bindung aber zu den spezifischen Teilen zählt, bleibt es dennoch erforderlich, für jede konkrete Ausprägung einer Rolle alle Bindungspunkte explizit zu definieren.

Die Kopplung zwischen Rolle und Basis ähnelt der Kopplung zwischen Unter- und Oberklasse, wie sie durch Vererbung entsteht. Im Gegensatz zur Vererbungsbeziehung ist die Rollenbeziehung allerdings weitaus flexibler gestaltet, gerade weil sie jegliche Abhängigkeiten zur Basis explizit über Bindungspunkte definiert. Die eigentliche Funktionalität einer ungebundenen Rolle wird ohne Abhängigkeit zur Basis definiert. Der Typ und damit das Interface der Rolle muss nicht konform sein zum Typ bzw. Interface der Basis. Eine Rolle kann somit auch für Basisklassen unterschiedlicher Interfaces spezialisiert werden, ohne dass dies Einfluss auf das Interface der Rolle hat. In idealisierter Form erfüllt OT das Kriterium der Obli-

viousness in beiden Richtungen: weder kennt eine Basis ihre adaptierenden Rollen, noch kennt eine ungebundene Rolle die Basisklassen, welche sie in Gestalt von gebundenen Subrollen adaptiert. Die konkrete Ausprägung einer Rolle (und damit deren Kopplung) entsteht erst durch die Bindung.

Die Definition der Bindung zwischen Rolle und Basis macht einen wesentlichen Anteil am Code und Programmieraufwand in OT aus. Wie wir im Folgenden erläutern werden, schmälert die rigide Form der Bindungsdefinition allerdings den Nutzen der Modularisierung und die Möglichkeiten zur Wiederverwendung.

Beim Entwurf einer allgemein anwendbaren Rolle definiert der Programmierer eine Menge von Bindungspunkten, im Fall der Subjekt-Rolle im Observer Pattern ist dies beispielsweise eine Callin-Bindung für die Methode `update`. Für die Bindung betrachtet der Programmierer i.A. lediglich eine Abstraktion der in Frage kommenden Basisklassen. Er berücksichtigt nur jene Merkmale der Basis (d.h. Eigenschaften oder Verhalten), die für die Rollenfunktionalität relevant sind. Die konkrete Ausprägung dieser Merkmale ist dabei für ihn nicht von Interesse. Stattdessen besitzt der Programmierer ein gedankliches Modell, welches die relevanten Merkmale einer Basis identifiziert und den Bindungspunkten zuordnet, z.B. „wähle all jene Methoden für die Callin-Bindung, welche darstellungsrelevante Eigenschaften verändern“. Allerdings existiert bislang keine Möglichkeit, diese gedankliche Abstraktion direkt im Code auszudrücken. Stattdessen ist der Programmierer gezwungen, sein Gedankenmodell selbst auf jede konkrete Basisklasse anzuwenden, um die relevanten Merkmale zu selektieren und zu binden – im angegebenen Beispiel ist die Selektion der Setter-Methoden der Klassen `Circle` und `Rectangle` für die Callin-Bindung erforderlich. Das gilt insbesondere für solche Bindungen, die einen direkten semantischen Zusammenhang zu anderen Bindungen aufweisen, der auf die Basis übertragen werden muss. Dieser Fall ist z.B. gegeben, wenn zwei Callout-Methoden auf dasselbe Feld der Basis zugreifen sollen, die eine lesend, die andere schreibend – hier besteht eine inhaltliche Abhängigkeit zwischen den Bindungen.

In OT existiert bislang keinerlei Abstraktion zur Definition der Bindung zwischen Rolle und Basis. Es fehlt allgemein die Möglichkeit, Bindungen auf deklarative Weise zu definieren, indem Aussagen über Struktur und Merkmale der Basis formuliert werden. Diese Einschränkung führt zu den nachfolgend aufgeführten Problemen, deren Lösung das Ziel dieser Arbeit ist:

1. **Modularisierung und Wiederverwendung.** Die fehlende Ausdrucks- bzw. Generalisierungsmöglichkeit der Selektionskriterien, welche die Bindung von Basiselementen an die Rolle definieren, erfordert eine explizite und spezifische Bindung von Basisklassen und deren Merkmalen. Die Modularisierung einer gebundenen Rolle ohne direkte Abhängigkeiten zu einer konkreten Basisklasse ist somit nicht möglich. Darüber hinaus ist keine Wiederverwendung der Selektionskriterien möglich, da auch jede Übertragung der Rolle für die Anwendung mit einer anderen Basisklasse eine spezifische Anpassung durch den Programmierer erfordert.

2. **Robustheit und Wartbarkeit.** Die spezifische Bindung bedeutet eine starke Kopplung zwischen Rolle und Basis, welche die Robustheit des Programms beeinträchtigt. Jede Änderung an einer Basis kann unerwünschte Auswirkungen auf deren gebundene Rollen haben. Die Evolution einer Basisklasse erfordert daher stets ein Review aller gebundenen Rollen, was dem eigentlichen Ziel der Modularisierung widerspricht.

3. **Verständlichkeit.** Die fehlende Abstraktion der Selektionskriterien beeinträchtigt auch die Verständlichkeit des Codes. Der Programmierer ist auf zusätzliche Dokumentation angewiesen, um die Bindung zwischen Rolle und Basis nachvollziehen zu können, da nur das Ergebnis der Selektion, nicht aber

dessen Grundlage im Code ersichtlich ist. Dies gilt insbesondere, falls Abhängigkeiten zwischen den Elementen existieren, welche die Selektion beeinflussen.

Alle drei genannten Probleme finden sich in unserer Beispiel-Implementierung des Observer-Szenarios in Listing 3.1 wieder:

1. Es ist nicht möglich, das Selektionskriterium für die aufgeführten Subjekt-Klassen (Zz. 9 und 14) oder die Methoden der Callin-Bindungen (Zz. 12 und 17) im Code auszudrücken. Stattdessen müssen sowohl die Basisklassen als auch die Methoden vom Programmierer selbst identifiziert und explizit aufgeführt werden.

2. Für den Fall, dass eine der Basisklassen um eine darstellungsrelevante Eigenschaft erweitert wird (z.B. eine Eigenschaft `Brush`, welche die Art spezifiziert, wie der Hintergrund der Form gezeichnet werden soll), müssten alle zugehörigen Rollen ebenfalls angepasst werden. Andernfalls wäre die Observer-Funktionalität lückenhaft.

3. Bei reiner Betrachtung des Codes wird nicht ersichtlich, aus welchen Gründen die Methoden für die Callin-Bindung (Zz. 12 und 17) gewählt wurden. Auch wenn es in diesem Beispiel leicht zu ersehen ist, dass es sich um Setter-Methoden handelt, so lässt sich doch eigentlich nur mit tieferem Verständnis der Methode `update` der Teamklasse erschließen, welche Basismethoden zur Callin-Bindung herangezogen wurden.

3.1.3 Problemanalyse

Das in Abschnitt 3.1.1 vorgestellte Observer-Szenario findet breite Anwendung verschiedener Ausprägung in Forschung und Literatur. Die Charakteristika einer objektorientierten Lösung in Java wie auch die Verbesserungen durch eine aspektorientierte Lösung in AspectJ werden von Hannemann und Kiczales in [49] vorgestellt. Wie die Autoren ausführen, lässt sich die Struktur einer Design-Pattern-Implementierung in zwei Teile untergliedern: allgemeine Code-Anteile, die essentiell für alle Instanzen des Musters sind, und individuelle Code-Anteile, die spezifisch für jede Instanz sind. Die allgemeinen Teile des Observer Patterns sind:

1. Observer-Kontext mit Subject-Rolle
2. Verwalten der Subject-Observer-Relation
3. Allgemeiner Update-Mechanismus („notify“)

Die individuellen Teile des Observer Patterns sind:

4. Binden der Subject-Rolle an bestimmte Klassen und Objekte
5. Auslösen des Update-Mechanismus
6. Spezifische Update-Implementierung des Observers

Für die Wiederverwendung in weiteren Implementierungen können die allgemeinen Teile (Punkte 1–3) in eine abstrakte Oberklasse extrahiert werden. Die spezifischen Teile (Punkte 4–6) können dann als Verfeinerung durch eine Subklasse realisiert werden. Dies ist die übliche und allgemein akzeptierte Vorgehensweise in allen aufgeführten Ansätzen der OOP, AOP und ROP. Wir möchten allerdings an dieser Stelle entgegenhalten, dass wir in diesem Szenario (wie auch in vielen anderen) weiterführende Möglichkeiten der Wiederverwendung sehen, die über die Punkte 1–3 hinausgehen.

Bei genauerer Betrachtung der Observer-Klasse `ShapeDisplay` aus Listing 3.1 sehen wir, dass die Information zur Bindung der Rollen und zum Auslösen des Update-Mechanismus (Punkte 4 und 5) in der spezifischen Update-Implementierung zu finden ist (Punkt 6). Objekte, welche die Rolle des Subjekts annehmen sollen, sind solche, die in der `update`-Methode verwendet werden. Dabei handelt es sich um Elemente der Liste `shapes`. Die relevanten Basisklassen der Rolle sind somit `Shape` und ihre Subklassen (Punkt 4). Die Ausführung der Methode `update` führt zur Aktualisierung der Darstellung. Maßgeblich darin ist der Aufruf der Methode `Shape.paint`. Somit sind offenbar all jene Eigenschaften der `Shape`-Klassen für die Darstellung relevant, die in `update` bzw. `paint` gelesen werden. Das bedeutet, dass Änderungen genau dieser Eigenschaften den Update-Mechanismus auslösen sollten, was durch Bindung aller Methoden mit entsprechendem Schreibzugriff erreicht werden kann (Punkt 5).

Auf konzeptioneller Ebene ist es uns also möglich, die Punkte 4 und 5 allgemein zu beschreiben. Die Idee des Update-Mechanismus kann sprachlich einfach ausgedrückt werden, z.B.: „Löse das Update nach jeder Ausführung einer Methode aus, die potentiell eine Eigenschaft ändert, die innerhalb der Update-Methode gelesen wird.“ Die Aussage ist so allgemeingültig gehalten, dass sie ohne Anpassung auf ähnliche Observer-Szenarien übertragen werden kann, z.B. im Kontext einer Darstellungskomponente zur textuellen Repräsentation der Objekte. Bei der beschriebenen Abstraktion werden quantifizierende Ausdrücke verwendet. Die benötigten Informationen lassen sich aus der Programmstruktur, speziell der Implementierung von Punkt 6, ableiten - auch wenn hierfür eine durchaus komplexe Code-Analyse erforderlich ist. Die üblichen Sprachmittel bieten jedoch keine Möglichkeit, den beschriebenen Ausdruck direkt im Code abzubilden. Keine der bislang hier aufgeführten Programmiersprachen unterstützt Introspektion und Quantifizierung zu dem Grad, dass Abstraktion und Wiederverwendung der Punkte 4 und 5 in direkter Form umsetzbar wären.

3.2 Lösungsansatz

Die in Abschnitt 3.1 diskutierte Problemstellung beschreibt einen Mangel an Abstraktion und Ausdrucksmächtigkeit. Der Programmierer ist nicht in der Lage, sein gedankliches Modell direkt in der Programmiersprache abzubilden. Um dies zu ermöglichen, ist ein zusätzlicher Grad an Abstraktion für die Definition von Rolle-Basis-Beziehungen erforderlich. Im Kern geht es dabei um die Auswahl von Programmelementen der Basis, die mit der Rolle verbunden werden sollen. Anstelle einer direkten Aufzählung soll der Programmierer die relevanten Programmelemente über eine deklarative Beschreibung spezifizieren können. Ein entsprechender Mechanismus muss drei grundlegende Kriterien erfüllen:

1. **Introspektion**: zur Selektion der Programmelemente können Informationen über die Struktur des Programms herangezogen werden.
2. **Quantifizierung**: die Selektion der Programmelemente ist auf Basis von quantifizierenden Ausdrücken möglich.
3. **Generizität**: an den Bindungspunkten der Rolle ist nicht mehr zwingend eine direkte Aufzählung der Programmelemente gefordert, stattdessen kann dort auf das Ergebnis der Selektion verwiesen werden.

Unser Ziel ist die Erweiterung von OT um die beschriebene Fähigkeit zur Abstraktion von Rolle-Basis-Beziehungen. Herrmann beschreibt bereits in [55] die Idee der Integration einer Sprache zur Selektion von Joinpoints in OT. Nachfolgend diskutieren wir, welche Form der Introspektion, Quantifizierung und Generizität uns

geeignet für den Einsatz in OT erscheint. Wir greifen dabei einige von Herrmanns Überlegungen auf und konkretisieren sie zu spezifischen Anwendungen.

3.2.1 Introspektion

Die Frage nach den Möglichkeiten der Introspektion ist unmittelbar mit dem zugrundeliegenden Joinpoint-Modell bzw. dem Modell der Rollenbindung verknüpft. Eine Analyse des Programms muss Informationen bereitstellen, die zur Selektion der zu bindenden Programmelemente erforderlich sind. Dabei gelten für OT zwei wesentliche Rahmenbedingungen:

1. Die Möglichkeiten zur Rollenbindung in OT sind vergleichsweise einfach. Direkt können nur Klassen, Methoden und Felder für die Bindung zwischen Rolle und Basis eingesetzt werden. Darüber hinaus können auch Typen und Parameter relevant sein, da auch sie Bestandteil der Schnittstelle sind.

2. Das Joinpoint-Modell von OT basiert auf der Bindung von statischen Programmelementen. Dynamische Informationen können zwar zur Laufzeit zur Steuerung des Liftings verwendet werden (vgl. Teamaktivierung und Guard Predicates in Abschnitt 2.1.3), die eigentliche Bindung zwischen Rolle und Basis ist aber statischer Natur – analog zur Vererbung.

Die in Abschnitt 2.2.3 diskutierten Charakteristika dieses restriktiven Modells sind essentiell für die Auslegung von Rollen in OT. Die angestrebte Erweiterung von OT soll dies unverändert beibehalten. Allerdings soll eine Abstraktion der Bindungen in Gestalt deklarativer Ausdrücke ermöglicht werden. Obwohl die eigentliche Menge der Programmelemente, die zur Bindung in Frage kommen, restriktiv begrenzt bleibt, sollen dennoch möglichst umfassende Informationen zur Analyse dieser Programmelemente zur Verfügung stehen.

Die Unterscheidung zwischen den Programmelementen, die adaptiert werden können, und denen, die analysiert werden können, ist von besonderer Bedeutung in Hinblick auf den damit verbundenen Kapselungsbruch (siehe Abschnitt 2.2.3). Der tatsächliche Kapselungsbruch der Adaptierung bleibt durch die angestrebte Erweiterung unverändert und ist nach wie vor auf die Schnittstelle der Basisklasse beschränkt. Lediglich für die Analyse werden zusätzliche Informationen benötigt, da auf Details der Implementierung zurückgegriffen werden muss. Dies stellt aber keinen Kapselungsbruch im eigentlichen Sinne dar, da es sich um einen rein lesenden Zugriff zur Compile-Zeit handelt. Der Programmierer besitzt damit die Flexibilität, sich bei Bedarf auch auf strukturelle Details der Implementierung zu beziehen, die für den normalen Zugriff nicht verfügbar wären.

Zur technischen Umsetzung der Introspektion ist bei erster Betrachtung die Nutzung der Standard *Java Reflection API* ein naheliegender Kandidat. Tatsächlich sind dort die wesentlichen Programmelemente, die zur Rollenbindung verwendet werden, für den reflexiven Zugriff verfügbar. Darüber hinaus gibt es allerdings nur wenige Möglichkeiten zur Analyse, da die *Java Reflection API* nicht bis auf die Anweisungsebene herunterreicht. Eine solche Beschränkung ist für die Anforderungen zu strikt. Da der konkrete Informationsbedarf bei der Analyse potentiell nicht vorhersehbar ist, können wir keine allgemeingültigen Kriterien für eine geeignete Beschränkung der Analysefähigkeiten identifizieren. Deshalb erachten wir für unseren Ansatz den vollen Zugriff auf alle Details des Programms für sinnvoll. Aufgrund der Beschränkung auf statisch analysierbare Informationen erscheint uns daher der *abstrakte semantische Graph* (ASG) als geeignete Repräsentation des Programms naheliegend. Der ASG resultiert aus dem *abstrakten Syntaxbaum* (AST) des Programms, wobei einerseits über syntaktische Details abstrahiert wird (die unerheblich für die inhaltliche Analyse sind) und andererseits Knoten des Baums zusätzlich

miteinander verlinkt werden (was eine Anreicherung mit semantischen Informationen darstellt, welche die Analyse deutlich erleichtert). Auch Herrmann sieht den aufgelösten ASG als Voraussetzung für viele sinnvolle Anwendungen [55].

Der ASG soll Gegenstand der Introspektion sein, mittels derer der Programmierer eine Selektion der relevanten Programmelemente vornehmen kann. Zu diesem Zweck ist eine geeignete Abfragesprache erforderlich, mit welcher der Programmierer seine Analyse und Selektion prägnant formulieren kann. Eine wesentliche Anforderung dabei ist die Fähigkeit zur Quantifizierung.

3.2.2 Quantifizierung

Quantifizierung auf Basis von Introspektion ist eine der zentralen Techniken in der AOP (siehe Abschnitt 2.2). Die AOP scheint daher ein perfekter Kandidat zur Erfüllung unserer Anforderungen zu sein.

Quantifizierung von Aspekten, wie sie in AspectJ realisiert sind, wird über Pointcuts realisiert. Die Pointcuts werden auf der Basis eines umfassenden Joinpoint-Modells definiert. Ein wesentlicher Faktor der Ausdrucksmächtigkeit von Pointcuts sind Wildcards. Der Pointcut

```
execution(void SomeClass+.set*(..))
```

verwendet mehrere Wildcards, um die Ausführungspunkte aller Setter-Methoden einer Klasse und ihrer Subklassen in einer Anweisung zu erfassen. Der Pointcut zeigt ein typisches Anwendungsbeispiel von Quantifizierung. Die knappe Formulierung fördert die Lesbarkeit des Codes. Doch obwohl das prägnante Äußere des Pointcuts es nicht auf den ersten Blick vermuten lässt, so ist er doch keineswegs präzise formuliert. Der Code `set*` beschreibt lediglich ein lexikalisches Muster, das keinerlei semantische Bedeutung hat. Der Programmierer möchte damit alle Setter-Methoden erfassen, ohne dass er inhaltliche Kriterien für solche Methoden aufführt. Die Anwendbarkeit solcher Muster basiert daher allein auf Konventionen zur Struktur und Namensgebung. Diese Konventionen müssen eindeutig sein und konsequent eingehalten werden, andernfalls ist die Verwendung solcher Muster nicht praktikabel. Selbst wenn dies der Fall ist, so können trotzdem Konflikte auftreten. Beispielweise würde eine Methode `setup` aufgrund ihres Namens auch von dem Pointcut erfasst, obwohl dies vermutlich nicht vom Verfasser des Pointcuts beabsichtigt war. Dies kann zur nicht-intendierten („falsch positiv“) Selektion von Joinpoints führen, die zwangsläufig unerwünschte Auswirkungen von Aspekten zur Folge haben. Diese Problematik ist von besonderer Bedeutung für die Evolution von Programmen, da selbst einfache Refactorings, wie z.B. das Umbenennen einer Methode, dazu führen können, das ursprünglich korrekte Pointcuts nicht mehr wie gewünscht funktionieren [48, 73] (dabei sollte der Einsatz von Quantifizierung eigentlich gerade die Evolutionsfähigkeit von Programmen unterstützen). Das Prinzip der Obliviousness erschwert es dem Programmierer, die Tragweite seiner Änderungen vollständig zu überblicken. Die Fragilität von Pointcuts ist ein umfassend dokumentiertes Problem [44]. Der Wildcard-Ansatz zur Quantifizierung macht die Ausrichtung des Basiscodes in Hinblick auf Musterkompatibilität erforderlich, um in komplexen Szenarien bestehen zu können [46]. Ein weiteres Problem von Wildcards besteht darin, dass sie nicht dahingehend verknüpft werden können, dass sie an verschiedenen Stellen die gleichen Werte repräsentieren. Mit Wildcards ist es daher nicht möglich, Joinpoints in Kombination zu betrachten, d.h. Abhängigkeiten zwischen Joinpoints bei der Selektion zu berücksichtigen. Die Defizite der beschriebenen Joinpoint-Selektion haben zu einer Fülle von alternativen Ansätzen geführt, die eine ausdrucksmächtigere („semantische“) Pointcutsprache zum Ziel haben [3, 28, 80, 88, 105].

Aufgrund der Probleme Wildcard-basierter Ausdrücke streben wir für die Anfragesprache in OT eine andere Form der Quantifizierung an. Offensichtliche Alternati-

ven zur Analyse des ASG sind relationale Anfragesprachen wie die *Structured Query Language* (SQL) [66]. Relationale Algebren ermöglichen allerdings keine rekursiven Anfragen, d.h. es kann kein transitiver Abschluss der Relationen gebildet werden. Dies ist aber von wesentlicher Bedeutung für eine prägnante Ausdrucksform zur Traversierung des ASG. Es existieren durchaus ausdrucksmächtigere Formen der SQL, die auch rekursive Anfragen ermöglichen[10]. Nichtsdestotrotz erscheinen uns in SQL formulierte Anfragen, die Analysen auf einer tief verschachtelten Hierarchie von Knoten und Relationen ausdrücken, vergleichsweise komplex und wortreich. Dies gilt insbesondere, wenn mehrere Anfragen miteinander kombiniert werden müssen, um Abhängigkeiten zwischen verschiedenen Elementen zu berücksichtigen (was gerade in dem von uns in Abschnitt 3.1.3 beschriebenen Anwendungsfall zum Observer Pattern für die Quantifizierung eine wesentliche Rolle spielt). Beispielhaft ist (analog zum zuvor genannten Pointcut) eine SQL-Query zur Identifizierung aller Setter-Methoden der Klasse `SomeClass` und ihrer Subklassen skizziert:

```
SELECT * FROM methods
WHERE name LIKE 'set%'
AND class IN
  (SELECT class, parent FROM classes
   START WITH class = SomeClass
   CONNECT BY PRIOR class = parent);
```

Die Bedeutung eines verschachtelten *SELECT*-Ausdrucks ist schwer zu erfassen. Dies gilt umso mehr, wenn bei der Suche nicht nur eine, sondern gleich mehrere Dimensionen in Tiefe betrachtet werden und dabei evtl. sogar Querverweise zwischen diesen berücksichtigt werden müssen. Diese Problematik besteht grundsätzlich bei Ansätzen, bei denen die Navigation entlang von Pfaden in einem hierarchisch aufgebauten Graphen im Vordergrund steht.

Eine ausdrucksmächtigere und zugleich prägnantere Alternative für deklarative Anfragen bieten logische Programmiersprachen wie *Datalog* und *Prolog*. Datalog ist eine dedizierte Anfragesprache für Datenbanken, die mittlerweile auch verstärkt zur Programmanalyse eingesetzt wird [62]. Syntax und Funktionsweise sind denen von Prolog sehr ähnlich (vgl. Abschnitt 2.3). Datalog ist allerdings in einigen Punkten etwas einfacher gehalten, etwa darin, dass es keine komplexen Terme als Argumente von Prädikaten erlaubt. Grundsätzlich erscheint uns die logische Programmierung als besonders geeignet für die Graphanalyse, da Prädikate über Knoten und Kanten in einfacher Weise modularisiert und kombiniert werden können, ohne dass eine Schachtelung erforderlich ist. Dies trägt wesentlich zur Verständlichkeit komplexer Ausdrücke bei. Beispielhaft sei eine Prolog-Anfrage (für zuvor genannten Pointcut) skizziert:

```
methodOfClass(M, C),
isSetter(M),
inheritsFrom(C, someclass).
```

Die Anfrage setzt sich aus mehreren eigens definierten Prädikaten zusammen. Die Implementierung dieser Prädikate erfordert natürlich zusätzlichen Code – der Umfang des Codes, der insgesamt zur Formulierung einer Anfrage benötigt wird, ist im Vergleich zu anderen Ansätzen also nicht unbedingt geringer. Der Vorteil von Prolog zeigt sich vielmehr in der Modularisierung, da Anfragen sehr einfach und ohne Schachtelung in kleinere Einheiten aufgeteilt werden können und ebenso einfach – durch Unifikation der Variablen (wobei kein „Gluecode“ erforderlich ist!) – aus solchen Einheiten kombiniert werden können. Die ausgeprägte Fähigkeit zur

[10] In der Spezifikation SQL 3 aus dem Jahr 1999 wurde das Konstrukt *WITH RECURSIVE* ergänzt. Eine nicht dem Standard entsprechende – aber populäre – alternative Syntax verwendet das Konstrukt *CONNECT BY*.

(De-)Komposition trägt sehr zur Wiederverwendbarkeit der Anfragen bei. Die Formulierung von Prädikaten erscheint uns im Vergleich auch als die direkteste Form der Abbildung einer deklarativen Beschreibung. Das Prädikat `isSetter` zur Selektion von Setter-Methoden kann wie in den Beispielen zuvor als lexikalischer Mustervergleich implementiert werden, alternativ können aber auch inhaltlichere Kriterien zur Analyse herangezogen werden (ein konkretes Beispiel zur Umsetzung wird später in Abschnitt 4.2.3 präsentiert).

Bei der Wahl der konkreten Programmiersprache fiel unsere Entscheidung letztlich zugunsten von Prolog, da die Sprache Prädikate mit verschachtelten Termen erlaubt (was eine prägnantere Schreibweise ermöglicht) und die gezielte Terminierung von Anfragen ermöglicht (mittels Cut-Operator, was der Effizienz zuträglich ist). Diese Eigenschaften sind aber nicht zwingend erforderlich, weshalb auch Datalog hinreichend mächtig für die angestrebten Zwecke wäre.

3.2.3 Generizität

Eine generische Rollendefinition muss Variablen anstelle von konkreten Programmelementen an den Bindungspunkten erlauben, d.h. Variablen dürfen die Angabe zur Basisklasse, sowie zu den Basiselementen von Callin- und Callout-Bindungen, ersetzen. Da es sich hier nicht um „normale“ Variablen handelt, sondern um variable Platzhalter für statische Programmelemente, sprechen wir an dieser Stelle von *Metavariablen*. Das folgende Codefragment skizziert eine generische Rollenklasse, in der die Basisklasse und die Basismethode eines Callins als Metavariablen repräsentiert sind:

```
class GenericRole playedBy <metavar1> {
  void foo() {...}
  foo <- after <metavar2> ;
}
```

Eine generische Rolle entspricht somit einer parametrisierbaren Template-Klasse, deren Code erst durch die konkrete Belegung der Metavariablen ausführbar wird. Die Definition einer Klasse mit Metavariablen ist eine Form der Metaprogrammierung.

Die Belegung der Metavariablen kann entweder imperativ oder deklarativ erfolgen. Eine imperative Belegung würde lediglich eine alternative Lokalisierung der Bindungsdefinition ermöglichen, da diese nun zwar an anderer Stelle, aber prinzipiell in der gleichen Form wie bisher definiert würde. Darin liegt kein erkennbarer Mehrwert. Die deklarative Belegung der Metavariablen entspricht dagegen dem Wunsch nach einer deklarativen Beschreibung der Bindung. Die Metavariablen können unmittelbar mit Prolog-Variablen assoziiert werden. Eine Prolog-Anfrage kann somit direkt zur Definition der Metavariablen verwendet werden.

3.2.4 Integration

Die Erweiterung von OT um generische Rollen nennen wir *Generic Object Teams*[11], (GOT). Mit dem beschriebenen Ansatz zur Metaprogrammierung wird ein fundamental neues Konzept in die Sprache OT eingeführt. Ein wesentliches Ziel bei der Integration ist die Abwärtskompatibilität, so dass bestehender Code, Werkzeuge und Dokumentation davon nicht betroffen sind. Ferner soll der Charakter der Sprache beibehalten werden, um OT-Programmierern den Einstieg zu erleichtern. Damit lassen sich drei Design-Prinzipien (nach MacLennan [81]) benennen, die bei der Integration im Vordergrund stehen:

[11] Besten Dank an Günter Kniesel, der diesen Namen vorgeschlagen hat.

- **Einfachheit**. Die Zahl der neu einzuführenden Sprachmittel soll auf das notwendige Minimum beschränkt werden. Die Mechanismen zur Metaprogrammierung sollen derart in die Sprache eingebettet werden, dass sie auch von Programmierern mit geringen Kenntnissen von Prolog (zumindest rudimentär) eingesetzt werden können.

- **Orthogonalität**. Die Spracherweiterung soll abwärtskompatibel zur Originalsprache sein, d.h. jedes gültige OT-Programm ist auch ein gültiges GOT-Programm. Die neuen Sprachmittel sollen keine unerwünschte Interferenz mit den bestehenden Sprachmitteln aufweisen. Im Gegenteil, die Sprachmittel sollen – wo es sinnvoll ist – in möglichst intuitiver Weise miteinander kombiniert werden können.

- **Konsistenz**. Sowohl die Syntax als auch die Struktur (bzw. die Modularisierung) der neuen Sprachmittel sollen möglichst nah an denen der bestehenden Sprachmittel angelegt sein.

Die genannten Prinzipien sind dabei alle dem Prinzip der *Abstraktion* untergeordnet, das die Hauptmotivation für den GOT-Ansatz darstellt. Die unterschiedlichen Dimensionen der Qualität einer Programmiersprache stehen nicht selten im Zielkonflikt zueinander – im Fall von GOT sind dies speziell Verständlichkeit und Komplexität gegenüber Abstraktion und Ausdrucksmächtigkeit. Letztendlich muss beurteilt werden, inwieweit der Zugewinn an Abstraktions- und Ausdrucksvermögen die Abstriche in anderen Dimensionen rechtfertigt.

Kapitel 4

Die Spracherweiterung Generic Object Teams

Zur Umsetzung von Generic Object Teams wird die Sprache Object Teams um Sprachmittel zur Metaprogrammierung erweitert. Konkret werden mit GOT drei neue Sprachmittel eingeführt:

1. **Metavariablen.** Metavariablen sind generische Programmelemente. Sie dienen zur Parametrisierung des Codes von Rollen- und Teamklassen – in erster Linie zur Ersetzung statischer Referenzen auf Basisklassen und deren Methoden an den Bindungspunkten zwischen Rolle und Basis.

2. **Queries.** Queries dienen zur Definition von Metavariablen. Mithilfe von Queries kann in deklarativer Form die Menge der Programmelemente beschrieben werden, mit der die Metavariablen belegt werden sollen.

3. **Per-Blöcke.** Per-Blöcke dienen als Container für generischen Code. Ein Per-Block umgibt ein Codefragment, das als Template fungiert, und deklariert eine Menge von Metavariablen als Parameter. Innerhalb des Codefragments können die Metavariablen als Variationspunkte eingesetzt werden. Per-Blöcke werden durch einen Transformationsprozess umgewandelt, dabei werden alle Vorkommen generischer Programmelemente durch konkrete Programmelemente ersetzt. In der Literatur können Per-Blöcke allgemein als *Fragment Boxes* und Metavariablen als *Hooks* eingeordnet werden, der Transformer wird auch *Composer* genannt (vgl. Aßmann [4] Kapitel 4).

Die Verarbeitung eines GOT-Programms besteht in dessen Transformation in ein normales OT-Programm. Bei der Transformation werden alle generischen Code-Anteile expandiert und entfernt. Das transformierte Programm kann anschließend vom Standard-OT-Compiler kompiliert werden (dargestellt in Abbildung 4.1).

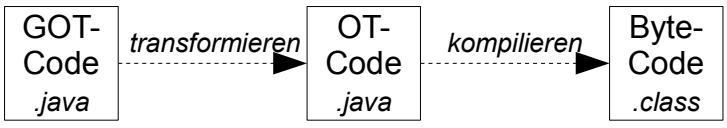

Abbildung 4.1: Der Verarbeitungsprozess eines GOT-Programms

Der Vorgang ist analog zum Verarbeitungsprozess von *Java Generics* aufgebaut. Dort wird die ursprünglich generische Version eines Java-Programms auf eine nichtgenerische Version in der Originalsprache abgebildet. Dabei werden alle generischen

Anteile ersetzt bzw. entfernt (man spricht von *Erasure*). Neue Programmelemente, sogenannte *Bridge Methods* (auch *synthetische Methoden*), werden eingeführt, um die Semantik des Programms zu erhalten [16].

Auf dem aktuellen Stand ist die Transformation von GOT als Quellcode-Transformation realisiert, womit der Programmierer direktes Feedback über das Ergebnis der Transformation erhält. Der Transformationsprozess eines GOT-Programms ist dem weiteren Verarbeitungsprozess durch den OT-Compiler vorgelagert. Der bestehende OT-Compiler bleibt somit von der Erweiterung unberührt und kann unabhängig davon weiterentwickelt werden.

In den folgenden Abschnitten wird die Umsetzung von GOT im Detail beschrieben. In Abschnitt 4.1 wird zunächst die Repräsentation von Programmen in Prolog erläutert. Anschließend wird in Abschnitt 4.2 die Integration von Prolog über Metavariablen und Queries beschrieben. In Abschnitt 4.3 wird Generizität durch Per-Blöcke in Teamklassen eingeführt und deren Transformation erläutert. Die Wiederverwendbarkeit solcher generischer Programmteile wird in Abschnitt 4.4 behandelt. Abschließend wird in Abschnitt 4.5 die Korrektheit der Transformation eines GOT-Programms in ein OT-Programm diskutiert.

4.1 Faktendarstellung

Zur Analyse eines Programms mit Prolog wird eine Repräsentation von dessen ASG in Form von Prolog-Fakten benötigt. Das Modell des ASG entspricht im Wesentlichen dem Metamodell der Programmiersprache. Prinzipiell kann bei der Abbildung als Faktenbasis die Menge verfügbarer Information auf einen relevanten Abstraktionsgrad reduziert werden. Für unsere Zwecke haben wir allerdings eine vollständige Abbildung gewählt, so dass aus der Faktenrepräsentation eines Programms auch wieder ein semantisch äquivalentes Programm generiert werden kann.

Im Allgemeinen wird jeder Knoten des ASG durch einen Fakt repräsentiert, der sämtliche Attribute des Knotens beschreibt. Jeder Fakt bekommt eine eindeutige ID zugewiesen, um die Referenzierung der Fakten untereinander zu ermöglichen. Die ID selbst ist eine Konstante ohne weitere Bedeutung und kann automatisch generiert werden.

Jeder Knotentyp des ASG wird durch ein bestimmtes Prädikat abgebildet; in unserem Modell existieren ca. 110 verschiedene Prädikate zur Repräsentation eines GOT-Programms. Zur Abbildung eines Programms in Prolog wird für jedes konkrete Programmelement ein Fakt erzeugt, dessen Kopf einem dieser Prädikate entspricht[12]. Ein Prädikat definiert dabei eine Relation zwischen konstanten Werten, die je nach Art in Anzahl und Bedeutung variieren. Eine Methodendeklaration wird beispielsweise durch das zehnstellige Prädikat `methodT` repräsentiert. Es hat die folgende Struktur (wobei eine vorangestellte Raute ein Zahl-Argument, einfache Anführungsstriche ein String-Argument und eckige Klammern eine Liste von Argumenten definieren):

methodT(#id, #class, 'name', [#parameters], [#exceptions], #body, #returnType, ['modifiers'], [#annotations], [#typeParameters]).

Die Argumente haben folgende Bedeutung:

- **#id**: das erste Argument ist – wie bei allen anderen Prädikaten zur Knotenrepräsentation auch – die ID, die einen konkreten Fakt eindeutig identifiziert.
- **#class**: ist die ID des Fakts, der die Klasse oder das Interface repräsentiert, die bzw. das die Methode deklariert. Der referenzierte Fakt gehört zum Prä-

[12] Übertragen auf die OOP: Prädikate entsprechen Klassen, Fakten entsprechen Instanzen.

dikat *classT*, das in ähnlicher Form die Deklaration einer Klasse oder eines Interfaces beschreibt.

- **'name'**: ist der Bezeichner der Methode.
- [**#parameters**]: ist eine Liste von IDs der Fakten, welche die Parameter der Methode repräsentieren. Das entsprechende Prädikat *parameterT* enthält die Informationen zu Bezeichner und Typ des Parameters.
- [**#exceptions**]: ist eine Liste von IDs der Fakten für die deklarierten Exceptions der Methode.
- **#body**: ist die ID des Fakts für den Methodenrumpf. Der Methodenrumpf ist als Fakt des Prädikats *blockT* repräsentiert, das lediglich als Container für eine Sequenz von IDs dient, womit auf die inneren Elemente des Blocks verwiesen wird.
- **#returnType**: ist die ID des Fakts für den deklarierten Rückgabetyp der Methode. Der referenzierte Fakt gehört zum Prädikat *typeTerm*, das allgemein zur Repräsentation von Typen verwendet wird.
- [**'modifiers'**]: ist eine Liste von Modifikatoren der Methode. Damit wird die Menge der statischen Modifikatoren erfasst, die in Java bei der Deklaration von Programmelementen zum Einsatz kommen können (z.B. `public` und `static`).
- [**#annotations**]: ist eine Liste von IDs der Fakten für die Annotationen der Methode. Eine Annotation wird durch Fakten des Prädikats *annotationT* repräsentiert, die neben dem Bezeichner auch Ausdrücke und Schlüssel-Wert-Paare der Annotation erfassen.
- [**#typeParameters**]: ist eine Liste von IDs der Fakten für die Typparameter der Methode. Das zugehörige Prädikat *typeParamT* erfasst den Bezeichner und Angaben zur Begrenzung des Typs.

Zum besseren Verständnis soll die Repräsentation der Fakten am Beispiel einer einfachen Methode veranschaulicht werden. Die Methode

```
public long triple(int x) {return 3*x;}
```

definiert eine simple Multiplikations-Funktion. Die Fakten-Repräsentation des ASG der Methode ist in Abbildung 4.2 skizziert. Die Knoten des Graphen erhalten jeweils eine eindeutige ID, die zur Referenzierung verwendet wird. Die Traversierung des Graphen erfolgt rekursiv, weshalb die Vergabe der IDs bei den Blättern des Graphen beginnt und sich zur Wurzel fortsetzt.

Die Referenz des Knoten mit ID 6 (dem rechten Operanden der Multiplikation) zum Knoten mit ID 3 (dem Parameter der Methode) verdeutlicht, dass es sich nicht um eine einfache Baumstruktur handelt, da mehrere Pfade existieren, die Knoten 10 und Knoten 3 verbinden. Tatsächlich enthält der ASG eines Programms zahlreiche Kreise.

Die vollständigen Fakten zur Darstellung der Methode `triple` in Prolog ist in Listing 4.1 aufgeführt. Entsprechend der Nummerierung ist das Wurzelelement unten in der Liste zu finden.

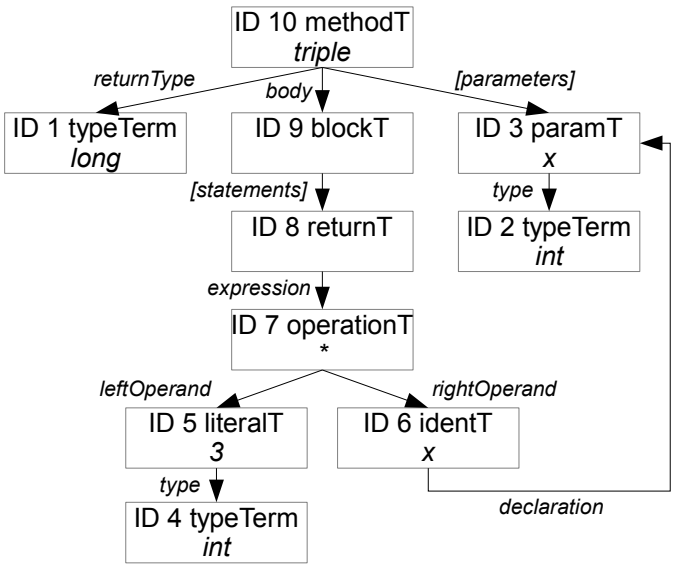

Abbildung 4.2: Fakten zur Abbildung des ASG der Methode `triple`

```
1  typeTerm(1,'basic',nil,'long',0,[],nil).
2  typeTerm(2,'basic',nil,'int',0,[],nil).
3  paramT(3,2,'x',[],[],0,'false').
4  typeTerm(4,'basic',nil,'int',0,[],nil).
5  literalT(5,10,4,'3').
6  identT(6,10,3,'x').
7  operationT(7,10,[5,6],'*','infix').
8  returnT(8,10,7).
9  blockT(9,10,[8]).
10 methodT(10,11,'triple',[3],[],9,1,['public'],[],[]).
```

Listing 4.1: Faktenrepräsentation der Methode `triple`

Der Fakt mit der ID 10 (die ID ist in diesem Beispiel stets gleich zur Zeilennummer) ist eine Klausel des Prädikats `methodT` und repräsentiert die Deklaration der Methode `triple`. Der Fakt zur deklarierenden Klasse ist mit der ID 11 referenziert (dieser Fakt ist aber nicht im Beispiel enthalten). Der Name *triple* und der einzige Modifikator *public* sind direkt genannt. Als einziger Parameter ist der Fakt mit ID 3 aufgeführt. Die Methode hat weder Exceptions, noch Annotationen oder Typparameter, die entsprechenden Listen sind folglich leer. Der Rückgabetyp der Methode ist mit ID 1 definiert. In gleicher Form sind auch der Typ des Parameters (ID 2) und der implizite Typ des Literals (ID 4) gegeben. Typen werden durch das Prädikat *typeTerm(#id, 'kind', #type, 'typeName', #dimension, [#typeArguments], #referenceType)* repräsentiert. Da es sich in diesem Beispiel ausschließlich um primitive Typen (*kind='basic'*) handelt, ist lediglich der Name des jeweiligen Typs aufgeführt, aber keine ID zu einem Fakt des Typs (*type=nil*). Bei Referenztypen wird an dieser Stelle die ID der zugehörigen Klasse bzw. des Interfaces angegeben. Die Argumente zur Dimension, den Typ-Argumenten und dem Referenztyp kommen bei komplexeren Typkonstruktionen zum Einsatz, z.B. bei Arrays, generischen Typparametern und Parametern, die Lifting deklarieren. Der Rumpf der Methode (ID 9) enthält als einzige Anweisung ein *return*-Statement (ID 8). Zurückgegeben wird das Ergebnis

eines Ausdrucks (ID 7), der ein Literal (ID 5) und eine Variable (ID 6) miteinander multipliziert. Alle Fakten des Methodenrumpfes besitzen als zweites Argument eine ID der umschließenden Methode. Diese Information ist zwar redundant, vereinfacht und beschleunigt aber das Abfragen der Faktenbasis. Auch an anderen Stellen des Modells existieren zu diesem Zweck gezielt Redundanzen. Da die Fakten automatisch generiert und normalerweise nicht manuell manipuliert werden, ist Redundanz hier i.A. unproblematisch.

4.2 Queries

Zur Programmanalyse in GOT benötigt der Programmierer eine Möglichkeit, Anfragen an die Faktenbasis zu stellen. Zu diesem Zweck wird die *Query* als neues Programmelement eingeführt. Eine Query deklariert und definiert jene Metavariablen, die im Rahmen der Metaprogrammierung zum Einsatz kommen. Queries sind strukturell ähnlich zu Methoden aufgebaut, realisieren dabei aber die Semantik von Prolog-Anfragen, wie in Abschnitt 2.3 beschrieben. Eine Query besteht aus einem Kopf zur Deklaration und einem Rumpf zur Definition.

4.2.1 Query-Deklaration

Der Kopf einer Query enthält deren Namen gefolgt von einer Liste von Parametern. Als Parameter dienen Metavariablen, die anschließend in Abschnitt 4.2.2 eingeführt werden. Im Unterschied zu einer Methode deklariert eine Query keinen Rückgabetyp, sondern führt stattdessen das Schlüsselwort `otquery` auf. Nachfolgend ist beispielhaft der Kopf einer Query `size` mit zwei Parametern gezeigt:

```
otquery size(?String +s, ?int ?len)
```

Die Notwendigkeit für das neue Schlüsselwort `otquery` ist durch die unterschiedliche Semantik von Queries und Methoden gegeben: trotz äußerlicher Ähnlichkeiten unterscheiden sich Rückgabe und Parameter erheblich.

Queries haben die Aufgabe eine passende Belegung für die Metavariablen der Parameter zu finden. Ihr Ergebnis ist daher stets eine Menge von Lösungen, die jeweils eine gültige Belegung der Metavariablen repräsentieren. Eine Rückgabe des Ergebnisses im klassischen Sinne ist dabei nicht vorgesehen, da die Lösungsmenge selbst nicht im Programm verwendet wird, sondern nur zur Definition der Belegung der Metavariablen dient. Die Metavariablen können direkt im Code als Platzhalter für konkrete Programmelemente verwendet werden (dies wird in Abschnitt 4.3.2 erläutert).

Die Eleganz der Queries im Prolog-Stil liegt in der Unifikation von Variablen über eine Menge von verknüpften Prädikaten. Eine strikte Trennung von Ein- und Ausgabeparametern ist für diesen Zweck hinderlich, stattdessen sollen die Query-Parameter direkt zur Ein- und Ausgabe genutzt werden, analog zu Variablen in Prolog. Wir übernehmen an dieser Stelle also die Prolog-Semantik. Dies ist ein wesentlicher Unterschied zur Java-Semantik; zum einen wegen der Quantifizierung, zum anderen sind Parameter in Java sonst stets Wertparameter, d.h. Argumente werden strikt *pass-by-value* übergeben. Letztere Restriktion ist allerdings weitgehend auf Java beschränkt. In vielen anderen populären OO-Sprachen, wie z.B. C#, ist es ohne Weiteres möglich, Parameter mit einer *pass-by-reference* Semantik zu verwenden (vgl. Deklaration mit `ref` und `out` in C#). Die Besonderheit der Prolog-Semantik besteht daher primär in der quantifizierenden Belegung der Metavariablen.

4.2.2 Metavariablen

Variablen, die Programmelemente repräsentieren, werden in GOT als *Metavariablen* bezeichnet. Metavariablen können ausschließlich als Parameter oder lokale Variablen von Queries deklariert werden. Im Unterschied zu Prolog, wo Variablen ungetypt sind, schließt die Deklaration hier einen Typ mit ein. Dies ermöglicht zum einen statisches Type-Checking und ist zum anderen konsistent zur Deklaration von normalen Variablen in Java. Die Begrenzung der Metavariablen auf einen Typ ist für unseren Anwendungsbereich auch aus praktischer Sicht sinnvoll, wie wir im Folgenden noch ausführen werden.

Zur Deklaration von Metavariablen kann nur eine begrenzte Menge von Typen, im Folgenden *Metatypen* genannt, herangezogen werden, die disjunkt zur Menge der „normalen“ Java-Typen ist. Um diese sowohl syntaktisch als auch visuell klar unterscheidbar zu machen, beginnen die Bezeichner aller Metatypen mit einem Fragezeichen. Die Metatypen repräsentieren die Programmelemente des ASG. Die Menge der Metatypen entstammt also im Wesentlichen dem Metamodell der Programmiersprache. Für jedes Prädikat, das zur Faktendarstellung definiert ist, existiert ein Metatyp, der die Menge aller Fakten dieses Prädikats beschreibt. Darüber hinaus existieren Metatypen für die primitiven Typen *String*, *Integer* und *Boolean*.

Der deklarierte Metatyp einer Metavariablen schränkt deren mögliche Belegung dahingehend ein, dass nur Fakten des Metatyps oder eines konformen Subtyps gewählt werden können. Der Metatyp definiert also im praktischen Sinne ein Prädikat, das implizit jeder Anfrage hinzugefügt wird, die zu der Metavariablen formuliert wird. Auf die Bedeutung der Metatypen für die Anwendung der Metavariablen gehen wir nachfolgend in Abschnitt 4.3 im Detail ein.

Wie in Abschnitt 4.2.1 erläutert, dienen die Parameter einer Query sowohl zur Ein- als auch zur Ausgabe. Um das Verständnis und die Handhabung dabei zu erleichtern, werden die Metavariablen von vornherein in drei Kategorien eingeteilt; die Zuordnung erfolgt jeweils durch das Voranstellen eines der Symbole „+“, „-“ und „?“ am Bezeichner (basierend auf einer gebräuchlichen Notation von Prolog). Die Kategorie gibt Auskunft über die erwartete Belegung der Metavariablen. Eine Metavariable mit dem Präfix „+“ muss beim Aufruf der Query zwingend mit einem Wert belegt sein; sie kann als klassischer Eingabeparameter interpretiert werden. Im Gegensatz dazu muss eine mit Präfix „-“ notierte Metavariable beim Aufruf zwingend frei (d.h. ohne Belegung) sein; sie entspricht damit einem Ausgabeparameter. Eine Metavariable mit dem Präfix „?“ darf beim Aufruf sowohl frei als auch belegt sein; sie dient also sowohl zur Ein- als auch zur Ausgabe. Hintergrund für die Unterteilung ist, dass die Aufrufe einer Query auf „sinnvolle“ Anwendungen eingeschränkt werden können. Nehmen wir als Beispiel die Query `size`:

```
otquery size(?String +s, ?int ?len) {...}
```

Die Query liefere als Ergebnis *true*, wenn die Länge des Strings des Parameters `+s` mit dem Wert des Parameters `?len` übereinstimmt. Würde die Query nun mit zwei freien Metavariablen aufgerufen, so wäre die Ergebnismenge unendlich groß, da sich eine unendliche Menge von Paaren finden lässt, für die diese Aussage gilt. Die Query wäre nicht auswertbar bzw. die Auswertung würde nicht terminieren. Auch wenn der Parameter für die Länge mit einen festen Wert n belegt würde, so würden immer noch a^n Strings gefunden (für a Zeichen des verfügbaren Alphabets) welche die Aussage erfüllen – ein Wert der praktisch oft nicht mehr händelbar ist. Um solche Aufrufe auszuschließen, wird der erste Parameter daher explizit als Eingabeparameter deklariert, womit beim Aufruf zwingend eine Belegung gegeben sein muss. Ziel der Query ist es also nicht, Strings zu generieren, sondern Aussagen zur Länge eines vorgegebenen Strings zu machen. Der Parameter zur Länge darf nach Bedarf belegt werden oder frei bleiben: im ersten Fall würde die Query prüfen,

ob die übergebene Länge der tatsächlichen Länge des String entspricht, im zweiten Fall würde die Metavariable mit der tatsächlichen Länge belegt.

Im Normalfall können alle Metavariablen ohne Bedenken als Ein- und Ausgabeparameter (Präfix „?“) deklariert werden. Es gibt nur wenige Ausnahmen, wie den zuvor beschriebenen Fall, in denen eine anderweitige Deklaration wirklich erforderlich ist. Nichtsdestotrotz kann die Kategorisierung der Metavariablen hilfreich sein, um ihre intendierte Anwendung für den Programmierer transparenter zu machen.

4.2.3 Query-Definition

Im Definitionsteil einer Query wird die Belegung der Metavariablen spezifiziert. Die Definition entspricht im Wesentlichen einer Prolog-Abfrage, die entscheidet, ob eine formulierte Aussage wahr ist bzw. die sämtliche Belegungen der freien Variablen findet, für die dies der Fall ist. Die Queries werden dementsprechend als Regeln in der Prolog-Faktenbasis abgebildet, wobei der Deklarationsteil den Kopf und der Definitionsteil den Rumpf der Regel bildet. Die Metavariablen werden als freie Prolog-Variablen abgebildet, die im Kopf der Regel als Argumente aufgeführt werden. Eine Query mit n Metavariablen resultiert somit zu einem n-stelligen Prädikat. Da Variablen in Prolog nicht typisiert sind, können die deklarierten Metatypen der Metavariablen nicht im Kopf einer Regel abgebildet werden. Stattdessen wandert diese Information in den Rumpf. Der Definitionsteil einer Query wird dabei implizit durch Verundung eines Prädikats zum deklarierten Metatyp jeder Metavariablen erweitert. Dies hat besondere Bedeutung für Eingabeparameter (Präfix „+“), die mit einem der ASG-Knoten-Metatypen deklariert sind. Durch das zusätzliche Prädikat werden die möglichen Belegungen von vornherein auf eine endliche Menge eingeschränkt. Selbst wenn die Anfrage also sonst keine Belegung für das Argument vorgibt, so ist sie trotzdem auswertbar. Die Deklaration als Eingabeparameter dient daher primär dem Programmierer zur Orientierung, technisch ist sie in diesen Fällen nicht erforderlich.

Die Definition von Queries in einem GOT-Programm kann auf zwei Arten erfolgen. Sie werden unterschieden in *High-Level-* und *Low-Level-Queries*.

High-Level-Queries

Bei einer High-Level-Query erfolgt die Definition der Query über einen logischen Ausdruck im Rumpf. Optional können vornan lokale Metavariablen deklariert werden. Der logische Ausdruck besteht aus einer Menge von Query-Aufrufen, die über logische Operatoren miteinander verknüpft werden können. Alternativ zum Aufruf einer Query können auch die Konstanten *true* oder *false* verwendet werden. Verfügbar sind die zweistelligen Operatoren „||“ (*or*) und „&&“ (*and*) und der einstellige Operator „!“ (*not*), ferner besteht die Möglichkeit zur Klammerung von Teilausdrücken. Es gilt die übliche Semantik von Prolog, sie ist lediglich syntaktisch auf die Sprache Java übertragen. Ein Beispiel für eine High-Level-Query ist mit der Query `changes` in Listing 4.2 gegeben.

Die Query `changes` macht eine Aussage über Relationen von jeweils einer Methode `?m` zu einem Feld `?f` (Z. 5). Sie soll aussagen, ob die Methode `?m` direkt oder indirekt eine Wertänderung eines Feldes `?f` auslösen kann; entweder auf direktem Weg, indem sie einen Schreibzugriff auf das Feld hat, oder auf indirektem Weg, indem sie eine andere Methode aufruft, in deren Kontrollfluss ein solcher Schreibzugriff vorkommen kann. Ein direkter Schreibzugriff sei durch die Query `writes` formuliert. Alle Methoden, die durch direkte oder indirekte Aufrufe von einer Methode aus erreichbar sind, seien über die Query `reachable` identifizierbar. Die Query `changes` formuliert nun, dass ein Schreibzugriff von `?m` auf `?f` gegeben ist (Z. 7) oder dass

von `?m` aus eine Methode `-callee` erreichbar ist, die wiederum einen Schreibzugriff auf `?f` besitzt (Z. 8).

```
1 public otquery writes(?Method ?m, ?Field ?f) {...}
2
3 public otquery reachable(?Method ?caller, ?Method
    ?callee) {...}
4
5 public otquery changes(?Method ?m, ?Field ?f) {
6   ?Method -callee;
7   writes(?m, ?f) ||
8   (reachable(?m, -callee) && writes(-callee, ?f))
9 }
```

Listing 4.2: Beispiel der High-Level-Query `changes`, die durch Kombination zweier Queries `writes` und `reachable` definiert wird.

Low-Level-Queries

Bei einer Low-Level-Query ist der Rumpf leer, stattdessen erfolgt die Definition der Query über die Angabe einer oder mehrerer Annotationen des Typs `@Prolog`. Eine Prolog-Annotation enthält stets eine Klausel in normaler Prolog-Syntax. Low-Level-Queries bilden das Verbindungselement zwischen den unterschiedlichen Sprachwelten von Java und Prolog. Sie erlauben die Verwendung nativer Prolog-Ausdrücke innerhalb eines GOT-Programms, in denen nun direkt Klauseln der Faktenbasis angesprochen werden können.

Mindestens eine der mittels Prolog-Annotationen definierten Klauseln muss vom Namen und der Stelligkeit des Prädikats her mit dem Namen der Query und der Zahl ihrer Parameter übereinstimmen. Dieses Prädikat ist die äquivalente Repräsentation der Query in Prolog. Ein Beispiel für eine Low-Level-Query ist mit der Query `writes` in Listing 4.3 gegeben.

```
1 @Prolog("writes(M, F) :-
2   assignopT(_, M, GF, _, _),
3   getFieldT(GF, M, _, _, F)"
4 )
5 public otquery writes(?Method ?m, ?Field ?f) {}
```

Listing 4.3: Beispiel für eine Low-Level-Query, die durch die Angabe einer Prolog-Annotation definiert wird.

Die Query `writes` spezifiziert analog zum vorigen Beispiel die Relationen von jeweils einer Methode `?m` zu einem Feld `?f` (Z. 5). Zur Definition ist eine Prolog-Annotation angegeben, die eine Regel enthält (Zz. 1–4). Der Kopf der Regel entspricht der Signatur der Query, wobei die Variable `M` auf die Metavariable `?m` und die Variable `F` auf die Metavariable `?f` abgebildet wird. Der Rumpf der Regel besteht aus der Verundung zweier Prädikate. Das Prädikat *assignopT* beschreibt den ASG-Knotentyp für eine Zuweisung mit folgender Struktur: *assignopT(#id, #enclosingMethod, #leftHandSide, 'operator', #rightHandSide)*. In der Query wird das Prädikat mit Wildcards anstelle der irrelevanten Argumente angewendet (Z. 2). Von Interesse ist allein, dass innerhalb der Methode `M` eine Zuweisung existiert, bei deren linker Seite (Variable `GF`) es sich um einen Feldzugriff handeln muss. Die Variable `GF` wird dazu mithilfe eines Prädikats *getFieldT* genauer spezifiziert (Z. 3). Das Prädikat beschreibt den ASG-Knotentyp für einen Feldzugriff mit der Struktur *getFieldT(#id, #enclosingMethod, #receiver, 'name', #field)*. Als letztes Argument

ist mit der Variablen `F` das fragliche Feld aufgeführt. Implizit wird der Ausdruck noch um ein Prädikat `methodT(M, _, _, _, _, _, _, _, _, _)` erweitert, dass die Belegung von `M` auf Methoden gemäß des deklarierten Metatyps reduziert; analog wird ein Prädikat für den Metatyp von `F` ergänzt. Letztlich sind die beiden Prädikate an dieser Stelle aber weitgehend redundant, da `M` und `F` bereits durch ihre Verwendung in angegebenen Prädikaten *assignopT* und *getFieldT* ausreichend eingeschränkt sind.

Low-Level-Queries ermöglichen den Zugriff auf alle Details der Faktenbasis und bieten dabei die volle Flexibilität und Mächtigkeit der logischen Programmierung. Allerdings erfordert die Definition einer Low-Level-Query auch Programmierkenntnisse in Prolog und ein gutes Verständnis der ASG-Struktur. Im Gegensatz dazu basieren High-Level-Queries auf der Kombination anderer Queries; sie bilden eine Abstraktionsebene, die im i.A. leichter verständlich ist, allerdings auch nicht so ausdrucksmächtig. Um den Programmierer eines GOT-Programms beim Verfassen von Queries zu unterstützen, sollten möglichst umfassende Bibliotheken von allgemeinen Queries vordefiniert sein. Sind ausreichend viele Grundbausteine für Queries vorhanden, so kann der Programmierer gängige Anwendungsfälle allein durch die Definition von High-Level-Queries umsetzen, ohne selbst gefordert zu sein, Low-Level-Queries zu verfassen.

Optionale Parameter

Prolog kann eine Query nur dann positiv beantworten, wenn eine Belegung für alle freien Variablen gefunden werden kann. In einem GOT-Programm möchten wir aber in einigen Fällen auch dann ein Ergebnis erhalten, wenn der ein oder andere Parameter nicht belegt ist. Beispielsweise möchten wir eine Menge von Klassen mit bestimmten Eigenschaften identifizieren und dazu alle deklarierten Felder der Klassen erhalten. Eine Lösung der entsprechenden Query bildet also ein Tupel von einer Klasse und einem Feld. In diesem Fall könnte eine Klasse, die zwar die gewünschten Eigenschaften besitzt, aber keine Felder deklariert, nicht in die Lösungsmenge aufgenommen werden, da keine positive Belegung für den Parameter, der das Feld repräsentiert, existiert. Um solche Anforderungen umsetzen zu können, erlauben wir die Belegung der Parameter mit Null-Werten. Eine Beispiel-Query für den dargestellten Fall ist in Listing 4.4 gezeigt. Die Query belegt die Metavariable `?f` durch `fieldOfClass` mit den Feldern der Klasse des Parameters `?c`. Darüber hinaus belegt sie `?f` durch `optionalField` mit einem Null-Wert. Somit ist sichergestellt, dass die Query in jedem Fall mindestens eine Belegung für `?f` findet.

```
1 public otquery classesAndFields(?Class ?c, ?Field ?f) {
2   fieldOfClass(?f, ?c) || optionalField(?f)
3 }
```

Listing 4.4: Eine Query mit einem optionalen Parameter

Null-Werte ermöglichen die partielle Belegung der transformationsrelevanten Metavariablen. Sie werden im weiteren Prozess der Transformation so behandelt, als wäre keine Belegung vorhanden.

Beispiel für eine komplexe Query

Die Mächtigkeit und Eleganz von Prolog-Abfragen in GOT möchten wir am Beispiel der Query

```
otquery reachable(?Method ?caller, ?Method ?callee)
```

die im Beispiel in Listing 4.2 verwendet wurde, demonstrieren. Die Query soll eine Aussage darüber machen, ob eine bestimmte Methode, die wir *Callee* nennen, im Kontrollfluss der Ausführung einer anderen Methode, genannt *Caller*, liegt. Dies ist der Fall, wenn die Methode *Caller* einen direkten Aufruf der Methode *Callee* enthält oder wenn sie eine andere Methode aufruft, die wiederum direkt oder über weitere Methoden zu einem Aufruf von *Callee* führen kann. Die Query erfordert also einen transitiven Abschluss. Der Informationsgehalt der Query entspricht dem eines statischen Call-Graphen!

Wir definieren die Query `reachable` als Low-Level-Query mit mehreren Prolog-Regeln. Alle dazu erforderlichen Regeln sind in Listing 4.5 aufgeführt.

```
1  calls(M1, M2) :-
2    callT(_, M1, _, _, _, M2, _, _).
3  ... % weitere calls-Regeln
4  callsDeep(Caller, Callee, Visited) :-
5    calls(Caller, Callee).
6  callsDeep(Caller, Callee, Visited) :-
7    calls(Caller, M),
8    not(member(M, Visited)),
9    callsDeep(M, Callee,  [M|Visited]).
10 reachable(Caller, Callee) :-
11   callsDeep(Caller, Callee, []).
```

Listing 4.5: Beispiel für eine komplexe Regel in Prolog

Eine notwendige Grundlage für komplexere Analysen ist zunächst einmal die einfache Information, ob eine Methode einen direkten Aufruf zu einer anderen Methode besitzt. Eine Aussage hierzu sei durch die zweistellige Regel `calls` realisiert (Z. 1):

```
calls(M1, M2) :- callT(_, M1, _, _, _, M2, _, _).
```

Die Regel ist über einen Fakt *callT*, der den ASG-Knoten für einen Methodenaufruf repräsentiert, definiert (Z. 2). Vom Prädikat *callT* sind hier lediglich zwei Argumente von Bedeutung: die aufrufende (zweites Argument) und die aufgerufene Methode (sechstes Argument). Der Fakt steht also für einen Aufruf an einer Methode `M2`, der innerhalb des Rumpfes einer Methode `M1` notiert ist. Die Regel `calls` erfasst in ihrer jetzigen Form nur die statisch deklarierte Version der Methode `M2`. Durch dynamisches Binden könnten an dieser Stelle jedoch auch jene Methoden, die `M2` überschreiben, zur Ausführung kommen. Es müssen also weitere Regeln zu `calls` formuliert werden, die das dynamische Binden abdecken – diese sind aber nicht mehr Bestandteil des Beispiels, der Fokus hier liegt in der Auflösung des Call-Graphen.

Um nun die Menge der Methoden, die sowohl direkt als auch indirekt von einer bestimmten Methode ausgehend aufgerufen werden, angeben zu können, wird das dreistellige Prädikat `callsDeep` formuliert. Die Schwierigkeit liegt hier darin, dass der Call-Graph Zyklen enthalten kann, da Methoden sich ohne Weiteres gegenseitig aufrufen können. Eine naive Baumtraversierung, wobei alle Elemente entlang eines Pfades aufgesammelt werden, führt daher nicht zum Erfolg. Bei der Traversierung des Call-Graphen muss zur Vermeidung von unendlichen Zyklen ein Suchzweig abgebrochen werden, sobald ein Knoten erreicht wird, der zuvor schon besucht wurde. Das Prädikat `callsDeep` wird daher mit drei Argumenten spezifiziert: neben den augenscheinlichen Argumenten `Caller` für die aufrufende und `Callee` für die erreichbare Methode gibt es das Argument `Visited`, das eine Liste aller bislang besuchten Knoten des Graphen enthält.

Wir definieren `callsDeep` mit insgesamt zwei Regeln für eine rekursive Suche. Die erste Regel (Zz. 4–5) ist definiert als:

```
callsDeep(Caller, Callee, Visited) :-
  calls(Caller, Callee).
```

Diese Regel deckt den Fall ab, dass ein direkter Aufruf an `Callee` innerhalb von `Caller` existiert. Die Liste der besuchten Knoten, `Visited`, ist für diesen Fall irrelevant. Die Regel bildet den Rekursionsanker für die rekursive zweite Regel.

Die zweite Regel (Zz. 6–9) ist definiert als:

```
callsDeep(Caller, Callee, Visited) :-
  calls(Caller, M),
  not(member(M, Visited)),
  callsDeep(M, Callee,  [M|Visited]).
```

Diese Regel deckt den Fall ab, dass in `Caller` ein Aufruf an einer beliebigen Methode `M` existiert – formuliert durch den ersten Term des Regelkörpers. `M` kann dabei nicht gleich `Callee` sein, da dieser Fall bereits durch die erste Regel, die eine höhere Ausführungspriorität besitzt, abgedeckt ist. Zur Vermeidung von Zyklen soll gelten, dass der Knoten `M` im Graph noch nicht besucht wurde, d.h. nicht Element der Liste der besuchten Knoten, `Visited`, ist (zweiter Term). Die Methode `M` dient in diesem Fall als Ausgangspunkt für die rekursive Fortführung der Tiefensuche nach einem Pfad zu `Callee`, wobei `Visited` entsprechend um `M` ergänzt wird (dritter Term).

Die eigentliche Regel `reachable` (Zz. 10–11) ist nun definiert als:

```
reachable(Caller, Callee) :-
  callsDeep(Caller, Callee, []).
```

Sie dient lediglich der Initialisierung der rekursiven Suche durch `callsDeep` mit einer anfangs leeren Liste von besuchten Knoten.

4.2.4 Query-Klassen

Nachdem Deklaration und Definition von Queries erläutert wurden, bleibt die Frage zu klären, in welcher Form Queries in die Modulstruktur integriert werden. Da Queries nur für die Code-Transformation relevant sind und danach keine Bedeutung mehr haben, sollten sie klar von den übrigen Bestandteilen eines Programms getrennt werden. Zugleich sollen sie aber die gewohnte Java-Struktur beibehalten. Es erscheint uns daher naheliegend, die Idee der Klasse, als übergeordnetes Modul für Methoden, auf Queries zu übertragen. Entsprechend führen wir Query-Klassen als neues Modul ein. Query-Klassen werden wie Query-Methoden mit dem Schlüsselwort `otquery` notiert. Von Query-Klassen können keine Instanzen gebildet werden. Sie sind vergleichbar mit Utility-Klassen, wie z.B. `java.lang.Math`, die als Bibliothek statischer Methoden dienen. Einzig erlaubter Bestandteil einer Query-Klasse sind Query-Methoden. Das folgende Listing 4.6 zeigt das Beispiel einer Query-Klasse `StdQ`, welche die zuvor vorgestellten Low-Level-Queries `calls` und `reachable` enthält.

```
1  public otquery class StdQ {
2    @Prolog("calls(M1, M2) :-
3      callT(_, M1, _, _, _, M2, _, _).")
4    public otquery calls(?Method ?caller, ?Method ?callee) {}
5
6    @Prologs({
7      @Prolog("callsDeep(Caller, Callee, Visited) :-
```

```
8         calls(Caller, Callee).");
9       @Prolog("callsDeep(Caller, Callee, Visited) :-
10        calls(Caller, M),
11        not(member(M, Visited)),
12        callsDeep(M, Callee,  [M|Visited]).");
13      @Prolog("reachable(Caller, Callee) :-
14        callsDeep(Caller, Callee, []).")
15    })
16    public otquery reachable(?Method ?caller, ?Method
          ?callee) {}
17  }
```

Listing 4.6: Eine Query-Klasse

Die einzige Möglichkeit zur Definition einer Query außerhalb von Query-Klassen besteht bei der Anwendung in einer generischen Teamklasse, wie nachfolgend in Abschnitt 4.3 erläutert wird.

4.2.5 Auswertung der Queries

Die Auswertung einer Query geschieht im Rahmen des Transformationsprozesses. Der Teilprozess der Auswertung ist in Abbildung 4.3 skizziert.

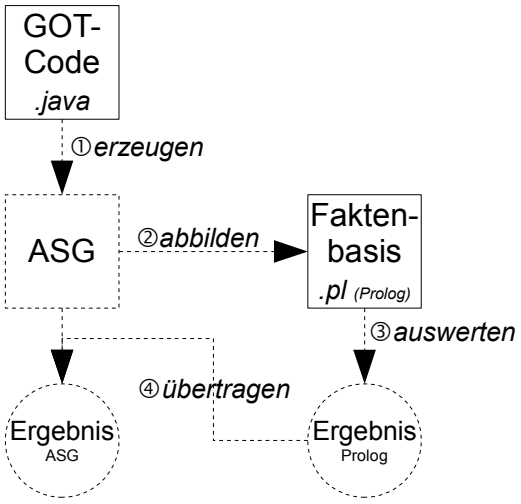

Abbildung 4.3: Auswertung der Queries im Rahmen der Transformation

1. Zu Beginn wird der ASG des GOT-Programms erzeugt. Der ASG wird durch Objekte abgebildet, welche die Elemente des Metamodells der Programmiersprache repräsentieren.

2. Der ASG wird auf Prolog-Fakten abgebildet, wie in Abschnitt 4.1 beschrieben. Dabei bekommen alle Objekte des ASG eine eindeutige ID zugeordnet. Queries werden in Form von Regeln abgebildet.

3. Die Regel, welche die in Frage kommende Query abbildet, wird mittels einer Prolog-Anfrage ausgewertet. Die freien Variablen der Regel werden dabei mit einer Menge von Werten belegt. Im Fall von ASG-Knoten handelt es sich also um eine Menge von IDs.

4. Die Ergebnisse werden von Prolog zurück nach GOT übertragen, wobei sie auf Werte aus dem Wertebereich der Metatypen der deklarierenden Metavariablen abgebildet werden. Hierbei handelt es sich entweder um Objekte, die ASG-Knoten der jeweiligen IDs repräsentieren, oder um einfache Werte der Typen Boolean, Integer oder String.

Die Bedeutung des Ergebnisses der Query-Auswertung wird nachfolgend in Rahmen der Beschreibung der Transformation in Abschnitt 4.3 erläutert.

4.3 Grundlegende Transformation

Die in Abschnitt 4.2 eingeführten Metavariablen finden Anwendung in generischen Teamklassen (die fortan auch einfach *generische Teams* genannt werden). Die übrigen Module werden von der Adaption durch GOT nicht beeinflusst. Ein generisches Team kann als Template für eine normale Teamklasse interpretiert werden, die neben dem normalen Code auch parametrisierte Code-Stellen enthält. Bei diesen Code-Stellen handelt es sich primär um die Bindungspunkte zwischen Rollen- und Basisklassen. Als Parameter dienen die Metavariablen. Bei der Transformation werden alle generischen Code-Anteile aufgelöst, d.h. jede generische Teamklasse wird zu einer nicht-generischen Teamklasse umgeformt. Die Teamklasse, auf welche die Transformation angewendet wird, wird fortan auch *Eingangsteam* genannt; die aus der Transformation resultierende Teamklasse wird entsprechend *Ausgangsteam* genannt.

In diesem Abschnitt wird die Transformation generischer Teams im Detail beschrieben. Zum besseren Verständnis wird die Beschreibung sowohl von einem konkreten Beispiel als auch von einer abstrakten Spezifikation in Object-Z [29] begleitet.

Zur Veranschaulichung wird das in Abschnitt 3.1.1 vorgestellte Beispiel zum Observer Pattern aufgegriffen. Ziel ist es, eine wiederverwendbare Implementierung der Observer-Funktionalität zu realisieren, so dass dessen Update-Mechanismus automatisch durch die relevanten Zustandsänderungen der Subjekte ausgelöst wird. Als konkretes Beispiel wird eine Teamklasse `MyShapeDisplay` implementiert, die eine Menge von Objekten der Klassen `Circle` und `Rectangle` grafisch darstellt (skizziert als Klassendiagramm in Abbildung 4.4).

Der Observer `MyShapeDisplay` besitzt eine Methode `update`, in der die Methode `paint` zum Zeichnen der Objekte aufgerufen wird. Wir nehmen dabei an, dass sowohl die spezifischen Attribute `radius` bzw. `width` und `height` der Klassen `Circle` und `Rectangle`, als auch das geerbte Attribut `color` in der jeweils überschriebenen Variante der Methode `paint` verwendet wird. Diese Attribute sind entsprechend jene, deren Zustandsänderungen ein Update des Observers auslösen sollen. Das geerbte Attribut `id` dagegen habe keinen Einfluss auf die grafische Darstellung. Eine Änderung hier soll daher nicht zum Update des Oberservers führen.

In diesem Abschnitt wird zunächst die grundlegende Abstraktion der Entwurfsmuster-Funktionalität in GOT mit Bezug auf den konkreten Anwendungsfall der Klasse `MyShapeDisplay` beschrieben. Im folgenden Abschnitt 4.4 wird diese Lösung dann verallgemeinert, so dass die Implementierung für andere Anwendungen des Entwurfsmusters wiederverwendet werden kann.

4.3.1 Teamquery

Eine Teamklasse ist generisch, wenn ihre Deklaration um eine anonyme Query ergänzt wird, die durch eines der beiden Schlüsselwörter `match` oder `declare` eingeleitet wird (der Unterschied ist an dieser Stelle noch nicht relevant, er wird in Abschnitt 4.4 erläutert). Diese Query wird im Folgenden als *Teamquery* bezeichnet.

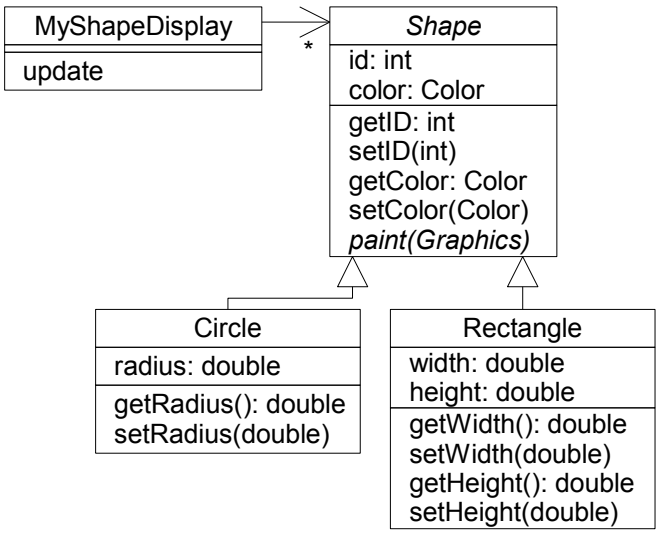

Abbildung 4.4: Eine konkrete Anwendung des Observer Patterns

Eine Teamquery bildet den Ausgangspunkt der Generizität in GOT. Sie deklariert und definiert Metavariablen, die im Rumpf der Teamklasse verwendet werden können. Die Auswertung der Teamqueries aller generischen Teams eines Programms erfolgt zu Beginn des Transformationsprozesses, wie in Abschnitt 4.2.5 beschrieben. Ein Beispiel für ein generisches Team ist in Listing 4.7 gezeigt.

```
1  team class MyShapeDisplay
2    match(?Class ?base, ?Method ?writer) {
3      ?Field -f;
4      ?Method -update;
5      StdQ.isAncestor(Shape.class, ?base) &&
6      StdQ.methodOfNameAndClass(-update, "paint", ?base) &&
7      StdQ.accesses(-update, -f) &&
8      StdQ.changes(?writer, -f) &&
9      StdQ.memberMethod(?writer, ?base)
10   }
11 {
12   ...
13 }
```

Listing 4.7: Ein generisches Team mit einer Teamquery

Von der Teamklasse `MyShapeDisplay` wird an dieser Stelle zunächst nur die Teamquery dargestellt – eingeleitet durch das Schlüsselwort `match`. Das Ziel der Teamquery ist es, all jene Programmelemente zu definieren, die über quantifizierende Ausdrücke in die Implementierung des Teams, d.h. vor allem zum Zwecke des Rollenspiels, integriert werden sollen. Im konkreten Beispiel zum Observer Pattern ist Quantifizierung in dreierlei Hinsicht erforderlich:

1. Zunächst müssen alle Klassen, welche die Rolle des Subjekts spielen sollen, ausgewählt werden.

2. Von diesen Klassen sind all jene Felder zu identifizieren, die zum relevanten Zustand (für den beobachtenden Observer) gehören.

3. Nun sind jene Methoden für den Callin der Methode `update` auszuwählen, die Änderungen an besagten Feldern hervorrufen können.

Punkt 2 ist hierbei nur ein Zwischenergebnis. Für die eigentliche Implementierung der Rolle `Subject` ist die Auswahl der Basisklassen (Punkt 1) und der Methoden (Punkt 3) hinreichend. Entsprechend deklariert die Teamquery die beiden Metavariablen `?base` und `?writer`, die Paare von Basisklassen und -methoden bilden.

Die Teamquery besteht aus der Deklaration zweier lokaler Metavariablen (Zz. 3–4) und fünf miteinander ver*und*eter Queries (Zz. 5–9). Die verwendeten Queries sind abgesehen von den Queries `accesses` und `changes` trivial, weswegen wir hier auf eine Angabe der Implementierung verzichten. Die Query

```
isAncestor(Shape.class, ?base)
```

bekommt als erstes Argument die konkrete Klasse `Shape` vorgegeben, welche die Oberklasse für Subjekte unter Beobachtung darstellt. Sie belegt das zweite Argument, die Metavariable `?base`, entsprechend mit den Subklassen von `Shape`. Im Beispiel sind dies die Klassen `Circle` und `Rectangle`. Mit dieser Query ist Punkt 1 bereits erledigt.

Zur Umsetzung von Punkt 2 muss zunächst die Update-Methode vorgegeben werden. Im Beispiel handelt es sich um die Methode `paint`, da die Klasse `MyShapeDisplay` im Rahmen eines Updates die Shape-Objekte neu zeichnen soll. Die Methode wird in der Query

```
methodOfNameAndClass(-update, "paint", ?base)
```

anhand ihres Namens ausgewählt und an die lokale Metavariable `-update` gebunden. Wichtig an dieser Stelle ist, dass jeweils die spezifische `paint`-Methode der jeweiligen Basisklasse gewählt wird, da sich die Implementierungen hier natürlich unterscheiden können. Auf Basis dieser Methode werden mit der Query

```
accesses(-update, -f)
```

alle Felder identifiziert, die im Rahmen der Ausführung der Methode `-update` auf direktem oder indirektem Wege gelesen werden können. Dies sind also genau die Felder, deren Werte Einfluss auf die Ausführung des Updates haben; damit ist Punkt 2 behandelt. Die Query `accesses` sei dabei analog zur Query `changes`, deren Implementierung in Abschnitt 4.2.3 vorgestellt wurde, für lesende Zugriffe umgesetzt.

Darauf aufbauend werden mit der Query

```
changes(?writer, -f)
```

all jene Methoden identifiziert, welche einen direkten oder indirekten Schreibzugriff auf diese Felder haben, d.h. eine Zustandsveränderung bewirken können.

Letztlich müssen diese Methoden zur Sicherheit noch auf die Zugehörigkeit zur Basisklasse gefiltert werden, da sie ansonsten natürlich nicht für eine Callin-Bindung herangezogen werden können. Dies geschieht mithilfe der Query

```
memberMethod(?writer, ?base)
```

Diese Einschränkung hat allerdings nur Auswirkungen, falls nicht-private Felder involviert sind. Solche Felder müssten gesondert behandelt werden, worauf wir an dieser Stelle aber verzichten, da eine derartige Konstellation den üblichen Konventionen widerspricht und eine Ausnahme darstellt.

Im Ergebnis wird die Teamquery der Klasse `MyShapeDisplay` Paare von Klassen und Methoden liefern. Bei den Klassen werden Subklassen der Klasse `Shape` ausgewählt, bei den Methoden werden solche ausgewählt, die eine Zustandsänderung an einem Attribut bewirken, das für die Methode `paint` relevant ist. Im Beispiel

ist dies für die Attribute `color`, `radius`, `width` und `height` der Fall. Die Ergebnismenge der Teamquery besteht somit aus den folgenden fünf Paaren von Klassen und Methoden:

1. *(Circle, setRadius)*
2. *(Circle, setColor)*
3. *(Rectangle, setWidth)*
4. *(Rectangle, setHeight)*
5. *(Rectangle, setColor)*

Nachfolgend geben wir eine formale Spezifikation generischer Teams in Object-Z an. Dafür führen wir zunächst die Basistypen

$$[MVAR, MTYPE, MVALUE]$$

ein. $MVAR$ repräsentiere die Menge der Metavariablen und $MTYPE$ die Menge der Metatypen, mit denen eine Metavariable deklariert werden darf. $MVALUE$ sei die Menge aller Belegungen für Metavariablen (im praktischen Sinn entspricht dies der Menge der ASG-Knoten eines Programms, zzgl. der Werte für Basistypen wie `?int` und `?string`). Zwischen den Mengen $MVAR$, $MTYPE$ und $MVALUE$ werden Relationen definiert, welche die übliche Bedeutung statischer Typisierung spezifizieren, skizziert in Abbildung 4.5.

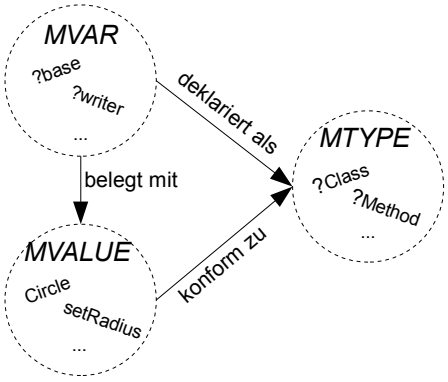

Abbildung 4.5: Die Mengen $MVAR$, $MTYPE$ und $MVALUE$ zur Spezifikation statischer Typisierung

Jede Metavariable einer Query ist deklariert mit einem Metatyp, es gilt daher eine Abbildung

$$query : MVAR \nrightarrow MTYPE$$

Die Funktion ist partiell mit endlichem Definitionsbereich ($\nrightarrow$), da eine Query nur eine begrenzte Menge von Metavariablen deklariert.

Das Ergebnis einer Query besteht aus einer Menge von Lösungen. Jede dieser Lösungen entspricht einer Abbildung jeder Metavariablen der Query auf jeweils eine konkrete Belegung. Eine Lösung kann daher definiert werden als eine Funktion

$$SOLUTION == MVAR \nrightarrow MVALUE$$

die eine konkrete Belegung für eine Metavariable liefert. Die Funktion ist ebenfalls partiell, da eine Lösung nur für die Menge der deklarierten Metavariablen der zugehörigen Teamquery definiert ist. Beispielsweise ist

$\{?base \mapsto Circle, ?writer \mapsto setRadius\}$

eine Lösung aus der Ergebnismenge der Teamquery in Listing 4.7. Die dadurch gegebene partielle Funktion bildet Werte aus der Definitionsmenge $\{?base, ?writer\}$, die den Parametern der Query entspricht, auf Werte der Wertemenge $\{Circle, setRadius\}$ ab.

Für die Belegung von Metavariablen gelte die allgemeine Konformitätsbeziehung

$$conform : MVALUE \leftrightarrow MTYPE$$

welche die nominelle Konformität zwischen einem Wert und einem Metatyp definiert.

Ein generisches Team sei damit folgendermaßen spezifiziert[13]:

GenericTeam

$$teamquery : MVAR \twoheadrightarrow MTYPE$$
$$solutions : \mathbb{F}\, SOLUTION$$

$$\forall sol : solutions \bullet$$
$$(\mathrm{dom}\, sol = \mathrm{dom}\, teamquery \land$$
$$\forall (var, value) : sol \bullet value \in \mathrm{dom}(conform \rhd \{teamquery(var)\}))$$

Das Zustandsschema der Klasse beschreibt die Teamquery als eine Abbildung, die eine begrenzte Menge von Metavariablen mit ihren deklarierten Metatypen assoziiert. Die Menge der Lösungen nach Auswertung der Teamquery sei in der Zustandsvariablen *solutions* erfasst. Die Lösungsmenge einer Query ist grundsätzlich endlich, da sie eine Auswahl aus einer endlichen Menge von Programmelementen repräsentiert. Für jede dieser Lösungen gilt, dass sie vollständig und konform in Bezug zur Deklaration der Teamquery ist:

- **Vollständig**. Die Lösung definiert für jede Metavariable der Teamquery eine Belegung, da für beide Abbildungen die gleichen Definitionsbereiche gefordert sind ($\mathrm{dom}\, sol = \mathrm{dom}\, teamquery$).
- **Konform**. Jede Belegung einer Metavariablen der Lösung ist konform zu deren deklariertem Metatyp ($conform \rhd \{teamquery(var)\}$ beschreibt die Beschränkung der Menge *conform* auf jene Abbildungen $x \mapsto y$, für die gilt $y = teamquery(var)$, d.h. y ist der deklarierte Metatyp der Metavariable)[14].

4.3.2 Generizität

Im Folgenden wird erläutert, in welcher Gestalt generische Programmelemente im Code eines GOT-Programms vorkommen dürfen und wie diese transformiert werden. Zum besseren Verständnis betrachten wir dazu eine abstrahierte Form der Java bzw. GOT-Syntax, in der eine Klasse als eine simple Folge von Anweisungen beschrieben wird. Die Struktur der Anweisungen ist dargestellt als EBNF:

```
statement = simple-statement | block-statement ;
simple-statement = code , ";" ;
block-statement = code , "{" , {statement} , "}" ;
code = ? ... ? ;
```

[13] Zur besseren Lesbarkeit erlauben wir in der Notation eine Kurzschreibeweise zur impliziten Variablendeklaration durch Tupel. Sei beispielsweise $R : \mathbb{P}(X_1 \times ... \times X_n)$, dann stehe die Deklaration $(x_1, ..., x_n) : R$ implizit für $x_1 : X_1;\ ...;\ x_n : X_n \mid (x_1, ..., x_n) \in R$.

[14] Der Object-Z-Operator $\rhd$ dient der Restriktion des Wertebereichs (*Range Restriction*). Sei $S : X \leftrightarrow Y$ und $R : \mathbb{P}\, Y$, dann gelte $S \rhd R = \{x : X, y : Y \mid y \in R \land x \mapsto y \in S\}$.

Bei Anweisungen (*statement*) wird lediglich zwischen einfachen (*simple-statement*) und hierarchisch aufgebauten Anweisungen (*block-statement*) unterschieden.

Eine einfache Anweisung besteht aus einem Stück Code, das von einem Semikolon abgeschlossen wird (nach dem Grundsatz *„Ein Semikolon beendet eine Anweisung“*). Dabei kann es sich z.B. um eine Zuweisung oder einen Methodenaufruf handeln. Auch die Deklaration eines Attributs oder einer Variablen zählt als einfache Anweisung.

Eine hierarchisch aufgebaute Anweisung (nachfolgend auch *Blockanweisung* genannt) besteht aus einem Stück Code, gefolgt von einem Block von Anweisungen. Zu den Blockanweisungen zählen z.B. Klassen- und Methodendefinitionen sowie Schleifen. Die Abstraktion der Blockanweisung gilt auch für Anweisungen, die syntaktisch mehrere Blöcke (und Code-Stücke) definieren können, wie z.B. *if-then-else*-Anweisungen. Für die nachfolgende Betrachtung zur Generizität ist lediglich die Schachtelung von Anweisungen von Bedeutung, ob dabei eine Aufteilung in einen oder mehrere Blöcke vorliegt ist unerheblich. Gleiches gilt auch für die Gestalt eines Code-Stücks (*code*), das ganz allgemein jegliche Formen von Definitionen und Ausdrücken repräsentiert. Auf die syntaktischen Details und Besonderheiten einzelner Anweisungen wird im Rahmen der allgemeinen Betrachtung nicht weiter eingegangen, da sie für den grundlegenden Mechanismus nicht relevant sind.

Zur Steuerung der Transformation wird in GOT der *Per-Block* als Kontrollstruktur eingeführt. Ein Per-Block umschließt eine Sequenz von Anweisungen im Rumpf eines generischen Teams und dient als Rahmen zur Anwendung von Generizität. Er wird durch das Schlüsselwort `per` eingeleitet, gefolgt von einer nicht-leeren Liste von Argumenten. Als Argumente sind ausschließlich Metavariablen erlaubt, die von der Teamquery deklariert werden. Innerhalb eines Per-Blocks können generische Programmelemente eingesetzt werden. Es werden konkret drei Arten von generischen Elementen unterschieden:

- *Metavariablen*: sie können anstelle von sonst konkret zu nennenden Programmelementen wie z.B. Klassen, Methoden und Feldern aufgeführt werden. Primär kommen hier die Bindungsrelationen zwischen Rolle und Basis in Frage. Es dürfen nur solche Metavariablen verwendet werden, die als Argument eines umschließenden Per-Blocks genannt werden.
- *Generische Entitäten*: eine Rollenklasse, eine Methode, ein Attribut oder eine Variable dürfen innerhalb des Per-Blocks als *generisch* deklariert werden, indem der Bezeichner um das Präfix „-“ erweitert wird – wie bei ungebundenen Metavariablen. Sollte das Code-Stück der entsprechenden Anweisung zur Deklaration der Entität eine Metavariable enthalten, dann muss die Entität zwingend generisch deklariert werden. Generische Entitäten sind neben ihrer Bedeutung für die Transformation semantisch äquivalent zu ihren nichtgenerischen Gegenstücken, mit der Ausnahme, dass ihr Gültigkeitsbereich auf den umschließenden Per-Block beschränkt ist.
- *Generische Anweisungen*: jede Anweisung, deren Code eine generische Entität oder eine Metavariable involviert, wird als generische Anweisung aufgefasst. Für Blockanweisungen gilt dies nur, sofern sich das generische Element im Code-Stück der Anweisung selbst befindet, d.h. außerhalb des Blocks. Generische Elemente, die in Anweisungen innerhalb des Blocks auftauchen, machen die Blockanweisung nicht generisch. Ein Per-Block zählt immer zu den generischen Blockanweisungen.

Die verschiedenen Arten von generischen Programmelementen sind beispielhaft im folgenden Programmfragment zu sehen, das zwei ineinander verschachtelte Per-Blöcke zeigt:

```
per (?base) {
  class -Subject playedBy ?base {
    void notify() {
      System.out.println("Notify by " + ?base.class);
      update();
    }
    per(?writer) {
      notify <- after ?writer;
    }
  }
}
```

Das Programmfragment besteht aus insgesamt sieben Anweisungen, deren Struktur in Abbildung 4.6 dargestellt ist. Jede Anweisung ist mir ihrem Code abgebildet. Abgesehen von den Anweisungen 3 und 5 handelt es sich bei allen um generische Anweisungen.

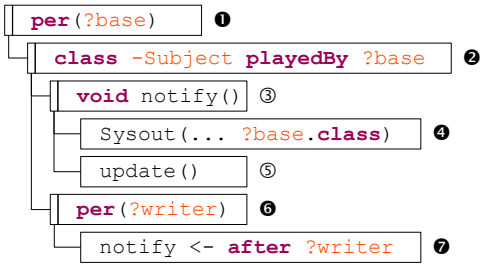

Abbildung 4.6: Ein Beispiel generischer und nicht-generischer Anweisungen

Zu den Anweisungen im Einzelnen:

1. Der äußere Per-Block ist eine Blockanweisung. Die Anweisung ist generisch (wie alle Per-Blöcke), da sie die Metavariable `?base` als Argument besitzt.

2. Als einzige Anweisung innerhalb des Per-Blocks wird eine generische Rollenklasse `-Subject` deklariert, ersichtlich am Präfix „-“. Anstelle einer konkreten Basisklasse wird die Metavariable `?base` genannt. Da bei der Deklaration der Rolle eine Metavariable zum Einsatz kommt, muss die Rolle zwingend als generische Entität deklariert werden.

3. Im Rumpf der Rollenklasse wird eine Methode `notify` definiert. Die Methode ist im Gegensatz zur Rolle nicht als generische Entität deklariert. Das ist auch nicht erforderlich, da bei ihrer Deklaration keine Metavariablen oder generische Entitäten involviert sind. Die Methode `notify` ist somit eine nicht-generische Blockanweisung. Im Rumpf der Methode, der mit einer Sequenz von zwei Anweisungen definiert ist, dürfen trotzdem generische Anweisungen enthalten sein.

4. Der Methodenaufruf `System.out.println` (kurz als `Sysout` dargestellt) besitzt als Argument einen Ausdruck, der die Metavariable `?base` involviert. Somit handelt es sich um eine einfache, generische Anweisung.

5. Der Methodenaufruf `update` wird ohne den Einsatz generischer Programmelemente definiert und gilt daher als einfache, nicht-generische Anweisung.

6. Im Rumpf der Rollenklasse ist ein weiterer Per-Block mit dem Argument `?writer` definiert, analog zu Anweisung 1.

7. Innerhalb des inneren Per-Blocks gibt es eine Callin-Anweisung. Diese Anweisung gilt als generisch, da zu ihrer Definition die Metavariable `?writer` eingesetzt wird.

4.3.3 Rollout

Nicht-generische Anweisungen eines Eingangsteams werden im Rahmen der Transformation nicht verändert, sie werden in identischer Form in das Ausgangsteam übernommen. Generische Anweisungen dagegen werden transformiert und durch konkrete, nicht-generische Anweisungen ersetzt. Ihr Vorkommen ist ausschließlich auf die Per-Blöcke beschränkt, außerhalb eines Per-Blocks dürfen sie grundsätzlich nicht aufgeführt werden. Der Einfluss der Transformation ist so klar begrenzt. Ein Per-Block bildet ein Template, das eigenständig transformiert wird. Der Transformationsprozess eines Per-Blocks wird als *Rollout* bezeichnet. Wir beschreiben den Rollout zunächst informell und werden ihn im Anschluss mit Object-Z spezifizieren.

Kernidee des Rollouts ist, dass für jede unterschiedliche Belegung der Argumente eines Per-Blocks, dessen generische Anweisungen einmal expandiert werden. Der Per-Block stellt also einen Ausdruck dar, der über seine Argumente quantifiziert. Die konkrete Belegung der Metavariablen ist durch die Teamquery gegeben (spezifiziert als $solutions : \mathbb{F}\, SOLUTION$). Im Beispiel (aus Abschnitt 4.3.1) erhalten wir für das Metavariablen-Paar `(?base, ?writer)` die Ergebnismenge

$$S = \{S1, S2, S3, S4, S5\}$$

mit $S : \mathbb{F}\, SOLUTION$ und

$$\begin{aligned}
\text{S1} &= \{?base \mapsto Circle, ?writer \mapsto setRadius\} \\
\text{S2} &= \{?base \mapsto Circle, ?writer \mapsto setColor\} \\
\text{S3} &= \{?base \mapsto Rectangle, ?writer \mapsto setWidth\} \\
\text{S4} &= \{?base \mapsto Rectangle, ?writer \mapsto setHeight\} \\
\text{S5} &= \{?base \mapsto Rectangle, ?writer \mapsto setColor\}
\end{aligned}$$

Der folgende Per-Block eines Eingangsteams

```
per (?base) {
  class -Subject playedBy ?base {...}
}
```

beschreibt, dass je gefundener Basisklasse `?base`, der Inhalt des Per-Blocks einmal expandiert werden soll, d.h. im Ausgangsteam eine Rollenklasse nach dem Schema der Klasse `-Subject` erzeugt werden soll. Dabei wird das Vorkommen der Metavariablen `?base` durch die konkrete Belegung ersetzt. In der Ergebnismenge S existieren zwei unterschiedliche Belegungen für `?base`: die Klassen `Circle` und `Rectangle`. Der Per-Block soll entsprechend zwei Rollenklassen im Ausgangsteam generieren. Der generierte Inhalt einer einzelnen Expansion des Per-Blocks wird als *Instanz* bezeichnet. In jeder Instanz werden die Bezeichner generischer Entitäten durch einen eindeutigen, nicht-generischen Bezeichner ersetzt. Damit ergeben sich die folgenden beiden Rollenklassen:

```
class CircleSubject playedBy Circle {...}
class RectangleSubject playedBy Rectangle {...}
```

Der Rollout des Per-Blocks ist abgeschlossen, wenn alle Instanzen generiert wurden.

In der Literatur wird diese Form der Generizität als *heterogen* bezeichnet [16]. Sie ist dadurch gekennzeichnet, dass für jede Belegung der Parameter eine Kopie des generischen Codefragments erzeugt wird, die dann entsprechend transformiert wird.

Allgemein formuliert muss für den Rollout eines Per-Blocks die Ergebnismenge S in *charakteristische Teilmengen* für die Metavariablen des Per-Blocks unterteilt werden. Eine charakteristische Teilmenge S_C in Bezug auf eine Menge von Metavariablen sei jene Teilmenge von S, deren Lösungen alle die gleiche Belegung für diese Metavariablen besitzen. Eine formale Definition geben wir nachfolgend im Rahmen der Object-Z-Spezifikation. Im Beispiel ergeben sich zwei charakteristische Teilmengen von Lösungen für die Metavariable `?base`:

$$\begin{aligned} S_{Circle} &= \{sol : S \mid sol(?base) = Circle\} &&= \{S1, S2\} \\ S_{Rectangle} &= \{sol : S \mid sol(?base) = Rectangle\} &&= \{S3, S4, S5\} \end{aligned}$$

Je charakteristischer Teilmenge wird der Per-Block einmal expandiert. Ein innerer Per-Block wird im Rahmen des umschließenden Per-Blocks transformiert. Bei der Expansion eines inneren Per-Blocks kommt anstatt der allgemeinen Ergebnismenge S die jeweils charakteristische Teilmenge der äußeren Instanz zur Anwendung. Das folgende Codefragment des Eingangsteams

```
per (?base) {
  class -Subject playedBy ?base {
    void notify() {
      System.out.println(?base.class);
      update();
    }
    per(?writer) {
      notify <- after ?writer;
    }
  }
}
```

generiert somit den folgenden Code im Ausgangsteam

```
class CircleSubject playedBy Circle {
  void notify() {
    System.out.println(Circle.class);
    update();
  }
  notify <- after setRadius;
  notify <- after setColor;
}
class RectangleSubject playedBy Rectangle {
  void notify() {
    System.out.println(Rectangle.class);
    update();
  }
  notify <- after setWidth;
  notify <- after setHeight;
  notify <- after setColor;
}
```

Der äußere Per-Block bildet (wie zuvor erläutert) zwei Instanzen. Eine Instanz besteht aus einer tiefen Kopie der Klasse `-Subject`, der einzigen Anweisung innerhalb des Blocks. Die Klasse selbst enthält einen Block mit zwei inneren Anweisung (die Methode `notify` und ein Per-Block) die ebenfalls kopiert werden. Die generischen Entitäten einer Instanz erhalten einen eindeutigen Bezeichner, die Metavariablen werden durch ihre Belegung ersetzt. Der innere Per-Block, der die generische Callin-Anweisung umschließt, wird wiederum in Abhängigkeit für jede der beiden Instan-

zen auf Basis der charakteristischen Teilmengen expandiert. Im ersten Fall werden somit zwei Instanzen des Callins gebildet, im zweiten Fall drei Instanzen.

An diesem Beispiel wird auch leicht ersichtlich, warum eine Entität, zu deren Definition eine Metavariable verwendet wird, zwingend generisch sein muss. Von der Klasse `-Subject` werden zwei Instanzen gebildet. Jede davon benötigt einen eindeutigen Bezeichner, d.h. der Bezeichner muss transformiert werden. Dies ist erlaubt, da die Klasse als generisch deklariert wurde. Wäre die Klasse dagegen nicht generisch, so dürfte nur eine Instanz der Klasse (mit unverändertem Bezeichner) existieren – die *playedBy*-Beziehung dieser Klasse wäre aber aufgrund der möglichen Belegung mit mehreren Werten nicht eindeutig definiert.

Von besonderer Bedeutung ist auch, dass ausschließlich generische Anweisungen expandiert werden. Nicht-generische Anweisungen eines Per-Blocks bleiben erhalten, ohne dass von ihnen Instanzen erzeugt werden. Mehrere Instanzen der gleichen Anweisung wären nicht sinnvoll und im Fall von Definitionen auch nicht eindeutig (da ein Bezeichner ambivalent definiert wäre) und damit schon syntaktisch nicht erlaubt. Der folgende Per-Block

```
per (?base) {
  void print() {
    System.out.println(?base);
  }
}
```

ist daher äquivalent zur Definition

```
void print() {
  per (?base) {
    System.out.println(?base);
  }
}
```

Die Methode `print` ist nicht als generische Entität deklariert. Es ist daher unerheblich, ob sie vom Per-Block umschlossen wird oder nicht. Beide Codefragmente werden zu folgendem Resultat transformiert:

```
void print() {
  System.out.println(Circle.class);
  System.out.println(Rectangle.class);
}
```

Für die Spezifikation des Rollouts in Object-Z beschreiben wir ein generisches Team in abstrakter Form als eine Folge von Anweisungen, die sequentiell und hierarchisch aufgebaut sind – entsprechend der in Abschnitt 4.3.2 vorgestellten Abstraktion von Anweisungen. Zunächst wird der Basistyp

$$[CODE]$$

eingeführt, der allgemein das eigentliche Codefragment einer Anweisung repräsentiert. Die Funktion

$$isGeneric : CODE \rightarrow \mathbb{B}$$

gibt Auskunft darüber, ob diese Anweisung eine *generische Anweisung* im Sinne der Definition aus Abschnitt 4.3.2 darstellt, d.h. ob zur Definition dieser Anweisung Metavariablen oder generische Entitäten verwendet werden (vgl. auch Abbildung 4.6). Die Funktion

$$transform : CODE \times SOLUTION \rightarrow CODE$$

definiert allgemein die Abbildung eines Codefragments auf eine nicht-generische Darstellung, wobei dessen generische Elemente entsprechend einer konkreten Belegung – gegeben als *SOLUTION* – transformiert werden. Im Fall, dass es sich nicht um generischen Code handelt, liefert die Funktion als Ausgabe den identischen Code zur Eingabe. Beispielsweise wird die generische Anweisung

```
update <- after ?writer;
```

von der Funktion *transform* für die Lösung $\{?base \mapsto Circle, ?writer \mapsto setRadius\}$ zur nicht-generischen Anweisung

```
update <- after setRadius;
```

transformiert. Dabei werden die Metavariablen entsprechend der durch die Lösung definierten Abbildung ersetzt.

Eine einfache Anweisung (im Gegensatz zu einer Blockanweisung) wird nun durch die Klasse *Statement* repräsentiert.

Statement

$code : CODE$

transformCode

$\Delta(code)$
$solutions? : \mathbb{F}\, SOLUTION$

$\exists\, sol : solutions? \bullet code' = transform(code, sol)$
$\neg\, isGeneric(code')$

$rollout \mathrel{\widehat=} transformCode$

Ein Statement besitzt als Zustand lediglich ein Codefragment *code*. Im Rahmen des Rollouts wird dieses Codefragment mittels der Funktion *transformCode* transformiert. Sie erwartet als Eingabeparameter eine charakteristische Teilmenge von Lösungen, für welche die Anweisung transformiert werden soll. Im Anschluss an die Transformation ist sichergestellt, dass diese Anweisung nicht generisch ist.

Die Methode *rollout* bündelt den gesamten Prozess der Transformation einer Anweisung. Sie ist für einfache Anweisungen äquivalent zur Methode *transformCode*, soll aber von Subklassen verfeinert werden.

Im Rahmen eines Rollouts wird für jede charakteristische Teilmenge jeweils eine Instanz einer generischen Anweisung generiert (durch Klonen des Originals) und transformiert. Anschließend wird die generische Anweisung durch die gebildeten Instanzen ersetzt (skizziert in Abbildung 4.7). Die Reihenfolge, in der die neu gebildeten Anweisungen eingefügt werden, ist irrelevant, da sie voneinander unabhängig sind. Ist die Menge der Instanzen leer, so wird die generische Anweisung ohne Ersatz gelöscht.

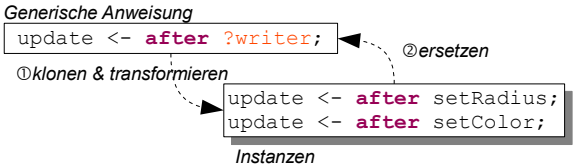

Abbildung 4.7: Rollout einer generischen Anweisung

Die Transformation generischen Codes wurde in der Klasse *Statement* spezifiziert. Das Klonen und Ersetzen eines Statements muss allerdings für dessen Elternelement spezifiziert werden, da die neu erzeugten Statements als dessen Kinder eingefügt werden. Das Elternelement ist die übergeordnete Blockanweisung, in deren Block sich das Statement befindet. In der Spezifikation werden solche Blockanweisungen durch die Klasse *BlockStatement* repräsentiert. Blockanweisungen können als Spezialisierung von einfachen Anweisungen betrachtet werden, entsprechend wird *BlockStatement* als Subklasse von *Statement* spezifiziert.

BlockStatement

$$
\begin{array}{l}
Statement \\
\hline
inner : \operatorname{seq} \downarrow Statement \\
rolledout : \downarrow Statement \times SOLUTION \pfun \downarrow Statement \\
\hline
\forall (r, sol, s) : rolledout \bullet \\
\quad s \in \operatorname{ran} inner \land isGeneric(s.code) \land \\
\quad transform(s.code, sol) = r.code
\end{array}
$$

Init

$$
\begin{array}{l}
rolledout = \varnothing
\end{array}
$$

Ein *BlockStatement* definiert mit der Zustandsvariablen *inner* eine Sequenz von *Statements*, die den Rumpf der Blockanweisung repräsentiert. Im Rahmen des Rollouts können mehrere Instanzen für jede generische Anweisung im Rumpf gebildet werden. Diese Instanzen werden vorübergehend in einer Funktion *rolledout* erfasst. Die Funktion bildet jede Instanz und die jeweilige Lösung, mit der diese Instanz erzeugt wurde, auf die generische Original-Anweisung ab, für die diese Instanz generiert wurde. Mithilfe dieser Abbildung kann die Original-Anweisung später durch ihre Instanzen ersetzt werden.

Der Vorgang des Klonens einer generischen Anweisung ist mit der Funktion

$$clone : \downarrow Statement \to \downarrow Statement$$

beschrieben, die ein Statement auf eine tiefe Kopie (im Sinne der dargestellten Schachtelung der Anweisungen) des Statements abbildet. Zum Generieren der Instanzen wird der Rollout eines *BlockStatements* wie folgt redefiniert:

BlockStatement(Fortsetzung)

$$
\begin{array}{l}
Statement\ [\textbf{redef}\ rollout] \\
\hline
\textit{rollout_inner} \\
\Delta(rolledout) \\
solutions? : \mathbb{F}\, SOLUTION \\
\hline
\exists\, sol : solutions?;\ add : \downarrow Statement \pfun \downarrow Statement \bullet \\
\quad \operatorname{dom} add = \{s : \operatorname{ran} inner \mid isGeneric(s.code)\} \land \\
\quad (\forall\, s : \operatorname{ran} inner \mid isGeneric(s.code) \bullet \\
\qquad \exists\, c, r : \downarrow Statement \bullet \\
\qquad\quad c = clone(s) \land c\ \underline{rollout}\ r \land add(s) = r) \land \\
\quad rolledout' = rolledout \cup \{(s, r) : add \bullet (r, sol, s)\} \\
\hline
rollout \mathrel{\widehat=} rollout_inner \land transformCode
\end{array}
$$

Der Rollout eines *BlockStatements* umfasst nun die Transformation der inneren Anweisungen (*rollout_inner*) sowie die Transformation des eigenen Codefragments (*transformCode*). In *rollout_inner* werden die inneren Anweisungen ebenfalls

für eine charakteristische Teilmenge von Lösungen (*solutions*?) transformiert. Die Abbildung *add* dient als Hilfsfunktion, um die erzeugten und transformierten Instanzen temporär zu erfassen, bevor sie in die Abbildung *rolledout* übernommen werden können. Für jede generische, innere Anweisung s wird jeweils ein Klon c erzeugt ($c = clone(s)$). Der Rollout wird zusammen mit der Eingabe *solutions*? an die geklonte Anweisung c delegiert, so dass die Anweisung für diese Lösungen zur nicht-generischen Anweisung r transformiert wird (c *rollout* r). Durch die Delegation wird der Rollout rekursiv an alle inneren Anweisungen propagiert (dabei wird die Eingabe *solutions*? ggf. in weitere charakteristische Teilmengen unterteilt, was nachfolgend in der Klasse *PerBlock* spezifiziert wird). Die Anweisung r wird in der Abbildung *add* als transformierte Instanz von s vermerkt. Schlussendlich werden alle erzeugten Instanzen aus der Abbildung *add* in Relation zur betrachteten Lösung *sol* in die Abbildung *rolledout* übernommen.

Das finale Ersetzen der generischen Anweisungen durch die dazu gebildeten Instanzen wird mit den Methoden *replace* und *wrapup* spezifiziert:

BlockStatement(*Fortsetzung*)

keep_nongeneric

$s? : \downarrow Statement$

$\neg\ isGeneric(s?.code)$

replace_generic

$\Delta(inner, rolledout)$

$s? : \downarrow Statement$

$isGeneric(s?.code)$

$\exists\, head, tail, insert : \operatorname{seq} \downarrow Statement \bullet$

$\quad inner = head \frown \langle s? \rangle \frown tail \land$

$\quad \operatorname{ran} insert = \{(r, t, s) : rolledout \mid s = s? \bullet r\} \land$

$\quad inner' = head \frown insert \frown tail$

$rolledout' = rolledout \rhd \{s?\}$

$replace \mathrel{\widehat=} \fatsemi s : \operatorname{ran} inner \bullet (keep_nongeneric [] replace_generic)[s/s?]$

$wrapup \mathrel{\widehat=} replace \fatsemi (\fatsemi s : \operatorname{ran} inner \bullet s.wrapup)$

Die Methode *replace* prüft nacheinander jede Anweisung des Blocks: ist die Anweisung nicht generisch, so bleibt sie unverändert im Block enthalten (*keep_nongeneric*), andernfalls muss die Anweisung ersetzt werden (*replace_generic*). In der Methode *replace_generic*[15] [16] werden alle zum Original-Statement s? gebildeten Instanzen r der Menge *rolledout* entnommen. Die Instanzen bilden eine (untereinander beliebige) Sequenz von Anweisungen *insert*, die anstelle des Original-Statements in die Sequenz der inneren Anweisungen *inner* übernommen werden. Dabei muss die ursprüngliche Reihenfolge der inneren Anweisungen beibehalten werden (*head* und *tail*).

Die Methode *wrapup* bildet den Abschluss der Transformation: sie führt *replace* aus und propagiert den Aufruf an die Anweisungen des Blocks. Für die Oberklasse *Statement* sei *wrapup* dabei als leere Methode definiert, da dieser Teil des Rollouts nur für hierarchisch aufgebaute Anweisungen von Bedeutung ist.

[15] Sequenzen werden in Object-Z als Abbildung der natürlichen Zahlen repräsentiert. Es gilt $\operatorname{seq} X == \{f : \mathbb{N} \pfun X \mid \operatorname{dom} f = 1..\#f\}$. Zur Kurzschreibweise werden die Klammern $\langle\rangle$ verwendet, z.B. $\langle a, b\rangle = \{1 \mapsto a, 2 \mapsto b\}$.

[16] Der Object-Z-Operator $\rhd$ dient der Substraktion eines Wertebereichs (*Range Substraction*). Sei $S : X \leftrightarrow Y$ und $R : \mathbb{P}\, Y$, dann gelte $S \rhd R = \{x : X, y : Y \mid y \notin R \land x \mapsto y \in S\}$.

Die eigentliche Durchführung des Rollouts wird durch den Per-Block gesteuert. Er wird mit der Klasse *PerBlock* als Spezialisierung von *BlockStatement* spezifiziert:

PerBlock

BlockStatement [**redef** *rollout*]

$$args : \mathbb{F}\, MVAR$$
$$args \neq \varnothing$$

distinctiveSubsets

$$solutions? : \mathbb{F}\, SOLUTION$$
$$subsets! : \mathbb{F}\,\mathbb{F}\, SOLUTION$$
$$solutions? = \bigcup subsets!$$
$$\forall\, subset1, subset2 : subsets! \bullet$$
$$\quad \forall\, sol1 : subset1;\ sol2 : subset2 \bullet$$
$$\quad\quad (\forall\, var : args \bullet sol1(var) = sol2(var)) \Leftrightarrow (subset1 = subset2)$$

$$rollout \mathrel{\widehat=} (distinctiveSubsets\, ⨟$$
$$\quad (⨟ subset : subsets! \bullet rollout_inner[subset/solutions?])$$
$$\quad ⨟ wrapup \land transformCode) \setminus (subsets!)$$

Die Klasse *PerBlock* ergänzt das Zustandsschema um eine nicht-leere Menge von Metavariablen *args*, welche die Argumente des Per-Blocks repräsentiert. Die Methode *distinctiveSubsets* liefert die Menge *subsets*! der charakteristischen Teilmengen einer Eingabemenge *solutions*? in Bezug auf die Argumente. Als Ausgabe liefert sie also eine Menge von Lösungsmengen. In jeder charakteristischen Teilmenge müssen alle Lösungen die gleichen Belegungen für die Metavariablen der Argumente des Per-Blocks aufweisen, die übrigen Belegungen sind für diesen Per-Block irrelevant.

Der Rollout wird nun für den Per-Block vollständig definiert: zunächst werden für jede charakteristische Teilmenge die inneren Anweisungen des Blocks einmal expandiert (*rollout_inner*). Im Anschluss werden die generischen Anweisungen darin durch die gebildeten Instanzen ersetzt und entfernt (*wrapup*). Zuletzt wird der äußere Code des Per-Blocks transformiert (*transformCode*). Die Transformation sieht in diesem Fall vor, dass der äußere Teil des Per-Blocks vollständig entfernt wird, im Ausgangsteam bleiben ausschließlich die inneren Anweisungen des Blocks erhalten.

Abschließend kann die Klasse *GenericTeam* nun ebenfalls als eine Spezialisierung von *BlockStatement* spezifiziert werden.

GenericTeam(*Redefinition*)

BlockStatement

$$teamquery : MVAR \twoheadrightarrow MTYPE$$
$$solutions : \mathbb{F}\, SOLUTION$$
$$\forall\, sol : solutions \bullet$$
$$\quad (\mathrm{dom}\, sol = \mathrm{dom}\, teamquery \land$$
$$\quad \forall (var, value) : sol \bullet value \in \mathrm{dom}(conform \rhd \{teamquery(var)\}))$$

$$transform \mathrel{\widehat=} rollout[solutions/solutions?]$$

Das generische Team stellt den Ausgangspunkt der Transformation dar, die nun einfach über den Rollout einer hierarchischen Anweisung definiert wird, wobei als initiale Eingabemenge das Ergebnis der Teamquery, repräsentiert durch die Menge *solutions*, zum Einsatz kommt.

Zusammengefasst beschrieben besteht der Transformationsprozess, um ein Eingangsteam in ein Ausgangsteam zu überführen, aus drei Phasen:

1. Auswertung der Teamquery
2. Rollout der Per-Blöcke
3. Ersetzen und Entfernen generischer und übriger GOT-Anweisungen

Vollständiges Beispiel

Die Beispiel-Implementierung des Observer Patterns sei an dieser Stelle noch einmal vollständig aufgeführt. Das generische Eingangsteam `MyShapeDisplay` ist folgendermaßen definiert:

```
1  team class MyShapeDisplay
2    match(?Class ?base, ?Method ?writer) {
3      ?Field -f;
4      ?Method -update;
5      StdQ.isAncestor(Shape.class, ?base) &&
6      StdQ.methodOfNameAndClass(-update, "paint", ?base) &&
7      StdQ.accesses(-update, -f) &&
8      StdQ.changes(?writer, -f) &&
9      StdQ.memberMethod(?writer, ?base)
10   }
11 {
12   private List<Shape> shapes;
13
14   public void update() {
15     for (Shape s : shapes)
16       {s.paint(graphics);}
17   }
18
19   per (?base) {
20     protected class -Subject playedBy ?base
21       base when (shapes.contains(base))
22     {
23       per(?writer) {
24         update <- after ?writer;
25       }
26     }
27   }
28 }
```

Listing 4.8: Ein vollständiges generisches Team

Nach Abschluss der Transformation auf Basis der Ergebnismenge

$$\{\{?base \mapsto Circle, ?writer \mapsto setRadius\},$$
$$\{?base \mapsto Circle, ?writer \mapsto setColor\},$$
$$\{?base \mapsto Rectangle, ?writer \mapsto setWidth\},$$
$$\{?base \mapsto Rectangle, ?writer \mapsto setHeight\},$$
$$\{?base \mapsto Rectangle, ?writer \mapsto setColor\}\}$$

wird das Ausgangsteam mit der folgenden Gestalt generiert:

```
team class MyShapeDisplay
{
  private List<Shape> shapes;

  public void update() {
    for (Shape s : shapes)
      {s.paint(graphics);}
  }

  protected class CircleSubject playedBy Circle
      base when (shapes.contains(base))
  {
    update <- after setRadius;
    update <- after setColor;
  }
  protected class RectangleSubject playedBy Rectangle
      base when (shapes.contains(base))
  {
    update <- after setWidth;
    update <- after setHeight;
    update <- after setColor;
  }
}
```

Listing 4.9: Das Team `MyShapeDisplay` nach der Transformation

Die Per-Blöcke wurden dahingehend transformiert, dass die generischen Anweisungen durch generierte Instanzen ersetzt wurden. Die übrigen GOT-Elemente, wie z.B. die Teamquery, wurden vollständig entfernt. Das Ausgangsteam entspricht somit einer gültigen OT-Klasse und kann im Anschluss an die Transformation vom OT-Compiler weiterverarbeitet werden.

4.3.4 Gültigkeit generischer Anweisungen

In Abschnitt 4.3.2 wurde die Restriktion beschrieben, dass generische Entitäten nur innerhalb eines Per-Blocks zur Anwendung kommen dürfen. Dies sichert die Gültigkeit von Anweisungen, bei deren Definition generische Entitäten involviert sind, auch nach der Transformation. Ziel ist es, im Vorhinein bereits all jene Fälle zu verbieten, bei denen die Transformation ungültigen oder uneindeutigen Code generieren würde. Nehmen wir als Beispiel den folgenden Per-Block, der eine generische Methode `-m` definiert.

```
per (?mv) {
  void -m() {...}
  void ok() {-m()}
}
void nok() {-m()} // verboten!
```

Innerhalb des Per-Blocks ist die Verwendung der Methode `-m` sicher, z.B. in einem Aufruf in der nicht-generischen Methode `ok`, da für jede Expansion des Per-Blocks eine eindeutige Instanz für `-m` generiert wird, die alle Vorkommen von `-m` im Ausgangsteam ersetzt. Es existiert eine 1:1-Beziehung zwischen generischen Entitäten und Anweisungen innerhalb des Blocks. Die Verwendung von `-m` außerhalb des Per-Blocks, dargestellt durch einen Aufruf in der Methode `nok`, ist nicht erlaubt, da bei dieser Anweisung nicht eindeutig definiert ist, auf welche Instanz von `-m` der Aufruf

verweisen soll. Der Methodenrumpf von `nok` ist nicht Bestandteil der Expansion des Per-Blocks, eine eindeutige 1:1-Beziehung ist somit nicht gegeben und `-m` daher nicht im Gültigkeitsbereich. An dieser Stelle ist nicht einmal sichergestellt, dass überhaupt eine Instanz von `-m` gebildet werden wird – ein Fall, den wir gesondert betrachten.

Umgang mit leeren Ergebnismengen und Null-Werten

Die Einschränkung des Gültigkeitsbereichs generischer Entitäten ist universell formuliert, so dass sie auch den Fall abdeckt, dass keine Belegung für die Argumente eines Per-Blocks gefunden wird. Dies kann der Fall sein, wenn entweder die Ergebnismenge insgesamt leer ist oder wenn einzelne Metavariablen mit einem Null-Wert belegt wurden, da sie, wie in Abschnitt 4.2.3 beschrieben, als optionale Parameter definiert wurden. Ein Per-Block, für den keine charakteristische Teilmenge existiert oder dessen Argumente darin mit mindestens einem Null-Wert belegt sind, wird keine Instanzen generieren. Das abschließende Entfernen der generischen Anweisungen bleibt davon unberührt. Der Per-Block

```
per (?mv) {
  void -m() {...}
  void ok() {-m()}
}
```

wird für eine leere Ergebnismenge also zu folgendem Code transformiert

```
void ok() {}
```

Übrig bleiben allein die nicht-generischen Anteile. Da sichergestellt ist, dass keine generischen Entitäten des Per-Blocks von außerhalb referenziert werden und innerhalb alle generischen Anweisungen gleichermaßen entfernt werden, ist für das Ausgangsteam auch in dieser Form die Gültigkeit des Codes gewährleistet.

Umgang mit eindeutigen Ergebnismengen

Die Einschränkung des Gültigkeitsbereichs kann in manchen Anwendungsfällen hinderlich für den Programmierer eines GOT-Programms sein, da die Grenzen des Per-Blocks undurchdringlich für generische Entitäten sind. Allerdings können auch Fälle auftreten, wo der Programmierer selbst die Eindeutigkeit von generischen Anweisungen zusichern kann, etwa dann, wenn er in der Teamquery eine Metavariable explizit mit einer festen Belegung vorgibt. Für solche Fälle ist es dem Programmierer erlaubt, eine Metavariable in der Teamquery mit `@UniqueMatch` zu annotieren. Die Annotation `@UniqueMatch` kann auf jede Metavariable angewendet werden. Sie versichert dem Compiler, dass für die Metavariable genau eine Belegung (ungleich eines Null-Werts) gefunden werden wird. In diesem Fall besteht eine 1:1-Abbildung zwischen der betreffenden Entität des Eingangsteams mit der transformierten Entität im Ausgangsteam. Entitäten eines Per-Blocks, dessen Argumente alle mit `@UniqueMatch` annotiert sind, müssen daher nicht generisch deklariert werden. Dies ist beispielhaft in folgendem Per-Block dargestellt, wobei die Metavariable `?base` mit `@UniqueMatch` annotiert sei:

```
per (?base) {
  class MyRole playedBy ?base() {...}
}
MyRole role = ...
```

Innerhalb des Per-Blocks wird eine Rolle `MyRole` mithilfe der Metavariablen definiert. Im Normalfall müsste die Rolle generisch deklariert werden. Da aber si-

chergestellt ist, dass es genau eine Belegung für `?base` gibt, wird auch durch die Transformation genau eine Instanz der Rolle generiert. In diesem Fall darf die Rolle nicht-generisch deklariert werden. Damit darf sie wiederum auch außerhalb des Per-Blocks verwendet werden.

Der Programmierer ist selbst für die korrekte Anwendung der Annotation verantwortlich (analog zu Anweisungen zum Type-Casting). Sollte bei Auswertung der Queries keine eindeutige Belegung für eine mit `@UniqueMatch` annotierte Metavariable gefunden werden, so wird das Problem durch eine Fehlermeldung angezeigt und die Transformation abgebrochen.

4.3.5 Gültigkeit und Korrektheit des Einsatzes von Metavariablen

Metavariablen dürfen im Code nur innerhalb von Per-Blöcken vorkommen, zu deren Argumenten sie zählen. Darüber hinaus ist ihr Einsatz in Abhängigkeit ihres deklarierten Metatyps auf bestimmte Stellen im Code beschränkt. Dabei handelt es sich in erster Linie um solche Elemente, die an den Bindungspunkten zwischen Rolle und Basis eingesetzt werden. Die folgenden Einsatzgebiete sind vorgesehen:

- Der Metatyp `?Class` erlaubt den Einsatz einer Metavariable als Basisklasse in einer playedBy-Beziehung.
- Der Metatyp `?Type` (ein Obertyp von `?Class`) erlaubt den Einsatz einer Metavariable als Typ im Rahmen der Deklaration einer Entität, dies sind formale Parameter, Rückgaben, Attribute oder Variablen.
- Der Metatyp `?Method` erlaubt den Einsatz einer Metavariable auf der Basisseite von Callin- und Callout-Bindungen.
- Der Metatyp `?Field` erlaubt den Einsatz einer Metavariable auf der Basisseite von Callout-Bindungen.

Darüber hinaus ist es erlaubt, Metavariablen auch an anderen Stellen des Quellcodes einzusetzen. Allerdings beschreiben nur wenige der Metatypen Programmelemente, die im Code eines GOT-Programms tatsächlich durch Metavariablen ersetzt werden können. An diesen Stellen funktionieren sie wie normale Variablen, die eine Objektreferenz halten. Eine Konformitätsbeziehung zwischen dem Metatypen und einem normalen Typ regelt den Zugriff:

- `?Class` ist konform zu `java.lang.Class`
- `?Type` ist konform zu `java.lang.reflect.Type`
- `?Method` ist konform zu `java.lang.reflect.Method`
- `?Field` ist konform zu `java.lang.reflect.Field`
- `?String` ist konform zu `java.lang.String`
- `?int` ist konform zu `int` bzw. `java.lang.Integer`
- `?boolean` ist konform zu `boolean` bzw. `java.lang.Boolean`

Eine Metavariable eines Metatyps kann an all jenen Code-Stellen eingesetzt werden, an denen auch eine Variable des konformen Typs erlaubt wäre. Im Rahmen der Transformation wird die Metavariable durch eine normale Variable ersetzt und für die Initialisierung mit einem Wert entsprechend der Belegung gesorgt. Das folgende Beispiel demonstriert das Verhalten.

```
per (?base) {
  System.out.println(?base.toString());
}
```

Die Metavariable `?base` sei deklariert als `?Class`. Sie wird innerhalb des Per-Blocks wie eine Variable vom Typ `java.lang.Class` verwendet. Entsprechend der Konformität kann daran auch die Methode `toString` aufgerufen werden. Die Transformation wird die Metavariable durch die entsprechenden Referenzen ersetzen:

```
void print() {
  System.out.println(Circle.class.toString());
  System.out.println(Rectangle.class.toString());
}
```

Erweiterte statische Korrektheitsprüfung

Durch das Zusammenspiel mehrerer Metavariablen in gemeinsamen Strukturen ergeben sich implizit Bedingungen an deren Belegung, die vom klassischen Type-Checking nicht abgedeckt werden können. So gilt beispielsweise im folgenden Per-Block

```
per (?base) {
  class -Subject playedBy ?base
  {
    per(?writer) {
      update <- after ?writer;
    }
  }
}
```

implizit die Bedingung, dass Methoden der Metavariable `?writer` zur korrespondierenden Klasse `?base` gehören müssen, aufgrund ihres gemeinsamen Einsatzes in der Rolle `-Subject`. Andernfalls wäre das Ausgangsteam nach der Transformation ungültig, da die generierte Callin-Anweisung auf Methoden verweisen würde, die nicht im Gültigkeitsbereich der Basisklasse liegen. Der Zusammenhang zwischen diesen beiden Metavariablen und dessen Einschränkung der gültigen Belegungen lässt sich nicht durch Typisierung erfassen. An dieser Stelle ist eine Vorbedingung für die Transformation gegeben. Im Gegensatz zu Vorbedingungen im Allgemeinen lassen sich solche strukturell implizierten Vorbedingungen aber ohne Weiteres statisch absichern – durch eine Ergänzung der Teamquery. Die Teamquery muss lediglich um ein Prädikat erweitert werden, dass die Erfüllung der Vorbedingung gewährleistet. Im Beispiel wird dies durch das Prädikat

```
memberMethod(?writer, ?base)
```

realisiert. Die Anzahl struktureller Abhängigkeiten ist aufgrund der Begrenzung der Bindungspunkte überschaubar. Eine Menge von Prädikaten zur Prüfung kann daher fest vorgegeben und in der statischen Korrektheitsprüfung verankert werden. Zur Designzeit kann somit die Notwendigkeit und Integration ergänzender Prädikate in der Teamquery ausgewertet und dem Programmierer aufgezeigt werden. Die statische Korrektheitsprüfung kann so im Vorhinein gewährleisten, dass der Code nach der Transformation einer gültigen Teamklasse entspricht.

4.4 Vererbung und erweiterte Transformation

Generizität und Transformation in GOT, wie sie in Abschnitt 4.3 vorgestellt wurden, bieten eine neue Abstraktionsebene für die Implementierung. Generische Teamklassen können Teile des Codes, die bisher nur spezifisch formuliert werden konnten, in einer für die Wiederverwendung geeigneten Form ausdrücken. Bis hierhin wurde aber noch nicht vorgestellt, wie ein generisches Team in unterschiedlichen Anwendungen tatsächlich wiederverwendet werden kann. In diesem Abschnitt werden Komposition und Wiederverwendung generischer Teams erläutert.

4.4.1 Anforderungen zur Wiederverwendung generischer Teams

Der grundlegende Ansatz zur Wiederverwendung ist, wie in Abschnitt 2.4 diskutiert, die Trennung von allgemeinen (wiederverwendbaren) und spezifischen (nichtwiederverwendbaren) Teilen des Codes in separaten Modulen. Das vorherrschende Konzept zur Komposition dieser Module in der OOP ist Vererbung. Dies soll sich durch die Erweiterungen von GOT nicht ändern: auch für generische Teams soll Vererbung nach wie vor das Mittel zum Zweck der Wiederverwendung sein. Dabei stellt sich die Frage, wie zwei unterschiedliche Mechanismen der Wiederverwendung – Vererbung einerseits und Generizität inklusive Transformation andererseits – miteinander in Einklang gebracht werden können. Das Minimalziel bei der Integration eines neuen Mechanismus in eine Programmiersprache ist die Vermeidung unerwünschter Interferenzen mit bestehenden Mechanismen, so dass zumindest eine „friedliche“ Koexistenz gewährleistet ist. Die Zielsetzung für GOT geht darüber hinaus: wir möchten bei der Kombination beider Mechanismen sowohl bestehende Strukturen erhalten als auch erweitern, um damit Synergien zum Vorteil des Entwicklers zu gewinnen. Nachfolgend definieren wir zunächst fünf Anforderungen an die Wiederverwendung generischer Teams.

Anforderung 1

Ziel der Vererbung ist die Trennung allgemeiner und spezifischer Anteile des Codes. In Bezug auf den Rumpf generischer Teams ist damit vor allem die Trennung generischer und nicht-generischer Code-Anteile in Super- und Subteamklasse verbunden. Ein primärer Zweck generischer Programmelemente ist schließlich die Wiederverwendbarkeit. Bei der Vererbung muss allerdings auch die Teamquery berücksichtigt werden. Die spezifische Ausprägung eines Teams wird durch die individuelle Parametrisierung des generischen Codes erreicht – definiert in der Teamquery. Die Subteamklasse eines generischen Teams muss daher auch in der Lage sein, die Teamquery zu verfeinern. Die Teamquery ist analog zu einer virtuellen Methode zu behandeln. Die Subteamklasse kann dann durch Verfeinerung der Teamquery die für einen spezifischen Anwendungskontext gegebenen Bedingungen zur Teamquery beisteuern.

A1: Die Teamquery eines generischen Teams muss von einem erbenden generischen Team verfeinert werden können.

Anforderung 2

Die Verfeinerung einer Teamquery hat in gleicher Weise Einfluss auf die Belegung der Metavariablen, wie eine überschriebene Methode – aufgrund dynamischen Bindens – Einfluss auf die Methodenausführung hat. Im Unterschied zum dynamischen Binden hat die Verfeinerung der Teamquery allerdings Auswirkungen auf die statische Struktur der transformierten Klasse. Je nach Belegung der Metavariablen müssen unterschiedliche Ausprägungen der Klasse als Ausgangsteam generiert werden,

veranschaulicht in Abbildung 4.8. Beispielsweise könnte das generische Team, das die Observer-Funktionalität implementiert, für mehrere Zwecke (z.B. unterschiedliche Views) in derselben Applikation zum Einsatz kommen.

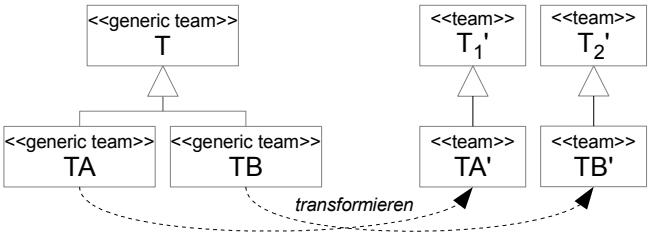

Abbildung 4.8: Transformation verfeinerter generischer Teams

Der Umgang mit veränderlichen Klassenstrukturen geht über den Verfeinerungsmechanismus des dynamischen Bindens hinaus. Für generische Teams muss der Mechanismus der Verfeinerung erweitert werden, so dass für unterschiedliche Belegungen einer Teamquery auch unterschiedliche Teamklassen generiert werden. Dies hat verschiedene Konsequenzen (vgl. nachfolgende Anforderungen), welche die Verwendbarkeit generischer Klassen einschränken. Wir teilen hier die Auffassung von Ernst [34], der für die Kombination von Vererbung und Parametrisierung eine Beschränkung der Verfeinerungsmechanismen fordert, so dass die statische Typsicherheit gewährleistet bleibt.

A2: Jede Verfeinerung eines generischen Teams muss eine einzigartige Ausprägung dieses Teams (mit eindeutigem Namen) als Ausgangsteam generieren.

Anforderung 3

Die Anforderung A2 stellt ein ernstes Problem in Bezug auf die übliche Semantik von Klassen dar. Aus einem generischen Eingangsteam können nun im Rahmen der Transformation mehrere Ausgangsteams erzeugt werden. Eine 1:1-Beziehung zwischen Ein- und Ausgangsteam ist an dieser Stelle nicht mehr gegeben. Der Typ, der von der Klasse eines solchen Eingangsteams konstituiert wird, besitzt somit keine eindeutige Abbildung (vgl. Abbildung 4.8: der Typ T wird durch die Transformation auf die Typen T_1' und T_2' abgebildet). Jede Referenz auf die Klasse könnte daher ambivalent interpretiert werden. Es gibt lediglich zwei Bereiche, in denen der Typ eindeutig abgebildet werden kann: innerhalb der generischen Teamklasse selbst oder innerhalb der verfeinernden Subteamklasse – hier erfolgt die Abbildung auf den Typ des jeweils zugehörigen Ausgangsteams. Der Typ darf dabei nicht über die Schnittstelle der Klasse nach außen dringen. Innerhalb der verfeinernden Subteamklasse ist die Verwendung des Typs ausschließlich für die Deklaration der Vererbungsbeziehung erforderlich. An allen anderen Stellen kann (u.a. aufgrund der Anforderung 4) der eigene Typ stattdessen verwendet werden.

A3: Der Typ einer generischen Teamklasse, zu der eine verfeinernde generische Subklasse existiert, ist nur innerhalb der eigenen Klasse und zur Deklaration einer Vererbungsbeziehung verfügbar.

Sollte der Programmierer einen gemeinsamen Obertyp für alle generierten Ausgangsteams eines Eingangsteams wünschen, so kann er dies über die Deklaration eines Interfaces erreichen. Ein nicht-generisches Interface wird von der Transformation nicht verändert.

Anforderung 4

Aus der zu A3 beschriebenen Problematik folgt auch, dass von einem generischen Team, das als Oberklasse zur Verfeinerung dient, keine Objekte instanziiert werden dürfen. Allgemein kann formuliert werden, dass ein generisches Team abstrakt sein muss, wenn es zur Verfeinerung genutzt wird. Dies sollte im Normalfall keine Einschränkung darstellen, da ein solches Team kaum spezifisch genug wäre, als dass Objekte davon für eine sinnvolle Verwendung genutzt werden könnten.

A4: Die Erzeugung eines Objekts als Instanz einer generischen Teamklasse, zu der verfeinernde Subklassen existieren, ist nicht erlaubt.

Anforderung 5

In Abschnitt 2.1.4 wurde das Prinzip der impliziten Rollenvererbung bzw. Copy Inheritance vorgestellt. Die Rollen eines Teams gelten als virtuelle Klassen. Bei der Vererbung zwischen Teams entscheidet die Gleichheit der Bezeichner der deklarierten Rollenklassen, ob die Rolle neu eingeführt wird oder implizit eine geerbte Rolle redefiniert – analog zum Überschreiben von Methoden. Dieser Mechanismus ist von großer Bedeutung für die Modularisierung und Spezialisierung von Teamklassen und muss entsprechend auch von generischen Teams unterstützt werden.

A5: Eine generische Entität muss von einer Subklasse in gleicher Weise verfeinert werden können, wie dies für nicht-generische Entitäten der Fall ist.

4.4.2 Wiederverwendung generischer Teams

Die Realisierung der Anforderungen A1 bis A5 in GOT erfordert eine differenziertere Betrachtung generischer Teams. Fortan wird unterschieden zwischen *verfeinerbaren generischen Teams* (VGTs) und *angewendeten generischen Teams* (AGTs). Der Umstand, dass ein generisches Team verfeinert werden darf, wird direkt in die Deklaration des Teams aufgenommen: die Teamquery eines VGTs wird mit dem Schlüsselwort *declare* eingeleitet, wohingegen die Teamquery eines AGTs mit dem Schlüsselwort *match* beginnt.

Ein VGT dient zur Definition der wiederverwendbaren Anteile einer Implementierung. Sie ist im Normalfall unvollständig, insbesondere was die Definition der Teamquery betrifft. Ein AGT wird dagegen für die konkrete Definition der spezifischen Anteile einer Implementierung verwendet. Ein AGT ist eine vollständig ausgestaltete Teamklasse inklusive Teamquery, von der Objekte erzeugt werden können. Sowohl ein VGT als auch ein AGT dürfen optional von einem VGT erben, wobei die Teamquery verfeinert werden darf. Wird von einem AGT geerbt, so kann dessen Teamquery nicht verfeinert werden. Nach wie vor bleibt natürlich auch das Erben von einer nicht-generischen Klasse gestattet. Alle bisher in dieser Arbeit gezeigten Beispiele generischer Teams sind AGTs, die ohne Vererbungsbeziehung deklariert wurden.

Umsetzung der Anforderung 1

Die Anforderung A1 verlangt die Möglichkeit zur Verfeinerung der Teamquery eines VGTs. Dies wird analog zur Verfeinerung von Methoden über einen *super*-Aufruf realisiert. Im Unterschied zu Methoden dürfen die Signaturen der Teamqueries, d.h. die Deklaration der Parameter, durchaus voneinander abweichen. Beispielhaft dient das VGT

```
abstract team class VGT
  declare(?Class ?base) {...}
{...}
```

als Oberklasse für das AGT

```
team class AGT extends VGT
  match(?Class ?base, ?Method ?m) {
    super(MyBase.class) && ...
  }
{...}
```

wobei das AGT die Teamquery durch einen super-Aufruf mit der Angabe einer konkreten Klasse als Argument spezifiziert. Die Teamquery des AGTs wurde zudem um eine Metavariable `?m` erweitert.

Die Teamquery eines VGTs kann (im Normalfall) allein nicht sinnvoll ausgewertet werden, erst in Kombination mit der Teamquery eines erbenden AGTs ergibt sich eine vollständige Query. Für die Transformation eines Programms werden daher ausschließlich die Teamqueries der AGTs ausgewertet, wobei diese die geerbten Teamqueries der Oberklassen inkludieren. Die Signaturen aller beteiligten Teamqueries werden dabei vereinigt, wobei gleiche Bezeichner für dieselben Metavariablen stehen. Die Bedingungen der Teamqueries werden über den super-Aufruf miteinander verknüpft. Die verfeinerte Teamquery des AGTs ist also eine Verknüpfung seiner Teamquery mit den Teamqueries all seiner Oberklassen. Eine Lösung der Ergebnismenge enthält daher für jede Metavariable aller beteiligten Teamqueries eine Belegung. Das Ergebnis der Auswertung wird für alle beteiligten generischen Teams verwendet.

Umsetzung der Anforderung 2

Die Anforderung A2 fordert, dass für jede Verfeinerung eines generischen Teams ein individuell ausgeprägtes Ausgangsteam generiert wird. Im Rahmen der Transformation wird für jedes Eingangs-AGT genau ein Ausgangsteam generiert. Für AGTs besteht also eine 1:1-Beziehung zwischen Ein- und Ausgangsteams. Da mehrere AGTs dabei allerdings von demselben VGT erben können, muss das VGT in Abhängigkeit der spezifischen AGTs mehrfach transformiert werden. Ein AGT stellt im praktischen Sinne die Wurzel der Transformation dar. Für jedes VGT, das von einem AGT verfeinert wird, wird ein Ausgangsteam als spezifische Ausprägung in Bezug auf das AGT generiert. Insgesamt wird für jedes AGT also eine Anzahl von Ausgangsteams generiert, die eins plus der Anzahl der verfeinerten VGT-Oberklassen entspricht. Das folgende Beispiel zeigt den Effekt der Transformation anhand zweier AGTs, die von demselben VGT erben:

```
abstract team class VGT
  declare(?Class ?base) {...}
{...}
team class AGT_A extends VGT
  match(?Class ?base) {super(A.class)}
{...}
team class AGT_B extends VGT
  match(?Class ?base) {super(B.class)}
{...}
```

Nach der Transformation wird für jedes der beiden AGTs ein entsprechendes Ausgangsteam erstellt. Das VGT wird individuell für jedes AGT ebenfalls zu zwei Ausgangsteams ausgeprägt:

```
abstract team class VGT_A {...}
team class AGT_A extends VGT_A  {...}
abstract team class VGT_B {...}
team class AGT_B extends VGT_B  {...}
```

Die ursprüngliche Klasse `VGT` ist nach der Transformation nicht mehr vorhanden. Es existieren nur noch die konkreten Ausprägungen `VGT_A` und `VGT_B`. Die Vererbungsbeziehung zwischen den Ausgangsteams wurde entsprechend angepasst, so dass der Zusammenhang unter den generierten Klassen eines gemeinsamen Rollouts erhalten bleibt.

Die Information, für welches generische Team die Auswertung einer Teamquery erfolgt, kann auch relevant für die Definition der Query selbst sein. Zu diesem Zweck wird eine vordefinierte Query

```
StdQ.rolledoutGenericTeam(?Class ?agt)
```

angeboten, die das Argument mit der Teamklasse des AGTs belegt, für das die aktuelle Auswertung erfolgt. Auf diese Weise kann der Programmierer auch in der Teamquery eines VGTs Informationen über die spezifische Ausprägung der Transformation nutzen.

Umsetzung der Anforderung 3

Die Anforderung A3 sieht vor, dass der Typ eines VGTs außerhalb der Klasse selbst ausschließlich zum Zwecke der Definition einer Vererbungsbeziehung verwendet werden darf. Hierfür ist eine Einschränkung der Sichtbarkeit eines VGTs erforderlich. Mit den Standard-Zugriffsmodifikatoren von Java lässt sich dies nicht direkt erreichen. Dem Schlüsselwort *declare* fällt daher auch eine Bedeutung zur Zugriffskontrolle zu. Die Prüfung muss für VGTs gesondert erfolgen, kann aber ohne Weiteres in den üblichen statischen Prozess der Korrektheitsprüfung integriert werden. Der originale Zugriffsmodifikator eines Eingangsteams bleibt dabei erhalten. Er wird durch die Transformation nicht verändert und entsprechend in das Ausgangsteam übernommen. Für das Eingangsteam wird die Sichtbarkeit aber darüber hinaus eingeschränkt, so dass die Teamklasse im praktischen Sinne privat ist – einzige Ausnahme stellt die Deklaration der Vererbungsbeziehung eines generischen Teams dar. Ausschließlich in diesem Rahmen gilt der eigentliche – über den Zugriffsmodifikator definierte – Gültigkeitsbereich.

Umsetzung der Anforderung 4

Die Anforderung A4 verlangt, dass keine Objekte als Instanzen eines VGTs erzeugt werden können. Dies lässt sich mit den Standard-Mitteln von Java realisieren. Ein VGT muss zwingend mit dem Schlüsselwort *abstract* als abstrakte Klasse deklariert werden. Damit ist sichergestellt, dass die Klasse nicht instanziiert werden darf.

Umsetzung der Anforderung 5

Die Anforderung A5 sieht vor, dass geerbte, generische Entitäten in gleicher Weise wie nicht-generische Entitäten redefiniert werden können. Die Umsetzung ist naheliegend: die Gleichheit der Bezeichner ist ausschlaggebend für die Entscheidung, ob eine geerbte Entität überschrieben wird – dies muss also auch für Bezeichner gelten, deren Präfix „-“ eine Deklaration als generisch kennzeichnet. Hier gilt allerdings die Besonderheit, dass die generischen Bezeichner durch die Transformation verändert werden. Die Redefinitionsbeziehung muss daher auch nach der Transformation bestehen. Da generische Entitäten in Abhängigkeit der Argumente der umschließenden Per-Blöcke transformiert werden, können hier mehrere Entitäten generiert werden,

die jeweils die ursprüngliche Redefinitionsbeziehung beibehalten müssen. An dieser Stelle müssen daher ebenfalls die Argumente der Per-Blöcke berücksichtigt werden. Zwei gleiche generische Bezeichner aus den eingehenden Teams sollen genau dann auf die gleichen nicht-generischen Bezeichner in den ausgehenden Teams abgebildet werden, wenn der Rollout für eine gleiche Belegung der relevanten Argumente erfolgt.

Im folgenden Beispiel überschreibt das AGT die geerbte Rolle `-R` des VGTs. Auch wenn es sich bei der Rolle `-R` um eine generische Rolle handelt, so gilt trotzdem – aufgrund gleicher Bezeichner – die implizite Rollenvererbung.

```
abstract team class VGT declare(?Class ?base) {...}
{
  per(?base) {
    class -R playedBy ?base {...}
  }
}
team class AGT extends VGT match(?Class ?base) {...}
{
  per(?base) {
    class -R playedBy ?base {...}
  }
}
```

Von der Transformation wird erwartet, dass die implizite Rollenvererbung auch nach Umwandlung der generischen Bezeichner noch gilt. Entsprechend muss die Abbildung der generischen Bezeichner in beiden Teamklassen analog erfolgen, z.B. für die Belegung von `?base` mit den Klassen `A` und `B`:

```
abstract team class VGT_AB
{
  class RA playedBy A {...}
  class RB playedBy B {...}
}
team class AGT extends VGT
{
  class RA playedBy A {...}
  class RB playedBy B {...}
}
```

Die implizite Rollenvererbung der Eingangsteams wurde analog in die Ausgangsteams übernommen.

Die Transformation der Bezeichner generischer Entitäten stellt ein Detail der in Abschnitt 4.3.2 abstrakt eingeführten Methode *transformCode* dar, das nun genauer spezifiziert werden soll. Die Bezeichner werden im Rahmen der Codetransformation auf nicht-generische Bezeichner abgebildet. Da die Möglichkeit besteht, dass für eine generische Entität mehrere Instanzen gebildet werden, existiert keine direkte Abbildung, sondern eine 1:n-Relation zwischen einer generischen Entität im Eingangsteam mit den dazu gebildeten Instanzen im Ausgangsteam. Ein Per-Block erzeugt jeweils eine Instanz pro charakteristischer Teilmenge der Lösungen. Das Ziel bei der Transformation der Bezeichner besteht daher darin, dass sie für jede charakteristische Teilmenge eindeutig ist, d.h. gleiche generische Bezeichner werden für gleichartige Lösungen auf dieselben nicht-generischen Bezeichner abgebildet. Eine Funktion, die generische auf nicht-generische Bezeichner abbildet, muss also neben einer Lösung mit konkreten Belegungen der Metavariablen auch die Argumente der involvierten Per-Blöcke einbeziehen, um zu entscheiden, welche der Belegungen Einfluss auf die Ausprägung des Bezeichners haben.

Zur Ergänzung der formalen Spezifikation führen wir hier die Basistypen *GENERIC_ID* und *NONGENERIC_ID* ein, welche die Mengen der gültigen generischen bzw. nicht-generischen Bezeichner repräsentieren.

$$[GENERIC_ID, NONGENERIC_ID]$$

Beispielsweise gehört der Bezeichner $-R$ zu *GENERIC_ID* und die Bezeichner *RA* und *RB* zu *NONGENERIC_ID*.

Die Funktion t beschreibt die Transformation der Bezeichner als eine Abbildung eines generischen Bezeichners auf ein nicht-generisches Gegenstück. Um die beschriebene Eindeutigkeit des Ergebnisses zu gewährleisten, sind die Menge der relevanten Metavariablen und eine konkrete Lösung mit Belegungen als zusätzliche Eingaben erforderlich.

$$t : GENERIC_ID \times \mathbb{F}_1\, MVAR \times SOLUTION \rightarrow NONGENERIC_ID$$

$$\begin{aligned}
&\forall\, sol1, sol2 : SOLUTION;\ args : \mathbb{F}_1\, MVAR;\ ident1, ident2 : GENERIC_ID \bullet \\
&\quad args \subseteq \operatorname{dom} sol1 \land args \subseteq \operatorname{dom} sol2 \Rightarrow \\
&\qquad ((ident1 = ident2 \land \forall\, var : args \bullet sol1(var) = sol2(var)) \\
&\qquad\quad \Leftrightarrow t(ident1, args, sol1) = t(ident2, args, sol2))
\end{aligned}$$

Für die Funktion t ist spezifiziert, dass für jede charakteristische Teilmenge von Lösungen, die sich aus der Eingabe ergibt, eine eindeutige (von anderen unterscheidbare) Ausprägung des übergebenen Bezeichners liefert.

4.4.3 Vollständiges Beispiel

Die Vererbung zwischen generischen Teams erlaubt nun die Implementierung einer direkt wiederverwendbaren Teamklasse zur Umsetzung der Observer-Funktionalität. Wie in Abschnitt 3.1.3 diskutiert, lassen sich die Anteile einer solchen Implementierung in sechs Kategorien unterteilen:

1. Observer-Kontext mit Subject-Rolle
2. Verwalten der Subject-Observer-Relation
3. Allgemeiner Update-Mechanismus („notify“)
4. Binden der Subject-Rolle an bestimmte Klassen und Objekte
5. Auslösen des Update-Mechanismus
6. Spezifische Update-Implementierung des Observers

Bei einer nicht-generischen Implementierung konnten lediglich die Punkte 1 bis 3 auf wiederverwendbare Weise implementiert werden. Eine generische Implementierung soll nun auch die Wiederverwendung der Anteile 4 und 5 ermöglichen. Die Umsetzung in GOT wird mit der Klasse `GenericObserver` demonstriert, dargestellt in Listing 4.10.

Das VGT `GenericObserver` definiert die wiederverwendbaren Anteile der Observer-Funktionalität. Auf die Umsetzung der einzelnen Punkte wird nachfolgend im Detail eingegangen:

1. Das Team `GenericObserver` selbst repräsentiert den Observer-Kontext. Die Rolle des Subjekts ist mit der generischen Rolle `-Subject` implementiert, deren Basisklasse durch eine Metavariable `+base` definiert wird. Das Grundgerüst der Rolle ist somit vollständig definiert.

2. Eine explizite Verwaltung der Subject-Observer-Relation ist in OT obsolet, da dies bereits durch die Rollenbeziehung abgedeckt ist, die in OT fest in den Sprachmechanismen verankert ist. Die Angabe der *playedBy*-Relation ist zur Realisierung hinreichend.

```
1  abstract team class GenericObserver
2    declare(?Class +base, ?Method ?writer) {
3      ?Field -f;
4      ?Method -update;
5      ?Class -agt;
6      StdQ.rolledoutGenericTeam(-agt) &&
7      StdQ.methodOfNameAndClass(-update, "update", -agt) &&
8      StdQ.accesses(-update, -f) &&
9      StdQ.changes(?writer, -f) &&
10     StdQ.memberMethod(?writer, +base)
11   }
12 {
13   public abstract void update();
14
15   per (+base) {
16     protected class -Subject playedBy +base
17     {
18       per(?writer) {
19         update <- after ?writer;
20       }
21     }
22   }
23 }
```

Listing 4.10: Ein VGT zur Implementierung der Observer-Funktionalität

3. Der allgemeine Update-Mechanismus wird über einen Callin (Z. 19), der auf eine abstrakte Methode `update` verweist, implementiert. Der Umweg über eine Methode `notify` und die Methode selbst sind in der OT-Lösung obsolet, da direkt auf die Methode `update` zugegriffen werden kann. Die Methode `update` selbst muss vom konkreten Observer zur Umsetzung der spezifischen Funktionalität überschrieben werden (Punkt 6). Die Basismethode, durch die der Callin ausgelöst wird, wird generisch in Form der Metavariable `?writer` angegeben.

4. Die Bindung der Rolle `-Subject` an bestimmte Klassen ist bereits über die Metavariable `+base` definiert. Die Angabe der konkreten Basisklasse(n) zählt zu den spezifischen Anteilen. Effektiv muss nur noch die Belegung der Metavariable `+base` definiert werden (in der Teamquery der erbenden AGTs). Für jede Belegung wird automatisch eine konkrete Ausprägung der Rolle generiert.

5. Der wesentliche Mehrwert der Wiederverwendbarkeit dieser Lösung liegt beim Auslösen des Update-Mechanismus. Die spezifische Belegung der Metavariable `?writer` bestimmt, wann das Update ausgelöst werden wird. Die Belegung wird wiederum durch die Teamquery in Abhängigkeit einer Methode `-update` vollständig definiert. Die Metavariable `-update` wird automatisch mit der vom jeweiligen AGT überschriebenen Variante der abstrakten Methode `update` belegt (Zz. 6–7). Die Prädikate `accesses` und `changes` der Teamquery (Zz. 8–9) analysieren diese Methode dahingehend, dass all jene Methoden `?writer` als Auslöser ausgewählt werden, die eine für `-update` relevante Zustandsveränderung bewirken. Somit werden durch die Analyse der Update-Methode alle relevanten Statusänderungen als Auslöser identifiziert, ohne dass der Programmierer manuell Anpassungen daran vornehmen muss. Dieser Anteil der Implementierung ist vollständig wiederverwendbar.

6. Die Realisierung der Methode `update` des Observers bleibt als einziger Bestandteil der Implementierung abstrakt. Diese Methode stellt die spezifische Funktionalität im Observer Pattern zur Verfügung und muss zwingend von der Subklasse überschrieben werden.

Die ehemals nicht wiederverwendbaren Anteile unter Punkt 5 konnten in dieser Lösung dank Generizität und statischer Analyse vollständig wiederverwendbar implementiert werden. Punkt 4 konnte in wesentlichen Teilen wiederverwendbar implementiert werden. Eine konkrete Implementierung des Observer Patterns erfordert nun lediglich noch die Angabe einer Basisklasse, die Ausprägung konkreter Rollen erfolgt automatisch.

Eine konkrete Implementierung der Klasse `MyShapeDisplay` des Beispiels 3.1 ist in Listing 4.11 in Form eines AGTs dargestellt.

```
1  team class MyShapeDisplay extends GenericObserver
2    match(?Class ?base, ?Method ?writer) {
3      super(?base, ?writer) &&
4      StdQ.isAncestor(Shape.class, ?base)
5    }
6  {
7    private List<Shape> shapes;
8
9    @Override
10   public void update() {
11     for (Shape s : shapes)
12       {s.paint(graphics);}
13   }
14
15   per (?base) {
16     protected class -Subject playedBy ?base
17       base when (shapes.contains(base)) {}
18   }
19 }
```

Listing 4.11: Ein AGT zur Verfeinerung der Observer-Funktionalität

Das AGT `MyShapeDisplay` erbt vom VGT `GenericObserver`. Die Teamquery wird dahingehend verfeinert, dass als Basisklassen `?base` die Subklassen der Klasse `Shape` vorgegeben werden. Das Team selbst überschreibt die abstrakte Update-Methode mit spezifischer Funktionalität zur Darstellung von Shape-Objekten (Zz. 9–13). Damit ist das AGT bereits vollständig implementiert. Darüber hinaus sind Verfeinerungen der geerbten Anteile möglich. Eine sinnvolle Verfeinerung ergibt sich für die Rollenklasse `-Subject`, die um ein Guard Predicate

```
base when (shapes.contains(base))
```

erweitert wird (Z. 17). Mit diesem Guard Predicate wird die Subjekt-Rolle auf solche Basisobjekte eingeschränkt, die Elemente der Liste `shapes` sind. Auf diese Weise kann das Update nur von den relevanten Elementen ausgelöst werden – überflüssige Updates, ausgelöst von anderen Objekten, werden verhindert.

Das AGT `MyShapeDisplay` ist Ausgangspunkt für die Transformation. Das AGT wird gemeinsam mit seiner Oberklasse `GenericObserver` transformiert. Die Teamqueries der beiden Klassen werden kombiniert und ausgewertet. Auf Basis der Ergebnismenge

$\{\{?base \mapsto Circle, ?writer \mapsto setRadius\},$
$\{?base \mapsto Circle, ?writer \mapsto setColor\},$
$\{?base \mapsto Rectangle, ?writer \mapsto setWidth\},$
$\{?base \mapsto Rectangle, ?writer \mapsto setHeight\},$
$\{?base \mapsto Rectangle, ?writer \mapsto setColor\}\}$

werden zwei Ausgangsteams generiert. Zunächst wird eine konkrete Ausprägung des VGTs `GenericObserver` erzeugt, dargestellt in Listing 4.12.

```
1  abstract team class GenericObserver_MyShapeDisplay {
2    public abstract void update();
3
4    protected class CircleSubject playedBy Circle {
5      update <- after setRadius;
6      update <- after setColor;
7    }
8    protected class RectangleSubject playedBy Rectangle {
9      update <- after setWidth;
10     update <- after setHeight;
11     update <- after setColor;
12   }
13 }
```

Listing 4.12: Das Ausgangsteam zur Klasse `GenericObserver` nach der Transformation für das AGT `MyShapeDisplay`

Die erzeugte Teamklasse hat nun einen neuen, eindeutigen Bezeichner `GenericObserver_MyShapeDisplay`. In der Klasse sind zwei konkrete Subjekt-Rollen für die Basisklassen `Circle` und `Rectangle` mit den jeweils spezifischen Callins definiert.

Die konkrete Ausprägung des VGTs wird nun als Oberklasse für das Ausgangsteam des AGTs `MyShapeDisplay` verwendet, dargestellt in Listing 4.13.

```
1  team class MyShapeDisplay extends
       GenericObserver_MyShapeDisplay {
2    private List<Shape> shapes;
3
4    @Override
5    public void update() {
6      for (Shape s : shapes)
7        {s.paint(graphics);}
8    }
9
10   protected class CircleSubject playedBy Circle
11       base when (shapes.contains(base)) {}
12   protected class RectangleSubject playedBy Rectangle
13       base when (shapes.contains(base)) {}
14 }
```

Listing 4.13: Das Team `MyShapeDisplay` nach der Transformation

Da es sich bei einem AGT um eine 1:1-Beziehung zwischen Ein- und Ausgangsteam handelt, bleibt der Bezeichner `MyShapeDisplay` von der Transformation unverändert. Der einzige generische Anteil des AGTs besteht in einer Verfeinerung der Subjekt-Rolle. Entsprechend werden hier verfeinerte Rollenklassen für die Basisklassen `Circle` und `Rectangle` generiert. Die generierten Bezeichner entsprechen

denen der geerbten Rollenklassen, da sie auf Grundlage der gleichen Argumente gebildet wurden. Durch die Verfeinerung wird lediglich ein Guard Predicate ergänzt. Der Rumpf der Klassen bleibt dank impliziter Rollenvererbung erhalten.

4.4.4 Rekursive Strukturen

Für die wiederverwendbare Implementierung von Rollen ergibt sich eine besondere Anforderung, wenn diese Rollen in einer gemeinsamen Hierarchie verbunden sein sollen, die spiegelbildlich zur Hierarchie der Basisklassen aufgebaut ist. Ein solches Szenario ist in Abbildung 4.9 skizziert: für eine Hierarchie von Basisklassen mit der Wurzel `B0` existiert im Team `T` eine korrespondierende Baumstruktur von Rollenklassen mit Wurzel `R0`, in der die Rollenklassen in der gleichen Vererbungsbeziehung zueinander stehen wie ihre jeweiligen Basisklassen.

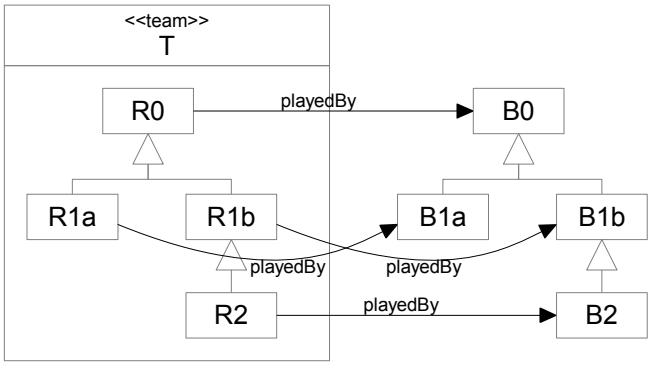

Abbildung 4.9: Eine Hierarchie von Rollen- und Basisklassen

Alle Rollenklassen erben von einem gemeinsamen Wurzelelement, das die wesentliche Schnittstelle der Funktionalität definiert. Die Subrollen ergänzen oder verfeinern diese Funktionalität. Ein solches Szenario tritt durchaus häufiger auf, da die Rollenfunktionalität einer Basisklasse oft analog zur Hierarchie der Basisklassen verfeinert werden muss. Durch Smart Lifting wird dabei zur Laufzeit immer die speziellste der in Frage kommenden Rollen ausgewählt.

Eine solche Rollenstruktur generisch abzubilden, stellt eine besondere Herausforderung dar. Zwar kann die Hierarchie der Basisklassen mit Queries analysiert und entsprechend in Metavariablen erfasst werden, doch existiert dabei eine Abhängigkeit unter den Basisklassen, die nicht ohne Weiteres beim Rollout berücksichtigt werden kann. Das Szenario erfordert, eine generische Rolle zu konstruieren, die für Basisklassen aller Hierarchieebenen anwendbar ist. Zum Zwecke der Modularisierung ist es durchaus sinnvoll, die generische Rolle auf Basis einer nicht-generischen Oberklasse zu definieren. Die nicht-generische Rolle funktioniert in diesem Fall als Wurzel, um die generische Hierarchie von Subrollen mit einem nicht-generischen Typ zu verankern. Der Ansatz ist in Abbildung 4.10 skizziert.

Die nicht-generische Rolle `R` ist die Oberklasse der generischen Rolle `-RG`. Die Rolle `R` wird von der Transformation unverändert übernommen. `-RG` dient als Template für die Rollen aller (!) Basisklassen der Hierarchie; im Ausgangsteam können entsprechend mehrere Instanzen von `-RG` gebildet werden. Die besondere Herausforderung hier ist nun, dass diese Instanzen nicht unabhängig voneinander sind, sondern über eine Vererbungsbeziehung – korrespondierend zur Hierarchie der Basisklassen – miteinander verknüpft werden müssen. Da alle Instanzen auf derselben generischen Rolle `-RG` basieren, erfordert die Abbildung dieser Beziehung eine

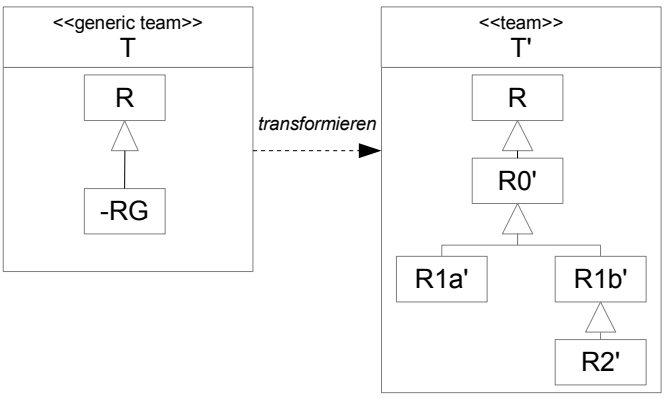

Abbildung 4.10: Ein generisches Team zur Generierung von Rollenhierarchien

selbstbezügliche Definition der Rolle, denn diese muss im praktischen Sinne „sich selbst“ als Oberklasse referenzieren. Um die Eindeutigkeit der Transformation sicherzustellen, muss der Selbstbezug natürlich für jeden Rollout auf eine konkrete Instanz verweisen. Dabei ist jeweils jene Instanz der generischen Rolle gemeint, die zur Oberklasse der aktuellen Basisklasse gebildet wird.

Die Möglichkeiten zur Definition einer solchen selbstbezüglichen Referenz werden vom Metatyp der jeweiligen Metavariable bereitgestellt. Der Metatyp `?Class` bietet die Methode `parentRole` an, die als Parameter eine Rollenklasse erwartet. Die Methode liefert die jeweils spezifischste Rollenklasse, die zur Basisklasse der Metavariablen existiert und eine Subklasse des Arguments darstellt, mit Ausnahme der Instanz, die im aktuellen Rollout gebildet wird. Existiert keine solche Rollenklasse, so wird das Argument selbst zurückgegeben. Die Methode `parentRole` ermöglicht so eine variable Definition der Vererbungsbeziehung relativ zur eigenen Position in der Hierarchie. Die Auswertung erfolgt rekursiv, wobei das Argument die Wurzel der Hierarchie repräsentiert und damit als Anker für die Rekursion dient. Die Anwendung ist in Listing 4.14 skizziert.

```
1  team class T
2    match (?Class ?base) {...}
3  {
4    protected class R {...}
5
6    per (?base) {
7      protected class -RG extends ?base.parentRole(R.class)
           playedBy ?base
8      {...}
9    }
10 }
```

Listing 4.14: Eine rekursiv definierte Rollenhierarchie

Die generische Rolle `-RG` definiert die Vererbungsbeziehung durch Verwendung der Methode `parentRole` an der Metavariable `?base`. Als Wurzel dient die nichtgenerische Rolle `R`. Für das in Abbildung 4.10 skizzierte Beispiel würde die folgende Rollenstruktur im Ausgangsteam generiert:

```
class R {...}
class RG_B0 extends R playedBy B0 {...}
class RG_B1a extends RG_B0 playedBy B1a {...}
class RG_B1b extends RG_B0 playedBy B1b {...}
class RG_B2 extends RG_B1b playedBy B2 {...}
```

Die Hierarchie der Rollen spiegelt die Hierarchie der Basisklassen 1:1 wider. Auf diese Weise können derartige Klassenhierarchien in beliebiger Vererbungstiefe mit konstantem Implementierungsaufwand abgebildet werden. Eine konkrete Anwendung dieser Technik wird im Rahmen der Evaluation für das Design Pattern *Memento* in Abschnitt 6.2.2 gezeigt.

4.5 Korrektheit der Transformation

GOT ist mit einer transformationellen Semantik relativ zur Zielsprache OT definiert. Der Sprachumfang von GOT wird durch Code-Transformation auf den Sprachumfang von OT reduziert, GOT-Konstrukte werden entsprechend auf OT-Konstrukte abgebildet. Die Code-Transformation nimmt ein GOT-Programm als Eingabe und liefert ein OT-Programm als Ausgabe. Wir betrachten ein GOT-Programm als gültig, wenn es durch den spezifizierten Transformationsprozess in ein gültiges OT-Programm umgewandelt wird. In diesem Abschnitt diskutieren wir die Integrität der Transformation dahingehend, dass das generierte Ausgabeprogramm darüber hinaus auch die Erwartungen des Programmierers erfüllt, indem es eine äquivalente Repräsentation seines Eingabeprogramms darstellt.

Die Klassen eines GOT-Programms lassen sich in drei Kategorien unterteilen: Query-Klassen, generische Teamklassen und alle anderen, nicht-generischen Klassen.

1. Query-Klassen (vgl. 4.2.4) sind einfache Utility-Klassen. Sie dienen ausschließlich als Bibliotheken für Queries und sind somit nur für die Definition anderer Query-Klassen und generischer Teamklassen relevant. Nach Abschluss der Transformation haben sie keine Bedeutung mehr für das Programm, da sie weder Verhalten noch Zustand definieren. Alle Referenzen auf Queryklassen und deren Queries werden im Rahmen der Transformation restlos beseitigt. Im Ausgabeprogramm können sie daher komplett entfernt werden.

2. Nicht-generische Klassen – seien es Teams oder normale Klassen – werden von der Transformation unverändert übernommen. Ihre Abbildung im Ausgabeprogramm ist identisch zum Original des Eingabeprogramms. Dies ist durch die Restriktionen gewährleistet, die für generische Entitäten und Klassen gelten.

   Generische Entitäten dürfen ausschließlich im Rahmen des umschließenden Per-Blocks referenziert werden, außerhalb des Per-Blocks ist der Zugriff nicht erlaubt (vgl. 4.3.2). Die Verwendung von Per-Blöcken wiederum ist ebenfalls auf generische Teams begrenzt. In nicht-generischen Klassen sind daher grundsätzlich keine Referenzen auf generische Entitäten, die eine Transformation der Klassen erfordern würden, möglich.

   Generische Teamklassen können in Form von VGTs oder AGTs existieren. Für VGTs gilt (vgl. 4.4.2), dass deren Typ außerhalb der Klasse selbst ausschließlich von anderen generischen Teams zur Deklaration einer Vererbungsbeziehung genutzt werden kann. Eine Referenz seitens einer nicht-generischen Klasse ist somit nicht erlaubt. AGTs dagegen dürfen auch von nicht-generischen Klassen referenziert werden. Für AGTs gilt eine 1:1-Abbildung zwischen dem Ein- und Ausgabeprogramm, die Referenz ist daher eindeutig und wird von

der Transformation nicht verändert. Gleiches gilt auch für den Zugriff auf die nicht-generischen Attribute und Methoden des AGTs. AGTs dienen im praktischen Sinne als Brücke zwischen den generischen und den nicht-generischen Teilen eines Programms. Sie stellen die einzige Verbindung zwischen den ansonsten strikt getrennten Bereichen dar.

3. Generische Teamklassen inklusive ihrer Rollen sind die einzigen Klassen, die von der Transformation tatsächlich verändert werden. Die eigentliche Transformation ist dabei auf die von Per-Blöcken umschlossenen Codefragmente begrenzt. Nur in diesem Rahmen kommt in GOT Metaprogrammierung zum Einsatz. Die Metaprogrammierung ist dabei auf den Einsatz von Metavariablen begrenzt, deren Belegung für den konkreten Rollout eines Per-Blocks entscheidend ist. Der Einsatz von Metavariablen ist wiederum in Abhängigkeit des Metatyps auf bestimmte Stellen im Code, wie PlayedBy- und Callin-Beziehungen, begrenzt (vgl. 4.3.5). An anderen Stellen können die Metavariablen nur als Repräsentant einer korrespondierenden Instanz eines Typs des Pakets `java.lang.reflect` o.ä. eingesetzt werden. Die Verwendung der Metavariablen ist also insgesamt auf klar definierte Einsatzzwecke begrenzt.

   Aus dem Zusammenspiel innerhalb der statischen Struktur können sich weitere Bedingungen an den korrekten Einsatz von Metavariablen ergeben (vgl. 4.3.5). Solche strukturell implizierten Abhängigkeiten müssen von der Teamquery abgedeckt werden, erzwungen durch die statische Korrektheitsprüfung.

   Die Kapselung generischer Codefragmente sichert zu, dass das Ergebnis der Transformation eindeutig ist. Jede Referenz auf eine generische Entität innerhalb eines Per-Blocks ist eindeutig definiert; für jeden Rollout wird der Entität ein eindeutiger Bezeichner gegeben (vgl. 4.3.3). Da kein Zugriff auf eine generische Entität von außerhalb des umschließenden Per-Blocks gestattet ist, ist es ausgeschlossen, dass an dieser Stelle ein Bezeichner mit mehrdeutiger (falls mehrere charakteristische Teilmengen für die Argumente des Per-Blocks existieren) oder undefinierter (falls keine charakteristische Teilmenge existiert) Bedeutung verwendet wird.

Zusammengefasst bilden die klare Trennung generischer und nicht-generischer Code-Anteile, die strikte Begrenzung der Metaprogrammierung und die erweiterte statische Korrektheitsprüfung ein Paket von Maßnahmen, das sowohl als Hilfestellung zur Beherrschung der Spracherweiterung GOT dienen als auch Vertrauen in die Korrektheit der Transformation schaffen soll. Metaprogrammierung in GOT ist *strukturiert*: im Gegensatz zu textbasierter Metaprogrammierung wird auf frei wählbare Textbausteine verzichtet. Der Template-Code ist vollständig statisch analysierbar. Die Transformationsregeln sichern zu, dass aus einer korrekten Eingabe stets ein syntaktisch korrektes OT-Programm generiert wird.

Kapitel 5

Technische Umsetzung

Die Spracherweiterung GOT wurde prototypisch als Plugin für die Entwicklungsumgebung Eclipse 4.2 [30] auf Basis des bestehenden *Object Teams Development Toolings* (OTDT[17], Version 2.1) mit einer Anbindung an SWI-Prolog[18] realisiert. In diesem Kapitel wird kurz auf die technische Umsetzung des Prototyps eingegangen. Wir geben einen Überblick über die Herausforderungen und die Erkenntnisse, die sich bei der Entwicklung ergeben haben.

5.1 Realisierung

Für die Integration von GOT in OT muss das OTDT erweitert werden. Das OTDT ist eine Erweiterung des *Java Development Toolings* (JDT), das als Standard in der Eclipse Plattform integriert ist. Das JDT besteht aus einzelnen Modulen, sogenannten *Plugins*, die zum einen den Compiler und zum anderen die Benutzerschnittstelle realisieren. Das OTDT erweitert das JDT, indem es direkt dessen Quellcode übernimmt und verändert. Diese Art der Umsetzung ist durchaus kritisch zu sehen, da somit eine starke Kopplung zwischen den Modulen existiert und jede Änderung am JDT im OTDT nachgezogen werden muss. Im Sinne guter Modularisierung soll die GOT-Erweiterung daher versuchsweise vollständig separat vom bestehenden OTDT entwickelt werden, indem sie über die Mittel der Rollenadaption integriert wird. Auf diese Weise werden die Abhängigkeiten zwischen den Werkzeugen reduziert, was sich positiv auf die Evolutionsfähigkeit auswirken sollte.

Die GOT-Erweiterung besteht im Wesentlichen aus zwei Teilen (skizziert in Abbildung 5.1): erstens, einer Adaption des bestehenden OT-Compilers und dessen Datenstrukturen, so dass er die erweiterte Syntax und Struktur der GOT-Programme verarbeiten kann, und zweitens, einem Transformer, der die Integration von Prolog realisiert und ein GOT-Programm in ein OT-Programm transformiert. Im Gegensatz zum Compiler kann der Transformer nicht auf einer bestehenden Implementierung aufbauen, sondern ist eine eigenständige Entwicklung. Essentieller Bestandteil sowohl des Compilers als auch des Transformers sind Datenstrukturen zur Repräsentation des Programms. Diese werden zwar als „AST“ bezeichnet, können aber derart mit Informationen angereichert werden, dass sie auch einen ASG darstellen. Die Module und Datenstrukturen werden in den folgenden Abschnitten näher erläutert.

[17] Siehe auch `http://www.eclipse.org/objectteams/`
[18] Siehe auch `http://www.swi-prolog.org/`

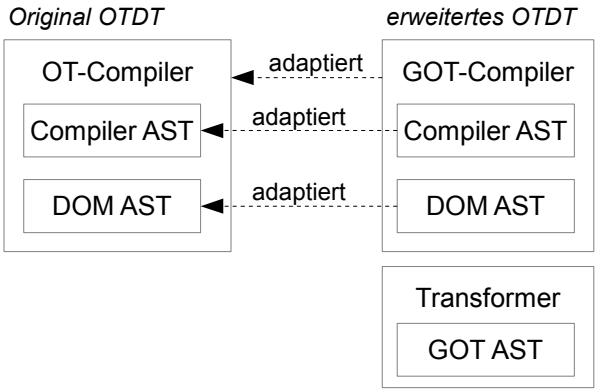

Abbildung 5.1: Integration von GOT durch Adaption des OTDT

5.1.1 Compiler und Anbindung an Eclipse

Der erste Schritt zur Umsetzung der GOT-Erweiterung war die Anpassung des Compilers und der zugehörigen Datenstrukturen. Details dazu sind in der Diplomarbeit von Hoffmann nachzulesen [61]. Wesentliche Bestandteile des Compilers sind der Scanner und der Parser, die für die Erkennung der Syntax und die Umwandlung von Quellcode in einen AST zuständig sind. Der Scanner musste um die neuen Schlüsselwörter für GOT ergänzt werden. Der Parser wird zu Teilen automatisch durch einen Parsergenerator generiert. Grundlage ist die OT-Grammatik, die um die neuen Sprachmittel von GOT erweitert wurde. Die erweiterte Grammatik ist in Auszügen in Anhang A dargestellt.

In Eclipse existieren mehrere Datenstrukturen zur Repräsentation des AST eines Programms. Resultat des Parsers ist der *Compiler AST*, der u.a. für die statische Korrektheitsprüfung und im weiteren Verlauf für die Generierung von Java Bytecode verwendet wird. Ähnlich aufgebaut ist der *DOM AST*, der allerdings eine vielseitigere API zur Verfügung stellt. Der DOM AST erlaubt z.B. das nachträgliche Hinzufügen, Manipulieren und Entfernen von Knoten. Änderungen am DOM AST können direkt in den Quellcode zurückgeschrieben werden. Auf diese Weise werden in Eclipse automatisierte Codeanpassungen wie Refactorings und Quickfixes realisiert. Der Compiler AST und der DOM AST werden mithilfe der Datenstrukturen *Binding* und *Scope* um Querverweise und zusätzliche Informationen ergänzt, so dass sie einen ASG repräsentieren (auch wenn sie nach wie vor als „AST“ bezeichnet werden). Durch Bindings und Scopes werden Typen und Namen aufgelöst und entsprechende Knoten der ursprünglichen Baumstruktur untereinander verlinkt. Für viele Anwendungen ist ein vollständiger AST bzw. ASG jedoch zu träge und eine abstraktere Darstellungsform ausreichend. Zu diesem Zweck existiert das *Java Model*, das nur die Strukturen bzw. Schnittstellen erfasst und Details der Anweisungsebene ausspart. Das Java Model wird z.B. in der grafischen Benutzerschnittstelle für die Darstellung der Outline und des Package Explorers verwendet.

Da GOT im Wesentlichen nur Spezialisierungen von den bestehenden Datenstrukturen des Compiler AST bzw. DOM AST ergänzt (z.B. Metavariablen, Query-Klassen und Methoden, generische Teams, Per-Blöcke), aber keine völlig neuartigen Strukturen einführt, wurde die Erweiterung der Datenstrukturen durch die Adaptierung der bereits bestehenden Klassen durch Rollen vorgenommen. Beispielsweise wird eine Query durch eine Rolle repräsentiert, welche die „normale“ Datenstruktur zur Repräsentation einer Methode adaptiert. Eine Query hat viele Eigenschaften mit einer Methode gemeinsam. Die Rolle übernimmt diese Anteile der Basisklas-

se und ergänzt bzw. adaptiert Zustand und Verhalten derart, dass die spezifischen Merkmale einer Query damit realisiert werden. Der Parser wurde so angepasst, dass eine Query zunächst wie eine normale Methode im ASG abgebildet wird, das entsprechende Objekt aber im Anschluss direkt zu der Query-Rolle geliftet wird. Mit den anderen GOT-Elementen wird in gleicher Weise verfahren. Mit Ausnahme der ohnehin generierten Code-Anteile des Parsers konnte die Erweiterung des OTDT somit weitgehend durch Teams und Rollen realisiert werden, ohne dass die Original-Implementierung dafür verändert werden musste.

Auf Basis der erstellten Datenstrukturen erfolgt im weiteren Verlauf die Transformation des GOT-Codes zu OT-Code. Alle Prozessschritte des Compile-Vorgangs, die im Anschluss an die Transformation stattfinden, können daher unverändert im OTDT beibehalten werden. Die eigentliche Erzeugung von Java Bytecode muss also nicht adaptiert werden.

5.1.2 Anbindung an Prolog

Zur Repräsentation des Programms in Prolog wurde eine eigene Datenstruktur, der *GOT AST*, entwickelt. Der GOT AST wird analog zu den anderen Datenstrukturen als „AST“ bezeichnet, obwohl er ebenfalls im eigentlichen Sinne bereits die Informationen eines ASG enthält. Er besteht im Kern aus ca. 110 Klassen zur Abbildung der unterschiedlichen Knotentypen des Programmgraphen. Die Entwicklung erfolgte modellgetrieben auf Basis des *Eclipse Modeling Frameworks* (EMF[19]). Somit kann die Implementierung zu großen Teilen automatisch generiert werden. Die Struktur des GOT AST ähnelt im Aufbau dem DOM AST, ist aber spezialisiert auf die spezifische Funktionalität zur Abbildung in Prolog und zur Transformation. Zur Erzeugung des GOT AST wird der um Binding- und Scope-Informationen angereicherte DOM AST von Eclipse verwendet.

Alle Informationen, die im ASG enthalten sind, können vollständig in Prolog abgebildet werden. Jeder Knotentyp des GOT AST wird durch ein bestimmtes Prolog-Prädikat repräsentiert. Ein GOT AST kann auf dieser Grundlage direkt in eine textbasierte Faktenbasis überführt werden und umgekehrt, wie in Abschnitt 4.1 erläutert. Neben der eigentlichen Funktion von GOT bietet sich dem Programmierer somit zusätzlich die Möglichkeit, über die Benutzerschnittstelle von Prolog auf die Faktenbasis zuzugreifen und das Werkzeug zur individuellen Codeanalyse zu benutzen.

Da in GOT potentiell sehr umfangreiche Faktenbasen analysiert werden müssen, ist Performanz ein wichtiges Kriterium bei der Wahl des Prolog-Interpreters. Der Prototyp wurde u.a. aus diesem Grund mit SWI-Prolog realisiert. Im Prinzip kann GOT aber mit beliebigen Prolog-Interpretern zusammenarbeiten. Zu diesem Zweck wurde die Anbindung an Prolog in einer allgemeinen Schnittstelle abstrahiert. Details zur technischen Umsetzung der Prolog-Anbindung in Eclipse können der Diplomarbeit von Gómez Esperón entnommen werden [42]. Bei der Umsetzung von GOT als Produktivsystem bestünde ein relevanter Bedarf zur Optimierung der Performanz. Abbildung und Evaluierung in Prolog verursachen einen deutlichen Mehraufwand im Verarbeitungsprozess eines Programms. Ein wesentlicher Fortschritt gegenüber dem Prototyp kann hier sicherlich durch den Einsatz inkrementeller Verfahren, wie sie auch im JDT Compiler zum Einsatz kommen, erreicht werden.

5.1.3 Transformation

Die Transformation eines GOT-Programms in ein OT-Programm ist im Prototyp als Quellcode-Transformation umgesetzt. Die Quellcode-Repräsentation des Pro-

[19] Siehe auch `http://www.eclipse.org/modeling/emf/`

gramms kann gleichermaßen wie die Prolog-Faktenbasis direkt aus dem GOT AST generiert werden. Die Transformation des Programms kann somit durch das Hinzufügen, Manipulieren und Entfernen von Knoten des GOT AST realisiert werden, ohne direkt den Quellcode bearbeiten zu müssen (analog zur Codemanipulation beim DOM AST). Potentiell könnte der Schritt zur Erzeugung von Quellcode aber auch ausgelassen werden und das Programm direkt vom Compiler weiterverarbeitet werden.

Der Transformations-Prozess ist dem eigentlichen Compiler vorgelagert und wurde in einem Modul *Transformer* realisiert. Der Transformer erzeugt den GOT AST des Programms und importiert diesen als Faktenbasis in Prolog. Die Teamqueries generischer Teams werden auf Prolog-Queries abgebildet und evaluiert (vgl. Abschnitt 4.2). Die Ergebnisse bilden die Grundlage für die Transformation. Nacheinander werden die Per-Blöcke der generischen Teamklassen expandiert und im Anschluss die übrigen GOT-Programmelemente entfernt (vgl. Abschnitte 4.3 und 4.4). Da der GOT AST und die Faktenbasis inhaltlich äquivalente Darstellungsformen sind, kann die Transformation für beide Darstellungsformen gleichermaßen realisiert werden. Im Prototyp ist die Transformation als Manipulation der Faktenbasis umgesetzt, aus der dann der GOT AST des transformierten Programms erzeugt wird. Auf diese Weise kann der Programmierer bei Bedarf manuell in den Transformationsprozess eingreifen und über die Benutzerschnittstelle von Prolog weitere Analysen oder Manipulationen durchführen. Dies bietet eine zusätzliche Option zum Testen.

5.2 Fazit und gewonnene Erkenntnisse

Insgesamt konnte der gewählte Ansatz zur Realisierung von GOT erfolgreich in einem Prototyp umgesetzt werden. Alle in dieser Arbeit vorgestellten, konkreten GOT-Beispiele sind (mit geringen Unterschieden in einigen syntaktischen Details) mit dem Prototyp evaluiert worden. Die Erweiterung des OTDT mit den Mitteln von OT zu modularisieren ist dabei allerdings nicht vollständig gelungen, auf die Probleme sowie die Vor- und Nachteile wird im Folgenden eingegangen.

Grundsätzlich war es durch den Einsatz von OT möglich, die Erweiterung fast vollständig in eigenständigen Modulen zu separieren. Dabei durften zunächst etliche Teile der komplexen Eclipse-Implementierung ausgespart werden, da diese nach wie vor mit dem unveränderten Basisprogramm verknüpft waren. Beispielsweise konnte auf eine Adaptierung des Java Models komplett verzichtet werden. Das Java Model wird aus dem Compiler AST generiert. Da dieser keine neuen Basisklassen für Knoten enthält, sondern die bestehenden Knoten nur durch Rollenklassen adaptiert werden, erzeugt er nach wie vor eine gültige Instanz des Java Models. Darin fehlen zwar die durch GOT ergänzten Detailinformationen, diese sind aber nur für die grafische Darstellung relevant und können in der prototypischen Umsetzung vernachlässigt werden. Um die Kernfunktionalität zu implementieren, mussten nur ausgewählte Bereiche durch Rollenadaption angepasst werden. Auf diese Weise war es möglich, vergleichsweise schnell zu ersten sichtbaren Ergebnissen zu kommen. Die strikte Modularisierung war auch – wie erhofft – förderlich für die Evolutionsfähigkeit der Implementierung. Tatsächlich war die Aktualisierung auf eine neue Version des OTDT innerhalb von kurzer Zeit mit nur geringem Aufwand möglich.

Negativ ist bei dem gewählten Ansatz anzumerken, dass die Adaption durch Rollen in diesem Szenario sowohl konzeptionell als auch technisch nur eingeschränkt anwendbar ist. Konzeptionell ist fraglich, ob die Idee des Rollenspiels für die Knoten des Compiler AST bzw. DOM AST passend ist. Können die durch GOT hinzukommenden Knoten als kontextabhängige Ausprägungen der bestehenden Knoten interpretiert werden? Eine generische Teamklasse unterscheidet sich inhärent von einer

nicht-generischen Teamklasse – unabhängig vom Kontext. Entsprechend einfacher für das Verständnis wäre es, wenn diese Knotentypen auf der gleichen Ebene und in der gleichen Form, d.h. als eigenständige Klassen in einer Vererbungshierarchie, realisiert werden.

Auf der technischen Seite haben sich bei der Adaption durch Rollen ebenfalls Probleme gezeigt. Der Compiler AST und DOM AST – die maßgeblich betroffenen Datenstrukturen – sind äußerst umfangreich und komplex und deren einzelne Bestandteile sind eng miteinander verzahnt. Bei der Implementierung lag der Fokus u.a. auf der Optimierung der Performanz, was allerdings zu Lasten der Modularisierung und Flexibilität ging. An vielen Stellen werden explizit Fallunterscheidungen anhand von dynamischer Typprüfung getroffen. In einem solchen Szenario wirkt sich eine zusätzliche Dimension der Polymorphie durch Rollen eher negativ auf die Verständlichkeit aus. Ferner existieren zahlreiche Sonderfälle, für deren Adaption keine geeigneten Bindungspunkte in der eigentlichen Schnittstelle zur Verfügung stehen. Die notwendigen Hilfskonstruktionen führen zu einer weiteren Verkomplizierung der Kontrollflüsse. An einigen Stellen war die Adaptierung durch Rollen gar nicht möglich, in diesen Fällen musste das Basisprogramm direkt manipuliert werden. Der Parser verwendet z.B. Tabellen mit statischen, numerischen Konstanten – es ist unmöglich diese mit Rollen zu adaptieren, da hier keinerlei Delegation stattfindet.

Die Datenstrukturen des JDT bzw. OTDT sind allgemein nicht auf eine Erweiterung um neue Knoten ausgelegt, da die Module stark untereinander gekoppelt sind und sich jede Änderung auf zahlreiche Stellen auswirkt. Dies gilt nicht nur für die Adaption durch Rollen, sondern in vielen Punkten auch für die alternative Erweiterung durch Vererbung. Insgesamt erscheint uns daher der Ansatz der direkten Manipulation, der bereits für die Entwicklung des OTDT auf Basis des JDT gewählt wurde, am geeignetsten für eine Umsetzung, auch wenn dies mit starker Kopplung und hohem Aufwand zur Evolution verbunden ist. Die komplexen Details der Datenstrukturen und Kontrollflüsse im JDT stehen der guten Modularisierung einer Erweiterung im Wege.

Kapitel 6

Evaluation

In diesem Kapitel werden die Auswirkung von GOT im Hinblick auf Modularisierung und Wiederverwendung untersucht. Als Gegenstand der Untersuchung wurden die Entwurfsmuster des klassischen Design Pattern Katalogs [39] ausgewählt — aus den folgenden Gründen:

1. Entwurfsmuster sind allgemein bekannt und finden breite Anwendung. Sie beschreiben anerkannt gute Lösungen für häufig auftretende Problemstellungen und dienen somit unmittelbar der Wiederverwendung. Die direkte Wiederverwendung der konkreten Implementierung eines Entwurfsmusters ist dabei allerdings nicht oder nur eingeschränkt möglich, da sie an vielen Stellen spezifische Details enthält.

2. Entwurfsmuster beschreiben oftmals kontextspezifische Kollaborationen von Objekten und definieren somit natürlicherweise Rollen. Eine Programmiersprache mit Rollenunterstützung unterstützt die direkte Abbildbarkeit solcher Entwürfe in die Implementierung.

3. Entwurfsmuster werden in der Literatur von mehreren Autoren für vergleichende Studien zu aspektorientierten Programmiersprachen herangezogen [17, 40, 43, 49, 87, 91, 96]. Im Rahmen solcher Studien werden Entwurfsmuster als eine signifikante Stichprobe betrachtet, da sie eine Vielfalt wiederholt auftretender, relevanter Problemstellungen behandeln. Es ist anzunehmen, dass sich die hier gewonnenen Erkenntnisse auf eine große Anzahl von Anwendungsfällen verallgemeinern lassen.

Sollte der Leser allgemein nicht mit Entwurfsmustern vertraut sein, so empfehlen wir vorab die Lektüre von [39]. Eine kurze Beschreibung der einzelnen Entwurfsmuster ist aber auch im Rahmen der jeweiligen Diskussion gegeben.

6.1 Aufbau und Kriterien

In den folgenden Abschnitten 6.2 bis 6.4 wird die Realisierung aller 23 Entwurfsmuster des Katalogs in GOT bzw. OT allgemein oder anhand von konkreten Beispielen diskutiert. Abschnitt 6.5 gibt einen kurzen Ausblick auf andere Anwendungen. In Abschnitt 6.6 werden die Ergebnisse im Kontext vergleichbarer Studien zu Entwurfsmustern diskutiert. Hannemann und Kiczales evaluieren in [49] AspectJ, Monteiro und Gomes tun dies in [87] für OT. Unsere Arbeit bietet nach unserem Kenntnisstand neben der Arbeit von Monteiro und Gomes die einzige umfassende Betrachtung von OT in Bezug auf Design Patterns - wir kommen dabei zu deutlich unterscheidbaren Erkenntnissen.

Terminologie

Im Kontext von Design Patterns wird der Begriff der *Rolle* für Klassen (bzw. Interfaces) oder konkrete Objekte verwendet. Zur besseren Unterscheidung von Rollen in OT werden diese Rollen der Entwurfsmuster fortan als *Musterrollen* bezeichnet. Für die Analyse der verschiedenen Muster ist es sinnvoll, die beteiligten Musterrollen nach zwei Arten zu unterscheiden [49]:

- *Ergänzende Musterrollen* werden Klassen zugeordnet, die eine eigene Funktionalität – unabhängig vom Entwurfsmuster – besitzen. Ein Beispiel für ergänzende Musterrollen sind die Rollen *Subject* und *Observer* im Observer Pattern. Die eigentliche Aufgabe der Klasse ist außerhalb des Musters definiert. Die Klasse nimmt mit ihrer Musterrolle eine zusätzliche Sekundärfunktionalität an, die entweder mit der Primärfunktionalität kombiniert wird oder diese in unabhängiger Form erweitert. Idealerweise werden Primär- und Sekundärfunktionalität in getrennten Modulen definiert.

- *Eigenständige Musterrollen* werden vollständig über das Entwurfsmuster definiert und besitzen keine Funktion außerhalb des Musters. Ein Beispiel dafür ist die Musterrolle *Facade* im gleichnamigen Muster. Eine Klasse zur Realisierung einer eigenständigen Musterrolle wird erst mit der Anwendung des Entwurfsmusters im Programm eingeführt.

Die Evaluation in diesem Kapitel wird u.a. anhand von Szenarien und Beispielen erläutert. Als *Szenario* wird ein konkreter Anwendungsfall beschrieben – meist in Form einer Anforderung bzw. Funktionalität, die im Kontext einer umfangreicheren Anwendung umzusetzen ist. Die Implementierung eines Entwurfsmusters zur Umsetzung dieser Anforderung des Szenarios wird als *Beispiel* bezeichnet.

Kriterien für den sinnvollen Einsatz von Teams und Rollen

Im Vordergrund der Evaluierung steht die Frage, inwieweit durch die Sprachmittel von GOT bzw. OT eine Verbesserung der Modularisierung und der Wiederverwendbarkeit gegenüber Java erreicht werden kann.

Eine Verbesserung der Modularisierung ergibt sich aus einer geeigneten Entkopplung verschiedener Anforderungen in getrennten Modulen. Eine grundlegende Voraussetzung ist hierbei die *Lokalität* des Codes, d.h. die Lokalisierung des gesamten Codes zur Umsetzung einer bestimmten Anforderung in einem eigenständigen Modul, das wiederum keinen anderen Code enthält. Die Funktion des Moduls soll unabhängig von anderen Modulen verständlich sein und nur von seiner unmittelbaren Umgebung beeinflusst werden. Die Lokalität ist auch in den vergleichbaren Studien [49, 87] das wesentliche Kriterium zur Beurteilung der Modularisierung. Diese Studien heben noch weitere Facetten (Composition Transparency und Un-/Pluggability) der Modularisierung gesondert hervor. Bei unserer Diskussion fließen diese Kriterien direkt in die qualitative Beurteilung der Modularisierung ein. Eine gelungene Modularisierung ist Voraussetzung für die Wiederverwendung.

Eine Verbesserung der Wiederverwendbarkeit ist dann gegeben, wenn ein Modul einen sinnvollen Beitrag zur Lösung von mehreren Problemstellungen leisten kann. Hannemann und Kiczales nennen in ihrer Studie zu AspectJ das Vorhandensein eines abstrakten Aspekts (analog zu einer abstrakten Klasse) als wesentliches Kriterium für die Wiederverwendbarkeit. Wir stehen dieser Auffassung kritisch gegenüber: das Vorhandensein eines abstrakten bzw. verfeinerbaren Moduls ist vielleicht ein notwendiges aber noch kein hinreichendes Kriterium für die Wiederverwendbarkeit. Der Einsatz eines abstrakten Moduls reduziert nicht zwangsläufig den Implementierungsaufwand einer Lösung. Wenn der eigentliche Mehrwert des Moduls gering ist,

so kann sich dessen Einsatz durch die Aufwände zur Wiederverwendung (Abstraktion, Selektion, Spezialisierung, Integration) insgesamt negativ auf die Produktivität auswirken (vgl. Abschnitt 2.4 zur Wiederverwendung). Monteiro und Gomes führen aus, dass ihre Kriterien zur Wiederverwendbarkeit „ein bisschen anspruchsvoller“ seien, dahingehend, dass sie den Einsatz des abstrakten Moduls in mindestens zwei Beispielen verlangen. Ob dadurch ein tatsächlicher Mehrwert gegeben ist, lassen auch sie dahingestellt. Wir werden die Beurteilung zur Wiederverwendbarkeit beider Studien in Abschnitt 6.6 detailliert diskutieren. Bei unserer Beurteilung der Wiederverwendbarkeit wird daher explizit der *sinnvolle* Beitrag zu mehreren Lösungen in unterschiedlichen Szenarien verlangt. Dieser ist gegeben, wenn ein verfeinerbares Modul einen relevanten Teil der Funktionalität zu einer Lösung beisteuert (*Sharing*), mit dem sich der Programmierer nicht im Detail befassen muss (*Information Hiding*).

Ob der Einsatz von Teams und Rollen gegenüber einer rein klassenbasierten Implementierung in Java tatsächlich einen Mehrwert bietet, entscheidet sich an der Frage, ob dadurch eine sinnvolle Umsetzung kontextabhängiger Funktionalität ermöglicht wird. Zur Beurteilung verwenden wir konkret zwei qualitative Kriterien:

1. *Kontextabhängiges Verhalten.* Dient eine Rolle der Spezialisierung einer Basis, indem sie kontextabhängigen Methoden-Dispatch (durch Callins) definiert?

2. *Kollaboration im Kontext.* Bildet ein Team ein Modul, das größer ist als eine einzelne Klasse, indem es den Kontext für die Kollaboration mit einer Rolle oder von Rollen untereinander repräsentiert?

Quantitativ werden diese Kriterien dahingehend beurteilt, ob sie für eine relevante Menge von Lösungen (und nicht lediglich in Ausnahmefällen) erfüllt werden. Die folgenden beiden Arten von Anwendungen stufen wir auf dieser Basis explizit als ungeeignet ein:

1. *Pseudo-Rollen.* Eine Rolle, die das Interface der Basis abbildet, ohne eine relevante Anpassung oder Ergänzung vorzunehmen, und die zur Implementierung lediglich Callouts (zur simplen Weiterleitung der Methodenaufrufe) definiert, stellt in unseren Augen keinen sinnvollen Anwendungszweck für eine Rolle dar. Es findet weder eine Adaptierung noch eine Entkopplung statt. Die Basis kann in solchen Fällen genauso gut für sich selbst stehen.

2. *Pseudo-Teams.* Ein Team, dessen Rollen weder kontextabhängiges Verhalten, noch eine Kollaboration untereinander oder mit dem Team selbst aufweisen, repräsentiert nach unserem Verständnis keinen Kontext. Die spezifische Funktionalität des Rollenspiels kommt hier nicht zum Einsatz. Eine derartige Struktur unterscheidet sich nicht relevant von einem Verbund normaler Klassen bzw. Objekte.

Sowohl für Pseudo-Rollen als auch für Pseudo-Teams existieren in Ausnahmefällen sinnvolle Anwendungen, i.A. sehen wir darin allerdings einen Code-Smell und halten den dadurch entstehenden Overhead für nicht gerechtfertigt.

Für die weiterführende Beurteilung, ob generische Lösungen in GOT einen Mehrwert gegenüber reinen OT-Lösungen bieten, sind zwei Kriterien ausschlaggebend, die in Abschnitt 3.2 ausführlich erläutert wurden:

1. *Introspektion.* Trägt Introspektion sinnvoll zur Lösung bei?

2. *Quantifizierung.* Trägt Quantifizierung sinnvoll zur Lösung bei?

6.2 Verbesserung durch generische Teams

In diesem Abschnitt werden jene Entwurfsmuster-Lösungen vorgestellt, bei denen durch den Einsatz von GOT ein sinnvoller Beitrag zur Modularisierung und ggf. auch zur Wiederverwendbarkeit geleistet wird. Es handelt sich um die Entwurfsmuster Visitor, Memento, Abstract Factory, Observer und Mediator.

6.2.1 Visitor

Als erstes Beispiel wird das Entwurfsmuster *Visitor* präsentiert. Das Visitor Pattern wird angewendet auf Objektstrukturen, an denen viele Klassen mit eventuell unterschiedlichen Interfaces beteiligt sind, z.B. Baumstrukturen mit verschiedenen Knotentypen. Ziel des Entwurfsmusters ist es, Operationen auf der Objektstruktur in eigenständigen Modulen statt der Struktur selbst zu implementieren. Damit bleibt die Struktur überschaubar und kann um neue Funktionalität erweitert werden, ohne dass die Klassen der Struktur geändert werden müssen. Das Entwurfsmuster definiert die Musterrolle des *Elements* für die Objekte der Struktur (z.B. für die Knoten eines Baums) im Kontext eines *Visitors*. Der Visitor besucht die Elemente, um auf ihnen eine Operation auszuführen. Das Muster realisiert Double Dispatch (eine Variante von Multiple Dispatch) zwischen den Typen `Element` und `Visitor` in einer Sprache wie Java, die eigentlich nur Single Dispatch unterstützt.

Obwohl das Visitor Pattern nach einer Trennung von Operationen und Struktur strebt, so erfordert der Kernmechanismus doch zunächst eine Erweiterung des Element-Typs um eine Methode zum Double Dispatch mit dem Visitor. Dazu implementiert jede einzelne Element-Klasse eine Methode `accept`, welche am übergebenen Visitor eine `visit`-Methode aufruft. Die Implementierung der Methode `accept` ist spezifisch für jede Element-Klasse, da sie vom Typ des Self-Arguments abhängt, und kann daher nicht geerbt werden.

OT bietet eine bessere Modularisierung der Visitor-Funktionalität dahingehend, dass keinerlei Anpassung der Element-Klassen erforderlich ist. Die gesamte Visitor-Funktionalität kann in einem Modul gekapselt werden. Ein Beispiel, das bereits zusätzliche Verbesserungen durch GOT enthält, ist in Listing 6.1 dargestellt. Im Rahmen des Beispiels gehen wir davon aus, dass eine Baumstruktur existiert, die auf einer Klasse `TreeElement` als allgemeiner Oberklasse für Elemente des Baums basiert. Im Beispiel ist mit `TreeNode` als Subklasse von `TreeElement` ein solches Element dargestellt. In OT spielen die Baumelemente die Rolle `Visitable` (Zz. 17–20) im Kontext eines Visitors (der Team-Klasse). Der Visitor vergibt explizit die `Visitable`-Rolle an Element-Objekte durch Declared Lifting in der Methode `acceptMe` (Z. 13). Die Methode `acceptMe` ruft die Methode `accept` an der Rolle auf (Z. 14), die ihrerseits eine `visit`-Methode des Visitors aufruft (Z. 19) — die Aufrufkette realisiert den Double Dispatch. Die eigentliche Operation des Visitors wird in den `visit`-Methoden implementiert (im Code durch „...“ abstrahiert), wobei jeweils eine überladene Variante für jeden Element-Typ existiert. Wenn der Visitor dabei die Kind-Elemente eines Elements besuchen soll, so wird dies durch den Aufruf der Methode `acceptMe` umgesetzt; beispielhaft dargestellt für einen Knoten `TreeNode` mit zwei Kind-Elementen *Left* und *Right* (Zz. 7 und 9).

Die einzige Aufgabe der Rolle `Visitable` ist es, eine Methode `accept` zur Verfügung zu stellen, die den Rückruf auf eine `visit`-Methode am Visitor macht. Die richtige Variante der überladenen `visit`-Methoden wird dabei durch den statischen Typ des Self-Arguments ermittelt, der in diesem Fall automatisch dem durch Lowering gewählten statischen Typ der Basisklasse der Rolle entspricht. Die allgemeine Rolle `Visitable` kann an dieser Stelle allerdings nur den allgemeinen Typ `TreeElement` liefern. Für jede Element-Klasse wird daher eine spezifische Subrolle

benötigt, welche die Methode `accept` überschreibt, um an dieser Stelle ein spezifisches Self-Argument für die Auswahl der korrekten `visit`-Methode zu übergeben.

```
1  public team class Visitor
2    match(?Class ?base)
3    {StdQ.isAncestor(TreeElement.class, ?base)}
4  {
5    public void visit(TreeElement element) {...}
6    public void visit(TreeNode node) {
7      acceptMe(node.getLeft());
8      ...
9      acceptMe(node.getRight());
10   }
11   ... // weitere überladene visit-Methoden
12
13   public void acceptMe(TreeElement as Visitable v) {
14     v.accept(this);
15   }
16
17   protected class Visitable playedBy TreeElement {
18     protected void accept(Visitor v)
19     {v.visit(this);}
20   }
21
22   per (?base) {
23     protected class -Visitable extends Visitable playedBy
         ?base {
24      protected void accept(Visitor v)
25      {v.visit(this);}
26     }
27   }
28 }
```

Listing 6.1: Das Entwurfsmuster *Visitor* für eine Baumstruktur in GOT

Die aufwändige Gestalt der Rückruf-Implementierung ist zwingend erforderlich für den Double Dispatch Mechanismus, unabhängig davon ob die Implementierung in Java oder OT erfolgt. Hierbei handelt sich um einen Crosscutting Concern; eine Lösung mittels Quantifizierung in GOT ist daher naheliegend. Dazu wird lediglich eine generische Rolle `-Visitable` als Subklasse von `Visitable` benötigt (Zz. 23–26). Die Rolle definiert als Basis die Metavariable `?base` (Z. 23), die von der Teamquery an alle Nachfahren der Klasse `TreeElement` gebunden wird (Z. 3). Die Rolle ist in einen Per-Block eingeschlossen, der für jede Belegung von `?base` ausgerollt wird. Durch die Code-Transformation wird also eine Instanz der Rolle `-Visitable` pro Basisklasse generiert. Die überschriebene Methode `accept` bedarf keiner Änderung, da sie bereits implizit durch den veränderten statischen Typ des Arguments `this` angepasst wurde. Aufgrund von Smart Lifting wird bei Ausführung von `acceptMe` (Z. 12) automatisch die spezifischste Subrolle aus der Hierarchie gewählt, die zum dynamischen Typ des Elements passt.

Fazit *Visitor*. OT ermöglicht im gegebenen Beispiel eine vollständige Trennung der kontextbezogenen Implementierung des Visitors von der Implementierung der Element-Klassen. Der Crosscutting Concern kann in einem Modul gekapselt werden. GOT verbessert dabei die Entkopplung von Rollen- und Basisklassen. Zur

Umsetzung der generischen Anteile wird lediglich eine Metavariable benötigt. Die generische Implementierung vermeidet duplizierten Code in Gestalt gleichartiger Rollen und `accept`-Methoden. Dies dient u.a. der Wartbarkeit des Codes. Möchte man z.B. die `accept`-Methode um zusätzliche Argumente erweitern, etwa das aktuelle Eltern-Element oder die aktuelle Tiefe des Baums, so muss dies nur einmalig an einer zentralen Stelle des Codes implementiert werden. Auch die Wiederverwendbarkeit wird durch Sharing und Information Hiding gefördert. Sowohl die generische Rolle `-Visitable` als auch die nicht-generische Rolle `Visitable` (mit Ausnahme der Bindung der Basisklasse) können hier zur Wiederverwendung in ein VGT ausgelagert werden. Bei der Wiederverwendung muss der Programmierer sich nicht mehr mit den Details der Rollen auseinandersetzen.

6.2.2 Memento

Ein anspruchsvolleres Beispiel wird mit dem Entwurfsmuster *Memento* gegeben. Das Muster kommt zum Einsatz, wenn der interne Zustand eines Objekts – *Originator* genannt – vorübergehend gespeichert werden soll, damit er zu einem späteren Zeitpunkt wiederhergestellt werden kann. Dies kann z.B. im Kontext einer *Undo*-Funktionalität erforderlich sein. Eine wesentliche Problemstellung dabei ist, dass das Objekt normalerweise seinen Zustand kapselt, und dieser daher nicht durch externe Zugriffe ausgelesen werden kann. Eine Aufweichung der Kapselung zu diesem Zweck ist nicht erwünscht. Lösen lässt sich das Problem durch Verwendung eines Memento-Objekts, das eine Momentaufnahme des Zustands eines Originator-Objekts speichert. Das Originator-Objekt selbst erzeugt das Memento-Objekt und nutzt es, um den gespeicherten Zustand wiederherzustellen. Das Interface des Memento-Objekts sollte dabei externe Zugriffe soweit möglich nur vonseiten des Originator-Objekts erlauben.

Das Entwurfsmuster definiert die Musterrolle *Memento* (Objekt, das den Zustand speichert), die gespielt wird von einem *Originator* (Objekt, dessen Zustand gespeichert wird) im Kontext eines *Caretakers* (Objekt, das ein Memento vom Originator benötigt). Die Umsetzung mit OT bietet sich hier in besonderer Weise an, da aufgrund von Gradual Decapsulation [57] eine Aufweichung der Kapselung in OT nicht erforderlich ist. Darüber hinaus kann der Zugriff auf das Memento durch das Prinzip von *Confined Roles* [54] strikt auf den Kontext beschränkt werden.

Wir werden zwei verschiedene Varianten für eine wiederverwendbare Implementierung der allgemeinen Anteile des Musters in GOT zeigen. Die erste Variante zeigt den einfachen Fall, dass die Memento-Funktionalität für genau eine Basisklasse benötigt wird, die für die Anwendung durch das AGT konkret vorgegeben wird. In der zweiten Variante wird die Memento-Funktionalität für beliebige Klassen inklusive ganzer Hierarchien von Subklassen verallgemeinert.

Memento für eine spezifische Basisklasse

Listing 6.2 zeigt die erste Variante zur Umsetzung des Musters. Das VGT `GenericCaretaker` definiert eine Rolle

```
protected class Memento playedBy ?base {...}
```

(Z. 14), deren Basisklasse durch eine Metavariable repräsentiert wird. Diese Metavariable wird erst durch ein erbendes AGT mit einer konkreten Klasse belegt. Wir fordern in diesem Fall eine eindeutige Belegung der Metavariablen `?base` (definiert durch die Annotation `@UniqueMatch` in der Teamquery in Z. 2), da das Team darauf ausgelegt ist, dass genau eine Rolle existiert. Mit der geforderten Eindeutigkeit der Basisklasse ist die Existenz der Rolle `Memento` gesichert: durch die zwingend eindeutige Belegung gibt es nach der Transformation in jedem Fall genau eine Ausprägung

der Rolle. Die Rolle muss daher nicht generisch deklariert werden.

Ein Klient kann durch Aufruf der Methoden `save` und `restore` explizit den Zustand eines Objekts der Basisklasse in der Rolle speichern und wiederherstellen. Beide Methoden haben einen Typparameter `B` für den noch unbekannten Typ der Basisklasse, z.B.

```
public <B base Memento> void save(B as Memento p)
```

Der Modifikator `base` bindet den Typparameter relativ zum Typ der Rolle und sichert zu, dass der Typ zu `Memento` geliftet werden kann. Der Typ `Memento` darf außerhalb des Per-Blocks verwendet werden, da er nicht generisch deklariert ist. Diese Voraussetzung vereinfacht die Struktur der Implementierung, da somit explizit Rollenobjekte an dieser Stelle durch Declared Lifting erzeugt werden können. Ein erbendes AGT, das eine Definition der Basisklasse liefert, kann statisch die Typsicherheit von Aufrufen an den Methoden prüfen.

```
1  abstract team class GenericCaretaker
2    declare (@UniqueMatch ?Class ?base, ?Type ?tf, ?Field ?f) {
3    StdQ.fieldOfClass(?f, ?base) && StdQ.typeOfField(?tf, ?f)
4  }
5  {
6    public <B base Memento> void save(B as Memento p) {
7      p.saveState();
8    }
9    public <B base Memento> void restore(B as Memento p) {
10     p.resetState();
11   }
12
13   per (?base) {
14     protected class Memento playedBy ?base {
15       per (?f, ?tf) {
16         private ?tf -memState;
17
18         abstract ?tf -getState();
19         -getState -> get ?f;
20         abstract void -setState(?tf s);
21         -setState -> set ?f;
22
23         public void saveState() {
24           -memState = -getState();
25         }
26         public void resetState() {
27           -setState(-memState);
28         }
29       }
30     }
31   }
32 }
```

Listing 6.2: Beispielimplementierung für das Entwurfsmuster *Memento* in GOT

Den Kern der Implementierung bildet der Inhalt der Rolle `Memento`. Die Rolle enthält einen Per-Block (Zz. 15–29), dessen Metavariablen `?f` und `?tf` von der Teamquery mit allen Feldern der Basis und deren zugehörigen Typen belegt werden (Z. 3). Die Rolle dupliziert jedes Feld der Basisklasse mit dem Feld `-memState`

(Z. 16). Dazu definiert sie die Methoden `-getState` und `-setState` (Zz. 18–21), die den Zugriff per Callout auf das korrespondierende Feld der Basisklasse erlauben. Die Methoden `saveState` und `resetState` (Zz. 23–28) dienen zum Speichern und Wiederherstellen des Zustands. Die Methoden sind selbst nicht generisch und können somit außerhalb des Per-Blocks sicher verwendet werden (Zz. 7 und 10). Innerhalb ihres Methodenrumpfes sind nun generische Anweisungen definiert, die den Wert eines jeden Feldes der Basis im duplizierten Feld der Rolle speichern (Z. 24) bzw. den gespeicherten Wert wiederherstellen (Z. 27).

Listing 6.3 zeigt die Implementierung eines spezifischen Caretakers. Das AGT `PointCaretaker` erbt vom VGT `GenericCaretaker`. Die einzig notwendige Verfeinerung, um einen vollständigen Caretaker zu implementieren, besteht in der Spezifizierung einer eindeutigen Basisklasse für die Memento-Rolle. In diesem Fall geben wir die Klasse `Point` an (Z. 4).

```
1 public team class PointCaretaker extends GenericCaretaker
2   match (?Type ?tf, ?Field ?f)
3 {
4   super(Point.class, ?tf, ?f)
5 }
6 {}
```

Listing 6.3: Eine vollständige Caretaker-Implementierung für die spezifische Basisklasse `Point`

Der Programmierer kann das AGT `PointCaretaker` nun direkt instanziieren und damit arbeiten. Ferner hat er natürlich die Möglichkeit weitere Verfeinerungen und Erweiterungen daran vorzunehmen, wie es bei Vererbung gegeben ist.

Der transformierte Code für das Beispiel wird in Listing 6.4 dargestellt. Es zeigt die vollständige Implementierung der Klasse `GenericPointCaretaker` als konkrete Ausprägung des VGTs `GenericCaretaker` für das AGT `PointCaretaker`. Die Rolle Memento (Zz. 9–30) ist für die Basisklasse `Point`, die zwei Felder x und y enthalte, definiert. Die Klasse `PointCaretaker` erbt nun von der spezifischen Klasse `GenericPointCaretaker` (Z. 33) und ist ansonsten leer, da sie ausschließlich zur Vervollständigung der Teamquery der Oberklasse verwendet wurde und keine eigene Funktionalität beisteuert.

Zusammenfassend betrachtet ermöglicht die Klasse `GenericCaretaker` die Wiederverwendung der Memento-Funktionalität für verschiedene Basisklassen. Der gezeigte Ansatz ist auch auf anspruchsvollere Szenarien übertragbar. Eine differenziertere Abbildung der Felder einer Basisklasse kann durch eine verfeinerte Teamquery erreicht werden, wobei z.B. auch Annotationen an den Merkmalen der Basis einbezogen werden können.

Die vorgestellte Variante hat allerdings auch eine bedeutende Einschränkung: die Rolle `Memento` kann keine geerbten Felder berücksichtigen, deren Zugriff durch Kapselung beschränkt ist (was dem Normalfall entspricht). Ursache hierfür ist, dass der Callout auf ein Feld (zu sehen in Listing 6.4, Zz. 12, 14, 18 und 20) nur für zugreifbare Felder der Basisklasse funktioniert. Ein Bruch dieser Kapselung ist in OT nicht erlaubt. Sollte also die Rolle auf eine Basisklasse angewendet werden, die Attribute von einer anderen Klasse erbt, so wären diese Teile des Zustands von der Funktionalität ausgeschlossen. Entsprechend werden in der Teamquery von vornherein ausschließlich die Felder der Basisklasse ausgewählt (mit `StdQ.fieldOfClass`), ohne geerbte Felder mit einzuschließen. Wie kann nun die Memento-Funktionalität auf Klassen in einer Vererbungshierarchie angewendet werden, wie z.B. die Klasse `Circle` aus dem eingangs aufgeführten Beispiel in Abbildung 3.1 (Seite 40)?

```
1  abstract team class GenericPointCaretaker {
2    public <B base Memento> void save(B as Memento p) {
3      p.saveState();
4    }
5    public <B base Memento> void restore(B as Memento p) {
6      p.resetState();
7    }
8
9    protected class Memento playedBy Point {
10     private double memStateX;
11     abstract double getStateX();
12     getStateX -> get x;
13     abstract void setStateX(double s);
14     setStateX-> set x;
15
16     private double memStateY;
17     abstract double getStateY();
18     getStateY -> get y;
19     abstract void setStateY(double s);
20     setStateY-> set y;
21
22     public void saveState() {
23       memStateX = getStateX();
24       memStateY = getStateY();
25     }
26     public void resetState() {
27       setStateX(memStateX);
28       setStateY(memStateY);
29     }
30   }
31 }
32
33 public team class PointCaretaker extends
       GenericPointCaretaker
34 {}
```

Listing 6.4: Der transformierte Code für die Anwendung des AGTs `PointCaretaker`

Memento für beliebige Basisklassen und Subklassen-Hierarchien

Die Implementierung der vorigen Variante kann derart erweitert werden, dass sie die ganze Vererbungshierarchie einer Basisklasse einschließt. Dabei muss die Rollen-Hierarchie analog zur Hierarchie der Basisklassen aufgebaut werden. Auf diese Weise ist jedes Feld einer Basis in der zugeordneten Rolle ohne Kapselungsbruch für den Callout verfügbar. Theoretisch könnte man jede Ebene der Vererbungshierarchie manuell durch eine generische Rolle abbilden. Dies ist allerdings aufwändig und abgesehen von der eigentlichen Vererbungsbeziehung zwischen den Rollen redundant. Alternativ ist es möglich, eine Subklassen-Hierarchie beliebiger Tiefe mit Quantifizierung zu erfassen. Dies erfordert eine selbstbezügliche Klassendefinition, die in einem rekursiven Prozess aufgelöst wird (vgl. Abschnitt 4.4.4).

Die Implementierung dieser Variante ist in Listing 6.5 dargestellt. Die Umsetzung erfordert drei Änderungen gegenüber der vorigen Variante:

```
1  abstract team class UniversalCaretaker
2    declare (?Class ?base, ?Type ?tf, ?Field ?f) {
3    StdQ.fieldOfClass(?f, ?base) && StdQ.typeOfField(?tf, ?f)
4  }
5  {
6    public <B base MementoRoot> void save(B as MementoRoot
       p) {
7      p.saveState();
8    }
9    public <B base MementoRoot> void restore(B as
       MementoRoot p) {
10     p.resetState();
11   }
12
13   protected class MementoRoot {
14     public void saveState() {};
15     public void resetState() {};
16   }
17
18   per (?base) {
19     protected class -Memento extends ?base.parentRole(
         MementoRoot.class) playedBy ?base {
20       per (?f, ?tf) {
21         private ?tf -memState;
22
23         abstract ?tf -getState();
24         -getState -> get ?f;
25         abstract void -setState(?tf s);
26         -setState -> set ?f;
27
28         public void saveState() {
29           super.saveState();
30           -memState = -getState();
31         }
32         public void resetState() {
33           super.resetState();
34           -setState(-memState);
35         }
36       }
37     }
38   }
39 }
```

Listing 6.5: Universell anwendbare Variante des Entwurfsmusters *Memento* in GOT

1. Das Team wird für eine variable Anzahl von Memento-Rollen ausgelegt. Die Template-Rolle zur Realisierung muss daher generisch deklariert werden: als `-Memento` (Z. 19). Als Ersatz für die eindeutige Rolle, die in den Methoden `save` und `restore` anzugeben ist, wird die Rolle `MementoRoot` eingeführt (Zz. 13-16). Die Rolle bietet eine Schnittstelle mit einer leeren Implementierung und erfüllt so die Mindestanforderung zur Typsicherheit für die nicht generischen Anteile des Teams. Alle generisch aus `-Memento` erzeugten Rollen nutzen `MementoRoot` als direkte oder indirekte Oberklasse und spezialisieren

deren Funktionalität. Smart Lifting sorgt in diesem Fall wieder dafür, dass die jeweils spezifischste Rolle für ein Basisobjekt zum Einsatz kommt. Da nun auf die Eindeutigkeit der Basisklasse verzichtet wird, muss die Metavariable `?base` nicht mehr als `@UniqueMatch` annotiert werden (Z. 2) .

2. Die selbstbezügliche Vererbungsbeziehung der Rolle `-Memento` wird nun durch die Anweisung

   ```
   extends ?base.parentRole(MementoRoot.class)
   ```

   hergestellt (Z. 19). Die Transformation wird an dieser Stelle die jeweils spezifischste Subrolle von `MementoRoot` wählen (analog zum Smart Lifting, unter Ausschluss der zu definierenden Rolle), die für diese Basis in Frage kommt – die gewählte Rolle wird als Oberklasse der neu zu erstellenden Rolle definiert. Ist keine solche Rolle vorhanden, so wird an dieser Stelle direkt `MementoRoot` als Oberklasse gewählt.

3. Die Funktionalität eines Mementos ergibt sich nun kumuliert über die Ebenen der Vererbungshierarchie. Entsprechend müssen die Methoden `saveState` und `resetState` um einen `super`-Aufruf erweitert werden (Zz. 29 und 33).

Als Beispiel für eine konkrete Anwendung zeigen wir in Listing 6.6 die Umsetzung des Memento Patterns für die auf der Klasse `Shape` basierende Klassenhierarchie (vgl. Abbildung 3.1 auf Seite 40). Das AGT `ShapeCaretaker` muss lediglich eine konkrete Angabe zu den gewünschten Basisklassen, für die Memento-Rollen erstellt werden sollen, machen. Die Auswahl der Klassen `Shape` und all ihrer Unterklassen kann durch eine simple Verfeinerung der Teamquery erfolgen (Z. 4), die übrige Implementierung bedarf keiner Anpassung.

```
1 team class ShapeCaretaker extends UniversalCaretaker
2   match (?Class ?base, ?Type ?tf, ?Field ?f) {
3   super(?base, ?tf, ?f) &&
4   (Shape.class=?base || isAncestor(Shape.class, ?base))
5 }
6 {}
```

Listing 6.6: Ein AGT als Subklasse des VGTs `UniversalCaretaker`

Das Resultat der Transformation des VGTs `UniversalCaretaker` ist in Listing 6.7 skizziert. Der wesentliche Unterschied liegt in der Definition der Vererbungsbeziehung der Rollen. Die Rollenhierarchie spannt nun einen Baum mit der Rolle `MementoRoot` als Wurzelelement auf. Die Hierarchie der Subrollen spiegelt die Hierarchie der Basisklassen wider.

```
1 abstract team class UniversalCaretaker_Shape {
2   ...
3   protected class MementoShape extends MementoRoot
        playedBy Shape {...}
4   protected class MementoCircle extends MementoShape
        playedBy Circle {...}
5   protected class MementoRectangle extends MementoShape
        playedBy Rectangle {...}
6 }
```

Listing 6.7: Der transformierte Code des `UniversalCaretaker` für eine Anwendung auf `Shape` inklusive aller Subklassen

Fazit *Memento*. OT ermöglicht für das Memento Pattern eine vollständige Modularisierung der kontextbezogenen Funktionalität in einem Team. Zur Umsetzung ist keine Anpassung der Basisklassen erforderlich. Mit GOT wird die Implementierung für verschiedene Anwendungen wiederverwendbar. Durch Quantifizierung und generische Anweisungen ist die Struktur so flexibel gestaltet, dass sie auf Basisklassen unterschiedlicher Art anwendbar ist. Die Implementierung kann auch derart erweitert werden, dass sie die verschiedenen Basisklassen inklusive ganzer Vererbungshierarchien einschließt.

6.2.3 Abstract Factory

Das Entwurfsmuster *Abstract Factory* dient dazu, eine Gruppe von Fabrikmethoden zu bündeln, die Objekte einer gemeinsamen Familie von Klassen erzeugen. Die abstrakte Fabrik besitzt ein generisches Interface, das von den spezifischen Subklassen der Fabrik so implementiert wird, dass sie mit den Fabrikmethoden jeweils unterschiedliche Klassen instanziieren. Ein konkretes Anwendungsbeispiel ist eine abstrakte Fabrik zur Erzeugung von grafischen Steuerelementen für eine Benutzerschnittstelle. Es existieren unterschiedliche Arten von Steuerelementen, wie z.B. *Buttons* und *Labels*, die bei der Gestaltung der Schnittstelle zum Einsatz kommen. Eine mögliche Anforderung dabei ist, dass die Steuerelemente alle einen bestimmten Stil bzw. Look-and-feel unterstützen. In Java existieren beispielsweise mehrere weitgehend unabhängige Bibliotheken für grafische Schnittstellen, u.a. AWT und Swing, die eigene Klassenhierarchien für Steuerelemente auf Basis einer allgemeinen Oberklasse (`java.awt.Component` bzw. `java.swing.JComponent`) definieren. Um in solchen Fällen die Anwendung unabhängig von der Wahl einer konkreten Klassenfamilie zu gestalten, kann eine abstrakte Fabrik zum Einsatz kommen. Die Fabrik muss zur Erzeugung der konkreten Steuerelemente jeweils eine Fabrikmethode, wie etwa `createButton` und `createLabel`, anbieten. Als Rückgabetyp wird ein Interface eingesetzt, dass von den konkreten Klassen implementiert werden muss. Ein Beispiel für das Interface einer solchen Fabrik ist `IFactory`:

```
public interface IFactory {
  public IButton createButton();
  public ILabel createLabel();
}
```

Die Anwendung kann nun unabhängig von den spezifischen Steuerelement-Klassen entwickelt werden. Zur konkreten Ausgestaltung muss der Anwendung lediglich eine spezifische Fabrik übergeben werden. Abbildung 6.1 zeigt beispielhaft zwei Klassenhierarchien `X` und `Y`, die dazu in Frage kommen.

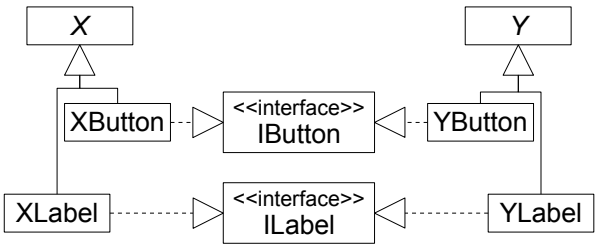

Abbildung 6.1: Szenario zur Anwendung des Entwurfsmusters *Abstract Factory*

Die Implementierung einer konkreten Fabrik stellt ein Quantifizierungsproblem dar. Die Struktur ist stets die gleiche, vorgegeben durch `IFactory`, nur die konkre-

ten Klassen, die zur Instanziierung verwendet werden, unterscheiden sich. Sofern die Konstruktion der Objekte in standardisierter Weise erfolgt, kann das Problem in GOT in universeller Weise gelöst werden. Das generische Team `GenericFactory`, dargestellt in Listing 6.8, bietet eine Implementierung aller Fabrikmethoden, wobei automatisch die zu instanziierenden Klassen der jeweiligen Interfaces identifiziert und zugeordnet werden. Die Auswahl der Klassen erfolgt in der Teamquery auf Basis einer anzugebenen Oberklasse `+root`, welche die Wurzel einer Klassenhierarchie repräsentiert. Alle ausgewählten Klassen müssen Subklassen der Klasse `+root` sein (definiert mit der Query `isAncestor`, Zz. 3 und 5). Die konkrete Klasse für ein bestimmtes Steuerelement, wie z.B. *Button*, wird dann eindeutig darüber identifiziert, dass sie das jeweilige Interface, z.B. `IButton`, implementiert (Query `implementsInterface`, Zz. 4 und 6). Die Implementierung der einzelnen Fabrikmethoden kann nun generisch erfolgen. In jeder solchen Methode wird eine Instanz der jeweils ausgewählten Klasse erzeugt.

```
1  abstract team class GenericFactory implements IFactory
2    declare(?Class +root, @UniqueMatch ?Class
       -btn, @UniqueMatch ?Class -lbl) {
3    StdQ.isAncestor(+root, -btn) &&
4    StdQ.implementsInterface(-btn, IButton.class) &&
5    StdQ.isAncestor(+root, -lbl) &&
6    StdQ.implementsInterface(-lbl, ILabel.class)
7  }
8  {
9    per(-btn) {
10     public IButton createButton() {
11       try {
12         return (IButton) -btn.newInstance();
13       } catch (Exception e) {...}
14     }
15   }
16   per(-lbl) {
17     public ILabel createLabel() {
18       try {
19         return (ILabel) -lbl.newInstance();
20       } catch (Exception e) {...}
21     }
22   }
23 }
```

Listing 6.8: Variante des Entwurfsmusters *Abstract Factory* in GOT

Das VGT `GenericFactory` kann als Oberklasse für konkrete Fabriken verwendet werden. Ein AGT muss lediglich eine Angabe zur Klasse `+root` machen, um die Implementierung zu vervollständigen. In Listing 6.9 ist beispielhaft das AGT `XFactory` dargestellt, das eine Fabrik für Steuerlemente der `X`-Familie implementiert.

```
1  public team class XFactory extends GenericFactory
2    match(?Class -btn, ?Class -lbl) {super(X.class, -btn, -lbl)}
3  {}
```

Listing 6.9: Eine konkrete Fabrik für Steuerelemente der `X`-Familie

Fazit *Abstract Factory*. Die Mittel zur Metaprogrammierung in GOT ermöglichen die Implementierung abstrakter Fabriken, die zu wesentlichen Teilen wiederverwendbar sind. Dies gilt zumindest in Fällen, in denen die Fabrikmethoden und die Konstruktoren eine standardisierte Form besitzen. Die Anzahl der Variationspunkte kann hier auf einen begrenzt werden, unabhängig davon, wie umfangreich die Familie von Klassen ist. Die Lösung ist aus technischen Gründen zwar auf Teamklassen beschränkt, könnte aber potentiell auch ohne den Team-Mechanismus realisiert werden, da in der dargestellten Variante keine Rollen zum Einsatz kommen (es handelt sich ausnahmsweise um ein „Pseudo-Team"). Eine OT-spezifische Alternative bestünde in einer zentralen Implementierung der Steuerelement-Klassen und deren spezifischer Ausgestaltung über Rollen. Eine solche Variante kann z.B. mit dem Bridge Pattern (vorgestellt in Abschnitt 6.3.3) realisiert werden.

6.2.4 Observer und Mediator

Die GOT-Lösung zum *Observer* Pattern wurde bereits in Abschnitt 4.4.3 mit einem vollständigen Beispiel detailliert vorgestellt. Das Observer Pattern beschreibt eine Anforderung, bei der sowohl Introspektion als auch Quantifizierung einen sinnvollen Beitrag zur Realisierung leisten. Das generische Team, das die Funktionalität umsetzt, ist in vielen unterschiedlichen Szenarien einsetzbar. Die Umsetzung dieses Entwurfsmuster ist daher das beste Beispiel für verbesserte Modularisierung und Wiederverwendbarkeit durch generische Teams.

Das *Mediator* Pattern definiert ein Objekt, das als zentraler Vermittler die Interaktion zwischen einer Menge von Objekten – den *Kollegen* – steuert. Das Muster dient der Entkopplung, da die Kollegen sich untereinander nicht referenzieren müssen und trotzdem über den Vermittler miteinander kommunizieren können. Die Kommunikation zwischen Kollegen und Vermittler kann mit dem Observer Pattern realisiert werden. Der Beitrag von GOT zu diesem Entwurfsmuster wäre entsprechend analog. Für die Besonderheiten des Entwurfsmusters, die über den Observer-Mechanismus hinausgehen, leistet GOT dagegen keinen zusätzlichen Beitrag. Aus unserer Sicht bietet die Einordnung des Mediator Patterns an dieser Stelle daher durchaus Anlass zu Kritik. Da aber sowohl Hannemann und Kiczales als auch Monteiro und Gomes sich in ihren Studien bei der Beurteilung zur Wiederverwendbarkeit ihrer Mediator-Lösung auf die Observer-Funktionalität beschränkt haben, verfahren wir an dieser Stelle genauso.

6.3 Verbesserung durch nicht-generische Teams

In diesem Abschnitt werden die Entwurfsmuster-Lösungen vorgestellt, bei denen die Sprachmittel von OT einen sinnvollen Beitrag zur Modularisierung und ggf. auch zur Wiederverwendbarkeit leisten. Einen sinnvollen Einsatz von GOT können wir im allgemeinen Fall dagegen nicht identifizieren. Eine charakteristische Ursache dafür ist, dass in den Anwendungen durchgängig Bezug auf szenariospezifische Funktionalität und Schnittstellen genommen werden muss, weshalb keine Möglichkeit zur Abstraktion durch Quantifizierung bzw. Introspektion existiert. Der Bezug auf spezifische Elemente beeinträchtigt oft auch die Wiederverwendbarkeit der Lösungen.

6.3.1 Chain of Responsibility

Das Entwurfsmuster *Chain of Responsibility* dient zur Delegation von Anfragen, die nicht direkt an ein konkretes Objekt, sondern indirekt an eines oder mehrere Objekte aus einer Menge gerichtet werden. Ein typisches Beispiel ist die Behandlung von

Eingabeereignissen (Tastatur, Maus etc.), die an eine Fenster-Anwendung gerichtet sind. Welche der grafischen Komponenten ein bestimmtes Ereignis behandelt, wird aufgrund einer komplexen Hierarchie entschieden, die sich durch die Verschachtelung der Komponenten und die Art des Ereignisses ergibt.

Das Muster beschreibt die Musterrolle des *Handlers*, dessen Schnittstelle Anfragen in Form von Methodenaufrufen ermöglicht. Der Handler kann einen anderen Handler als seinen Nachfolger definieren. An den Nachfolger wird eine Anfrage weitergeleitet, wenn der ursprüngliche Handler selbst die Anfrage nicht abschließend behandelt.

Die Implementierung der Delegation zwischen den Handlern in den Handler-Klassen selbst, wie es in Java erforderlich ist, kann aus Sicht der Modularisierung nachteilig sein. Der Code zur Verwaltung von Nachfolgern und zur Delegation der Anfragen trägt nicht zur Kernfunktionalität des Handlers bei. OT bietet hier eine Alternative, indem die Verkettung der Handler und das Weiterleiten der Anfragen kontextspezifisch erfasst werden können. Dies ist besonders nützlich, wenn unterschiedliche Handler-Strukturen abgebildet werden sollen und diese eventuell sogar zeitgleich aktiv sind. In OT kann die Verkettung der Handler unabhängig von der physischen Schachtelung der Komponenten erfolgen.

Eine Anwendung des Chain of Responsibility Patterns für Komponenten des *Abstract Window Toolkits* (AWT) ist in Listing 6.10 gegeben. Das dargestellte Team definiert die Rolle `Handler` für beliebige Komponenten. Eine Komponente kann in Abhängigkeit des Kontexts eine Nachfolger-Beziehung zu einer anderen Komponente definieren (Methode `setSuccessor`, Zz. 4–6), die vom Team zentral verwaltet wird. Die Delegation ist in der Methode `process` realisiert (Zz. 11–15). Wurde das Ereignis bis jetzt noch nicht behandelt (`isConsumed`), dann wird es an den Nachfolger des aktuellen Handlers – sofern einer definiert ist – weitergeleitet. Als Auslöser für die Delegation dient ein Callin. Im Beispiel werden alle Mauseingaben als Auslöser behandelt (Z. 17).

```
1  import java.awt.Component;
2  public team class ComponentChainOfResponsibility {
3
4    public void setSuccessor(Component as Handler handler,
       Component successor) {
5      handler.successor = successor;
6    }
7
8    protected class Handler playedBy Component {
9      protected Component successor;
10
11     private void process(InputEvent e) {
12       if (successor != null && !e.isConsumed()) {
13         successor.dispatchEvent(e);
14       }
15     }
16
17     void process(InputEvent e) <- after void
         processMouseEvent(MouseEvent e);
18   }
19 }
```

Listing 6.10: Umsetzung des Chain of Responsibility Patterns für AWT Komponenten

Ob die Implementierung in diesem Beispiel signifikant von GOT profitieren kann, hängt primär mit der Frage zusammen, ob eine Quantifizierung über die Auslöser sinnvoll ist. Dies kann der Fall sein, wenn mehrere Methoden für die Callin-Bindung in Frage kommen. Ansonsten ist der Mechanismus mit wenigen – allerdings auch spezifischen – Elementen umgesetzt, so dass nur ein geringer invarianter Anteil vorhanden ist, der sich zur allgemeinen Wiederverwendung eignet.

Fazit *Chain of Responsibility.* OT ermöglicht eine gute Modularisierung des Chain of Responsibility Patterns, da die Verkettung der Handler und das Weiterleiten der Anfragen kontextspezifisch erfasst werden kann. Die Lösung ist aber bereits mit den Sprachmitteln von OT sehr knapp und prägnant, so dass der Einsatz von GOT nur in speziellen Fällen einen Mehrwert bieten dürfte.

6.3.2 Composite

Mit dem *Composite* Pattern können Objekte, die eine Teil-Ganzes-Hierarchie repräsentieren, in einer gemeinsamen Baumstruktur mit universeller Schnittstelle erfasst werden. Intern wird zwischen Blättern (Knoten ohne Kinder) und Verbundobjekten (*Composites*, Knoten mit Kindern) unterschieden. Nach außen hin ist der strukturelle Unterschied nicht relevant. Für Klienten ist nur von Bedeutung, dass beide Arten von Knoten die gleiche Funktionalität anbieten, z.B. in Form einer gemeinsamen Schnittstelle.

Zur Erläuterung greifen wir direkt das in [49] bzw. [40] vorgestellte Anwendungsbeispiel zum Composite Pattern auf. Das Szenario beschreibt ein Dateisystem, in dem Dateien in Verzeichnissen abgelegt werden, die wiederum selbst in Verzeichnissen liegen können. Eine typische Anforderung ist, dass für Dateien wie für Verzeichnisse die (kumulierte) Speichergröße berechnet werden kann. Eine Datei wird von der einfachen Klasse

```
public class File {
  public int getSize() {...}
}
```

repräsentiert. Die Speichergröße ist direkt dem Zustand eines Datei-Objekts zu entnehmen. Ein Verzeichnis wiederum kann eine Menge von Dateien und anderen Verzeichnissen enthalten, die Gesamtgröße ergibt sich aus der Summe der Einzelgrößen und der eigenen Größe.

```
1  abstract team class AbstractComposite {
2    protected abstract class Child {}
3    protected List<Child> children = new SomeList<Child>();
4
5    public <B base Child> void addChild(B as Child c) {
6      if (!children.contains(c)) {
7        children.add(c);
8      }
9    };
10   public <B base Child> void removeChild(B as Child c) {
11     children.remove(c);
12     this.unregisterRole(c); // Rollenobjekt löschen
13   };
14 }
```

Listing 6.11: OT-Lösung für Composite

In OT wird ein Composite naheliegender Weise als Teamklasse realisiert, da es einen Verbund von Objekten darstellt. Verschiedene Klassen, auch die Teamklasse selbst, können die Rollen von Kindern des Composites spielen. Die grundlegende Struktur eines Composites kann in OT für viele Fälle allgemein realisiert werden, implementiert mit der abstrakten Teamklasse `AbstractComposite` in Listing 6.11. Zur allgemeingültigen Struktur gehört eine Rolle `Child` zur Repräsentation von Kindknoten, sowie deren Verwaltung in einer Liste `children` inklusive der Methoden `addChild` und `removeChild` zum Hinzufügen bzw. Entfernen.

Die konkrete Umsetzung einer Teamklasse `Directory` zur Repräsentation von Verzeichnissen ist in Listing 6.12 dargestellt. Das Team erweitert die Rolle `Child` um die gewünschte Funktionalität `getSize` (Z. 3). Die Klasse `FileChild` wird als Subrolle der Rolle `Child` definiert (Zz. 6–7). Die Methode `getSize` ist für diese Rolle nicht explizit definiert, sondern wird per inferiertem Callout an die entsprechende Methode der Basis delegiert. Die Teamklasse selbst wird ebenfalls als Basisklasse einer Subrolle von `Child` definiert (Zz. 9–10). Dies ist erlaubt, solange eine Instanz der Teamklasse nicht als Basisobjekt für deren eigene Rolleninstanzen herangezogen wird. Für `Directory` wird die Methode `getSize` spezifisch unter Einbezug der Kindknoten implementiert (Zz. 12–18). Der dazugehörige Callout der Rolle wird wie in der Klasse `FileChild` inferiert.

```
1  public team class Directory extends AbstractComposite {
2    protected abstract class Child {
3      public abstract int getSize();
4    }
5
6    protected class FileChild extends Child playedBy File
7      base when (children.contains(base)) {}
8
9    protected class DirectoryChild extends Child playedBy
       Directory
10     base when (children.contains(base)) {}
11
12   public int getSize() {
13     int size = ...; // Basisgröße eines Verzeichnisses
14     for (Child c: children) {
15       size += c.getSize();
16     }
17     return size;
18   }
19 }
```

Listing 6.12: Eine konkrete Anwendung von `AbstractComposite`

Die dargestellte OT-Lösung weist mehrere Besonderheiten gegenüber einer herkömmlichen Umsetzung des Composite Patterns auf:

1. Die Klassen `File` und `Directory` müssen kein gemeinsames Interface implementieren. Das ist nicht erforderlich, da die Klassen dieselbe Rolle innerhalb des Teams spielen und auf diesem Wege bereits eine gemeinsame Schnittstelle teilen. Dies und die Tatsache, dass diese Schnittstelle vom Kontext gekapselt wird, tragen wesentlich zur Reduzierung der Kopplung bei. Für den universellen Zugriff eines Klienten kann natürlich trotzdem ein gemeinsames Interface ergänzt werden – zur universellen Behandlung von Kindknoten innerhalb der Composite-Klasse ist es aber nicht notwendig.

2. Sollte ein Composite nicht allgemein alle Elemente einer Hierarchie, sondern nur solche eines bestimmten Typs enthalten dürfen, so kann diese Beschränkung normalerweise nicht durch das Typsystem abgesichert werden (aufgrund des gemeinsamen Interfaces). In OT ermöglicht dagegen die dedizierte Zuweisung von `Child`-Rollen an spezifische Klassen eine individuelle Beschränkung, die bereits von der statischen Typprüfung durchgesetzt wird.

Fazit *Composite*. OT bietet bei der Umsetzung des Composite Patterns Möglichkeiten zur Entkopplung, die bei einer herkömmlichen Lösung nicht gegeben sind. Die OT-Lösung enthält dabei sogar einen wiederverwendbaren Anteil. In einigen Fällen mag bei der Zuweisung der Rollen für Kindknoten ein sinnvoller Einsatz für Quantifizierung mit GOT möglich sein. Dies halten wir aber für eine Ausnahme; i.A. dürften die Details der Struktur und der Funktionalität einer generalisierenden Implementierung im Wege stehen.

6.3.3 Modularisierung mit gebundenen Rollen

In diesem Abschnitt werden weitere Entwurfsmuster aufgeführt, die von einer Modularisierung durch OT profitieren können. Wir beschränken die Diskussion auf eine qualitative Erörterung und verzichten auf die Angabe konkreter Implementierungsbeispiele, da sie nur wenig Neues zu den bereits gezeigten Beispielen beisteuern würden.

Adapter

Das Entwurfsmuster *Adapter* (auch als *Wrapper* bezeichnet) dient zur Anpassung der Schnittstelle einer Klasse, so dass diese kompatibel für einen bestimmten Klienten ist. Man unterscheidet zwei Varianten das Musters: entweder die Adapter-Klasse verfeinert die zu adaptierende Klasse (*Class Adapter*) oder sie referenziert deren Instanz (*Object Adapter*) und sorgt für die Delegation der Methodenaufrufe an das Original. Das Pattern eignet sich in besonderer Weise für eine Umsetzung mit OT. Die Adaptierung der Originalklasse erfolgt als Rolle, im einfachsten Fall direkt im Kontext des Klienten. Bei dieser Lösung ist keine Unterscheidung zwischen Object Adapter und Class Adapter notwendig, die Rolle realisiert die Vorteile beider Varianten simultan. Eine Wiederverwendung der Adapter-Klasse ist generell nicht möglich, da diese notwendigerweise für spezifische Schnittstellen definiert ist.

Decorator

Das *Decorator* Pattern wird eingesetzt, um die Funktionalität einzelner Objekte zu spezialisieren oder zu erweitern, ohne dass andere Instanzen derselben Klasse davon betroffen sind. Ziel des Musters ist genau eine solche Flexibilität, wie sie durch Rollen in OT realisiert werden kann. Die Adaptierung durch Rollen kann durch Guard Predicates feingranular auf einzelne Objekte und bestimmte Situationen beschränkt werden. Die statische Struktur der adaptierten Klasse muss dazu nicht angepasst werden. Die Modularisierungsmöglichkeiten von OT erscheinen uns besonders geeignet zur Realisierung dieses Entwurfsmuster. Die Wiederverwendung scheitert hier abermals am Bezug auf spezifische Funktionalität und Schnittstellen.

Bridge

Das Entwurfsmuster *Bridge* hat zum Ziel, eine Abstraktion und deren Implementierung dahingehend zu entkoppeln, dass sie unabhängig voneinander variieren können.

Abstraktion und Implementierung werden dabei jeweils durch eine eigene Schnittstelle repräsentiert. Die Abstraktionsklasse besitzt eine Referenz auf die Implementierungsklasse und nutzt diese zur Realisierung ihrer Funktionalität. Da Abstraktion und Implementierung in verschiedenen Klassen umgesetzt sind, können diese auch individuell durch Vererbung verfeinert werden und damit eigene Klassenhierarchien bilden. Aus N Abstraktionen und M Implementierungen lassen sich somit $N \times M$ unterschiedliche Lösungen realisieren.

Effektiv dient das Bridge Pattern der mehrdimensionalen Zerlegung. Diese Form der Zerlegung wird von OT direkt unterstützt. Die Abstraktion kann als Rolle realisiert werden, deren Basis durch die Implementierung gestellt wird. Beide Klassenhierarchien können unabhängig voneinander verfeinert werden, skizziert in Abbildung 6.2.

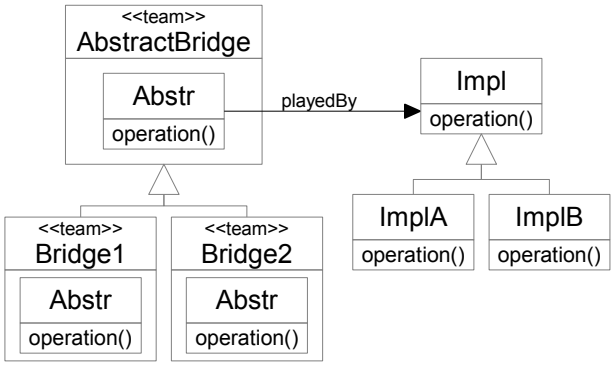

Abbildung 6.2: Das Entwurfsmuster *Bridge* in OT

Eine Modularisierung in OT erlaubt sogar, unterschiedliche Klassenhierarchien von Implementierungen auf der Basisseite auf eine gemeinsame Rollenstruktur abzubilden. Eine gemeinsame Schnittstelle für verschiedene Varianten der Implementierung ist nicht mehr zwingend erforderlich. Die Implementierung ist somit völlig entkoppelt von der Funktionalität der Abstraktion.

Eine Wiederverwendung von Bridge-Implementierungen ist allerdings nicht möglich, da eine Abstraktion eine spezifische Schnittstelle definiert und diese mit der spezifischen Schnittstelle einer Implementierung verknüpfen muss.

Factory Method

Eine *Fabrikmethode* definiert eine Schnittstelle zur Erzeugung eines Objekts. Anstelle eines direkten Konstruktor-Aufrufs wird die Fabrikmethode aufgerufen, welche die Erzeugung des Objekts kapselt. Im Gegensatz zu einem Konstruktor kann die Fabrikmethode von Subklassen verfeinert werden, die Methode repräsentiert einen virtuellen Konstruktor. Die Entscheidung, welche konkrete Klasse instanziiert wird, kann so individuell in der Subklasse getroffen werden. Anstelle der Implementierung einer Subklasse zum Überschreiben der Fabrikmethode kann in OT die Fabrikmethode kontextabhängig über den Callin einer Rolle redefiniert werden, was die Kopplung der Anwendung mit der spezifischen Subklasse vermeidet. Der Wechsel zwischen verschiedenen Fabrikmethoden kann in OT ohne Weiteres dynamisch zur Laufzeit geschehen. Sind die zu erzeugenden Objekte als Rollen realisiert, so ist das Pattern generell obsolet, da Rollen virtuelle Klassen sind und somit bereits der Konstruktor eine kontextabhängige Objekterzeugung bewirkt. Das Muster besitzt keine wiederverwendbaren Anteile, da es im Wesentlichen auf die spezifische Auswahl der zu erzeugenden Klasse beschränkt ist.

6.3.4 Modularisierung mit ungebundenen Rollen

Die Entwurfsmuster *Iterator*, *Interpreter*, *State*, *Strategy* und *Builder* teilen die Gemeinsamkeit, dass sie alle eine eigenständige Musterrolle definieren, die in einem bestimmten Kontext aktiv ist (dargestellt in Tabelle 6.1).

Entwurfsmuster	Musterrolle des Kontexts	Eigenständige Musterrolle
Iterator	*Aggregate*	*Iterator*
Interpreter	*Context*	*Expression*
State	*Context*	*State*
Strategy	*Context*	*Strategy*
Builder	*Director*	*Builder*

Tabelle 6.1: Eigenständige Musterrollen mit Bezug zu einem Kontext

Mit Ausnahme des Iterators wird die eigenständige Musterrolle dabei in Form von Subklassen in unterschiedlichen Ausprägungen spezialisiert. In Java-Lösungen wird die Musterrolle als Klasse bzw. Klassenhierarchie realisiert. Die Klasse, die den Kontext repräsentiert, wird entweder direkt oder situationsgebunden (über Methodenparameter) mit der eigenständigen Musterrolle assoziiert.

Eine Modularisierung in OT unterscheidet sich in diesen Fällen nur geringfügig von einer Modularisierung in Java. Die eigenständige Musterrolle kann als ungebundene Rolle eines Teams, das den Kontext abbildet, realisiert werden. Ein solches Modul weist eine hohe Kohäsion auf, was vorteilhaft für die Kapselung ist. Dieselbe Struktur ist allerdings auch in Java mit inneren Klassen umsetzbar (innere Klassen bilden schließlich die Basis für Rollen in OT). Der eigentliche Mehrwert einer OT-Lösung liegt hier im Family Polymorphism und der Eigenschaft, dass Rollen als virtuelle Klassen überschrieben werden können. Potentiell bietet sich damit eine zusätzliche Option zur Modularisierung, die in Java nicht vorhanden ist. Ob dies tatsächlich bei einer konkreten Anwendung eines Entwurfsmusters zum Tragen kommt, ist vom Einzelfall abhängig. Da es sich letztlich nur um eine Nuance der Modularisierung handelt, welche die grundsätzliche Struktur und Kollaboration der beteiligten Klassen nur wenig beeinflusst, kann der Mehrwert von OT hier sicherlich nur als gering eingestuft werden. Obwohl aus einer Modularisierung durch virtuelle Klassen theoretisch auch eine Verbesserung der Wiederverwendbarkeit abgeleitet werden könnte, so können wir im praktischen Sinn bei keinem der Entwurfsmuster eine wiederverwendbare Abstraktion extrahieren, die wir als sinnvoll und produktiv einstufen würden. Eine allgemein wiederverwendbare Implementierung, die den Kern der Musterfunktionalität vollständig von den spezifischen Details entkoppelt und auf andere Szenarien übertragen werden kann, ist in diesen Fällen nicht realisierbar. Ursache hierfür ist, dass ausschließlich eigenständige Musterrollen zum Einsatz kommen – deren Ausprägung ist natürlicherweise vom konkreten Szenario abhängig. Die Details der Entwurfsmuster werden nachfolgend kurz vorgestellt.

Iterator

Ein *Iterator* wird verwendet, um auf Elemente aus einem Aggregat in einer sequentiellen Ordnung zuzugreifen, ohne die spezifische Repräsentation des Aggregats dafür offenzulegen. Auf diese Weise können auch unterschiedliche Iteratoren für dasselbe Aggregat definiert werden. In Java existiert das Standard-Interface *java.util.Iterator* zur Umsetzung dieses Musters. Eine Iterator-Rolle in OT basiert gleichermaßen wie eine Java-Implementierung auf diesem Interface (oder einer vergleichbaren allgemeinen Schnittstelle). Die Möglichkeit, die Rolle zu überschreiben, ist hier durchaus sinnvoll, denn eine Spezialisierung des Aggregats erfordert ggf. auch eine Spezialisierung des Iterators, was so gemeinsam umgesetzt werden kann.

Interpreter

Das *Interpreter* Pattern definiert die Repräsentation für eine Grammatik einer einfachen Sprache, meist in Form eines Syntaxbaums bestehend aus unterschiedlichen Ausdrücken (Musterrolle *Expression*). Ein Interpreter nutzt diese Repräsentation, um Sätze der Sprache zu interpretieren. Das Pattern definiert im Wesentlichen ein Interface für Ausdrücke, über das die Interpreter-Funktionalität aufgerufen werden kann. Die Daten, die global für den Interpreter gelten, können zentral in einem Team verwaltet werden. Das Team bildet ein natürliches Modul, das den Kontext und die Rollenklassen miteinander kollaborieren lässt und dabei vor der Außenwelt verbirgt. Die Rollen eines Interpreter-Teams können in einfacher Weise in Subklassen verfeinert werden.

State

Mit der Anwendung des *State* Patterns lagert ein Objekt (der *Kontext*) einen Teil seiner Funktionalität in ein anderes Objekt (die eigenständige Musterrolle *State*) aus. Das Kontext-Objekt delegiert Methodenaufrufe an das State-Objekt und kann so sein Verhalten in Abhängigkeit seines Zustands verändern. Der Wechsel des State-Objekts wirkt nach außen hin so, als würde das Kontext-Objekt dynamisch seine Klasse wechseln. In OT können die State-Klassen innerhalb eines Teams realisiert werden. Das Team definiert die relevante Schnittstelle für Klienten und leitet Nachrichten an das aktuelle State-Objekt weiter, ein direkter Zugriff auf das State-Objekt von außerhalb des Teams ist nicht erforderlich.

Strategy

Das *Strategy* Pattern trennt den Kontext eines Algorithmus von dessen konkreter Implementierung. Die Funktionalität einer Familie von Algorithmen wird in Klassen mit einer gemeinsamen Schnittstelle, der *Strategy*, gekapselt. Der Kontext kann nun zwischen unterschiedlichen Ausprägungen der Strategie dynamisch wechseln.

Neben der zentralen Realisierung in einem Team bietet OT durch Implicit Invocation auch die Möglichkeit, das Strategy Pattern in einem Kontext anzuwenden, dessen Implementierung nicht explizit von der Strategie getrennt wurde. Eine alternative Strategie könnte in diesem Fall per Callin anstelle der Original-Implementierung ausgeführt werden. Geht man aber davon aus, dass Kontext und Strategie im Sinne der Entkopplung schon im Design voneinander getrennt werden, so bringt diese Variante keine Vorteile.

Builder

Mit dem *Builder* Pattern wird die Erzeugung komplexer Objekte von ihrer Repräsentation getrennt. Der *Builder* wird als dedizierte Schnittstelle definiert. Ein anderes Objekt, der *Director*, kann einem Builder-Objekt nun eine Sequenz unterschiedlicher Anweisungen senden, um nach dieser Vorgabe ein komplexes Objekt, das *Produkt*, zu erzeugen. Die Builder-Schnittstelle kann unterschiedlich implementiert werden, so dass derselbe Konstruktionsprozess unterschiedliche Repräsentationen des Produkts generiert. In OT kann der Builder als Rolle im Team des Directors realisiert werden, um ihn darin zu kapseln. Darüber hinaus kann OT aber keine Verbesserung zur Umsetzung dieses Entwurfsmusters bieten.

6.4 Keine Verbesserung durch Teams

Bei den folgenden Entwurfsmustern können wir keine Verbesserung der Modularisierung und Wiederverwendbarkeit durch den Einsatz von OT feststellen. Damit ist auch eine Verbesserung durch GOT ausgeschlossen.

Singleton

Mit dem *Singleton* Pattern wird erreicht, dass es nur eine einzige Instanz einer Klasse gibt und diese global verfügbar ist. Für dieses Entwurfsmuster sehen wir keine Verbesserung der Modularisierung durch OT. Der Umfang der Musterfunktionalität ist sehr gering – es gibt nur eine beteiligte Klasse, in der die Implementierung bereits sinnvoll modularisiert ist. Die Funktionalität kann zudem nicht ohne Weiteres von der Klasse separiert werden, da ein wesentlicher Teil der Umsetzung in der Beschränkung der Schnittstelle liegt – konkret im Verbergen des Konstruktors. Die Sprachprinzipien von OT sehen vor, dass ein Objekt erst nach dessen Erzeugung eine Rolle annehmen kann. Die Erzeugung selbst kann also mit dem Rollenmechanismus nicht beeinflusst bzw. beschränkt werden.

Flyweight

Das *Flyweight* Pattern wird eingesetzt, um eine große Anzahl gleichartiger Objekte effizient händeln zu können. Anstatt viele Objekte mit dem gleichen Zustand zu erzeugen, wird lediglich eine einzelne Instanz mit dem entsprechenden Zustand erzeugt und von vielen Klienten geteilt. Die Erzeugung und Verwaltung der Objekte wird dabei zentral über eine Fabrik gesteuert. Der Zustand eines Flyweight-Objekts ist unveränderlich. Ist die Funktionalität des Flyweight-Objekts auch vom Zustand des Kontexts abhängig, so muss dieser vom Klienten beim Methodenaufruf beigesteuert werden.

Die Realisierung des Flyweight Patterns kann in unseren Augen nicht von den OT-Sprachmitteln profitieren. Die Aufteilung des Zustands in intrinsische und extrinsische Anteile hat einen bedeutenden Einfluss auf die gesamte Struktur und Funktionalität der Klasse. Eine derartig tiefgreifende Designentscheidung kann nicht separat durch die Adaptierung einer Rolle realisiert werden. Ein wesentlicher Teil der Funktionalität besteht in der zentralen Verwaltung der Objekterzeugung, die – wie beim Singleton Pattern erläutert – nicht durch OT beeinflusst werden kann. Da ein primäres Ziel des Flyweight Pattern eine Reduzierung des Speicherbedarfs ist, wäre eine Adaptierung durch zusätzliche Rollenobjekte eventuell sogar kontraproduktiv.

Facade

Eine *Fassade* definiert eine universelle Schnittstelle zu einer Menge von heterogenen Schnittstellen eines Subsystems. Die Idee der Fassade ist die der klassischen Abstraktion: Ziel ist das Vereinheitlichen und Vereinfachen des Zugriffs auf ein komplexes Subsystem. Die Fassade definiert selbst keine zusätzliche Funktionalität. Per Definition ist hier keine wiederverwendbare Struktur denkbar (unabhängig von der Technologie), da es lediglich um die Verknüpfung von komplexen Interfaces geht, die notwendigerweise immer spezifisch sein müssen. Ein Team, das allgemein eine Einheit größer als eine Klasse repräsentiert, ist potentiell gut zur Abbildung einer Fassade geeignet. Da die Nachrichtendelegation nur in eine Richtung geht (von der Fassade zum Subsystem, aber nicht umgedreht), ergibt sich allerdings keine Anforderung zur Implicit Invocation. Rollen bieten also an dieser Stelle keinen Mehrwert – die Weiterleitung von Nachrichten zwischen Objekten ohne Rollenadaption erfüllt den Zweck gleichermaßen.

Proxy

Ein *Proxy* dient als Platzhalter für ein anderes Objekt, genannt *Subjekt*. Proxy und Subjekt teilen die gleiche Schnittstelle. Über den Proxy kann der Zugriff auf das Subjekt zentral kontrolliert werden. Zwar kann die Proxy-Funktionalität potentiell durch eine Rolle für das Subjekt realisiert werden, in vielen der üblichen Anwendungsszenarien ist dies aber aus technischen Gründen nicht umsetzbar:

- Ein *Virtual Proxy* kann ein Subjekt repräsentieren, auch wenn es noch nicht erzeugt wurde. Dies ist beispielsweise sinnvoll, wenn die Erzeugung des Subjekts mit hohem Aufwand verbunden ist und verzögert werden soll, bis erstmals ein Zugriff erfolgt. Dies ist mit Rollen in OT nicht umsetzbar, da ein gebundenes Rollenobjekt zwingend die Existenz eine Basisinstanz voraussetzt.

- Ein *Remote Proxy* dient als lokaler Repräsentant eines entfernten Subjekts. Ein klassisches Beispiel für die Anwendung dieses Musters ist *Java Remote Method Invocation (RMI)*. In OT kann eine solche Verbindung bislang nicht über ein Rollenverhältnis ausgedrückt werden.[20]

- Ein *Delegation Proxy* kann als permanente Referenz auf wechselnde Subjekte genutzt werden. Falls eine Anwendung an mehreren Stellen ein konkretes Subjekt referenziert, dieses Subjekt aber zur Laufzeit gegen ein anderes ausgetauscht werden soll, so ist die Aktualisierung der Referenzen aufwändig und fehleranfällig. Stattdessen kann ein permanenter Proxy referenziert werden, der Anfragen an das Subjekt weiterleitet. Der Austausch des Subjekts muss dann lediglich an dieser Stelle vorgenommen werden. Auch dieser Fall ist mit Rollen in OT nicht realisierbar, da ein Rollenobjekt sein Basisobjekt nicht austauschen kann.

Template Method

Eine *Template*-Methode definiert das Skelett eines Algorithmus. Details des Algorithmus in Form einzelner Methoden sind entweder gänzlich abstrakt gelassen oder mit einer Default-Implementierung versehen, so dass die Methoden von Subklassen überschrieben werden müssen bzw. können.

Sofern mindestens eine Default-Implementierung vorliegt, erlaubt OT die Adaptierung ohne Vererbung. Die grundlegende Trennung zwischen allgemeinen und speziellen Anteilen ist hier aber bereits in gleicher Weise über Vererbung realisiert. OT kann also in einigen Fällen eine Alternative, aber grundsätzlich keine Verbesserung der Modularisierung bieten.

Prototype

Ein *Prototyp*-Objekt dient als Vorlage zur Erzeugung neuer Objekte. Anstatt eines expliziten Konstruktor-Aufrufs wird der Prototyp geklont. In Java ist dazu bereits standardmäßig die virtuelle Methode `clone` am Typ `Object` definiert. Soll der Klon-Prozess kontextabhängig gestaltet werden und dazu ein spezialisiertes Interface erfordern, so kann OT hier eine geeignete Modularisierung bieten. In der Regel ist dies aber nicht erforderlich. Die Standard-Funktionalität von Java ist die denkbar prägnanteste und einfachste Umsetzung des Patterns. Der Einsatz von Rollen würde hier lediglich einen Mehraufwand bedeuten und die Struktur verkomplizieren.

[20] Die Möglichkeit des *Remote Role Playings* in OT ist allerdings aktueller Forschungsgegenstand [1].

Command

Mit dem *Command* Pattern werden Anfragen an eine Anwendung in Form von Objekten abgebildet. Die Anfrageobjekte können in einfacher Weise dazu genutzt werden, Steuerelemente zu parametrisieren. Die Ausführung der Anfragen kann protokolliert, verzögert oder rückgängig gemacht werden. Ausgangspunkt ist die Musterrolle des *Invokers* (das auslösende Steuerelement), die den Kontrollfluss an ein verknüpftes *Command*-Objekt (die ausgelöste Anfrage bzw. Aktion) weiterleitet. Das Command-Objekt sorgt dann für die Durchführung der Aktion, wobei ein oder mehrere *Receiver* (Objekte, auf die im Rahmen der Aktion zugegriffen wird) involviert sind. Die Musterrolle Receiver besitzt keine spezifischen Merkmale, weshalb sie normalerweise keinen Einfluss auf die Modularisierung hat.

Potentiell kann mit OT der Invoker vollständig von der Verknüpfung mit einem Command entkoppelt werden. Der auslösende Aufruf kann stattdessen über den Callin einer Rolle an ein Command weitergeleitet werden. Die Struktur für Command-Klassen bleibt davon unberührt. Es ist allerdings fraglich, ob diese Modularisierung mit OT wirklich vorteilhaft ist. Eine Entkopplung zwischen Invoker und Command hinterlässt einen Invoker, der selbst weder eine Aktion für eine Anfrage definiert noch ein Objekt referenziert, an das die Anfrage delegiert wird. Die Zuordnung des Commands an den Invoker geschieht in einer dritten Instanz, dem jeweiligen Team. Die Verknüpfung der Rollenobjekte und die Weiterleitung des Kontrollflusses sind dabei mit signifikantem Mehraufwand verbunden. Prinzipiell ist die Entkopplung einer Delegationsbeziehung über einen mit OT-Mitteln realisierten „Mittelsmann“ immer möglich. Da das Auslösen der Aktion die eigentliche Aufgabe des Invokers ist, kann diese Form der Entkopplung aber auch als Minderung der Lokalität bewertet werden. Das Team verkompliziert die ursprünglich einfache Struktur des Musters durch Hinzufügen eines vermittelnden Teams – letztlich kann dies als Anwendung des Mediator Patterns ausgelegt werden. In Anbetracht des damit verbundenen Aufwands bzw. der komplexen Struktur erscheint diese Art der Modularisierung im Regelfall unangemessen für den Zweck des Musters zu sein.

6.5 Andere Anwendungen

Die bisher gezeigte Evaluierung von GOT beschränkt sich nur auf Design Patterns. Selbstverständlich existieren noch andere Anwendungen außerhalb dieser Domäne. Gerade in komplexeren Szenarien scheint sich der Einsatz von Quantifizierung anzubieten. An dieser Stelle soll nur ein kurzer Ausblick darauf gegeben werden, anhand eines allgemeinen Beispiels zur Rollenadaption von Interfaces. In OT kann anstatt einer konkreten Basisklasse auch ein Interface zur Definition der Basis in einer Rollenbeziehung angegeben werden. Aus technischen Gründen ist bei einer solchen Basisrelation aber kein Callin möglich, was eine erhebliche Einschränkung der Anwendbarkeit bedeutet. Durch GOT kann dieses Problem in einfacher Weise gelöst werden – effektiv handelt es sich um eine reine Anforderung zur Quantifizierung. Beispielsweise soll das Interface

```
interface I {void foo();}
```

durch eine Rolle adaptiert werden. In Listing 6.13 wird die Realisierung durch ein generisches Team `T` skizziert. Das Team definiert eine Rolle `R` für das Interface `I` (Zz. 5–7). Die Rollenmethode `bar` soll per Callin nach der Interfacemethode `foo` ausgeführt werden. In der Rolle selbst ist die Angabe des Callins aufgrund der technischen Restriktion nicht erlaubt. Die Rolle kann allerdings einfach durch eine generische Subrolle `-R` erweitert werden, die eine Implementierung des Callins für alle Klassen vornimmt, die das Interface implementieren (Zz. 9–11). Die Query zur Selektion der konkreten Basisklassen ist trivial (Z. 2).

```
team class T match(?Class ?base) {
  StdQ.implementsInterface(?base, I.class)
}
{
  protected class R playedBy I {
    void bar() {...}
  }
  per (?base) {
    protected class -R extends R playedBy ?base {
      bar <- after foo;
    }
  }
}
```

Listing 6.13: Rollen für Interfaces in GOT

6.6 Diskussion vergleichbarer Studien

In diesem Abschnitt werden vergleichbare Untersuchungen zu Entwurfsmuster-Implementierungen in AspectJ und OT diskutiert. Ein besonderer Fokus liegt dabei auf der Beurteilung zur Wiederverwendbarkeit.

6.6.1 *Hannemann und Kiczales 2002*

In der Studie von Hannemann und Kiczales [49] werden Realisierungen der 23 klassischen Entwurfsmuster in Java und AspectJ qualitativ miteinander verglichen. Die Autoren erläutern, dass bei 17 der 23 Muster die AspectJ-Implementierung eine Verbesserung der Modularisierung bewirkt. Diese basiert primär auf der Lokalisierung der Muster-Funktionalität in einem eigenen Modul, getrennt von den beteiligten Klassen. Für zwölf dieser Muster sehen die Autoren auch eine Verbesserung der Wiederverwendbarkeit, da die Implementierung ein abstraktes Modul für eine oder mehrere generalisierte Musterrollen aufweist, das vermeintlich zur allgemeinen Wiederverwendung taugt.

Garcia et al. greifen in [40] die Ergebnisse von Hannemann und Kiczales auf und analysieren sie in einer quantitativen Studie. Auf Basis verschiedener Metriken kommen die Autoren zu dem Schluss, dass lediglich vier der Muster-Realisierungen mit AspectJ tatsächlich eine signifikante Verbesserung zur Wiederverwendbarkeit zeigen – dies sind die Muster Mediator, Observer, Composite und Visitor. Sie argumentieren, dass allein die Präsenz eines abstrakten Aspekts, der zur Vererbung verwendet werden kann, nicht notwendigerweise zur Wiederverwendbarkeit beiträgt. Dies ist erst durch eine Reduzierung des Programmieraufwands und des Codeumfangs gegeben. Bei acht der zwölf als wiederverwendbar eingestuften Implementierungen konnten die Autoren keine solche Reduzierung feststellen.

Wir teilen die Auffassung von Garcia et al. Wie bereits einleitend erwähnt bewerten wir eine Modularisierung erst dann als wiederverwendbar, wenn sich aus der Wiederverwendung ein Mehrwert für den Entwickler gibt – Wiederverwendung ist kein Selbstzweck. Die ausschlaggebenden Kriterien für die Beurteilung der OT- bzw. GOT-Implementierungen wurden am Anfang dieses Kapitels erläutert. Dass sich auch in anderen Fällen eine Abstraktion mit (evtl. auch generischen) Rollen realisieren lässt, ist eine Selbstverständlichkeit. Darin allein kann aber noch kein Mehrwert gesehen werden. Im Gegenteil, ein Produktivitätsgewinn durch den Einsatz zusätzlicher Sprachmittel ist nicht zu erwarten, wenn eine Anforderung auch mit einfacheren Mitteln in vergleichbarer Weise umgesetzt werden kann.

6.6.2 *Monteiro und Gomes 2010 und 2012*

Die Studie von Monteiro und Gomes [43, 87] untersucht die Modularisierung und Wiederverwendbarkeit von OT. Die Autoren nutzen zwei unterschiedliche Kataloge von Entwurfsmuster-Implementierungen als Grundlage: zum einen die AspectJ-Beispiele von Hannemann und Kiczales (vorgestellt in [49]), zum anderen Java-Beispiele von James Cooper (vorgestellt in [22]). Zu jedem der 23 klassischen Entwurfsmuster besitzen die Autoren somit jeweils zwei unabhängige Szenarien (zur Unterscheidung im folgenden Text werden die Szenarien in Kurzform referenziert mit *HK* für Hannemann und Kiczales und *Cooper* für James Cooper). Zu jedem Szenario erstellen die Autoren eine Lösung in OT und vergleichen diese mit der Original-Lösung. In vier Fällen (*Composite*, *Prototype*, *Memento* und *Visitor*), erstellen sie mindestens ein weiteres Beispiel zum HK-Szenario, um alternative Implementierungsmöglichkeiten in OT aufzuzeigen. Auf Basis ihrer Beobachtungen beurteilen sie die Fähigkeiten von OT zur Modularisierung und Wiederverwendung. Die Studie wird auch als Grundlage für eine quantitative Untersuchung der Modularisierung mit OT in [78] verwendet.

Ursprünglich wollten wir diese Studie als unabhängige Grundlage für einen Vergleich von OT mit GOT heranziehen. Nach genauerer Untersuchung mussten wir allerdings von diesem Vorhaben abrücken, da die vorgestellten OT-Lösungen aus unserer Sicht entscheidende Defizite aufweisen.

Modularisierung

Monteiro und Gomes behaupten, dass OT für alle 23 Entwurfsmuster eine erfolgreiche Modularisierung ermöglicht. Als Hauptursache nennen sie die Möglichkeit den Code aller Beteiligten in einem gemeinsamen Team zu lokalisieren. Die Entwurfsmuster Singleton und Template Method sind dabei die einzigen, die nicht von den OT-Sprachmitteln profitieren, weil sie bereits in Java vollständig modularisiert sind.

Die Beispiele der Autoren zu den verbleibenden Entwurfsmustern lassen nach unserer Auslegung nur in 15 der 21 Fälle auf eine Verbesserung der Modularisierung durch OT schließen:

- **Abstract Factory, Factory Method**. Diese beiden Entwurfsmuster nehmen eine Sonderstellung ein, da sie teilweise direkt durch die OT-Sprachmittel realisiert werden. Das Entwurfsmuster Factory Method kann durch polymorphe Konstruktoren realisiert werden, das Entwurfsmuster Abstract Factory durch virtuelle Klassen und Family Polymorphism. Dabei gilt die Einschränkung, dass diese Mechanismen nur für Teams und Rollen, nicht aber für normale Java-Klassen verfügbar sind. Wir stimmen mit den Autoren überein, dass diese Mittel als zuträglich für die Modularisierung der Entwurfsmuster bewertet werden.

- **Adapter, Bridge, Chain of Responsibility, Composite, Decorator, Memento, Visitor**. Auch bei diesen Entwurfsmustern stimmen wir mit der Beurteilung der Modularisierung weitgehend mit den Autoren überein (vgl. Abschnitte 6.2 und 6.3). Rollen bieten hier eine sinnvolle Alternative gegenüber Vererbung und Aggregation.

- **Iterator, Interpreter, State, Builder**. OT bietet in diesen Fällen nur geringfügig andere Möglichkeiten zur Umsetzung gegenüber Java, denn gebundene Rollen – das wesentliche Unterscheidungsmerkmal von OT – kommen hier nicht zum Einsatz. In den realisierten Lösungen werden im Wesentlichen Klassen mit Bezug zum Kontext als ungebundene Rollen eines Teams abgebildet. Ob diese Art der Modularisierung zu einer signifikanten Verbesserung

führt, muss im Einzelfall beurteilt werden. Grundsätzlich ist die Struktur aber für eine Modularisierung geeignet (vgl. Abschnitt 6.3.4).

- **Observer, Mediator**. Bei diesen beiden Entwurfsmustern sehen wir ebenfalls grundsätzlich Möglichkeiten zur Verbesserung der Modularisierung mit OT. Die umgesetzten Lösungen der Autoren können hier aber nur bedingt als geeignete Beispiele herangezogen werden. In der folgenden Diskussion zur Wiederverwendbarkeit werden wir detailliert auf diese Beispiele eingehen.

Bei den übrigen sechs Entwurfsmustern stimmen wir nicht mit der Auffassung der Autoren zur Modularisierung überein:

- **Strategy, Flyweight, Prototype**. Obwohl das Strategy Pattern strukturell ähnlich zu den Entwurfsmustern Iterator, Interpreter, State und Builder ist (vgl. Abschnitt 6.3.4), weicht die Implementierung der Autoren hier von der allgemeinen Struktur ab, da sie auch den Kontext in Gestalt einer Rolle definieren. Beim Flyweight Pattern ist die Rolle `Flyweight` eine Klasse ohne jeglichen Zustand und Methoden. Beim Prototype Pattern existiert ebenfalls keine spezifische Rollenfunktionalität. Aus den Kommentaren der Autoren geht hervor, dass sie selbst in ihren Lösungen wesentliche Kritikpunkte sehen. Dennoch interpretieren die Autoren ihre Lösungen als gut modularisiert und sogar als wiederverwendbar. Wir können diese Auffassung nach unseren Maßstäben nicht teilen. In der folgenden Diskussion zur Wiederverwendbarkeit werden wir detailliert auf die Ansätze der Autoren zu diesen Entwurfsmustern eingehen.

- **Command.** Die Autoren definieren eine abstrakte Teamklasse mit Rollenklassen für die beteiligten Musterrollen *Invoker*, *Receiver* und *Command*. Die wesentliche Funktionalität des Teams besteht lediglich in der Bereitstellung assoziativer Datenstrukturen zur Zuordnung von Objekten der Rolle `Command` zu Objekten der Rollen `Invoker` und `Receiver`. Der tatsächliche Nutzen ist uns allerdings nicht ersichtlich. Die Rolle `Receiver` wird in keinem der Beispiele, das die Autoren präsentieren, verwendet. *Command* wird von den Autoren in der Implementierung als ergänzende Musterrolle ausgelegt, obwohl es sich eigentlich um eine eigenständige Musterrolle handelt. Die Eigenständigkeit ist offensichtlich, denn für die konkrete Realisierung von Aktionen ist nach wie vor die allgemeine Klasse `AbstractCommand` erforderlich, an welche die Rolle `Command` gebunden wird. Ein primärer Zweck dieser Klasse, unabhängig von der Musterrolle, ist nicht vorhanden. `Command` fungiert lediglich als Pseudo-Rolle für diese Basisklasse. Der grundlegende Mechanismus zum Auslösen und Durchführen von Aktionen profitiert von der Rolle nicht. Wie bereits in Abschnitt 6.4 ausgeführt, sehen wir darin insgesamt eine Verkomplizierung der Struktur. Die Lösung wird dem einfachen Aufbau des Command Patterns nicht gerecht und stellt in unseren Augen keine geeignete Modularisierung dar.

- **Proxy**. Die Autoren präsentieren als Beispiel die Anwendung eines *Protection Proxys*. Dies ist einer der wenigen Anwendungsfälle eines Proxys, der durch eine Rolle realisiert werden kann. Wie in Abschnitt 6.4 erläutert, sehen wir allerdings für die überwiegende Mehrheit der Anwendungsfälle, die mit dem Muster assoziiert werden, keine geeignete Möglichkeit zur Umsetzung mit OT.

- **Facade**. Die realisierten Beispiele der Autoren zu diesem Muster enthalten aus unserer Sicht lediglich Pseudo-Rollen und Pseudo-Teams. Der Einsatz von OT dient hier dem reinen Selbstzweck. Einen Mehrwert gegenüber einer Java-Lösung können wir nicht erkennen. Im Gegenteil, der zusätzliche Code der Pseudo-Rollen zur Kopie von Schnittstellen hat einen negativen Einfluss auf die Verständlichkeit und Wartbarkeit des Programms.

Wiederverwendung

Monteiro und Gomes identifizieren insgesamt zehn Entwurfsmuster, für die sie wiederverwendbare Modularisierungen mit OT erstellen konnten. Die wiederverwendbaren Anteile werden dabei stets in Form einer Teamklasse realisiert, die ein Protokoll für die Funktionalität des Musters definiert. Die Autoren selbst verlangen den Einsatz dieses Teams in beiden Szenarien (HK und Cooper) als notwendiges Kriterium für eine Verbesserung der Wiederverwendbarkeit („'yes' for reusability whenever the module implementing the pattern is used in both scenarios" [87]). Die Analyse und Beurteilung der Wiederverwendbarkeit von OT-Lösungen, die die Autoren präsentieren, können wir dabei in etlichen Punkten nicht teilen. Wir haben die Beispiele genauer untersucht und werden nachfolgend im Detail darauf eingehen:

- **Observer.** Die Autoren definieren ein allgemeines Observer-Protokoll in Gestalt einer Teamklasse, in der Subjekt und Beobachter als Rollen realisiert sind. Die Verwaltung der Beziehung zwischen Subjekt und Beobachter ist mit einer Liste realisiert, deren Elemente durch explizite An- und Abmeldung bestimmt werden. Für beide Szenarien (HK und Cooper) muss eine Subklasse des Teams gebildet werden, in der die Rollenfunktionalität spezifisch ausgestaltet wird. Beide Lösungen entsprechen daher einer klassischen Java-Lösung, mit dem einzigen Unterschied, dass die Klassen für Subjekt und Beobachter nun als Rollen anstatt als Oberklassen integriert werden. Obwohl diese Herangehensweise bereits einen Fortschritt in Hinblick auf die Modularisierung darstellt, so wird sie in unseren Augen der Sprache OT nicht gerecht und ist auch nicht für die Wiederverwendung geeignet. Beim direkten Übertrag der Java-Lösung nach OT werden alternative Strukturen ignoriert. Aus unserer Sicht ist die Beibehaltung der klassischen Struktur des Observer Patterns schlichtweg überflüssig. Wie bereits am Beispiel in Abschnitt 4.4 demonstriert, kann der Beobachter selbst den Kontext für Subjekte repräsentieren. Die Sprachmittel von OT erlauben die Verwaltung der Subjekte als Rollen, was die manuelle Verwaltung der Beziehung in Form einer Liste überflüssig macht. Die Zuordnung der Subjekt-Rolle kann nicht nur explizit durch Declared Lifting, sondern auch implizit durch Guard Predicates erfolgen. Die wiederverwendbaren Teile des vorgestellten Observer-Protokolls sind daher in unseren Augen in wesentlichen Teilen obsolet für eine OT-Lösung. Die integrierten Sprachmechanismen decken die Observer-Funktionalität so weit ab, dass eine Implementierung in OT weitgehend auf spezifische Anteile reduziert werden kann und nicht signifikant von der Wiederverwendung profitiert.

- **Mediator.** Für das Mediator Pattern liefern die Autoren ausnahmsweise eine Version, in welcher die Musterrolle *Mediator* nicht als Rolle sondern als Team realisiert ist. Die einzige Rolle ist hier entsprechend der *Colleague*. Das Team `MediatorProtocol` besitzt eine Liste zur Verwaltung von Colleague-Objekten. Die allgemeinen Anteile der Colleague-Rolle beschränken sich auf eine Methode zur Realisierung eines Update-Mechanismus, wie er im Observer Pattern vorkommt. Der Einsatz des Observer Patterns im Rahmen des Mediator Patterns ist durchaus üblich. In der Umsetzung der Autoren ist die Funktionalität der vorgestellten Musterrollen des Mediator Patterns aber vollständig auf die des Observer Patterns reduziert: *Mediator* ist analog zu *Observer* und *Colleague* analog zu *Subject* aufgebaut. Dies ist insofern bemerkenswert, als die Autoren für das eigentliche Observer Pattern eine andere Umsetzung gewählt haben.

  Der Ansatz lässt darüber hinaus keinen Bezug zur spezifischen Bedeutung des Mediator Patterns erkennen. Die unterschiedliche Ausprägung der konkreten

Colleagues erfordert üblicherweise eine individuelle Referenzierung und Behandlung der Teilnehmer, die über eine pauschale Rollenzuordnung nicht realisierbar ist. Die charakteristische Mediator-Funktionalität fehlt nach unserer Auffassung in dieser Umsetzung. Diese grundsätzliche Kritik an der Reduzierung des Mediators auf einen Observer wurde bereits in 6.2.4 thematisiert. Dies zeigt sich auch konkret in den Beispielen der Autoren.

Im Beispiel zum HK-Szenario werden die Beteiligten als allgemeine Colleague-Objekte am Mediator registriert, ihre spezifische Bedeutung ist somit im Mediator nicht mehr bekannt. In der Update-Methode muss der Mediator daher in einer Fallunterscheidung die Referenzen der Colleague-Objekte mit externen Referenzen des Klienten abgleichen, um darüber deren Identität festzustellen und die damit verbundene Funktionalität auszulösen. Der Sinn der allgemeinen Rolle `Colleague` und des zentralen Update-Mechanismus ist somit äußerst fraglich, da sie ohne Rückgriff auf die spezifischen Informationen des Klienten keine Funktion erfüllen können.

Das Cooper-Szenario ist etwas komplexer als das HK-Szenario. Im Gegensatz zum HK-Beispiel referenziert der Mediator hier die einzelnen Colleague-Objekte individuell – die Autoren sehen hier also offenbar auch die Notwendigkeit dafür als gegeben an. Dies wiederum führt dazu, dass die Oberklasse `MediatorProtocol` keinen Beitrag mehr zur Lösung leistet. Wir konnten die Vererbungsbeziehung entfernen, ohne dass dies einen Effekt auf die Funktion des spezifischen Mediators hatte!

Im technischen Sinne verwenden die Autoren die Klasse `MediatorProtocol` also in beiden Szenarien, da diese aber im Cooper-Beispiel praktisch keinen Effekt hat, sehen wir das Kriterium der Autoren zur Wiederverwendbarkeit nicht erfüllt.

- **Strategy.** Die Autoren definieren eine abstrakte Teamklasse als Protokoll, die eine Rolle für den Kontext und eine Rolle für die Strategie definiert. Der einzige (und äußerst geringe) Mehrwert dieser Lösung besteht darin, dass der Kontext ein Attribut mit einer Setter-Methode für die Strategie besitzt und an Nachkommen vererbt. Dieser Mehrwert kommt wiederum nur zum Tragen, wenn die vorgegebene Struktur in dieser Form auf die konkrete Anwendung übertragen werden kann. Dies ist nur im Beispiel der Autoren zum HK-Szenario der Fall. Darin wird die Basisklasse für die Kontext-Rolle allerdings mit einer leeren Methode für die Hauptfunktion (Sortieren) implementiert, da diese Funktion nun über die Strategie-Rolle (Sortieralgorithmen) realisiert wird. Bei dieser Basisklasse kann daher nicht von einer geeigneten Modularisierung gesprochen werden, da sie ohne Rolle unvollständig ist. Die Implementierung einer leeren Methode `sort` widerspricht in unseren Augen der grundsätzlichen Idee des Rollenspiels. Die Basisklasse suggeriert eine vollständige Implementierung, da sie nicht als *abstract* gekennzeichnet ist, obwohl sie die erwartete Nachbedingung der Methode nicht erfüllt. Die eigentliche Semantik wird erst über eine Rolle beigesteuert – die Rolle ist damit keine kontextabhängige Ergänzung mehr, sondern muss permanent vorhanden sein, da ein Ablegen der Rolle zwangsläufig zu Fehlverhalten führen wird! Weder der Klient noch die Klasse selbst kann ersehen, ob überhaupt eine Strategie ausgewählt ist und ausgeführt wird. In diesem Fall widerspricht die Implementierung dem Prinzip der Lokalität, der Effekt auf die Modularisierung ist in unseren Augen negativ zu bewerten.

  Im Cooper-Beispiel ist die geerbte Struktur der Protokoll-Klasse von vornherein zu simpel und muss verfeinert werden. Damit hat die Oberklasse überhaupt keinen Effekt (wir konnten auch in diesem Fall die Vererbungsbeziehung aus

der Implementierung entfernen, ohne dass dies eine beobachtbare Auswirkung hatte).

Die Autoren selbst kommentieren zum Strategy Pattern, dass OT gegenüber klassischer OOP kaum zur Verbesserung von Lösungen beiträgt. Warum die Autoren auf dieser Basis das Strategy Pattern in OT zu den wiederverwendbaren Modularisierungen zählen, bleibt uns völlig unklar. Wir sehen auch für dieses Entwurfsmuster keine geeignete Abbildung der Beteiligten auf OT-Elemente. Bereits der Name der beteiligten Klasse `Context` (!) legt nahe, dass diese möglicherweise besser als Team denn als Rolle realisiert werden sollte.

- **Flyweight.** Die Autoren definieren eine abstrakte Teamklasse mit einer Rolle `Flyweight`. Die Rollenklasse besitzt sowohl in der Oberklasse als auch in den erbenden Subklassen der Beispiele weder Zustand noch Methoden. Sie ist als reine Wrapper-Klasse angelegt, die von den Subklassen des Teams an konkrete Basisklassen gebunden werden muss. Die Teamklasse selbst fungiert als Factory zur Instanziierung der Basis(!)objekte. Sie besitzt eine abstrakte Methode `createFlyweight`, die von den Subklassen überschrieben werden muss, so dass dort die Konstruktoren der Basisklassen aufgerufen werden. Die eigentlich wiederverwendbare Funktionalität der Teamklasse besteht in der Verwaltung der Flyweight-Objekte durch eine assoziative Datenstruktur, die über eine Methode der Signatur `Object getFlyweight(Object key)` verfügbar gemacht werden. Aus unserer Sicht bringt der Einsatz von OT für diese Lösung keinerlei Vorteile gegenüber einer reinen Java-Implementierung. Wir können keinen Mehrwert durch die Modularisierung in Rollen und Teams erkennen. Im Gegenteil: die Rollenklasse `Flyweight` dient keinem Zweck. Dagegen verkompliziert sie die Struktur und bedeutet überflüssigen Aufwand, da sie in jeder Teamklasse überschrieben werden muss. Trotz der Rolle `Flyweight` ist auch die Teamklasse an die konkreten Basisklassen gekoppelt, da ein direkter Aufruf der Konstruktoren und Casting erforderlich ist. Die Autoren liefern keine Begründung, warum sie für die Verwaltung in einer assoziativen Datenstruktur Lifting und Lowering eines Rollenobjekts einfordern, obwohl alternativ die direkte Verwaltung des Basisobjektes möglich wäre. Tatsächlich konnten wir ohne Weiteres die Rolle entfernen und die Teamklasse zu einer normalen Java-Klasse umbauen. Die Struktur wurde so bei gleichbleibender Funktionalität deutlich vereinfacht. Der Einsatz von OT erscheint uns in diesem Beispiel kontraproduktiv, da die vorgestellte Funktionalität mit Standardmitteln von Java in einfacherer Weise realisiert werden kann.

- **Prototype.** Zu diesem Muster definieren die Autoren eine Teamklasse mit einer Rolle für Prototypen. Die Rolle besitzt je eine Methode zur Erstellung einer flachen und einer tiefen Kopie. Die flache Kopie wird dabei als Default auf die Standard-Methode `clone` von `java.lang.Object` abgebildet. Der Ansatz erfordert, dass die konkrete Zuordnung der Rollen zu den Basisklassen und die Implementierung der clone-Funktionalität in den Subklassen des Teams erfolgt. Die clone-Funktionalität ist dabei spezifisch für jede Basisklasse auszugestalten. Eine allgemein rollen-spezifische Funktionalität ist nicht gegeben.

  Im HK-Szenario sehen die Autoren einen Vorteil darin, dass die clone-Funktionalität auf bestimmte Prototyp-Objekte beschränkt werden kann. Diese individuelle Beschränkungsmöglichkeit geht aber mit einer deutlich verkomplizierten Struktur einher, u.a. müssen die Objekte explizit als Rollen registriert werden. Darüber hinaus wird die Lokalität beeinträchtigt, da die clone-Funktionalität nicht in der dafür vorgesehen Standard-Methode, sondern extern in einer Rolle implementiert wird. Der konkrete Mehrwert dieser OT-Lösung

erschließt sich uns nicht, insgesamt könnte die Beschränkungsmöglichkeit in ähnlicher Weise auch mit Java-Mitteln umgesetzt werden.

Im Cooper-Beispiel sehen die Autoren gerade einen Vorteil darin, dass die clone-Funktionalität vom konkreten Prototyp-Objekt getrennt ist. Dies können wir nicht nachvollziehen, da die Funktionalität keineswegs kontextspezifisch ist. Unabhängig davon bleibt auch hier unklar, welchen Mehrwert Team und Rolle gegenüber einen reinen Java-Lösung beisteuern.

Wiederverwendungspotential bietet die präsentierte Lösung nicht. Im Cooper-Beispiel konnten wir abermals die Vererbungsbeziehung zur wiederverwendeten Teamklasse ohne Effekt entfernen.

- **Command.** Wie bereits bei der Beurteilung der Modularisierung ausgeführt, halten wir die vorgestellte Umsetzung des Command Patterns der Autoren für ungeeignet und damit auch nicht für wiederverwendbar.

- **Composite.** Die Autoren definieren eine abstrakte Teamklasse `CompositeProtocol`, die eine Rolle *Component* und davon erbende Rollen für die Beteiligten *Composite* und *Leaf* enthält. Der Mehrwert der Struktur besteht in der Verwaltung einer Liste von Kindknoten für jedes *Component*-Objekt (nicht *Composite*! - womit dies unverständlicherweise auch für Blatt-Knoten vorgesehen ist), inklusive des Hinzufügens und Entfernens von Knoten. Die Autoren liefern jedoch kein überzeugendes Beispiel, wo diese Struktur sinnvoll verwendet werden kann. Stattdessen kommen die Autoren selbst zu dem Schluss, dass diese Struktur ungeeignet ist und dass *Composite* besser als Team denn als Rolle realisiert werden kann. In den überarbeiteten Beispielen verzichten die Autoren daher auch auf Vererbung des Teams `CompositeProtocol`. Diese Erkenntnis der Autoren steht für uns in klarem Widerspruch zu der Tatsache, dass sie ihre Lösung zu den wiederverwendbaren Modularisierungen zählen. Eine aus unserer Sicht geeignete Form einer wiederverwendbaren Implementierung des Composite Patterns in OT wurde in Abschnitt 6.3.2 gezeigt.

- **Memento.** Die Autoren definieren eine Teamklasse mit Rollen für *Memento*, *Originator* und *Caretaker*. Alle Klassen und fast alle Methoden sind abstrakt, so dass sich die Wiederverwendung im Wesentlichen auf die Struktur beschränkt. Diese wiederum halten wir nur für bedingt geeignet, da sie teilweise bereits durch sehr spezifische Details (z.B. eine Methode `clearUndoList`) in ihrer Anwendbarkeit beschränkt wird, die Kernfunktionalität davon jedoch wenig profitiert. Die eigentliche Memento-Funktionalität ist in den Beispielen der Autoren fast ausschließlich spezifisch ausgestaltet. Unsere Interpretation des Memento Patterns, insbesondere die Zuordnung der Beteiligten zu Team-, Rollen- und Basisklassen, haben wir bereits in 6.2.2 ausführlich dargelegt.

- **Visitor.** Zur Realisierung dieses Musters definieren die Autoren eine Teamklasse als wiederverwendbares Visitor-Protokoll. Das Team deklariert je eine abstrakte Rolle für den Besucher und die besuchbaren Knoten. Die einzig konkrete Funktionalität ist mit der Realisierung der Methode `accept` gegeben. Demgegenüber stehen mehrere abstrakte Rollen und Methoden, die zwingend zu verfeinern sind. Die Autoren selbst kommentieren, dass die `accept`-Funktionalität und die abstrakte Struktur die einzigen Elemente des Musters sind, die sich zur Wiederverwendung herauslösen lassen, ohne dass dies einen wesentlichen Mehrwert bietet. Als Alternative zeigen sie spezifische Implementierungen für konkrete Anwendungsfälle ohne die Wiederverwendung der Visitor-Protokoll-Klasse und stufen diese Lösungen insgesamt als deutlich besser ein, speziell in Bezug auf die Modularisierung. Die Schlussfolgerung, aufgrund derer sie die Pattern-Implementierung zu den wiederverwendbaren

Modularisierungen zählen, bleibt uns daher unklar. Wir können in der vorgeschlagenen Protokoll-Klasse kein substantielles Potential für eine allgemeine Wiederverwendung entdecken. Bei den Vorteilen der Modularisierung des Visitor Patterns durch OT, die wir bereits detailliert in Abschnitt 6.2.1 diskutiert haben, stimmen wir mit den Autoren überein.

- **Chain of Responsibility.** Die Autoren definieren eine abstrakte Teamklasse mit Rollen für *Handler* und *Request*. Die Funktionalität der Teamklasse besteht im Weiterleiten der Request-Objekte an die Handler-Objekte und im Verketten derselben. Aus unserer Sicht ist die Rolle *Request* nicht unbedingt notwendig. Ein Request ist oftmals nicht als Objekt sondern als Methodenaufruf realisiert. Der Umgang mit den Request-Objekten gestaltet sich entsprechend umständlich, da Casting erforderlich ist, um sinnvolle Informationen über die Details des Requests zu erhalten. Nichtsdestotrotz ist die Lösung zu *Chain of Responsibility* das einzige Beispiel der Autoren, in dem wir ein gewisses Wiederverwendungspotential erkennen können. Einen detaillierten Lösungsansatz mit OT für dieses Design Pattern haben wir bereits in 6.3.1 vorgestellt.

Die Wiederverwendbarkeit einer Modularisierung ist ein Maß für den Mehrwert, den der Entwickler bei der Wiederverwendung genießt. In den von Monteiro und Gomes vorgestellten Beispielen können wir in neun von zehn Fällen kein signifikantes Wiederverwendungspotential erkennen – einzige Ausnahme ist die Realisierung des Musters *Chain of Responsibility*. Die Aussagen der Autoren zur Wiederverwendbarkeit können wir daher nicht nachvollziehen und kommen zu gegenteiligen Erkenntnissen. Bei den Beispielen zu den Mustern *Command*, *Flyweight*, *Strategy* und *Prototype* sehen wir nicht einmal den Einsatz von OT als gerechtfertigt. Ein grundlegendes Problem besteht darin, dass die wiederverwendbaren Anteile eines Musters von den Autoren meist als Protokoll in Form einer Teamklasse implementiert werden. Alle Beteiligten sind dabei (bis auf wenige Ausnahmen) als Rolle repräsentiert. Die Autoren ignorieren somit weitgehend die Möglichkeiten der eigentlichen Teamklasse zur Repräsentation des spezifischen Kontexts. Die Lösungen sind als bloße Ergänzung zu reinen Java-Implementierungen vorgesehen. Der universelle Einsatz von OT zur Gestaltung der Klassenstruktur scheint nicht in Betracht gezogen zu werden.

6.7 Fazit

Die Fähigkeiten von GOT zur Modularisierung und Wiederverwendung wurden anhand der 23 Entwurfsmuster des klassischen Design Pattern Katalogs [39] untersucht. Eine Übersicht der Ergebnisse ist in Tabelle 6.2 dargestellt.

Ursprünglich wollten wir die Studie von Monteiro und Gomes als unabhängige Grundlage für einen Vergleich von OT und GOT heranzuziehen. Monteiro und Gomes sehen durch den Einsatz von OT gegenüber Java eine Verbesserung der Modularisierung bei 21 von 23 Entwurfsmustern gegeben. Bei zehn dieser Entwurfsmuster soll damit auch eine Verbesserung der Wiederverwendbarkeit einhergehen. Wir können nach Analyse der präsentierten Beispiele die Aussagen zur Modularisierung in sechs von 21 Fällen und die Aussagen zur Wiederverwendbarkeit in neun von zehn Fällen nicht nachvollziehen (vgl. Tabelle 6.2). Aufgrund der diskutierten Defizite haben wir davon Abstand genommen, diese Studie als Grundlage zu verwenden, und unsere Evaluation stattdessen auf eigene Beispiele gestützt. Die Studie bestätigt allerdings unsere in Kapitel 3 diskutierte Auffassung, dass die konventionellen Methoden zur Wiederverwendung deutlich in ihrer Anwendbarkeit begrenzt sind.

Entwurfsmuster	Monteiro und Gomes		Unsere Studie	
	Modularisierung	Wiederverwendbarkeit	Modularisierung	Wiederverwendbarkeit
Abstract Factory	*in OT direkt unterstützt*		++	++
Adapter	+	−	+	−
Bridge	+	−	+	−
Builder	+	−	+	−
Chain of Responsibility	+	+	+	+
Command	+*	+*	−	−
Composite	+	+*	+	+
Decorator	+	−	+	−
Facade	+*	−	−	−
Factory Method	*in OT direkt unterstützt*		+	−
Flyweight	+*	+*	−	−
Interpreter	+	−	+	−
Iterator	+	−	+	−
Mediator	+	+*	++	++
Memento	+	+*	++	++
Observer	+	+*	++	++
Prototype	+*	+*	−	−
Proxy	+*	−	−	−
Singleton	−	−	−	−
State	+	−	+	−
Strategy	+*	+*	+	−
Template Method	−	−	−	−
Visitor	+	+*	++	++

Erläuterung:
− Keine Verbesserung durch OT ggü. Java
\+ Verbesserung durch OT ggü. Java
+* Verbesserung durch OT ggü. Java – wird von uns in Frage gestellt
++ Verbesserung durch GOT ggü. OT

Tabelle 6.2: Gesamtübersicht der Ergebnisse

Nach unserer Analyse konnten wir bei fünf Entwurfsmustern eine Verbesserung der Modularisierung und Wiederverwendbarkeit durch den Einsatz von GOT gegenüber OT feststellen. Die reinen OT-Lösungen stellen hier bereits eine Verbesserung gegenüber Java dar, profitieren aber nochmals (besonders in Hinblick auf die Wiederverwendbarkeit) deutlich von den Mitteln der Generizität in GOT. Unter den 18 übrigen Entwurfsmustern existieren nur zwei, die allein durch OT gegenüber Java eine Verbesserung der Wiederverwendbarkeit erfahren. Eine Verbesserung der Modularisierung ist hier bei elf weiteren Entwurfsmustern gegeben. Insgesamt hat sich gezeigt, dass besonders ergänzende Musterrollen von einer Realisierung als Rolle profitieren. Dies ist naheliegend, da die Basisklasse eine primäre Aufgabe erfüllt, die unabhängig von der Funktion der Rolle definiert ist. Für eigenständige Musterrollen ist eine Umsetzung in OT sinnvoll, wenn sie mit einem Kontext kollaborieren, der durch das umschließende Team abgebildet werden kann. In anderen Fällen beschränkt sich der Mehrwert der OT-Lösung auf die Kapselung der Rolle in einem größeren Team-Modul. Diese Form der Kapselung ist allerdings nicht ausschließlich von Vorteil, der Nutzen muss je nach Anwendungsfall bewertet werden.

Kapitel 7

Verwandte Arbeiten

Zur Motivation dieser Dissertation wurden bereits in den vorhergehenden Kapiteln einige verwandte Arbeiten zitiert. Neben diesen Publikationen gibt es noch weitere, die inhaltlich Bezugspunkte zu dieser Arbeit aufweisen und an dieser Stelle erwähnt werden sollen. Es werden Ansätze zur logischen Metaprogrammierung im Kontext der AOP, insbesondere zur Unterstützung von Generizität, sowie allgemein die Modularisierung von Rollen, Kontext und Design Patterns betrachtet.

Diese Arbeit zu GOT ist in der Folge von Herrmanns Überlegungen bezüglich der Integration einer Pointcut-Sprache in OT entstanden [55]. Herrmann diskutiert die fundamentalen Konzepte hinter Pointcuts und inwieweit sie bereits in OT abgedeckt sind. Eine Möglichkeit zur Quantifizierung auf Basis des ASG wird als sinnvolle Ergänzung von OT angesehen. Als geeignete Kandidaten für eine Query-Sprache werden funktionale Programmierung [31] und logische Metaprogrammierung (LMP) [46, 88] identifiziert. Mit dem GOT-Ansatz wurde die Integration von Queries durch LMP vorangetrieben.

7.1 LMP-Ansätze in der AOP

Mehrere Ansätze demonstrieren die Eignung von logischer Programmierung für die Analyse von Programmen [25, 47, 67, 84]. Die Fähigkeit zur Introspektion ist von besonderer Bedeutung für die Idee der Pointcuts in der AOP. De Volder und D'Hondt schlagen bereits in [107] den Einsatz von LMP als Basis für AOP vor. In der Folge sind verschiedene aspektorientierte Ansätze entstanden, die LMP zur Definition und Resolution von Pointcut-ähnlichen Strukturen nutzen. Der Ansatz von Gybels und Brichau [46] realisiert eine Pointcut-Sprache auf der Basis von Prolog. Pointcuts werden durch Prädikate abgebildet und per Unifikation aufgelöst. Ostermann et al. [88] gehen mit der Sprache *Alpha* noch einen Schritt weiter und bieten eine auf Prolog basierende Pointcut-Sprache mit umfassender Ausdrucksmächtigkeit. U.a. können in Alpha vergangene und sogar zukünftige Ereignisse als Joinpoints referenziert werden [70]. Die praktische Anwendbarkeit ist aufgrund der hohen Komplexität allerdings fraglich. Allan et al. [3] verwenden freie Variablen zur Selektion von Joinpoints auf Basis der Programmhistorie.

Der Fokus bei den zuvor genannten Ansätzen liegt auf einem ausdrucksmächtigeren Joinpoint-Modell. Der Modularisierung zum Zwecke der Wiederverwendbarkeit – realisiert durch Generizität – kommt hier keine besondere Bedeutung zu. Für De Volder et al. [26] sind Generizität und Parametrisierung dagegen grundlegend, um Aspekte zu schaffen, die in verschiedenen Kontexten wiederverwendbar sind. Kniesel und Rho argumentieren in gleicher Weise in [71], dass der Einsatz von Metavariablen nicht nur zur Definition von Pointcuts, sondern auch zur Definition der Aspekte

eingesetzt werden kann. Aus diesem ganzheitlich generischen Ansatz ergeben sich Vorteile für die Ausdrucksmächtigkeit, Evolutionsfähigkeit und Wiederverwendbarkeit von aspektorientierten Programmen. Diese Form der Generizität wird für die nächste Generation der AOP-Sprachen als Lösungsansatz vorgeschlagen, um die Defizite des Wildcard-Ansatzes zu beheben. Eine konkrete Entwicklung, die aus dieser Motivation resultiert, ist die Sprache *LogicAJ* [92], auf die im folgenden Abschnitt näher eingegangen wird.

7.2 Generizität durch strukturierte Metaprogrammierung

In der AOP wird die Komposition der Aspekte mit dem Basisprogramm durch das Einweben von Code-Fragmenten realisiert. Der eingewebte Code ist für den Programmierer allerdings nicht sichtbar, er muss die Komposition des Advice-Codes an den Joinpoints gedanklich nachvollziehen. Eine Alternative zur Komposition bietet die Metaprogrammierung, mit der das Programm direkt auf der Quellcode-Ebene transformiert wird. Dies eignet sich besonders für komplexere Manipulationen, wie z.B. der Ausgestaltung generischer Templates. Strukturierte Metaprogrammierungs-Ansätze sichern dabei die syntaktische Korrektheit generierter Programme zu – im Gegensatz zu Ansätzen auf Basis einer rein textbasierter Metaprogrammierung (vgl. Abschnitt 4.5).

Ein Ansatz, der vergleichbar zu GOT Generizität durch strukturierte Metaprogrammierung realisiert, ist *LogicAJ* [92], eine aspektorientierte Sprache im Stil von AspectJ. Metavariablen werden hier bei der Selektion von Joinpoints durch Prolog-ähnliche Prädikate und auch zur Definition von Advice-Code verwendet. Das Joinpoint-Modell ist unverändert von AspectJ übernommen. Die Realisierung von LogicAJ basiert auf *Conditional Transformations* [8] und dem *JTransformer* [72], einer Sprache und einem Werkzeug zur Transformation von Java-Quellcode. LogicAJ war eine wesentliche Inspiration für die Entwicklung von GOT. In LogicAJ werden – wie auch in Prolog – Metavariablen nicht deklariert. Die Zuordnung der Programmelemente ist Bestandteil der Prädikate der Query, z.B. beschreibt das Prädikat `class(?super)`, dass die Metavariable `?super` eine Klasse repräsentiert. In GOT liegt hier eine explizite Trennung in Deklaration und Definition vor, womit die eigentliche Query etwas überschaubarer wird. Die statische Prüfbarkeit ist in beiden Ansätzen gegeben. Generizität ist in LogicAJ nur auf Aspekte beschränkt, das Basisprogramm bleibt vollständig unabhängig („oblivious“) von den generischen Anteilen. In GOT sind generische Teams und Rollen dagegen ein direkt referenzierbarer Bestandteil des Programms, was eine dedizierte Unterscheidung von generischen und nicht-generischen Teilen erfordert. LogicAJ legt besonderen Wert auf die Ausdrucksmächtigkeit der Pointcutsprache ohne den Rückgriff auf Wildcards. GOT ist dagegen auf die Flexibilisierung der Rollenbindung und die Wiederverwendbarkeit fokussiert. Beide Sprachen reflektieren dabei die zugrunde liegende Philosophie des jeweiligen Paradigmas.

Meta-AspectJ [64] ist eine Sprache zur Generierung von AspectJ-Programmen auf der Basis von Code-Templates. Die Syntax des generierten Codes wird durch typisierte Strukturen repräsentiert. Fragmente generierten Codes können in Variablen gespeichert werden, deren Typ automatisch inferiert wird. Meta-AspectJ verzichtet auf die Integration einer dedizierten Querysprache, die Autoren sehen dies als unabhängige Problemstellung. Zur Selektion von Programmelementen nutzt Meta-AspectJ die vorhandenen Mittel von Java Reflection und Annotationen. Der Codeumfang der Queries in Meta-AspectJ beträgt oft ein Vielfaches des eigentlichen Modulcodes, was sich negativ auf die Verständlichkeit des Moduls auswirkt. Wir

teilen die Auffassung der Autoren, dass die Wahl der Querysprache ein eigenständiges Problemfeld darstellt. Allerdings ist dieses nicht gänzlich unabhängig, da die Querysprache erheblichen Einfluss auf die Ausdrucksmächtigkeit des Ansatzes hat. Durch die Wahl von Java Reflection wurde auch in Meta-AspectJ bereits eine Mindestanforderung an die Ausdrucksmächtigkeit festgelegt.

Vergleichbare Ansätze zur Generierung von Java-Programmen sind *TyRuBa* [106] und *SafeGen* [63]. Beide Ansätze setzen ebenfalls logische Programmierung zur Quantifizierung über Programmelemente ein.

Das *Compost* Framework [4] realisiert invasive Software Komposition über strukturierte Metaprogrammierung. Das Modell besitzt eine vordefinierte Menge von Joinpoints (implizite und deklarierte „Hooks“) und der Programmierer kann individuell weitere Joinpoints ergänzen. Es wird eine Vielzahl allgemeiner Kompositionsoperatoren angeboten. Im Gegensatz zu den anderen Ansätzen wird hier Java selbst als Kompositionssprache verwendet.

Ansätze wie Meta-AspectJ, TyRuBa, SafeGen und Compost unterstützen eine universelle Form der Metaprogrammierung. Sie sind strukturiert, erlauben aber zugleich den flexiblen Einsatz generischer Code-Bausteine, die durch explizite Operatoren mit dem Template kombiniert werden. Ansätze wie GOT und LogicAJ betten die Metaprogrammierung im Vergleich dazu enger in die Zielsprache ein. Sie sind dabei beschränkter aber auch fokussierter auf bestimmte Anwendungen und bieten eine homogenere Darstellung des Codes.

7.3 Rollen und Kontext

CaesarJ [9] ist eine auf Java basierende aspektorientierte Sprache mit einem besonderen Fokus auf Wiederverwendbarkeit. Die Sprache teilt etliche Konzepte mit OT. Es existieren Adapterklassen analog zu Rollen (genannt „Wrapper“), die Basisklassen („Wrappees“) in einem bestimmten Kontext repräsentieren. Die Adapterklassen werden dabei ebenfalls als Familien von virtuellen Klassen in größeren Modulen gebündelt. Zur Bindung der Adapter und der Basisklassen werden explizite Bindungsklassen definiert. Ein wesentlicher Unterschied besteht in der Definition der Bindung, CaesarJ verwendet hier AspectJ-ähnliche Pointcuts. Die Sprachmittel von CaesarJ sind im Vergleich zu OT insgesamt abstrakter ausgelegt und verlangen mehr expliziten Code für die Infrastruktur, beispielsweise muss das Lifting von Wrappee-Objekten zu Wrapper-Objekten konkret definiert werden. Dies gibt der Sprache aber zugleich ein breiter ausgelegtes Anwendungsspektrum. OT hat dagegen einen klaren Fokus auf das Konzept der Rolle, was in überschaubareren Ausdrucksmöglichkeiten resultiert. Die Pointcut-Notation erlaubt CaesarJ eine Quantifizierung der Bindungen für Wrapperklassen. Durch GOT ist diese Möglichkeit nun auch für Rollen gegeben. Die Ausdrucksmittel der logischen Programmierung vermeiden dabei aber die Probleme lexikalischer Mustervergleiche, wie sie bei CaesarJ zum Einsatz kommen (vgl. Abschnitt 3.2.2).

Sprachen mit expliziter Unterstützung von Rollen und Kontext sind *powerJava* [10] und *EpsilonJ* [100]. Viele der Konzepte sind mit denen von OT vergleichbar. Die Effekte des Liftings, die in OT zu großen Teilen implizit wirken, müssen allerdings in diesen Sprachen explizit im Code definiert werden. Eine Quantifizierung der Bindungen, wie sie GOT ermöglicht, ist nach unserem Wissen in diesen Sprachen nicht möglich.

Hannemann et al. beschreiben in [50] einen explizit rollenbasierten Ansatz, um Crosscutting Concerns zu modularisieren. Die Autoren demonstrieren den Wert des Rollen-Konzepts für die Abstraktion und die Modellierung. Für die Realisierung greifen sie jedoch auf die AOP zurück und müssen ihr Modell auf eine allgemeine Darstellung mit aspektorientierten Sprachmitteln abbilden.

gbeta [32] hat äußerlich nicht viel mit OT bzw. GOT gemein (die Syntax basiert auf der Programmiersprache BETA), beide Sprachen teilen aber wesentliche Konzepte zur Flexibilität von Modulen. gbeta besitzt keine explizite Unterstützung von Rollen, erlaubt aber ebenfalls das Verschachteln von Klassen, wobei innere Klassen auch verfeinert werden können (virtuelle Klassen inkl. Family Polymorphism). Klassen und Methoden können erweitert und verfeinert werden, indem generische Platzhalter (in Form von Code Hooks und Position Hooks) mit konkreten Werten gefüllt werden. Im Unterschied zu GOT gibt es aber keine Möglichkeit zur deklarativen Selektion von Programmelementen für diese Platzhalter.

Neben den genannten Ansätzen existieren u.a. mit der featureorientierten, der subjektorientierten und der kontextorientierten Programmierung weitere Paradigmen aus dem Umfeld der Aspektorientierung, die eine Modularisierung von Kollaborationen oder Kontext anstreben.

Die *featureorientierte Programmierung* [11] legt den Fokus auf die flexible Komposition von Features. Features bilden die Grundbausteine eines Programms. Sie sind implementiert als eine Kollaboration aus Klassenfragmenten [94]. Bei Ansätzen aus diesem Bereich, wie *FeatureC++* [6], steht die Kollaboration von Modulen im Vordergrund. FeatureC++ unterstützt dabei auch generische Programmierung mit Klassentemplates.

Bei der *subjektorientierte Programmierung* [51] steht die Komposition von Modulen im Vordergrund. Die Sprache *Hyper/J* [101] ermöglicht die mehrdimensionale Zerlegung von Anforderungen in einzelne Module, die mit verschiedenen Operatoren miteinander kombiniert werden können. Der Nachfolger von Hyper/J [52] integriert zusätzlich eine auf Unifikation basierende Query-Sprache zur Joinpoint-Selektion. Dabei werden Programmelemente allerdings ähnlich zu AspectJ anhand ihrer Art und anhand von lexikalischen Mustern (mit Wildcards) beschrieben.

Aspectual Collaborations [77] ist ein aspektorientierter Ansatz mit einem Fokus auf Kollaboration. Alternativ zu abstrakten Programmelementen, deren Definition über Vererbung beigesteuert wird, können hier die beteiligten Module ihre Schnittstelle mit erforderlichen (*expected*) und bereitgestellten (*provided*) Elementen deklarieren, die dann explizit in einem Kollaborationsmodul miteinander kombiniert werden. Sogenannte *aspektuelle* Methoden können dabei durch Method Call Interception mit anderen Methoden verknüpft werden, ohne dass die beiden Methoden eine direkte Abhängigkeit zueinander definieren.

Sowohl bei den featureorientierten als auch bei den subjektorientierten Ansätzen oder Aspectual Collaborations können Rollen indirekt als Features bzw. Module abgebildet werden. Die Komposition erfolgt aber im Wesentlichen statisch. Die Rolle als explizites Sprachmittel, insbesondere deren dynamischer Charakter, wird nicht direkt unterstützt.

Die *kontextorientierte Programmierung* [59] bietet eine direkte Unterstützung von kontextabhängiger Funktionalität. Es existieren zahlreiche kontextorientierte Ansätze, die u.a. auch Konzepte der AOP aufgreifen [7]. Das Konzept des *Kontexts* repräsentiert in diesem Zusammenhang eine Dimension zur (De-)Komposition von Funktionalität, die einen expliziten Variationspunkt in den Modulen des Programms bietet, der global aktiviert bzw. deaktiviert werden kann. Dem Konzept der *Rolle* kommt dabei keine Bedeutung zu.

7.4 Modularisierung von Design Patterns

Es existieren zahlreiche Arbeiten, die den Einfluss von neuen Sprachen bzw. Sprachmitteln auf die Implementierung von Design Patterns untersuchen. Ansätze aus dem Umfeld der Aspektorientierung legen einen besonderen Fokus auf verbesserte Modularisierung. Aus diesem Grund genießt die Evaluation anhand von Design Patterns in

diesem Bereich eine besondere Popularität. Die Studie von Hannemann und Kiczales zu AspectJ [49] hat viele vergleichbare Studien im aspektorientierten Umfeld initiiert oder beeinflusst. Dazu gehören Evaluationen der Sprachen *AspectJ* [17, 40], *Object Teams* [43, 87], *Eos* [91], *Ahead* [76] (bei diesen Studien wurden alle 23 Entwurfsmuster des Katalogs betrachtet), *CaesarJ* [96] (nur sieben Entwurfsmuster), *AspectS* [60] (nur Visitor und Decorator).

Die Evaluation anhand von Design Patterns wird auch außerhalb des Kontexts der Aspektorientierung durchgeführt, z.B. für die Sprache *Go* [93] und für funktionale Programmierung (*Origami*) [41].

Die direkt zu dieser Arbeit vergleichbaren Arbeiten zu OT [43, 87] und AspectJ [40, 49] wurden bereits ausführlich in Abschnitt 6.6 diskutiert.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Dissertation haben wir Probleme diskutiert, welche die Modularisierung und die Wiederverwendbarkeit von Rollen und Teams in OT beeinträchtigen, und einen Lösungsansatz dafür entwickelt.

Ausgangspunkt war eine Analyse der Modularisierung durch Rollen und Teams in Kapitel 3, die eine starke Kopplung von Rollen- und Basisklassen an den Bindungspunkten aufzeigte. Diese Kopplung wirkt sich negativ auf die Wiederverwendbarkeit, Wartbarkeit und Verständlichkeit der Software aus. Ursache dafür ist der Mangel zur Abstraktion der Bindungen durch eine deklarative Beschreibung, die dem gedanklichen Modell des Programmierers entspricht. Als Lösung wurde in Kapitel 4 der GOT-Ansatz entwickelt, der OT um Möglichkeiten zur generischen Programmierung erweitert. Kapitel 5 beschreibt kurz die technische Umsetzung eines Prototyps. Der Einfluss von GOT auf die Modularisierung und Wiederverwendbarkeit wurde schließlich in Kapitel 6 anhand von Beispielen zu Design Patterns evaluiert. Nachfolgend werden wir die Beiträge und Ergebnisse der Arbeit zusammenfassen und diskutieren.

8.1 Beiträge und Ergebnisse

Der Hauptbeitrag dieser Arbeit liegt in der Entwicklung des GOT-Ansatzes, der OT um Generizität auf Basis von logischer Metaprogrammierung erweitert. Mit GOT können generische Teamklassen erstellt werden, in denen Metavariablen anstelle von konkreten Programmelementen zum Einsatz kommen, womit die Beziehung zwischen Rollen- und Basisklassen abstrakt formuliert werden kann. Die Definition der Metavariablen erfolgt durch eine deklarative Beschreibung mit Prädikaten im Stil der logischen Programmierung. Anstatt der direkten Aufzählung konkreter Programmelemente kann der Programmierer so eine qualifizierte Auswahl durch Introspektion des Programms und Quantifizierung über dessen Elemente treffen.

Der GOT-Ansatz wurde als Spracherweiterung von OT mit Syntax und Semantik definiert. Im Wesentlichen wurden drei neue Sprachmittel zur Förderung der Generizität eingeführt:

- *Metavariablen* als Platzhalter für statische Programmelemente – primär für den Einsatz an den Bindungspunkten zwischen Rollen- und Basisklassen,
- *Queries* zur Definition von Metavariablen im deklarativen Stil der logischen Programmierung,
- *Per-Blöcke* zur kontrollierten Einbettung von Metavariablen im Code.

GOT ist mit einer transformationellen Semantik definiert, die ein GOT-Programm auf ein inhaltlich äquivalentes OT-Programm abbildet. Im Rahmen des Transformationsprozesses wird das Programm in eine Prolog-Faktenbasis überführt, damit dort die Queries ausgewertet werden können. Anschließend wird der Quellcode der generischen Klassen des Programms transformiert, wobei alle Per-Blöcke expandiert und Metavariablen durch ihre konkreten Belegungen ersetzt werden. Ein Prototyp für GOT wurde erfolgreich als Erweiterung des OTDT für Eclipse mit Anbindung an einen Prolog-Interpreter realisiert.

Ein weiterer Beitrag dieser Arbeit ist die Evaluation von GOT und OT für alle 23 Entwurfsmuster des klassischen Katalogs. Durch den Einsatz von GOT sehen wir bei fünf Mustern eine Verbesserung der Modularisierung und der Wiederverwendbarkeit. Bei den übrigen 18 Entwurfsmustern führt der Einsatz von OT allein (ohne Verwendung der GOT-Erweiterung) in elf Fällen zu einer verbesserten Modularisierung, in nur zwei dieser Fälle sehen wir auch eine Verbesserung der Wiederverwendbarkeit als gegeben. Für OT ist dies neben der Studie von Monteiro und Gomes ([87], vgl. Abschnitt 6.6.2) die einzige Evaluation dieser Art. Im Vergleich zu dieser Studie kommen wir insgesamt zu einer schlechteren Bewertung der Modularisierung und Wiederverwendbarkeit von OT-Lösungen. Die Ergebnisse von Monteiro und Gomes wurden von uns detailliert analysiert und diskutiert.

8.2 Diskussion

GOT hat als maßgebliches Ziel die Förderung der Modularisierung und der Wiederverwendbarkeit zum Zweck einer gesteigerten Produktivität bei der Softwareentwicklung. Zur Gesamtbeurteilung müssen die positiven und negativen Auswirkungen des GOT-Ansatzes gegenübergestellt werden. Unsere Diskussion beschränkt sich dabei auf eine qualitative Betrachtung der einzelnen Punkte, gestützt auf die gewonnenen Erkenntnisse aus der Evaluation der Entwurfsmuster. Eine aussagekräftige empirische Untersuchung über den Effekt von GOT in der Praxisanwendung hätte den Zeit- und Mittelrahmen dieser Arbeit überschritten.

Anhand der Evaluation der Entwurfsmuster konnte demonstriert werden, dass GOT tatsächlich eine positive Auswirkung auf die Modularisierung und Wiederverwendbarkeit besitzt. Insbesondere die Wiederverwendbarkeit konnte mit GOT deutlich gegenüber OT gesteigert werden. Die Gesamtzahl der wiederverwendbaren Lösungen liegt dabei in derselben Größenordnung, wie sie bei kritischer Betrachtung mit AspectJ erreicht wurde [40]. Auch die Evolutionsfähigkeit der Programme profitiert von GOT. Gelingt es, die Natur der Rolle-Basis-Beziehung deklarativ in einer Query zu erfassen, so ist auch im Fall der nachträglichen Manipulation einer Basisklasse keine Anpassung der adaptierenden Rolle mehr erforderlich.

Die Verbesserungen in GOT resultieren aus den erweiterten Möglichkeiten zur Abstraktion durch generische Programmierung. Die dafür eingeführten Sprachmittel gehören zu den Bereichen der logischen Programmierung und der Metaprogrammierung. Die Nutzung dieser Sprachmittel stellt höhere Anforderungen an den Programmierer und führt zu einer gesteigerten Komplexität der Programme. Um diese Auswirkungen möglichst gering zu halten, wurde den Grundsätzen Einfachheit, Orthogonalität und Konsistenz bei der Entwicklung von GOT besondere Aufmerksamkeit gewidmet:

- **Einfachheit.** Die neu eingeführten Sprachmittel beschränken sich auf Metavariablen, Queries und Query-Klassen sowie Per-Blöcke. Ferner können Entitäten innerhalb eines Per-Blocks als *generisch* deklariert werden. Die eigentliche Natur der Entität bleibt davon aber unberührt, für den Programmierer ist damit lediglich eine zusätzliche Beschränkung des Zugriffs verbunden. Die Abstraktion von Prolog-Prädikaten durch High-Level-Queries erleichtert es

Programmierern (besonders bei geringen Kenntnissen der logischen Programmierung) Queries zu verstehen und anzuwenden.

- **Orthogonalität.** Die GOT-Erweiterung führt zusätzliche Schlüsselwörter in die Sprache ein. Davon abgesehen wird die Originalsprache nicht verändert – die Anpassung von Syntax oder Semantik bestehender OT-Programme ist für den Einsatz von GOT nicht erforderlich. Die Verwendung der GOT-Mittel ist vollständig optional.

- **Konsistenz.** Mit der logischen Programmierung wurde ein von der OOP grundverschiedenes Paradigma integriert. Die Syntax der Queries und deren Modularisierung in Query-Klassen wurden dabei aber an die in Java üblichen Strukturen angenähert. Auch die Metaprogrammierung führt einen neuen Abstraktionsmechanismus in die Sprache ein, das bekannte Prinzip von Generalisierung und Spezialisierung durch Vererbung wurde dabei auf generische Klassen übertragen.

Trotz dieser Maßnahmen kann der GOT-Ansatz immer noch als „schwergewichtig“ bezeichnet werden. Zur Beherrschung der neuen Sprachmittel muss ein Programmierer sicherlich einen initialen Lernaufwand investieren. Entgegen der Intention kann sich der Einsatz der generischen Programmierung dann aufgrund der Komplexität immer noch negativ auf die Verständlichkeit des Programms auswirken. Damit verbunden ist natürlich auch entsprechendes Fehlerpotential.

Wie allgemein bei der Entwicklung wiederverwendbarer Software, so ist auch der Einsatz von GOT mit zusätzlichen Aufwänden verbunden, die sich negativ auf die Produktivität auswirken. Demgegenüber stehen positive Auswirkungen durch verbesserte Wiederverwendbarkeit und Evolutionsfähigkeit. Die Frage, ob der GOT-Ansatz daher insgesamt zu einer Produktivitätssteigerung führt, muss genauer untersucht werden. Wir vermuten, dass dies nicht pauschal beantwortet werden kann, sondern letztlich vom Anwendungsfall abhängt. Aufgrund unserer Erfahrungen erwarten wir, dass der sinnvolle Einsatz von GOT an zwei Bedingungen geknüpft ist: erstens, dass die Anwendung von einem gut qualifizierten Programmierer entwickelt wird, und zweitens, dass Anforderungen existieren, die durch eine Quantifizierung beschrieben werden. Letzteres ist eine klare Stärke des GOT-Ansatzes, die so weder in Java noch in OT zur Verfügung steht.

8.3 Ausblick

Die Entwicklung einer Joinpoint-Sprache für OT ist bereits seit vielen Jahren angedacht [55], blieb bislang aber eine offene Baustelle, da die Aufgabe mit einer erheblichen Komplexität verbunden ist und zudem fraglich war, inwieweit sich dieser Mechanismus aus der AOP für das Konzept des Rollenspiels eignet. Mit GOT besteht erstmals ein vollständiger Ansatz zur Realisierung dieser Anforderung.

Ausgehend von der vorherigen Diskussion erscheint es angebracht, die Praxistauglichkeit des GOT-Ansatzes weiter zu evaluieren. Allgemein ist – wie bei anderen Ansätzen aus dem Bereich der AOP – der Mehrwert der neuen Sprachmittel gegenüber der damit verbundenen Steigerung der Komplexität abzuwägen. Hierbei dürfte von Interesse sein, Problemfelder in der Praxis zu identifizieren, bei denen ein relevanter Bedarf zur Quantifizierung besteht, und betreffende Java-Projekte mit den Mitteln von GOT versuchsweise umzustrukturieren. Auf Basis solcher Untersuchungen kann dann besser eingeschätzt werden, inwieweit auch ein leichtgewichtigerer Ansatz zur Quantifizierung ohne den Einsatz logischer Metaprogrammierung in der Lage wäre, diese Anwendungsfälle zu realisieren.

Ein kritischer Faktor zur Anwendbarkeit von GOT ist die Unterstützung durch Werkzeuge. Die Integration eines ausgereiften Compilers in die Entwicklungsumgebung ist dafür eine selbstverständliche Notwendigkeit. Darüber hinaus kann der Transformationsprozess durch eine Steigerung der Performanz deutlich an Komfort und Transparenz gewinnen. Das ultimative Ziel wäre hier eine *Just-in-time*-Evaluation der Queries, womit der Programmierer bereits zur Design-Zeit ein direktes Feedback über die Auswirkungen der bevorstehenden Transformation erhalten kann. Die Notwendigkeit zur gedanklichen Vorhersage und nachträglichen Prüfung des Ergebnisses durch den Programmierer würde damit entfallen. Eine derartige Performanzsteigerung bietet sowohl technische als auch konzeptionelle Herausforderungen. Insbesondere ist die vollständige Neuberechnung aller Ergebnisse nach jeder Änderung schon bei einer Codebasis geringen Umfangs nicht mehr effizient. Zur Verbesserung werden in diesen Bereichen inkrementelle Verfahren eingesetzt, welche die Ergebnisse zwischenspeichern und nur solche Teile neu berechnen, die von Änderungen der Eingangsdaten betroffen sind. Der Programmierer modifiziert zu einem Zeitpunkt nur einen geringen Teil des Codes und nur dieser sowie die davon abhängigen Teile müssen neu verarbeitet werden. Für GOT wäre ein solches Vorgehen mit einer partiellen Anpassung und Auswertung der Faktenbasis in Prolog verbunden, Ansätze hierzu sind Gegenstand der aktuellen Forschung [14] und finden mit GOT einen zusätzlichen Anwendungsfall.

Anhang A

Grammatik

Für OT/J wird der LALR Parsergenerator *LPG* verwendet [18]. Die Grammatik wird in Form von BNF Regeln formuliert. Die vollständige Grammatik von GOT umfasst ca. 3600 Zeilen. Sie erweitert die ca. 3000 Zeilen umfassende Grammatik von OT/J[21]. Im Folgenden sind die wesentlichen Ergänzungen der Grammatik aufgeführt, gekürzt um die Makro-Anweisungen zur Steuerung des Parsers.

```
$Terminals

match
declare
otquery
per
-- anchoring metaidentifiers in grammar
MetaIdentifier
UnBoundMetaIdentifier
BoundMetaIdentifier

$Rules

-- OT enhancements

Goal ::= '...' GOTQueryMethodBlockStatementsopt

SimpleName -> 'MetaIdentifier'
SimpleName -> 'UnBoundMetaIdentifier'
SimpleName -> 'BoundMetaIdentifier'

ImportDeclaration -> GOTSingleQueryImportDeclaration
TypeDeclaration -> GOTQueryClassDeclaration

ClassHeader ::= ClassHeaderName ClassHeaderExtendsopt
   ClassHeaderImplementsopt ClassHeaderPlayedByopt
   Predicateopt GOTMatchOrDeclareopt
ClassHeaderName ::= ClassHeaderNm TypeParameters
ClassHeaderName -> ClassHeaderNm
ClassHeaderNm ::= Modifiersopt 'class' 'MetaIdentifier'
ClassHeaderNm ::= Modifiersopt 'class' '
   UnBoundMetaIdentifier'
```

[21] basierend auf OT-Version 2.1.0 – verfügbar unter `http://www.eclipse.org/objectteams/`

```
ClassHeaderNm ::= Modifiersopt 'class' '
   BoundMetaIdentifier'
ClassBodyDeclaration ::= GOTPerClassOpen
   ClassBodyDeclarationsopt GOTPerClassClose

VariableDeclaratorId ::= 'MetaIdentifier' Dimsopt
VariableDeclaratorId ::= 'UnBoundMetaIdentifier' Dimsopt
VariableDeclaratorId ::= 'BoundMetaIdentifier' Dimsopt

MethodHeaderName ::= Modifiersopt TypeParameters Type '
   MetaIdentifier' '('
MethodHeaderName ::= Modifiersopt TypeParameters Type '
   UnBoundMetaIdentifier' '('
MethodHeaderName ::= Modifiersopt TypeParameters Type '
   BoundMetaIdentifier' '('
MethodHeaderName ::= Modifiersopt Type 'MetaIdentifier'
   '('
MethodHeaderName ::= Modifiersopt Type '
   UnBoundMetaIdentifier' '('
MethodHeaderName ::= Modifiersopt Type '
   BoundMetaIdentifier' '('
ConstructorHeaderName ::=  Modifiersopt TypeParameters
   GOTUnBoundMetaIdentifier '('
ConstructorHeaderName ::=  Modifiersopt
   GOTUnBoundMetaIdentifier '('
CalloutHeaderLong ::= CalloutBindingLeftLong Modifiersopt
    MethodSpecShort
CallinHeaderLong ::= CallinBindingLeftLong CallinModifier
    Modifiersopt MethodSpecShort Predicateopt
CalloutFieldSpecLong ::= CalloutModifier Type
   GOTMetaReferenceType
BlockStatement ::= GOTPerMethodOpen BlockStatementsopt
   GOTPerMethodClose

-- Types

GOTType -> GOTPrimitiveType
GOTType -> GOTReferenceType
GOTPrimitiveType ::= GOTInt
GOTPrimitiveType ::= GOTBoolean
GOTInt ::= 'int'
GOTBoolean ::= 'boolean'
GOTMetaType -> GOTMetaReferenceType
GOTReferenceType ::= 'Identifier'
GOTMetaReferenceType ::= 'MetaIdentifier'
GOTUnBoundMetaIdentifier ::= 'Identifier'
GOTUnBoundMetaIdentifier ::= 'UnBoundMetaIdentifier'

-- Variable Declarators

GOTMetaVariableDeclarators -> GOTMetaVariableDeclarator
GOTMetaVariableDeclarators ::= GOTMetaVariableDeclarators
    ',' GOTMetaVariableDeclarator
```

```
GOTMetaVariableDeclarator ::= GOTMetaVariableDeclaratorId
    GOTEnterVariable GOTExitVariableWithoutInitialization
GOTMetaVariableDeclaratorId ::= 'MetaIdentifier'
   GOTDimsopt
GOTDimsopt -> Dimsopt
GOTEnterVariable ::= $empty
GOTExitVariableWithoutInitialization ::= $empty
GOTVariableDeclaratorId ::= 'Identifier' GOTDimsopt

-- Formal Parameterlist

GOTFormalParameterListopt ::= $empty
GOTFormalParameterListopt ::= GOTFormalParameterList
GOTFormalParameterList -> GOTFormalParameter
GOTFormalParameterList ::= GOTFormalParameterList ','
   GOTFormalParameter
GOTFormalParameter ::= GOTQueryModifiersopt GOTType
   GOTVariableDeclaratorId
GOTFormalParameter ::= GOTQueryModifiersopt GOTMetaType
   GOTMetaVariableDeclaratorId

-- Argumentlist

GOTArgumentListopt ::= $empty
GOTArgumentListopt -> GOTQueryArgumentList
GOTQueryArgumentList -> GOTQueryArgument
GOTQueryArgumentList ::= GOTQueryArgumentList ','
   GOTQueryArgument
GOTQueryArgument -> GOTName
GOTQueryArgument -> BooleanLiteral
GOTQueryArgument -> 'StringLiteral'
GOTQueryArgument -> 'IntegerLiteral'
GOTQueryArgument -> GOTElementLiteral
GOTName ::= Name
GOTElementLiteral -> GOTClassLiteral
GOTClassLiteral ::= Name '.' 'class'

-- Query Modifier

GOTQueryModifiersopt -> Modifiersopt

-- Query Class

GOTQueryClassDeclaration ::= GOTQueryClassHeader
   GOTQueryClassBody
GOTQueryClassHeader ::= GOTQueryClassHeaderName
GOTQueryClassHeaderName ::= GOTQueryModifiersopt otquery
   class Identifier
GOTQueryClassBody ::= '{'
   GOTQueryClassBodyDeclarationsopt '}'
GOTQueryClassBodyDeclarationsopt ::= $empty
GOTQueryClassBodyDeclarationsopt ->
   GOTQueryClassBodyDeclarations
```

```
GOTQueryClassBodyDeclarations ->
   GOTQueryClassBodyDeclaration
GOTQueryClassBodyDeclarations ::=
   GOTQueryClassBodyDeclarations
   GOTQueryClassBodyDeclaration
GOTQueryClassBodyDeclaration -> GOTQueryMethodDeclaration

-- Query Method

GOTQueryMethodDeclaration ::= GOTQueryMethodHeader
   GOTQueryMethodBody
GOTQueryMethodHeader ::= GOTQueryMethodHeaderName
   GOTFormalParameterListopt
   GOTQueryMethodHeaderRightParen
GOTQueryMethodHeaderName ::= GOTQueryModifiersopt otquery
    Identifier '('
GOTQueryMethodHeaderRightParen ::= ')'
GOTQueryMethodBody ::= GOTQueryNestedMethod
   GOTQueryMethodOpenBlock
   GOTQueryMethodBlockStatementsopt
   GOTQueryMethodCloseBlock
GOTQueryNestedMethod ::= $empty
GOTQueryMethodOpenBlock ::= '{'
GOTQueryMethodCloseBlock ::= '}'

-- Query Methodbody

GOTQueryMethodBlockStatementsopt ::= $empty
GOTQueryMethodBlockStatementsopt ->
   GOTQueryMethodBlockStatements
GOTQueryMethodBlockStatements ->
   GOTQueryMethodBlockStatement
GOTQueryMethodBlockStatements ::=
   GOTQueryMethodBlockStatements
   GOTQueryMethodBlockStatement
GOTQueryMethodBlockStatement ->
   GOTQueryLocalVariableDeclarationStatement
GOTQueryMethodBlockStatement ->
   GOTQueryExpressionStatement
GOTQueryLocalVariableDeclarationStatement ::=
   GOTQueryLocalVariableDeclaration ';'
GOTQueryLocalVariableDeclaration ::= GOTMetaType
   PushModifiers GOTMetaVariableDeclarators

-- Query Methodinvocation

GOTQueryMethodInvocation ::= Name '(' GOTArgumentListopt
   ')'
GOTQuerySuperInvocation ::= 'super' '('
   GOTArgumentListopt ')'

-- Expression

GOTQueryExpressionStatement ::= GOTQueryExpression
```

```
GOTQueryExpression -> GOTQueryAndExpression
GOTQueryAndExpression -> GOTQueryOrExpression
GOTQueryAndExpression ::= GOTQueryAndExpression '&&'
   GOTQueryOrExpression
GOTQueryOrExpression -> GOTQueryNotExpression
GOTQueryOrExpression ::= GOTQueryOrExpression '||'
   GOTQueryNotExpression
GOTQueryNotExpression -> GOTQueryPrimary
GOTQueryNotExpression ::= GOTPushNot GOTQueryPrimary
GOTPushNot ::= '!'
GOTQueryPrimary -> BooleanLiteral
GOTQueryPrimary -> GOTQueryMethodInvocation
GOTQueryPrimary -> GOTQuerySuperInvocation
GOTQueryPrimary ::= GOTPushLPAREN GOTQueryExpression
   GOTPushRPAREN
GOTPushLPAREN ::= '('
GOTPushRPAREN ::= ')'

-- Team

GOTMatchOrDeclareopt ::= $empty
GOTMatchOrDeclareopt ::= GOTMatchDeclaration
GOTMatchOrDeclareopt ::= GOTDeclareDeclaration

-- match

GOTMatchDeclaration ::= GOTMatchHeader GOTMatchBody
GOTMatchHeader ::= GOTMatchHeaderName
   GOTFormalParameterListopt GOTMatchHeaderRightParen
GOTMatchHeaderName ::=  GOTBeforeMatch 'match'
   GOTMatchHeaderLeftParen
GOTBeforeMatch ::= $empty
GOTMatchHeaderLeftParen ::= '('
GOTMatchHeaderRightParen ::= ')'
GOTMatchBody ::= GOTQueryNestedMethod GOTMatchOpenBlock
   GOTQueryMethodBlockStatementsopt GOTMatchCloseBlock
GOTMatchOpenBlock ::= '{'
GOTMatchCloseBlock ::= '}'

-- declare

GOTDeclareDeclaration ::= GOTDeclareHeader GOTDeclareBody
GOTDeclareHeader ::= GOTDeclareHeaderName
   GOTFormalParameterListopt GOTDeclareHeaderRightParen
GOTDeclareHeaderName ::=  GOTBeforeDeclare 'declare'
   GOTDeclareHeaderLeftParen
GOTBeforeDeclare ::= $empty
GOTDeclareHeaderLeftParen ::= '('
GOTDeclareHeaderRightParen ::= ')'
GOTDeclareBody ::= GOTQueryNestedMethod
   GOTDeclareOpenBlock GOTQueryMethodBlockStatementsopt
   GOTDeclareCloseBlock
GOTDeclareOpenBlock ::= '{'
GOTDeclareCloseBlock ::= '}'
```

```
-- per (Class)

GOTPerClassOpen ::= GOTPerClassHeader
   GOTPerClassBodyStart
GOTPerClassClose ::= GOTPerClassDeclaration
GOTPerClassBodyStart ::= GOTPerClassOpenBlock
GOTPerClassBodyEnd ::= GOTPerClassCloseBlock
GOTPerClassDeclaration ::= GOTPerClassBodyEnd
GOTPerClassHeader ::= GOTPerClassHeaderName
   GOTPerClassArgumentList GOTPerClassHeaderRightParen
GOTPerClassHeaderName ::=  'per'
   GOTPerClassHeaderLeftParen
GOTPerClassHeaderLeftParen ::= '('
GOTPerClassHeaderRightParen ::= ')'
GOTPerClassOpenBlock ::= '{'
GOTPerClassCloseBlock ::= '}'
GOTPerClassArgumentList ::= GOTPerClassArgument
GOTPerClassArgumentList ::= GOTPerClassArgumentList ','
   GOTPerClassArgument
GOTPerClassArgument ::= GOTName

-- per (Method)

GOTPerMethodOpen ::= GOTPerMethodBlockOpen
GOTPerMethodClose ::= GOTPerMethodBlock
GOTPerMethodBlockOpen ::= GOTPerMethodBlockHeader
   GOTPerMethodBlockOpenBlock
GOTPerMethodBlock ::= GOTPerMethodBlockCloseBlock
GOTPerMethodBlockHeader ::= GOTPerMethodBlockHeaderName
   GOTPerMethodBlockArgumentListopt
   GOTPerMethodBlockHeaderRightParen
GOTPerMethodBlockHeaderName ::=  'per'
   GOTPerMethodBlockHeaderLeftParen
GOTPerMethodBlockHeaderLeftParen ::= '('
GOTPerMethodBlockHeaderRightParen ::= ')'
GOTPerMethodBlockOpenBlock ::= '{'
GOTPerMethodBlockCloseBlock ::= '}'
GOTPerMethodBlockArgumentListopt ::= $empty
GOTPerMethodBlockArgumentListopt ::=
   GOTPerMethodBlockArgumentList
GOTPerMethodBlockArgumentList ::=
   GOTPerMethodBlockArgument
GOTPerMethodBlockArgumentList ::=
   GOTPerMethodBlockArgumentList ','
   GOTPerMethodBlockArgument
GOTPerMethodBlockArgument ::= GOTName

-- Import

GOTSingleQueryImportDeclaration ::=
   GOTSingleQueryImportDeclarationName ';'
GOTSingleQueryImportDeclarationName ::= 'import' 'otquery
   ' Name
```

Abbildungsverzeichnis

2.1 Beispiel für *Family Polymorphism* in OT 22

3.1 Eine Anwendung zur Darstellung von Formen, realisiert mit dem Entwurfsmuster *Observer* 40

4.1 Der Verarbeitungsprozess eines GOT-Programms 51
4.2 Fakten zur Abbildung des ASG der Methode `triple` 54
4.3 Auswertung der Queries im Rahmen der Transformation 62
4.4 Eine konkrete Anwendung des Observer Patterns 64
4.5 Die Mengen *MVAR*, *MTYPE* und *MVALUE* zur Spezifikation statischer Typisierung 66
4.6 Ein Beispiel generischer und nicht-generischer Anweisungen 69
4.7 Rollout einer generischen Anweisung 73
4.8 Transformation verfeinerter generischer Teams 83
4.9 Eine Hierarchie von Rollen- und Basisklassen 92
4.10 Ein generisches Team zur Generierung von Rollenhierarchien 93

5.1 Integration von GOT durch Adaption des OTDT 98

6.1 Szenario zur Anwendung des Entwurfsmusters *Abstract Factory* . . . 114
6.2 Das Entwurfsmuster *Bridge* in OT 121

Listingverzeichnis

2.1 Beispiel einer transparenten Rolle . . . 23
2.2 Ein Aspekt in AspectJ . . . 24
2.3 Inter-type Declarations in AspectJ . . . 26
3.1 Eine Teamklasse, die das Observer Pattern einsetzt . . . 41
4.1 Faktenrepräsentation der Methode `triple` . . . 54
4.2 Beispiel der High-Level-Query `changes`, die durch Kombination zweier Queries `writes` und `reachable` definiert wird. . . . 58
4.3 Beispiel für eine Low-Level-Query, die durch die Angabe einer Prolog-Annotation definiert wird. . . . 58
4.4 Eine Query mit einem optionalen Parameter . . . 59
4.5 Beispiel für eine komplexe Regel in Prolog . . . 60
4.6 Eine Query-Klasse . . . 61
4.7 Ein generisches Team mit einer Teamquery . . . 64
4.8 Ein vollständiges generisches Team . . . 77
4.9 Das Team `MyShapeDisplay` nach der Transformation . . . 78
4.10 Ein VGT zur Implementierung der Observer-Funktionalität . . . 89
4.11 Ein AGT zur Verfeinerung der Observer-Funktionalität . . . 90
4.12 Das Ausgangsteam zur Klasse `GenericObserver` nach der Transformation für das AGT `MyShapeDisplay` . . . 91
4.13 Das Team `MyShapeDisplay` nach der Transformation . . . 91
4.14 Eine rekursiv definierte Rollenhierarchie . . . 93
6.1 Das Entwurfsmuster *Visitor* für eine Baumstruktur in GOT . . . 107
6.2 Beispielimplementierung für das Entwurfsmuster *Memento* in GOT . 109
6.3 Eine vollständige Caretaker-Implementierung für die spezifische Basisklasse `Point` . . . 110
6.4 Der transformierte Code für die Anwendung des AGTs `PointCaretaker` 111
6.5 Universell anwendbare Variante des Entwurfsmusters *Memento* in GOT 111
6.6 Ein AGT als Subklasse des VGTs `UniversalCaretaker` . . . 113
6.7 Der transformierte Code des `UniversalCaretaker` für eine Anwendung auf `Shape` inklusive aller Subklassen . . . 113
6.8 Variante des Entwurfsmusters *Abstract Factory* in GOT . . . 115
6.9 Eine konkrete Fabrik für Steuerelemente der `X`-Familie . . . 115
6.10 Umsetzung des Chain of Responsibility Patterns für AWT Komponenten . . . 117
6.11 OT-Lösung für Composite . . . 118
6.12 Eine konkrete Anwendung von `AbstractComposite` . . . 119
6.13 Rollen für Interfaces in GOT . . . 127

Abkürzungsverzeichnis

AGT	Angewendetes generisches Team
AOP	Aspektorientierte Programmierung
ASG	Abstrakter semantischer Graph
AST	Abstrakter Syntax-Baum
BNF	Backus-Naur-Form
EBNF	Erweiterte Backus-Naur-Form
GOT	Generic Object Teams
JDT	Java Development Tooling
LMP	Logische Metaprogrammierung
LP	Logische Programmierung
MCI	Method Call Interception
OOP	Objektorientierte Programmierung
OT	Object Teams
OT/J	Object Teams/Java
OTDT	Object Teams Development Tooling
ROP	Rollenorientierte Programmierung
SQL	Standard Query Language
VGT	Verfeinerbares generisches Team

Literaturverzeichnis

[1] Al-Zaghameem, Abdullah O.: *Extending the Object Teams Programming Model into Distributed Environments.* Berlin, Deutschland : Dissertation, Technische Universität Berlin, 2012

[2] Aldrich, Jonathan: Open modules: modular reasoning about advice. In: Black, Andrew P. (Hrsg.): *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)* Bd. 3586. Berlin, Heidelberg : Springer, 2005 (Lecture Notes in Computer Science). – ISBN 978–3–540–27992–1, 978–3–540–31725–8, S. 144–168

[3] Allan, Chris ; Avgustinov, Pavel ; Christensen, Aske S. ; Hendren, Laurie ; Kuzins, Sascha ; Lhoták, Ondřej ; Moor, Oege de ; Sereni, Damien ; Sittampalam, Ganesh ; Tibble, Julian: Adding trace matching with free variables to AspectJ. In: *Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).* New York, NY, USA : ACM, 2005. – ISBN 1–59593–031–0, S. 345–364

[4] Assmann, Uwe: *Invasive Software Composition.* NJ, USA : Springer, 2003. – ISBN 3–540–44385–1

[5] Apel, Sven ; Batory, D.: How AspectJ is used: an analysis of eleven AspectJ programs. In: *Journal of Object Technology* 9 (2010), S. 117–142

[6] Apel, Sven ; Leich, Thomas ; Rosenmüller, Marko ; Saake, Gunter: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Glück, Robert (Hrsg.) ; Lowry, Michael (Hrsg.): *Proceedings of the 4th international conference on Generative Programming and Component Engineering (GPCE)* Bd. 3676. Berlin, Heidelberg : Springer, 2005. – ISBN 978–3–540–29138–1, 978–3–540–31977–1, S. 125–140

[7] Appeltauer, Malte ; Hirschfeld, Robert ; Haupt, Michael ; Lincke, Jens ; Perscheid, Michael: A comparison of context-oriented programming languages. In: *International Workshop on Context-Oriented Programming (COP).* New York, NY, USA : ACM, 2009. – ISBN 978–1–60558–538–3, S. 6:1–6:6

[8] Appeltauer, Malte ; Kniesel, Günter: Towards Concrete Syntax Patterns for Logic-based Transformation Rules. In: *Electronic Notes in Theoretical Computer Science* 219 (2008), November, S. 113 – 132. – ISSN 1571–0661

[9] Aracic, Ivica ; Gasiunas, Vaidas ; Mezini, Mira ; Ostermann, Klaus: An overview of CaesarJ. In: Rashid, Awais (Hrsg.) ; Aksit, Mehmet (Hrsg.): *Transactions on Aspect-Oriented Software Development I* Bd. 3880. Berlin, Heidelberg : Springer, 2006. – ISBN 978–3–540–32972–5, 978–3–540–32974–9, S. 135–173

[10] Baldoni, Matteo ; Boella, Guido ; Torre, Leendert van d.: Roles as a Coordination Construct: Introducing powerJava. In: *Electronic Notes in Theoretical Computer Science* 150 (2006), März, Nr. 1, S. 9–29. – ISSN 1571–0661

[11] Batory, Don ; Sarvela, Jacob N. ; Rauschmayer, Axel: Scaling step-wise refinement. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE)*. Washington, DC, USA : IEEE, 2003. – ISBN 0–7695–1877–X, S. 187–197

[12] Biggerstaff, Ted J.: A perspective of generative reuse. In: *Annals of Software Engineering* 5 (1998), Januar, S. 169–226. – ISSN 1022–7091

[13] Biggerstaff, Ted J. ; Richter, Charles: Reusability Framework, Assessment, and Directions. In: *IEEE Software* 4 (1987), März, Nr. 2, S. 41–49. – ISSN 0740–7459

[14] Bolz, Carl F. ; Leuschel, Michael ; Rigo, Armin: Towards just-in-time partial evaluation of Prolog. In: De Schreye, Danny (Hrsg.): *Proceedings of the 19th international symposium on Logic-Based Program Synthesis and Transformation (LOPSTR)* Bd. 6037. Berlin, Heidelberg : Springer, 2010 (Lecture Notes in Computer Science). – ISBN 978–3–642–12591–1, 978–3–642–12592–8, S. 158–172

[15] Bracha, Gilad ; Cook, William: Mixin-based inheritance. In: *Proceedings of the European Conference on Object-Oriented Programming and on Object-Oriented Programming Systems, Languages, and Applications (ECOOP/OOPSLA)*. New York, NY, USA : ACM, 1990. – ISBN 0–89791–411–2, S. 303–311

[16] Bracha, Gilad ; Odersky, Martin ; Stoutamire, David ; Wadler, Philip: Making the future safe for the past: adding genericity to the Java programming language. In: *Proceedings of the 13th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, NY, USA : ACM, 1998. – ISBN 1–58113–005–8, S. 183–200

[17] Cacho, Nelio ; Sant'Anna, Claudio ; Figueiredo, Eduardo ; Garcia, Alessandro ; Batista, Thais ; Lucena, Carlos: Composing design patterns: a scalability study of aspect-oriented programming. In: *Proceedings of the 5th international conference on Aspect-Oriented Software Development (AOSD)*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–300–X, S. 109–121

[18] Charles, Philippe ; Fisher, Gerald ; Fuhrer, Robert: *LALR Parser Generator (LPG)*. `http://lpg.sourceforge.net/`. – (Abruf 12.07.13)

[19] Clifton, Curtis ; Leavens, Gary T.: Obliviousness, modular reasoning, and the behavioral subtyping analogy. In: *Proceedings of 1st workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT)*, 2003

[20] Clocksin, William F. ; Mellish, Christopher S.: *Programming in Prolog: Using the ISO Standard*. 5th edition. Berlin Heidelberg : Springer, 2003. – ISBN 978–3–540–00678–7, 978–3–642–55481–0

[21] Constantinides, Constantinos ; Skotiniotis, Therapon ; Stoerzer, Maximilian: AOP considered harmful. In: *1st European Interactive Workshop on Aspect Systems (EIWAS)*, 2004

[22] Cooper, James W.: *Java design patterns: a tutorial*. Boston, MA, USA : Addison-Wesley, 2000. – ISBN 0–201–48539–7

[23] Coulange, Bernard: *Software reuse*. Springer, 1998. – ISBN 978–3–540–76084–9

[24] De Fraine, Bruno ; Südholt, Mario ; Jonckers, Viviane: StrongAspectJ: flexible and safe pointcut/advice bindings. In: *Proceedings of the 7th international conference on Aspect-Oriented Software Development (AOSD)*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–044–9, S. 60–71

[25] De Roover, Coen ; Noguera, Carlos ; Kellens, Andy ; Jonckers, Vivane: The SOUL Tool Suite for Querying Programs in Symbiosis with Eclipse. In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ)*. New York, NY, USA : ACM, 2011. – ISBN 978–1–4503–0935–6, S. 71–80

[26] De Volder, Kris ; Tourwe, Tom ; Brichau, Johan: Logic Meta Programming as a Tool for Separation of Concerns. In: *Proceedings of the ECOOP Workshop on Aspects and Dimensions of Concerns*, 2000

[27] Dijkstra, Edsger W.: On the role of scientific thought. In: *Selected writings on Computing: A Personal Perspective*. New York, NY, USA : Springer, 1982 (Texts and Monographs in Computer Science). – ISBN 978–1–4612–5697–7, 978–1–4612–5695–3, S. 60–66

[28] Douence, Rémi ; Fradet, Pascal ; Südholt, Mario: Composition, reuse and interaction analysis of stateful aspects. In: *Proceedings of the 3rd international conference on Aspect-Oriented Software Development (AOSD)*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–842–3, S. 141–150

[29] Duke, Roger ; Rose, Gordon ; Smith, Graeme: Object-Z: a Specification Language Advocated for the Description of Standards. In: *Computer Standards and Interfaces* 17 (1995), S. 511–533

[30] Eclipse Foundation: *Eclipse*. `http://www.eclipse.org/`. – (Abruf 09.08.13)

[31] Eichberg, Michael ; Mezini, Mira ; Ostermann, Klaus: Pointcuts as Functional Queries. In: Chin, Wei-Ngan (Hrsg.): *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS)* Bd. 3302. Berlin, Heidelberg : Springer (Lecture Notes in Computer Science). – ISBN 978–3–540–23724–2, 978–3–540–30477–7, S. 366–381

[32] Ernst, Erik: *gbeta - a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. Aarhus, Dänemark : PhD thesis, Department of Computer Science, University of Aarhus, 1999

[33] Ernst, Erik: Family Polymorphism. In: Knudsen, Jørgen L. (Hrsg.): *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)* Bd. 2072. Berlin, Heidelberg : Springer, 2001 (Lecture Notes in Computer Science). – ISBN 978–3–540–42206–8, 978–3–540–45337–6, S. 303–326

[34] Ernst, Erik: Inheritance versus parameterization. In: *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance (MASPEGHI)*. New York, NY, USA : ACM, 2013. – ISBN 978–1–4503–2046–7, S. 26–29

[35] Filman, Robert E. ; Friedman, Daniel P.: Aspect-Oriented Programming is Quantification and Obliviousness. In: *OOPSLA Workshop on Advanced Separation of Concerns*, 2000

[36] Fowler, Martin: *Refactoring: improving the design of existing code.* Boston, MA, USA : Addison-Wesley, 1999. – ISBN 0–201–48567–2

[37] Fowler, Martin: *Domain Specific Languages.* 1st edition. Addison-Wesley, 2010. – ISBN 0321712943, 978–0321712943

[38] Frakes, William B. ; Kang, Kyo: Software Reuse Research: Status and Future. In: *IEEE Transactions on Software Engineering* 31 (2005), Juli, Nr. 7, S. 529–536. – ISSN 0098–5589

[39] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns – Elements of Reusable Object-Oriented Software.* Boston, MA, USA : Addison-Wesley, 1995. – ISBN 978–0201633610

[40] Garcia, Alessandro ; Sant'Anna, Cláudio ; Figueiredo, Eduardo ; Kulesza, Uirá ; Lucena, Carlos ; Staa, Arndt von: Modularizing design patterns with aspects: a quantitative study. In: *Proceedings of the 4th international conference on Aspect-Oriented Software Development (AOSD).* New York, NY, USA : ACM, 2005. – ISBN 1–59593–042–6, S. 3–14

[41] Gibbons, Jeremy: Design patterns as higher-order datatype-generic programs. In: *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming (WGP).* New York, NY, USA : ACM, 2006. – ISBN 1–59593–492–8, S. 1–12

[42] Gómez Esperón, Daniél: *Entwicklung eines Moduls zur Integration von Prolog in das Generic Object Teams Framework.* Berlin, Deutschland : Diplomarbeit, Technische Universität Berlin, 2013

[43] Gomes, João ; Monteiro, Miguel P.: Design pattern implementation in Object Teams. In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC).* New York, NY, USA : ACM, 2010. – ISBN 978–1–60558–639–7, S. 2119–2120

[44] Greenwood, Phil ; Rashid, Awais ; Khatchadourian, Raffi T.: Contributing Factors to Pointcut Fragility. In: *3rd Workshop on Assessment of Contemporary Modularization Techniques (ACoM)*, 2009

[45] Griswold, William G. ; Sullivan, Kevin ; Song, Yuanyuan ; Shonle, Macneil ; Tewari, Nishit ; Cai, Yuanfang ; Rajan, Hridesh: Modular Software Design with Crosscutting Interfaces. In: *IEEE Software* 23 (2006), Nr. 1, S. 51–60. – ISSN 0740–7459

[46] Gybels, Kris ; Brichau, Johan: Arranging language features for more robust pattern-based crosscuts. In: *Proceedings of the 2nd international conference on Aspect-Oriented Software Development (AOSD).* New York, NY, USA : ACM, 2003. – ISBN 1–58113–660–9, S. 60–69

[47] Hajiyev, Elnar ; Verbaere, Mathieu ; Moor, Oege de: codeQuest: Scalable Source Code Queries with Datalog. In: Thomas, Dave (Hrsg.): *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)* Bd. 4067. Berlin, Heidelberg : Springer, 2006 (Lecture Notes in Computer Science). – ISBN 978–3–540–35726–1, 978–3–540–35727–8, S. 2–27

[48] Hanenberg, Stefan ; Oberschulte, Christian ; Unland, Rainer: Refactoring of aspect-oriented software. In: *4th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net. ObjectDays)*, 2003, S. 19–35

[49] Hannemann, Jan ; Kiczales, Gregor: Design pattern implementation in Java and AspectJ. In: *Proceedings of the 17th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, NY, USA : ACM, 2002. – ISBN 1–58113–471–1, S. 161–173

[50] Hannemann, Jan ; Murphy, Gail C. ; Kiczales, Gregor: Role-based refactoring of crosscutting concerns. In: *Proceedings of the 4th international conference on Aspect-Oriented Software Development (AOSD)*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–042–6, S. 135–146

[51] Harrison, William ; Ossher, Harold: Subject-oriented programming: a critique of pure objects. In: *Proceedings of the 8th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, NY, USA : ACM, 1993. – ISBN 0–89791–587–9, S. 411–428

[52] Harrison, William ; Ossher, Harold ; Tarr, Peri: General composition of software artifacts. In: Löwe, Welf (Hrsg.) ; Südholdt, Mario (Hrsg.): *Proceedings of the 5th international symposium on Software Composition (SC)* Bd. 4089. Berlin, Heidelberg : Springer, 2006 (Lecture Notes in Computer Science). – ISBN 978–3–540–37657–6, 978–3–540–37659–0, S. 194–210

[53] Herrmann, Stephan: Object Teams: Improving Modularity for Crosscutting Collaborations. In: Aksit, Mehmet (Hrsg.) ; Mezini, Mira (Hrsg.) ; Unland, Rainer (Hrsg.): *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World (NODe)* Bd. 2591. Berlin, Heidelberg : Springer, 2003 (Lecture Notes in Computer Science). – ISBN 978–3–540–00737–1, 978–3–540–36557–0, S. 248–264

[54] Herrmann, Stephan: Confinement and representation encapsulation in Object Teams / Technische Universität Berlin. 2004 (06). – Technischer Report

[55] Herrmann, Stephan: Are Pointcuts a First-Class Language Feature? In: *Proceedings of the 5th international workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2006

[56] Herrmann, Stephan: A precise model for contextual roles: The programming language Object Teams/Java. In: *Applied Ontology - Roles, an interdisciplinary perspective* 2 (2007), Nr. 2, S. 181–207. – ISSN 1570–5838

[57] Herrmann, Stephan: Gradual Encapsulation. In: *Journal of Object Technology* 7 (2008), Nr. 9, S. 47–68

[58] Herrmann, Stephan: Confined Roles and Decapsulation in Object Teams – Contradiction or Synergy? In: Aksit, Mehmet (Hrsg.) ; Mezini, Mira (Hrsg.) ; Unland, Rainer (Hrsg.): *Aliasing in Object-Oriented Programming. Types, Analysis and Verification* Bd. 7850. Berlin, Heidelberg : Springer, 2013. – ISBN 978–3–642–36945–2, 978–3–642–36946–9, S. 443–470

[59] Hirschfeld, Robert ; Costanza, Pascal ; Nierstrasz, Oscar: Context-oriented Programming. In: *Journal of Object Technology* 7 (2008), Nr. 3, S. 125–151

[60] Hirschfeld, Robert ; Lämmel, Ralf ; Wagner, Matthias: Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration. In: *Proceedings of the 3rd German GI Workshop on Aspect-Oriented Software Development*, 2003, S. 8

[61] Hoffmann, Jan M.: *Entwicklung der Werkzeugunterstützung für die Sprachерweiterung Generic Object Teams.* Berlin, Deutschland : Diplomarbeit, Technische Universität Berlin, 2011

[62] Huang, Shan S. ; Green, Todd J. ; Loo, Boon T.: Datalog and emerging applications: an interactive tutorial. In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD).* New York, NY, USA : ACM, 2011. – ISBN 978–1–4503–0661–4, S. 1213–1216

[63] Huang, Shan S. ; Zook, David ; Smaragdakis, Yannis: Statically safe program generation with SafeGen. 3676 (2005), S. 309–326. ISBN 978–3–540–29138–1, 978–3–540–31977–1

[64] Huang, Shan S. ; Zook, David ; Smaragdakis, Yannis: Domain-specific Languages and Program Generation with Meta-AspectJ. In: *ACM Transactions On Software Engineering and Methodology (TOSEM)* 18 (2008), November, Nr. 2, S. 6:1–6:32. – ISSN 1049–331X

[65] Hunt, Andrew ; Thomas, David: *The pragmatic programmer: from journeyman to master.* Boston, MA, USA : Addison-Wesley, 1999. – ISBN 0–201–61622–X

[66] International Organization for Standardization: *ISO/IEC 9075 standard: Information technology - Database languages - SQL.* `http://www.iso.org/`. – (Abruf 28.06.13)

[67] Janzen, Doug ; De Volder, Kris: Navigating and querying code without getting lost. In: *Proceedings of the 2nd international conference on Aspect-Oriented Software Development (AOSD).* New York, NY, USA : ACM, 2003. – ISBN 1–58113–660–9, S. 178–187

[68] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William G.: An Overview of AspectJ. In: Knudsen, Jørgen L. (Hrsg.): *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)* Bd. 2072. Berlin, Heidelberg : Springer, 2001 (Lecture Notes in Computer Science). – ISBN 978–3–540–42206–8, 978–3–540–45337–6, S. 327–354

[69] Kiczales, Gregor ; Irwin, John ; Lamping, John ; Loingtier, Jean-Marc ; Lopes, Cristina V. ; Maeda, Chris ; Mendhekar, Anurag: Aspect-Oriented Programming. In: Aksit, Mehmet (Hrsg.) ; Matsuoka, Satoshi (Hrsg.): *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)* Bd. 1241. Berlin, Heidelberg : Springer, 1997 (Lecture Notes in Computer Science). – ISBN 978–3–540–63089–0, 978–3–540–69127–3, S. 220–242

[70] Klose, Karl ; Ostermann, Klaus: Back to the Future: Pointcuts as Predicates over Traces. In: *Proceedings of the 4th international workshop on Foundations of Aspect-Oriented Languages (FOAL)*, 2005

[71] Kniesel, Günter ; Rho, Tobias: A Definition, Overview and Taxonomy of Generic Aspect Languages. In: *L'Objet* 11 (2006), Nr. 2–3

[72] Kniesel, Günter ; Hannemann, Jan ; Rho, Tobias: A comparison of logic-based infrastructures for concern detection and extraction. In: *Proceedings of the 3rd workshop on Linking Aspect Technology and Evolution (LATE)*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–655–4

[73] Koppen, Christian ; Stoerzer, Maximilian: Pcdiff: Attacking the fragile pointcut problem. In: *1st European Interactive Workshop on Aspects in Software (EIWAS)*, 2004

[74] Kristensen, Bent B. ; Osterbye, Kasper: Roles: conceptual abstraction theory and practical language issues. In: *Theory and Practice of Object Systems* 2 (1996), Dezember, Nr. 3, S. 143–160. – ISSN 1074–3227

[75] Krueger, Charles W.: Software reuse. In: *ACM Computing Surveys (CSUR)* 24 (1992), Juni, Nr. 2, S. 131–183. – ISSN 0360–0300

[76] Kuhlemann, Martin ; Rosenmüller, Marko ; Apel, Sven ; Leich, Thomas: On the duality of aspect-oriented and feature-oriented design patterns. In: *Proceedings of the 6th workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*. New York, NY, USA : ACM, 2007. – ISBN 978–1–59593–657–8

[77] Lieberherr, Karl ; Lorenz, David H. ; Ovlinger, Johan: Aspectual Collaborations: Combining Modules And Aspects. In: *The Computer Journal* 46 (2003)

[78] Lima, Arlindo ; Goulão, Miguel ; Monteiro, Miguel P.: Evidence-Based Comparison of Modularity Support Between Java and Object Teams. In: *Proceedings of Empirical Evaluation of Software Composition Techniques (ESCOT)*, 2010

[79] Liskov, Barbara: Keynote address - data abstraction and hierarchy. In: *Addendum to the proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, NY, USA : ACM, 1987. – ISBN 0–89791–266–7, S. 17–34

[80] Lopes, Cristina V. ; Dourish, Paul ; Lorenz, David H. ; Lieberherr, Karl: Beyond AOP: toward naturalistic programming. In: *ACM SIGPLAN Notices* 38 (2003), Dezember, Nr. 12, S. 34–43. – ISSN 0362–1340

[81] MacLennan, Bruce J.: *Principles of programming languages - design, evaluation, and implementation*. Holt, Rinehart and Winston, 1987. – ISBN 978–0–03–005163–0

[82] Madsen, O. L. ; Moller-Pedersen, B.: Virtual classes: a powerful mechanism in object-oriented programming. In: *Proceedings of the 4th annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, NY, USA : ACM, 1989. – ISBN 0–89791–333–7, S. 397–406

[83] McIlroy, M. D.: Mass Produced Software Components. In: Naur, Peter (Hrsg.) ; Randell, Brian (Hrsg.): *Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany*. Brussels, Belgium, 1969

[84] Mens, Kim ; Michiels, Isabel ; Wuyts, Roel: Supporting Software Development through Declaratively Codified Programming Patterns. In: *Elsevier Journal on Expert Systems with Applications* 23 (2002), Nr. 4, S. 405–431

[85] Meyer, Bertrand: *Object-Oriented Software Construction*. 2nd edition. Upper Saddle River, NJ, USA : Prentice-Hall, 1997. – ISBN 0136291554

[86] Mikhajlov, Leonid ; Sekerinski, Emil: A Study of The Fragile Base Class Problem. In: *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP)*, 1998, S. 355–382

[87] Monteiro, Miguel P. ; Gomes, João: Implementing design patterns in Object Teams. In: *Software: Practice and Experience* (2012). `http://dx.doi.org/10.1002/spe.2154`. – DOI 10.1002/spe.2154. – ISSN 1097–024X

[88] Ostermann, Klaus ; Mezini, Mira ; Bockisch, Christoph: Expressive Pointcuts for Increased Modularity. In: *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP)*. 2005, S. 214–240

[89] Parnas, D. L.: Information distribution aspects of design methodology. In: *Proceedings of IFIP Congress*, 1971, S. 339—-344

[90] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. In: *Communications of the ACM* 15 (1972), Dezember, Nr. 12, S. 1053–1058. – ISSN 0001–0782

[91] Rajan, Hridesh: Design pattern implementations in Eos. In: *Proceedings of the 14th Conference on Pattern Languages of Programs (PLOP)*. New York, NY, USA : ACM, 2007. – ISBN 978–1–60558–411–9, S. 9:1–9:11

[92] Rho, Tobias ; Kniesel, Günter: Uniform Genericity for Aspect Languages / Institut für Informatik III, Universität Bonn. 2004 (IAI-TR-2004-4). – Technischer Report

[93] Schmager, Frank ; Cameron, Nicholas ; Noble, James: GoHotDraw: evaluating the Go programming language with design patterns. In: *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*. New York, NY, USA : ACM, 2010. – ISBN 978–1–4503–0547–1, S. 10:1–10:6

[94] Smaragdakis, Yannis ; Batory, Don: Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. In: *ACM Transactions On Software Engineering and Methodology (TOSEM)* 11 (2002), April, Nr. 2, S. 215–255. – ISSN 1049–331X

[95] Soukup, Jiri: Pattern languages of program design. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1995. – ISBN 0–201–60734–4, Kapitel Implementing patterns, S. 395–412

[96] Sousa, Edgar ; Monteiro, Miguel P.: Implementing design patterns in CaesarJ: an exploratory study. In: *Proceedings of 6th workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT)*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–144–6, S. 6:1–6:6

[97] Steimann, Friedrich: On the representation of roles in object-oriented and conceptual modelling. In: *Data and Knowledge Engineering* 35 (2000), Oktober, Nr. 1, S. 83–106. – ISSN 0169–023X

[98] Steimann, Friedrich: The paradoxical success of aspect-oriented programming. In: *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–348–4, S. 481–497

[99] Sullivan, Kevin ; Griswold, William G. ; Song, Yuanyuan ; Cai, Yuanfang ; Shonle, Macneil ; Tewari, Nishit ; Rajan, Hridesh: Information hiding interfaces for aspect-oriented design. In: *Proceedings of the 10th European Software Engineering Conference (ESEC) held jointly with 13th ACM SIGSOFT international symposium on Foundations of Software Engineering (FSE)*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–014–0, S. 166–175

[100] Tamai, Tetsuo ; Ubayashi, Naoyasu ; Ichiyama, Ryoichi: An adaptive object model with dynamic role binding. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE)*. New York, NY, USA : ACM, 2005. – ISBN 1–58113–963–2, S. 166–175

[101] Tarr, Peri ; Ossher, Harold: Hyper/J: multi-dimensional separation of concerns for Java. In: *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*. Washington, DC, USA : IEEE, 2001. – ISBN 0–7695–1050–7, S. 821–822

[102] Tarr, Peri ; Ossher, Harold ; Harrison, William ; Sutton, Stanley M. Jr.: N degrees of separation: multi-dimensional separation of concerns. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE)*. New York, NY, USA : ACM, 1999. – ISBN 1–58113–074–0, S. 107–119

[103] TIOBE Software BV: *TIOBE Programming Community Index*. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`. Version: August 2013. – (Stand 09.08.13)

[104] Ullenboom, Christian: *Java ist auch eine Insel*. 10. Auflage. Galileo Computing, 2012 `http://openbook.galileocomputing.de/javainsel/`. – ISBN 978–3–8362–1802–3

[105] Vanderperren, Wim ; Suvée, Davy ; Cibrán, María A. ; Fraine, Bruno D.: Stateful Aspects in JAsCo. In: Geschwind, Thomas (Hrsg.) ; Assmann, Uwe (Hrsg.) ; Nierstrasz, Oscar (Hrsg.): *Proceedings of the 4th international workshop on Software Composition (SC)* Bd. 3628, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 978–3–540–28748–3, 978–3–540–28749–0, S. 167–181

[106] Volder, Kris de: *Type-Oriented Logic Meta Programming*. Brüssel, Belgien : PhD thesis, Programming Technology Laboratory, Vrije Universiteit Brussel, 1998

[107] Volder, Kris D. ; D'Hondt, Theo: Aspect-Oriented Logic Meta Programming. In: Cointe, Pierre (Hrsg.): *Proceedings of the 2nd International Conference on Meta-Level Architectures and Reflection* Bd. 1616. Berlin, Heidelberg : Springer, 1999 (Lecture Notes in Computer Science). – ISBN 978–3–540–66280–8, 978–3–540–48443–1, S. 250–272

[108] Wand, Mitchell ; Kiczales, Gregor ; Dutchyn, Christopher: A semantics for advice and dynamic join points in aspect-oriented programming. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 26 (2004), September, Nr. 5, S. 890–910. – ISSN 0164–0925

[109] Wimmer, Manuel ; Schauerhuber, Andrea ; Kappel, Gerti ; Retschitzegger, Werner ; Schwinger, Wieland ; Kapsammer, Elizabeth: A survey on UML-based aspect-oriented design modeling. In: *ACM Computing Surveys (CSUR)* 43 (2011), Oktober, Nr. 4, S. 28:1–28:33. – ISSN 0360–0300