UNIVERSITÄT
KOBLENZ · LANDAU

Das Dagstuhl Middle Metamodel im Kontext sprachunabhängigen Refactorings

Diplomarbeit

zur Erlangung des Grades eines
Diplom-Informatikers
im Studiengang Informatik

vorgelegt von

Steffen Flick

Betreuer: Prof. Dr. Jürgen Ebert
Dr. Andreas Winter

Universität Koblenz-Landau
Fachbereich Informatik
Institut für Softwaretechnik

Koblenz, im August 2006

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.

_____

Steffen Flick

Danksagung

Zunächst bedanke ich mich bei meiner lieben Freundin Sandra, die mir in all den Jahren mit großer Liebe und Vertrauen zur Seite stand. Vielen Dank dafür!
Ein weiteres Danke geht an meine Eltern, die mir diese Ausbildung durch ihre Unterstützung überhaupt erst ermöglichten.

Für die großartige Betreuung bedanke ich mich bei Prof. Dr. Jürgen Ebert und bei Dr. Andreas Winter. Die viele Zeit, die sie sich für mich genommen haben, das genaue Lesen der Arbeit, die Gespräche und Anregungen haben mir sehr geholfen.

Bei Rüdiger Frankenberger bedanke ich mich für das kritische und sehr genaue Korrekturlesen. Der Firma Sysco Netzwerktechnik GmbH&Co. KG danke ich für den zur Verfügung gestellten Arbeitsplatz und zuletzt bedanke ich mich bei allen Menschen, die mich über die ganze Zeit unterstützt haben.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Zielsetzung . . . 1
- 1.3 Vorgehensweise . . . 4

2 Grundlagen **5**
- 2.1 Das Dagstuhl Middle Metamodel . . . 5
- 2.2 GReQL . . . 9

3 Refactoring **11**
- 3.1 Einführung und Geschichte des Refactoring . . . 11
- 3.2 Auswahl geeigneter Refactorings . . . 12
- 3.3 Beschreibung der Auswahl . . . 13
  - 3.3.1 Methode verschieben . . . 15
  - 3.3.2 Attribut verschieben . . . 15
  - 3.3.3 Eigenes Attribut kapseln . . . 16
  - 3.3.4 Attribut kapseln . . . 16

4 Untersuchung ausgewählter Refactorings **17**
- 4.1 Methode verschieben . . . 18
  - A 10: Ober- und Unterklassen finden . . . 22
  - A 20: Ober- und Unterklassen auf die gleiche Methodensignatur überprüfen . . . 22
  - A 30: Benutzte Attribute finden . . . 25
  - A 40: Verschiebbarkeit der benutzten Attribute untersuchen . 26
  - A 50: Benutzte Methoden finden . . . 27
  - A 60: Verschiebbarkeit der benutzten Methoden untersuchen . 28
  - A 70: Methodenname prüfen . . . 29
  - A 80: Methode erstellen . . . 31
  - A 90: Benutzte Attribute der Ausgangsklasse behandeln . . . 32
  - A 100: Benutzte Methoden der Ausgangsklasse behandeln . . 37
  - A 110: Delegierende Methode einführen . . . 39
- 4.2 Attribut verschieben . . . 40
  - A 200: Sichtbarkeit des Feldes prüfen . . . 43
  - A 210: Klassenhierarchie der Zielklasse auf gleiche Feldnamen prüfen . . . 44
  - A 220: Feld in Zielklasse erstellen . . . 45
  - A 230: *Get*- und *set*-Methoden in Zielklasse erstellen . . . 46
  - A 240: Feld für Zielobjekt in der Ausgangsklasse erstellen . . 46
  - A 250: Referenzen auf das Feld in der Ausgangsklasse suchen 49
  - A 260: Zugriffe durch Aufruf der Zugriffs-Methoden ersetzen . 50
- 4.3 Eigenes Attribut kapseln und Attribut kapseln . . . 52

A300: Attribut als 'private' deklarieren 54
4.4 Zusammenfassung 54

5 Erkennung von Bad Smells auf Modellebene 55
BS 1 Duplizierter Code 55
BS 2 Lange Methoden 58
BS 3 Große Klassen 60
BS 4 Lange Parameterlisten 61
BS 5 Divergierende Änderungen 62
BS 6 Schrotkugeln herausoperieren 63
BS 7 Neid 63
BS 8 Datenklumpen 65
BS 9 Neigung zu elementaren Typen 65
BS 10 `Switch`-Befehle 67
BS 11 Parallele Vererbungshierarchien 67
BS 12 Faule Klasse 68
BS 13 Spekulative Allgemeinheit 70
BS 14 Temporäre Felder 70
BS 15 Nachrichtenketten 70
BS 16 Vermittler 71
BS 17 Unangebrachte Intimität 72
BS 18 Alternative Klassen mit verschiedenen Schnittstellen 73
BS 19 Unvollständige Bibliotheksklasse 74
BS 20 Datenklassen 74
BS 21 Ausgeschlagenes Erbe 76
BS 22 Kommentare 76

6 Grenzen des Sprachunabhängigen Refactoring 79
6.1 Kontextbedingungen und zentrale Annahmen zu unterstützten Sprachen 81
6.2 Würdigung des *Dagstuhl Middle Metamodel* 85
6.2.1 Änderungen des DMM 86
6.2.2 Berichtigungen am DMM 87
6.2.3 Fehlende Erweiterungen des DMM 87
6.3 Allgemeine Schwierigkeiten für sprachunabhängiges Refactoring 89
6.3.1 Zusammenfassung 91

7 Abschließende Betrachtungen 92

1 Einleitung

Ziel dieser Diplomarbeit ist eine Studie zur Überprüfung der Umsetzbarkeit von sprachunabhängigem Refactoring auf Basis des *Dagstuhl Middle Metamodel* (DMM).

1.1 Motivation

Wartungen an Softwaresystemen stellen im Softwarelebenszyklus häufig den größten Kostenfaktor des Gesamtsystems dar (vgl. [Bal00] S. 1094 ff.). Der Aufwand für diese Wartungen sollte daher minimiert werden.

Refactoring bezeichnet das *disziplinierte* Restrukturieren von bestehendem Code durch Verändern der internen Struktur ohne das Verhalten nach außen zu beeinflussen (vgl. [Fow05]). Ziel des Refactoring ist die Verbesserung der Lesbarkeit des Quellcodes und dadurch die Erleichterung der Wartungsarbeiten. Um dieses Ziel zu erreichen wird das Refactoring über den gesamten Softwarelebenszyklus durchgeführt.

Refactoringwerkzeuge finden sich heute bereits in vielen Entwicklungsumgebungen wieder. Diese Werkzeuge sind jedoch an bestimmte Programmiersprachen gebunden, wie beispielsweise Java im Falle der Entwicklungsumgebung Eclipse[1]. Dies führt dazu, dass im Falle neuer Refactorings alle diese Werkzeuge angepasst werden müssen, um das neue Refactoring durchführen zu können.
Aus diesem Grund erscheint es sinnvoll sprachunabhängige Lösungen zu entwickeln, die es ermöglichen, die bestehenden und neuen Refactorings sprachübergreifend durchzuführen.

1.2 Zielsetzung

Das Ziel der Sprachunabhängigkeit ist nur zu erreichen, wenn die Refactorings nicht, wie sonst üblich, auf der Quellcodeebene, sondern vielmehr auf einem höheren Abstraktionsniveau mit Hilfe von Modellen durchgeführt werden.

Kernpunkte dieser Arbeit sind die Zerlegung von ausgewählten Refactoring in kleinere Aktivitäten, die Formulierung von Anforderungen an ein Metamodell zur Durchführung von Refactorings auf Modellbasis und die anschließende Überprüfung des *Dagstuhl Middle Metamodels* auf die Erfüllung dieser Anforderungen.
Da Software üblicherweise in Form von Quellcode vorliegt, muss dieser zunächst in ein geeignetes Modell überführt werden. Ein Werkzeug für sprach-

[1] http://www.eclipse.org

unabhängiges Refactoring benötigt neben der Komponente, die für das Refactoring verantwortlich ist eine weitere, welche in der Lage ist, den Quellcode in ein Modell zu überführen. Auf diesem wird das Refactoring dann sprachübergreifend durchgeführt. Diese Komponente wird *Parser* genannt und steht dem sogenannten *Unparser* gegenüber, welcher das geänderte Modell wieder in die ursprüngliche Programmiersprache zurück überführt.
Eine weitere benötigte Komponente ist dafür zuständig, im Programmcode Stellen zu finden, auf welche die einzelnen Refactorings angewendet werden sollten. Eine solche Stelle könnte zum Beispiel eine Methode sein, die in zwei verschiedenen Klassen auftaucht und somit doppelt vorhanden, also redundant ist. Solche Stellen werden im Allgemeinen als *Bad Smells* (vgl. [Fow05]) bezeichnet, weshalb die Komponente im Folgenden auch als *Bad-Smell-Finder* bezeichnet wird. Aus den gleichen Gründen wie die Refactoring-Komponente sollte auch der Bad-Smell-Finder nicht an eine bestimmte Programmiersprachen gebunden sein, sondern vielmehr in der Lage sein, die Bad Smells auf Modellebene zu erkennnen.

Die Architektur und die Arbeitsabläufe eines solchen Werkzeugs sind in Abbildung 1 dargestellt, wobei das Hauptaugenmerk dieser Arbeit auf der Suche nach geeigneten Metamodellen für den Bad-Smell-Finder und das Refactoring-Tool liegt.

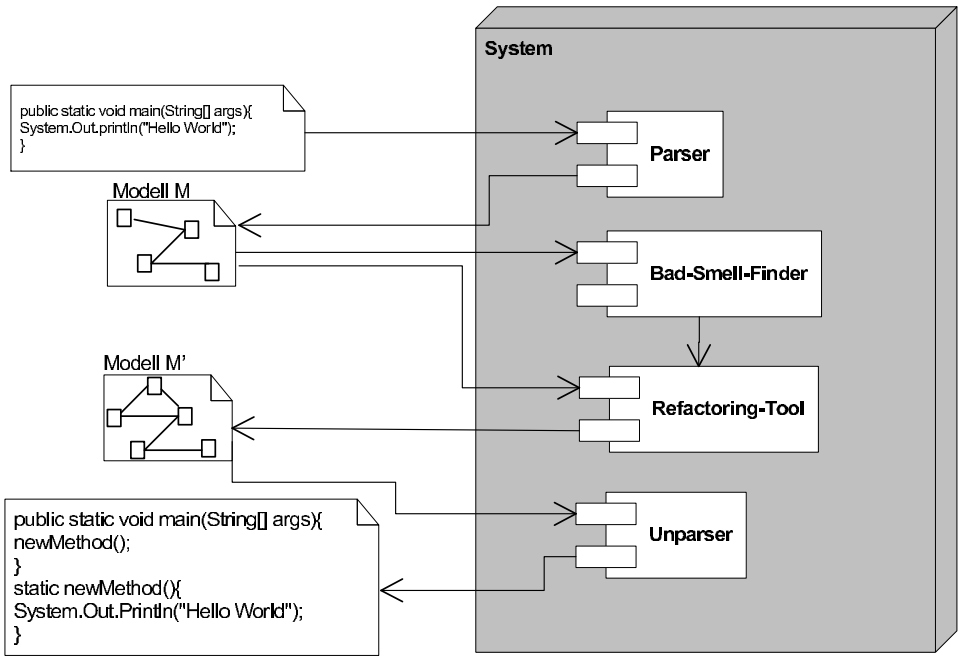

Abbildung 1: Architektur und Ablaufstruktur eines sprachunabhängigen Refactoringwerkzeugs

Da sich die Refactorings auf zum Teil völlig unterschiedliche Aspekte der Programmierung beziehen, als Beispiel sei hier das einfache Extrahieren eines Codeabschnitts in eine eigene Methode gegenüber der Manipulation komplexer Vererbungshierarchien genannt, werden zunächst kleinere Metamodelle für jedes einzelne Refactoring entwickelt und diese dann nach Möglichkeit zu einem Gesamtmodell integriert. Diese Integration ist sinnvoll, da die Anzahl der einzelnen Metamodelle direkten Einfluss auf die Anzahl der zu entwickelnden Parser und Unparser hat. Denn je weniger Metamodelle benötigt werden, um verschiedene Refactorings durchzuführen beziehungsweise die Bad Smells zu erkennen, desto weniger (Un-)Parser müssen auch entwickelt werden, um eine Software in die verschiedenen Metamodelle zu transformieren.

Für das Gesamtsystem bedeutet dies, dass es aus mehreren Blöcken gleich dem in Abbildung 1 besteht, und zwar im ungünstigsten Fall für jedes Refactoring ein Block mit je einem Parser, einem Bad-Smell-Finder, einem Refactoring-Service und einem Unparser.
Um dies zu vermeiden, müssen geeignete Metamodelle gefunden oder erarbeitet werden. Die Arbeit von Rainer Koschke in [Kos98] stellt dabei eine mögliche Modellierungsmöglichkeit dar. Er entwickelt dort eine sogenannte *Intermediate Representation*, also eine Code-Repräsentation auf Middle-Model-Niveau. Mit dieser kann durch verschiedene Sichten auf ein Programm, ein für ein bestimmtes Refactoring notwendiges Abstraktionsniveau abgebildet werden. Eine weitere Möglichkeit, welche auch für diese Arbeit verwendet wird, bietet das Dagstuhl Middle Metamodel (vgl. Kapitel 2.1 auf Seite 5), welches in der Lage ist, Informationen zu vielen gängigen Programmiersprachen wie beispielsweise C, C++ und Java zu repräsentieren (vgl. [LTP04]).

Eine verwandte Arbeit stellt die Diplomarbeit von Bodo Hinterwäller [Hin05] dar, in welcher er die metamodellbasierte Durchführung des Refactorings *Methode extrahieren* auf der Basis von Abstrakten Syntaxbäumen, sogenannten ASTs (*Abstract Syntax Tree*), beschreibt.
Die abstrakten Syntaxbäume sind als Modell für ein sprachunabhängiges Refactoring jedoch ungeeignet, da sie eine Instanz der Grammatik der verwendeten Programmiersprache, im Falle der Arbeit von Bodo Hinterwäller Java, darstellen. In dieser Diplomarbeit wird sich der Fragestellung aus einer anderen Richtung genähert, nämlich zunächst für je ein Refactoring ein geeignetes Metamodell zu finden, so dass dieses Refactoring tatsächlich sprachunabhängig durchgeführt werden kann. Diese einzelnen Modelle werden nach Möglichkeit integriert, um so ein möglichst komplettes Gesamtmodell zu erhalten. Als Basis für die Entwicklung solcher Metamodelle dient, wie bereits erwähnt, das *Dagstuhl Middle Metamodel*. Dies gilt dann für jede Programmiersprache, für die ein Parser entwickelt werden kann, der den Quellcode der Sprache in eine Instanz des jeweils benötigten Metamodells (also speziell

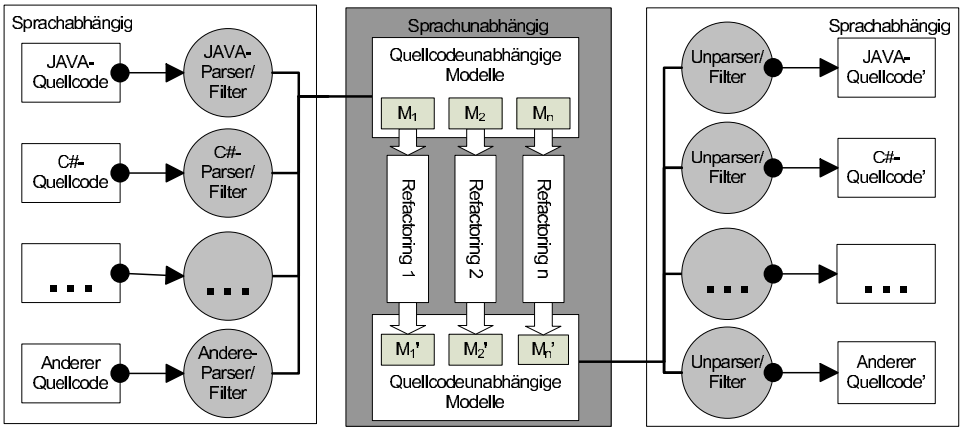

Abbildung 2: Visuelle Darstellung der Zielsetzung

für das eine Refactoring) überführen kann. Abbildung 2 verdeutlicht diese Zielsetzung noch einmal, wobei es aus den bereits zuvor genannten Gründen auch wünschenswert wäre, dass sich ein Metamodell für mehrere Refactorings eignet.
Im Folgenden wird die Gliederung der Arbeit näher erläutert.

1.3 Vorgehensweise

Auf diese Einleitung folgend werden zunächst in Kapitel 2 kurz die benötigten Grundlagen erläutert. Dies sind in Kapitel 2.1 das *Dagstuhl Middle Metamodel*, welches als Basis zur Entwicklung der Metamodelle für die einzelnen Refactoring und die Bad Semlls dient und in Kapitel 2.2 die Anfrage-Sprache GReQL.

Eine Einführung in das Refactoring selbst und auf dessen Besonderheiten im Rahmen dieser Arbeit bietet das Kapitel 3. Dabei folgt im Unterkapitel 3.2 eine Auswahl konkreter Refactorings, auf deren Basis der Rest der Diplomarbeit aufgebaut wird und in Kapitel 3.3 eine detailierte Beschreibung dieser Auswahl.

In Kapitel 4 werden die ausgewählten Refactorings in einzelne Aktivitäten zerlegt und basierend auf dieser Zerlegung die benötigten Strukturen der Metamodelle entwickelt.

Kapitel 5 geht auf die Bad Smells ein, wobei auch hier, sofern möglich, für jeden Bad Smell ein Metamodell entwickelt wird, um diesen auf Modellbasis erkennen zu können.

Kapitel 6 beleuchtet die Grenzen des sprachunabhängigen Refactorings.

Dabei werden im Kapitel 6.1 zentrale Sprachannahmen und Kontextbedingungen genannt, für die vor der Entwicklung eines konkreten Werkzeugs zu klären ist, ob sie unterstützt werden oder nicht. Kapitel 6.2 befasst sich noch einmal abschließend mit dem DMM. Speziell wird zusammenfassend auf Änderungen (6.2.1) und Berichtigungen (6.2.2) sowie fehlende Aspekte des DMM (6.2.3) eingegangen. In Kapitel 6.3 werden generelle Schwierigkeiten in Zusammenhang mit dem sprachunabhängigen Refactoring erläutert, bevor die Arbeit mit den abschließenden Betrachtungen in Kapitel 7 endet.

2 Grundlagen

In diesem Kapitel werden die in dieser Arbeit benötigten Grundlagen vorgestellt. Dabei handelt es sich um das *Dagstuhl Middle Metamodel* und die Anfragesprache GReQL.

2.1 Das Dagstuhl Middle Metamodel

Das Dagstuhl Middle Metamodel (DMM) stellt ein Schema zur Modellierung der statischen Struktur von Quellcode dar. Dabei soll mit diesem Ansatz zunächst ein mittleres Abstraktionsniveau erreicht werden. Das heißt, die Instanzen enthalten weder die komplette Syntax des Quellcodes noch lassen sich damit komplexere Architekturelemente, wie beispielsweise Client-Server-Architekturen, modellieren.
Das DMM wurde aus Arbeiten von verschiedenen Universitäten entwickelt (vgl. [TDD00], [Tic01], [Let01], [CEK+00]), welche auf dem *Dagstuhl Seminar on Interoperability of Reengineering Tools*[2] zum DMM[3] zusammengebracht wurden (vgl. [LTP04]).

Das DMM ist mit Hilfe von vier UML Klassendiagrammen beschrieben. Drei dieser Klassendiagramme beschreiben die Hierarchien der drei Top-Klassen *ModelObject*, *SourceObject* und *Relationship*. Das vierte der UML-Diagramme stellt die Zusammenhänge zwischen den drei Top-Klassen dar. Diese Zusammenhänge sind in Abbildung 3 dargestellt.

Beim DMM handelt es sich wie bereits erwähnt um ein so genanntes Middle Model. Das bedeutet, es wird nicht der komplette Sourcecode im Modell gespeichert, sondern im Modell werden einzelne Elemente des Sourcecodes, wie beispielsweise Klassen oder Methoden, abgebildet. Die Klassenhierarchie der *SourceObjects*, dargestellt in Abbildung 4, dienen dazu, bestimmte Teile des Sourcecode abzubilden. Die wichtigsten Teilelemente dieser Hierarchie bilden die Klassen *SourcePart* und *SourceFile*. Mit Hilfe dieser beiden Klas-

[2] vgl. http://www.dagstuhl.de/01041 - Stand: 14.02.2006
[3] damals wurde das DMM noch als Dagstuhl Middle Model bezeichnet

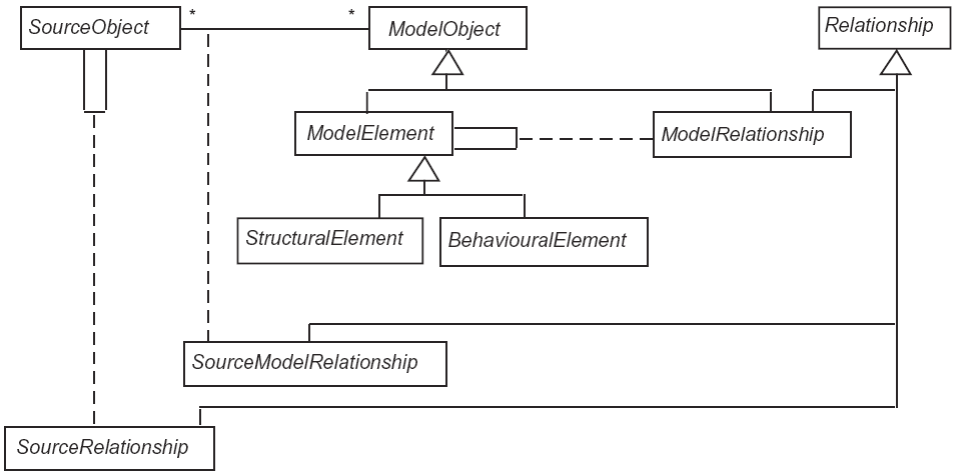

Abbildung 3: Top-Level-Klassen des DMM. [LTP04]

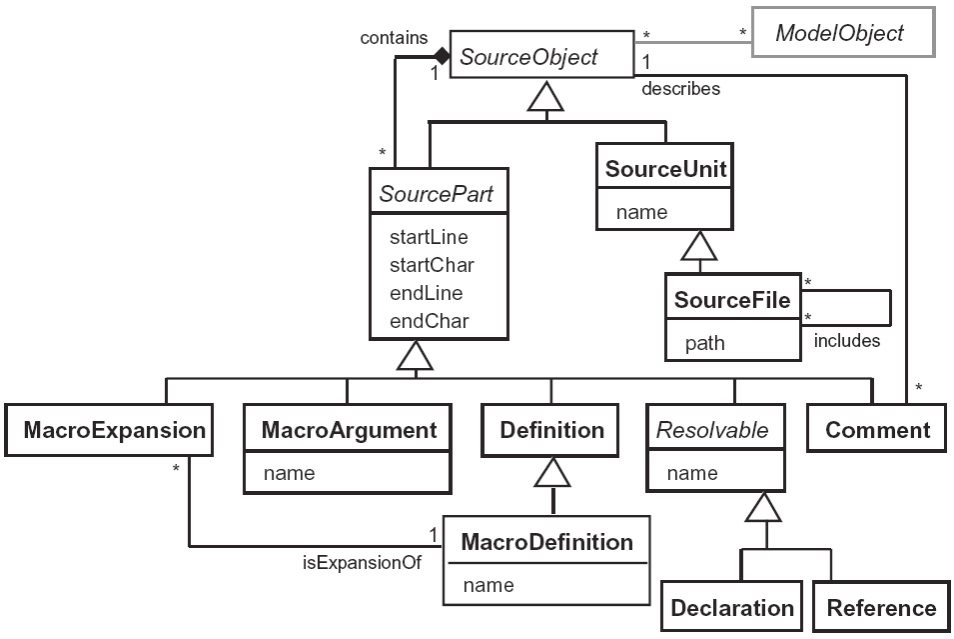

Abbildung 4: Source Object Hierarchie des DMM. [LTP04]

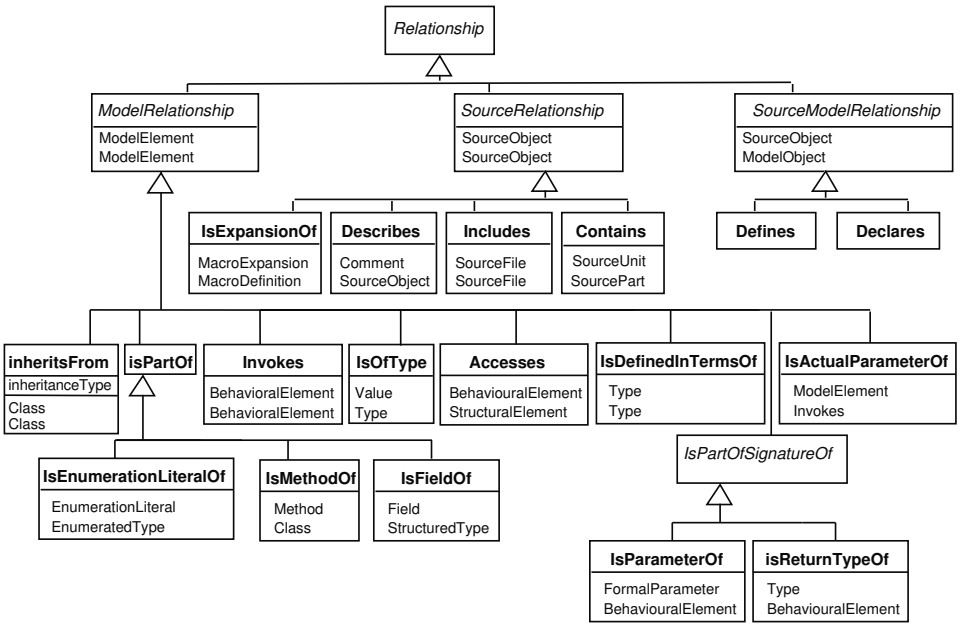

Abbildung 5: Relationship Hierarchie des DMM. [LTP04]

sen und den entsprechenden Relationen zu den *ModelObjects* lassen sich für alle *ModelObjects* Informationen über deren Quellcode modellieren.

Die Relationen des DMM sind in Abbildung 5 dargestellt. Sie verbinden zwei Modellelemente (*ModelRelationship*), zwei Objekte des Quellcodes (*SourceRelationship*) oder ein Modellelement mit einem Quellcodeelement (*SourceModelRelationship*). Die verschiedenen Verwendungszwecke der Relationen sind bereits durch die gewählten Namen zu erkennen, eine detailliertere Beschreibung findet sich in [LTP04], wobei die für diese Arbeit benötigten Elemente bei der ersten Verwendung erklärt sind.

Die dritte, in Abbildung 6 dargestellte Klassenhierarchie, sind die ModelObjects. In dieser wird zunächst zwischen verhaltensorientierten Elementen (*BehaviouralElements*, z.B. Methoden) und statischen Elementen (StructuralElements, z.B. Variablen und Klassen) unterschieden. Mit dieser Hierarchie ist es möglich, die logische Struktur des Quellcodes abzubilden, weshalb auch hier die meisten der in der Relationship-Hierarchie beschriebenen Relationen zu finden sind.

Das DMM wurde für diese Arbeit aus zwei Gründen gewählt. Die deutliche Trennung zwischen Quellcode und Modellelementen ist für das modellbasierte Refactoring gut geeignet, da es damit problemlos möglich ist, ein Stück Code mehreren Modellelementen zuzuordnen. Außerdem ist es dadurch wei-

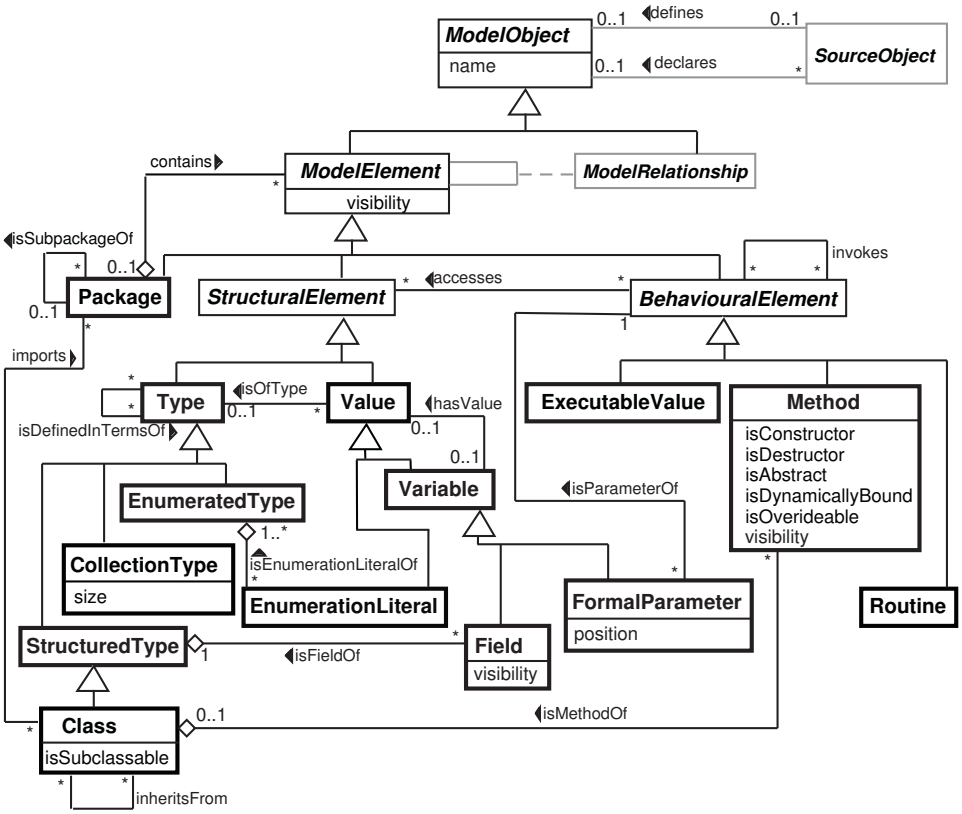

Abbildung 6: Model Object Hierarchie des DMM. [LTP04]

terhin möglich, trotz Abstraktion von einer konkreten Programmiersprache auf den Quellcode zuzugreifen.
Ein weiterer Grund ist die Erweiterbarkeit des DMM. Lethbridge spricht in [LTP04] von zwei Möglichkeiten der Erweiterung. Auf der einen Seite nennt er *konsistente* Erweiterungen, welche durch die Erweiterung des bestehenden DMM um weitere Klassen und Subklassen entstehen. Diese Erweiterungen heißen dann einfach *Extensions* (vgl. [LTP04]). Auf der anderen Seite ist es auch möglich, *inkonsistente* Erweiterungen des DMM durchzuführen. Diese entstehen dann, wenn bestehende Klassen des DMM geändert werden müssen, da sie beispielsweise nicht ausdrucksstark genug sind. Es handelt sich daher vielmehr um eine Änderung des DMM als eine Erweiterung, weshalb ein so geändertes DMM dann nicht mehr *Extension*, sondern *Variant* heißt.

In dieser Arbeit werden später beide Konzepte der Erweiterung genutzt, was somit insgesamt zu einer Variante des DMM führt. Allgemein ist es jedoch vorzuziehen, wenn man eine konsistente Erweiterung des DMM benutzt, da solche auch von bereits bestehenden Tools verarbeitet werden kann, was bei der inkonsistenten Variante nicht unbedingt gegeben sein muss.

2.2 GReQL

GReQL stellt eine allgemeine Anfragesprache zur Auswertung von TGraphen dar und wurde im Rahmen des GUPRO-Projekts (vgl. [EKW97]) an der Universität Koblenz-Landau entwickelt. In dieser Arbeit wird sie benutzt, um Anfragen an die in Graphen abgelegten Modellinstanzen der in der vorliegenden Arbeit entwickelten Variante des *Dagstuhl Middle Metamodells* zu stellen.
Detailliertere Information zu GReQL finden sich in [KK01], an dieser Stelle erfolgt nur eine kurze Einführung mit den für diese Arbeit relevanten Teilen.

Ein Anfrage in GReQL besteht im Wesentlichen aus folgenden drei Teilen: FROM, WITH und REPORT. Eine solche Anfrage ist in Abbildung 7 gezeigt. Hier werden aus allen Knoten vom Typ Method diejenigen geliefert, die die Bedingung `method.name = 'test'` erfüllen.

```
FROM   method : V{Method}
WITH method.name = 'test'
REPORT method END
```

Abbildung 7: Eine einfache GReQL-Anfrage

Neben der Gleichheit bietet GReQL auch noch andere Vergleichsoperationen wie $<$, $>$ und $<>$. Im WITH-Teil einer Anfrage lassen sich die logischen Ausdrücke über die Operatoren AND, OR und NOT verknüpfen.

Weiterhin besteht in GReQL die Möglichkeit, parametriesierbare Anfragen zu stellen. Diese benutzen dann die USING-Klausel wie in Abbildung 8, wo die Namensgleichheit nicht mehr auf 'test', sondern auf den zu übergebenden Parameter `myname` getestet wird.

```
USING myname
FROM   method : V{Method}
WITH  method.name = myname
REPORT method END
```

Abbildung 8: GReQL-Anfrage mit Parameter

Mit das wichtigste Sprachkonstrukt von GReQL stellen die Pfadausdrücke dar. Mit diesen lassen sich einerseits in der Anfrage Beziehungen zwischen Objekten ausdrücken, andererseits kann man damit Mengen von Objekten berechnen, die über einen anzugebenden Pfad von einem Objekt aus erreichbar sind. Ein Beispiel für eine solche Menge ist in Abbildung 9 zu sehen, wo für einen Klassenknoten die Mengen aller zugehörigen Methoden berechnet werden.

```
FROM   c : V{Class}
REPORT c (-->{isMethodOf}) END
```

Abbildung 9: GReQL-Anfrage mit Parameter

Viele Funktionen, die bei Anfragen häufig verwendet werden, finden sich in der GReQL-Bibliothek. Darin sind sowohl arithmetische Funktion wie +,-,*,/ enthalten als auch komplexere graphspezifische Funktionen wie beispielsweise *indegree*, mit welcher die Anzahl der eingehenden Kanten eines Knotens berechnet werden kann.
Im Folgenden werden diejenigen Funktionen aus der GReQL-Bibliothek erläutert, welche für diese Arbeit benutzt werden. Eine detailliertere Beschreibung der GReQL-Bibliothek ist in [KK01] zu finden.

indegreetype(n) Berechnet die Anzahl der eingehenden Kanten vom Typ `type` in den Knoten `n`.

outdegreetype(n) Berechnet die Anzahl der ausgehenden Kanten vom Typ `type` vom Knoten `n`.

cnt(FROM c:VClass REPORT c.name) Berechnet die Anzahl der Elemente einer Menge (bzw. Liste, Bag).

isIN(c, neighbours(x)) Überprüft, ob der Wert in `c` in der Menge der Nachbarn des Knotens `x` enthalten ist.

3 Refactoring

In den nächsten Unterkapiteln wird zunächst eine kurze Einführung in das Thema Refactoring gegeben (3.1), danach erfolgt die Auswahl einer Anzahl von Refactorings, welche für den Rest dieser Diplomarbeit als repräsentative Basis dienen (3.2). In Kapitel 3.3 wird schließlich darauf eingegangen, welchem Zweck die ausgewählten Refactorings dienen.

3.1 Einführung und Geschichte des Refactoring

Ein genauer Zeitpunkt für eine Geburtsstunde des Refactoring lässt sich heute nicht mehr bestimmen, da solche Programmtransformationen schon seit langem in der Informatik üblich sind. Die wahrscheinlich ersten, die den Begriff des Refactoring prägten, waren wohl Kent Beck und Ward Cunningham, die seit den 80er-Jahren an der Entwicklung eines Softwareentwicklungsprozesses gearbeitet haben, der heute unter dem Namen *Extreme Programming* (kurz XP) bekannt ist (vgl. [Bec00]). Innerhalb des XP kommt dem Refactoring eine große Bedeutung zu, da sich nur dadurch ein gutes Design erzielen lässt, denn beim Extreme Programming wird auf jeglichen Designentwurf verzichtet.
Martin Fowler differenziert in [Fow05] zwischen zwei verschiedenen Bedeutungen des Wortes Refactoring, nämlich einmal dem Substantiv Refaktorisierung und dem Verb refaktorisieren:

> **Refaktorisierung** (Substantiv): Eine Änderung an der internen Struktur einer Software, um sie leichter verständlich zu machen und einfacher zu verändern, ohne ihr beobachtbares Verhalten zu ändern.
>
> **Refaktorisieren** (Verb): Eine Software umstrukturieren, ohne ihr beobachtbares Verhalten zu ändern, indem man eine Reihe von Refaktorisierungen anwendet.

Beobachtbares Verhalten meint in Bezug auf Software und Refactoring, dass eine Software vor und nach einem Refactoring bei gleicher Eingabe auch die gleiche Ausgabe liefert.

Dass Programmierer ihren bereits geschriebenen Code nachträglich noch verändern ist unumstritten. Neu am Refactoring ist dabei, dass diese Änderungen explizit nicht das Verhalten oder die Funktionalität der Software ändern, sondern lediglich der Verbesserung der Lesbarkeit beziehungsweise der späteren Änderung oder Erweiterung der Software dienen sollen. Kent Beck spricht in diesem Zusammenhang von zwei Hüten, die der Programmierer während der Entwicklung trägt. Trägt man den einen Hut, darf man nur Funktionalität hinzufügen, was der konventionellen Programmiertätigkeit entspricht

und das System zur Fertigstellung führt. Trägt man den anderen Hut, darf man nur refaktorisieren, wobei dies nicht für die Anpassungen der Tests gilt, die aufgrund des Refactorings durchgeführt werden müssen und ansonsten fehlschlagen würden.

Gründe für Refactorings gibt es viele. Zum einen gibt es die konkreten *Bad Smells* (vgl. Kapitel 5), die auf schlecht entwickelten oder schlecht erweiterten Code schließen lassen. Zum anderen dient das Refactoring wie beim Extreme Programming der Verbesserung des Designs, da man, nach Ansicht der Befürworter von XP, vor dem Start eines Softwareprojekts das Design nicht korrekt entwerfen kann. Weiterhin trägt das Refactoring zur besseren Verständlichkeit und Lesbarkeit des Quellcodes bei, so dass spätere Erweiterungen und Wartungen daran einfacher und schneller durchgeführt werden können. Ein anderer nicht zu unterschätzender Punkt liegt im besseren Verständnis der Programmierer für ihren Code, da sie ihn durch das Refactoring öfters lesen müssen, weshalb auch Fehler schneller entdeckt und korrigiert werden können.

3.2 Auswahl geeigneter Refactorings

Aufgrund der großen Anzahl an Refactorings im Katalog[4] von Martin Fowler werden zunächst einige repräsentative ausgewählt. Anhand dieser werden dann Anforderungen an Metamodelle definiert und das DMM auf die Erfüllung dieser Anforderungen überprüft, um das jeweilige Refactoring modellbasiert durchführen zu können. Die Auswahl repräsentativer Refactorings sollte allerdings nicht willkürlich geschehen, da zwischen den Refactorings inhaltliche Zusammenhänge bezüglich der durchzuführenden Schritte bestehen. Das heißt, bestimmte Schritte könnten in mehreren Refactorings nötig sein, so dass diese, und dadurch auch das jeweilige Metamodell, wiederverwendbar sind.
Ein weiterer zu beachtender Punkt ist die teilweise gegenseitige Abhängigkeit der Refactorings voneinander. Es sollte also bei der Auswahl von Refactorings eine Gruppe gefunden werden, die nur innere Abhängigkeiten besitzt und die nicht weitere Refactorings außerhalb der Auswahl benötigt.
Eine erste Klassifizierung der einzelnen Refactorings in verschiedene Gruppen wurde von Martin Fowler selbst vorgenommen (vgl. [Fow05]) und ist in Abbildung 10 dargestellt. Die oberen sechs Gruppen wurden aufgrund der Quellcode-Objekte (Methoden, Objekte, Daten, Bedingte Ausdrücke) und deren Beziehungen zueinander gebildet (Methodenaufrufe, Generalisierung, Spezialisierung). Die Gruppe 'Enterprise Java' bündelt spezielle Refactorings für das Java-Umfeld und in 'Weiteres' finden sich solche Refactorings, die sich

[4] http://www.refactoring.com/catalog

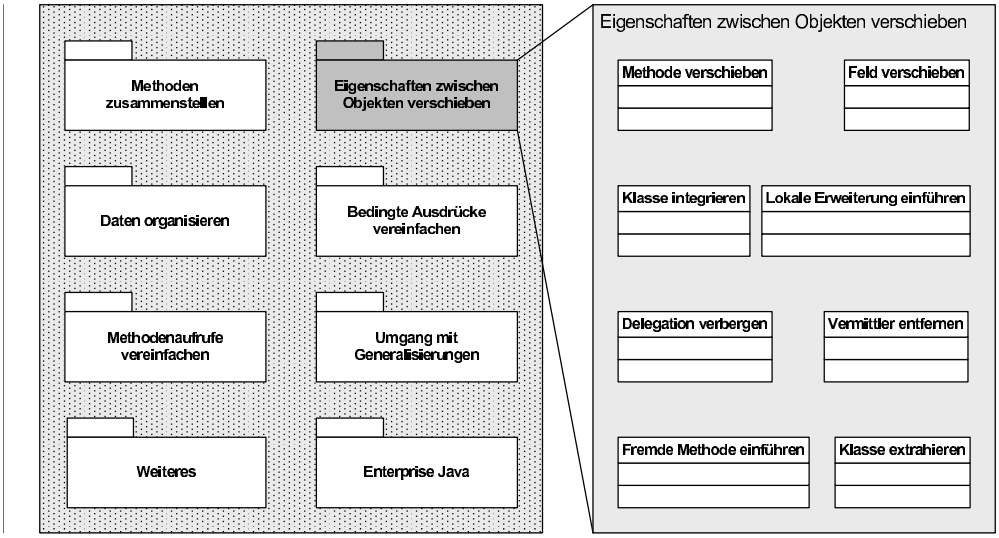

Abbildung 10: Einteilung der Refactorings in acht Gruppen

in keine der vorhandenen Gruppen einteilen lassen. Aufgrund der in dieser Arbeit nicht gewünschten Sprachabhängigkeit im Falle von 'Enterprise Java' und der geringen inneren Kopplung der Gruppe 'Weiteres' fallen diese beiden Gruppen bereits für die Auswahl weg.
Ein weiteres Kriterium bei der Auswahl der Refactorings sollte die Häufigkeit der Anwendung sein. Martin Fowler führt in [Fow05] eine Liste von Bad Smells auf, und welche Refactorings dagegen anzuwenden sind. Aus dieser Liste lässt sich erkennen, das die Refactorings *Methode verschieben* und *Attribut verschieben* sehr häufig angewendet werden. Aus diesem Grund werden diese beiden Refactorings mit den zusätzlich durch Abhängigkeiten benötigten als Grundlage für diese Arbeit ausgewählt. Die Abhängigkeiten der Refactoringgruppe 'Eigenschaften zwischen Objekten verschieben' und speziell die Abhängigkeiten der beiden ausgewählten Refactoring zu *Attribut kapseln* und *Eigenes Attribut kapseln* sind in Form von *uses*-Beziehungen in Abbildung 11 dargestellt. Für diese Arbeit werden also im Weiteren die Refactorings *Methode verschieben*, *Attribut verschieben*, und *Eigenes Attribut kapseln* relevant sein. Diese ausgewählten Refactorings werden im Folgenden kurz vorgestellt.

3.3 Beschreibung der Auswahl

Die nächsten Unterkapitel beschreiben die ausgewählten Refactorings mit dem Ziel, ein Grundverständnis darüber herzustellen, WAS die einzelnen Refactoring tun. Antworten auf die Frage nach dem WIE folgen erst danach im Kapitel 4, wo die Refactorings in kleinere Aktivitäten zerlegt werden. Die

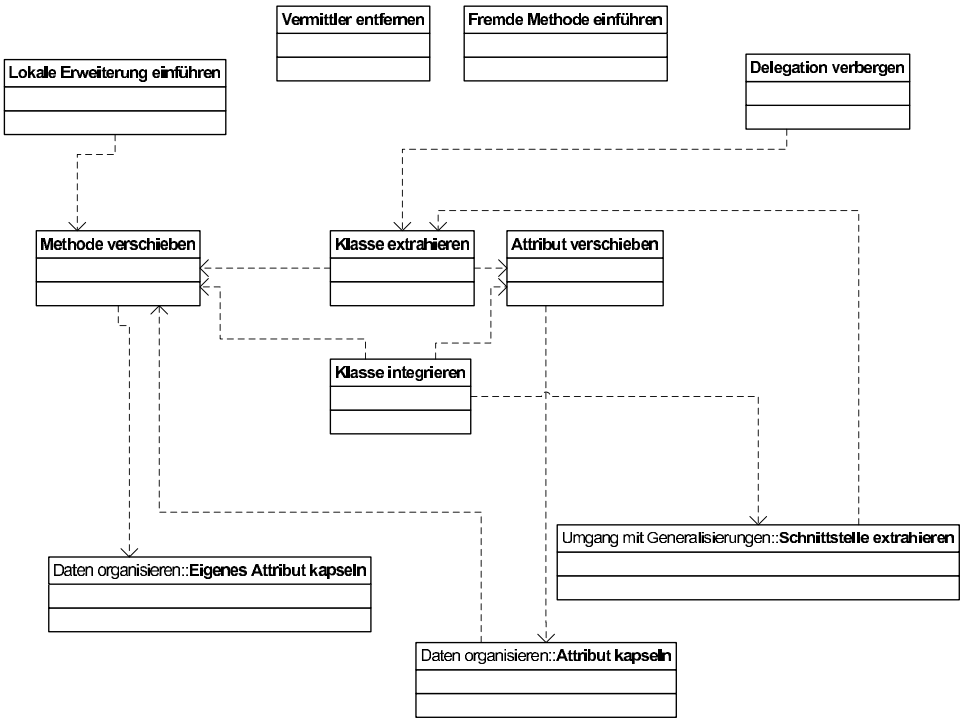

Abbildung 11: Abhängigkeiten (*uses*-Beziehungen) im Paket Eigenschaften zwischen Objekten verschieben

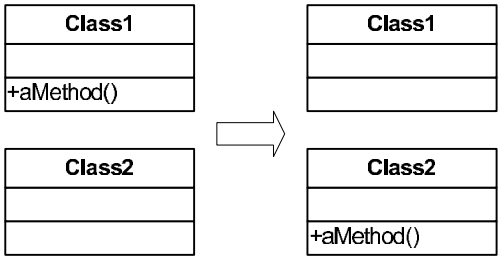

Abbildung 12: Methode verschieben

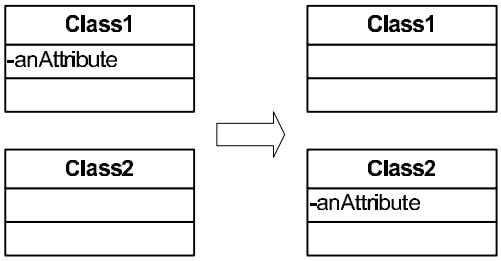

Abbildung 13: Attribut verschieben

Abbildungen und Beschreibungen fassen hier die wichtigsten Elemente der Ausführungen von Martin Fowler in [Fow05] zusammen. Um das Nachschlagen in diesem Werk zu erleichtern wurden die Beispiele daraus unverändert übernommen.

3.3.1 Methode verschieben

Während der Implementation und auch in der Wartungsphase ist eines der am häufigsten angewendeten Refactorings *Methode verschieben*. Dabei wird eine Methode aus einer Klasse in eine andere Klasse verschoben (vgl. Abbildung 12). Dies kann aus vielen Gründen sinnvoll oder notwendig sein. Ein Beispiel wäre das Verschieben von Methoden, weil man eine große Klasse in zwei kleinere aufteilen möchte und dazu die Methoden in die neue kleinere Klasse schieben muss. Eine andere sinnvolle Anwendung besteht, wenn eine Klasse die Methoden einer anderen Klasse sehr häufig benutzt und man diese enge Kopplung durch das Verschieben der Methode in die benutzende Klasse entfernen möchte.

3.3.2 Attribut verschieben

Aus den gleichen Gründen wie auch Methode verschieben kann das Refactoring *Attribut verschieben* angewendet werden. Dabei wird ein Attribut einer Klasse in eine andere Klasse verschoben (vgl. Abbildung 13). Häufig werden Methode verschieben und Feld verschieben gemeinsam angewandt, wenn zum

```
private int low, high;
boolean includes(int arg) {
       return arg >= low && arg <= high;
}
```

```
private int low, high;
boolean includes(int arg) {
       return arg >= getLow() && arg <= getHigh();
}

Int getLow() {return low;}
Int getHigh() {return high;}
```

Abbildung 14: Eigenes Attribut Kapseln

Beispiel eine Methode verschoben werden soll, die bestimmte Attribute einer Klasse exklusiv benutzt, können oder sollten diese Attribute gleich mit verschoben werden.

3.3.3 Eigenes Attribut kapseln

Wie in früheren Kapiteln bereits erwähnt, stellt die Möglichkeit der Kapselung einen großen Vorteil der Objektorientierung dar. Mit *Eigenes Attribut kapseln* wird der Zugriff auf ein privates Attribut einer Klasse durch Zugriffsmethoden gekapselt. Dazu wird für ein Attribut eine get-Methode für lesende Zugriffe erstellt und eine set-Methode für schreibende Zugriffe. Die Methoden der Klasse müssen dann so umgestellt werden, dass sie künftig nur noch über die entsprechenden Zugriffsmethoden auf die Attribute der Klasse zugreifen. In Abbildung 14 ist ein Codebeispiel angegeben.

3.3.4 Attribut kapseln

Handelt es sich um ein öffentliches Attribut einer Klasse, wird *Attribut kapseln* angewandt, um den Zugriff auf das Attribut zu steuern. Das Vorgehen ist analog zu dem in Kapitel 3.3.3, es müssen jedoch auch alle Attributzugriffe außerhalb der Klasse umgestellt werden, da es sich um ein öffentliches Attribut handelt. Abbildung 15 zeigt ein Codebeispiel.

Zusammenfassung

Dieses Kapitel sollte eine kurze Einführung in das Refactoring geben. Es wurde eine Anzahl von Refactorings zur weiteren Untersuchung ausgewählt und diese mit dem Ziel beschrieben, einen Überblick darüber zu geben, *was*

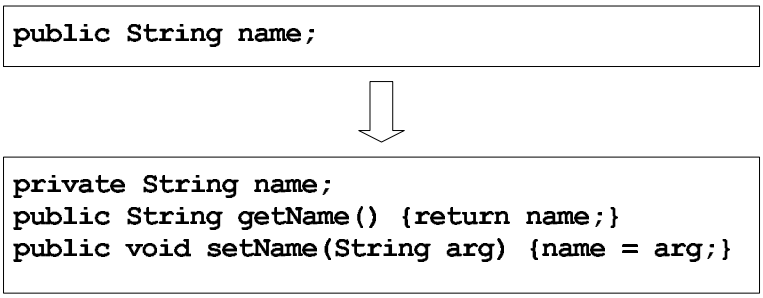

Abbildung 15: Attribut Kapseln

die Refactorings im einzelnen tun. Die Auswahl erfolgte dabei anhand der Häufigkeit der Benutzung der Refactorings und der gegenseitigen Abhängigkeit, so das zur weiteren Bearbeitung die Refactorings *Methode verschieben*, *Attribut verschieben*, *Eigenes Attribut kapseln* und *attribut kapseln* benutzt werden.
Im nächsten Kapitel erfolgt nun eine Aufteilung der Refactorings in ihre einzelnen Teilschritte, im folgenden Aktivitäten genannt, wodurch deutlich zu erkennen sein wird, *wie* die Refactorings arbeiten.

4 Untersuchung ausgewählter Refactorings

In diesem Kapitel werden die ausgewählten Refactorings genauer untersucht. Ziele dabei:

1. Die Refactorings zur Entwicklung der Metamodelle in kleinere Aktivitäten aufzusplitten.

2. Für andere Refactorings wiederverwendbare Aktivitäten zu identifizieren.

Zur Durchführung der Refactorings auf Modellebene wird eine geeignete Datenstruktur in Form von Metamodellen benötigt, welche von den für das Refactoring nicht relevanten Aspekten abstrahiert.

Um geeignete Metamodelle zur Durchführung der Refactorings zu finden, ist es zunächst notwendig, die Refactorings in kleinere Aktivitäten aufzusplitten. Anhand dieser Aktivitäten wird in den folgenden Unterkapiteln für jede einzelne ein dazu passendes Metamodell entwickelt. Diese einzelnen Metamodelle werden am Ende zu einem Gesamtmodell für alle hier untersuchten Refactorings integriert.
Die Entwicklung der Metamodelle orientiert sich an dem in Kapitel 2.1 vorgestellten Dagstuhl Middle Metamodell (DMM), so dass für jede Teilaktivität

untersucht wird, ob das DMM die dafür notwendigen Elemente bereits enthält oder entsprechend erweitert werden kann.

Bei der Aufteilung der einzelnen Refactorings in kleinere Aktivitäten ist außerdem darauf zu achten, ob bestimmte Aktivitäten in mehreren Refactorings angewendet werden müssen (dies gilt eventuell auch für sehr ähnliche Aktivitäten, wenn sie sich entsprechend anpassen lassen). Sind solche Aktivitäten vorhanden, können diese Aktivitäten und die dazu jeweils passenden Metamodelle in anderen Refactorings wiederverwendet werden.

Die Nummerierung der Aktivitäten erfolgt fortlaufend in Zehnerschritten, wobei jedes Refactoring mit einem neuen Hunderterblock begonnen wird. Dadurch ist es möglich, einzelne Aktivitäten nach Bedarf weiter aufzusplitten und die einzelnen Elemente dennoch in die bestehende Nummerierung einzugliedern.

Als Grundlage für die Diskussion der einzelnen Aspekte der Refactorings wird das in Abbildung 16 stehende Codebeispiel (Sprache C#) benutzt und an den entsprechenden Stellen darauf verwiesen.

4.1 Methode verschieben

In diesem Abschnitt wird das Refactoring *Methode verschieben* in einzelne Aktivitäten zerlegt. Diese werden dann näher beschrieben und in Bezug zum *Dagstuhl Middle Metamodel (DMM)* gesetzt.

Als Diskussionsgrundlage soll beispielsweise im Codebespiel in Abbildung 16 die Methode `calculateDiscount` aus der Klasse `Customer` in die Klasse `Customertype` verschoben werden. Dies ist beispielsweise dann sinnvoll, wenn der Rabatt nicht mehr für jeden Kunden selbst berechnet werden muss. Es könnte also sein, dass die Berechnung des Rabattes in Zukunft für jeden Kundentyp unterschiedlich ist und die Methode `calculateDiscount` somit in der Klasse `Customertype` besser aufgehoben wäre.

Die Methoden einer Klassen können nur selten isoliert betrachtet werden und sind meist Teil eines größeren Systems von Elementen wie Klassen, Attributen und anderen Methoden. Aus diesem Grund ist vor dem Verschieben einer Methode zunächst der Kontext zu betrachten, in dem die Methode steht und auf welche Elemente das Verschieben Auswirkungen haben könnte. In Abbildung 16 wird beispielsweise die Methode `calculateDiscount` aus der Methode `main` heraus aufgerufen. Damit dieser Aufruf auch nach dem Verschieben von `calculateDiscount` in die Klasse `Customertype` funktioniert, müsste der Methodenaufruf auf die neue Klasse abgeändert werden. Genauso könnte die Methode `calculateDiscount` eine andere Methode der

```
using System;

namespace myExample
{
        class testClass
        {
                private static int customernumber;

                [STAThread]
                static void Main(string[] args)
                {
                        Customer c = new Customer(new Customertype());
                        c.calculateDiscount();
                }
        }

        public class Customer
        {
                private double discount;
                private Customertype type;
                private double sales;
                private static double baseDiscount = 2;
                //some other attributes like a name and a unique number etc.

                public Customer(Customertype type)
                {
                        this.discount = 1.5;
                        this.type = type;
                        this.sales = 0;
                }

                public double calculateDiscount()
                {
                        return baseDiscount + this.discount + (this.sales / 1000000);
                }

                //.......... some other code ..........
        }

        public class Customertype
        {
                //.......... some other code ..........
        }
}
```

Abbildung 16: Ausgangssituation: Die Methode `calculateDiscount` soll in die Klasse Customertype verschoben werden

Klasse Customer aufrufen und auch diese müssen nach dem Verschieben noch erreichbar sein.
Es sind jedoch nicht nur Methoden untereinander durch gegenseitige Aufrufe verflochten, sondern das Verschieben einer Methode kann auch auf die Attribute einer Klasse Auswirkungen haben, beispielsweise wenn die zu verschiebende Methode ein Attribut der Ausgangsklasse lesen oder schreiben muss. In Abbildung 16 ist dies für die beiden Attribute `discount` und `sales` der Fall.
Ein weiterer Punkt, den es zu beachten gilt, ist die Stellung der Methode innerhalb der Vererbungshierarchie. Wird die Methode in einer Unterklasse überschrieben oder überschreibt die Methode selbst eine Methode ihrer Oberklasse, sollte das Refactoring nicht durchgeführt werden, da durch eine Vererbungshierarchie häufig semantisch wichtige Aspekte ausgedrückt sind, die durch das Herauslösen bestimmter Elemente verloren gingen.
Abbildung 17 zeigt die durchzuführenden Aktivitäten in Form eines Aktivitätsdiagramms. Eine detaillierte Beschreibung all dieser Aktivitäten erfolgt in den nächsten Unterabschnitten.
Es gilt zunächst in einer Vorbereitungsphase alle Ober- und Unterklassen auf gleiche Methodensignatur zu prüfen. Namensgleichheit als *weicheres* Suchkriterium reicht an dieser Stelle nicht aus, da einige modernere Programmiersprachen wie beispielsweise Java oder C# explizit die Koexistenz mehrerer Methoden mit gleichem Namen, aber unterschiedlicher Signatur, zulassen. Diese Technik wird als Überladen bezeichnet. In Abbildung 17 findet die Suche und die Überprüfung der Ober- und Unterklassen durch die Aktivitäten A10 + A20 statt.
Weiterhin müssen alle von der Methode benutzten Attribute (A30) und Methoden (A50) gefunden werden, und es muss weiter entschieden werden, ob sie gleich mitverschoben werden können (A40 + A60).
Als letzte Aktivität der Vorbereitungsphase gilt es noch zu prüfen, ob der Methodenname in der Zielkasse bereits existiert (A70). Ist dies der Fall, muss ein anderer Name gewählt werden (I10), was durch eine Interaktion mit dem Benutzer geschieht.

Nach der Vorbereitungsphase kann dann das eigentliche Refactoring durchgeführt werden. Die Methode wird verschoben (A80), und die Erreichbarkeit der Attribute und Methoden der Ausgangsklasse angepasst (A90 + A100). Zuletzt ist in der Ausgangsklasse noch eine delegierende Methode zu erstellen (A110), damit die verschobene Methode weiterhin über die Ausgangsklasse aufgerufen werden kann.
Die folgenden Abschnitte beschreiben die Aktivitäten detailliert, zeigen die Anforderungen an das Metamodell auf und stellen den Bezug dieser Anforderungen zum DMM her.

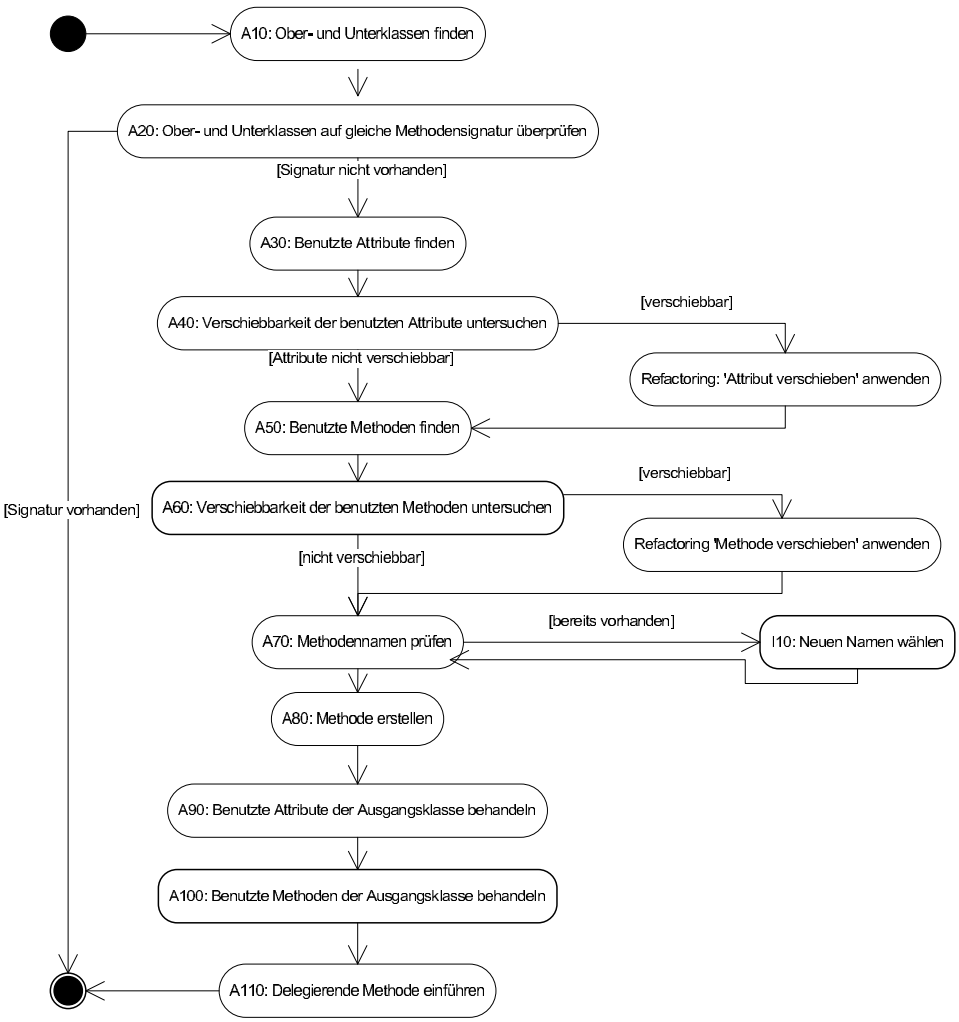

Abbildung 17: Aktivitätsdiagramm *Methode verschieben*

A 10: Ober- und Unterklassen finden

Wie bereits im vorherigen Abschnitt erwähnt, sollte das Refactoring nicht durchgeführt werden, wenn die zu verschiebende Methode eine Methode ihrer Oberklasse überschreibt oder selbst von einer Methode ihrer Unterklassen überschrieben wird. Dadurch kann unter Umständen ein Teil der Semantik, welche durch die Vererbungshierarchie ausgedrückt werden sollte, verloren gehen. Martin Fowler schreibt in [Fow05] zwar:

> „Gibt es weitere Deklarationen, so kann es sein, dass Sie die Verschiebung nicht durchführen können, es sei denn, Polymorphismus kann auch mit der anderen Klasse ausgedrückt werden."

Da die Entscheidung, ob der Polymorphismus auch mit der neuen Klasse ausgedrückt werden kann, nicht trivial anhand weniger Faktoren zu treffen ist, wird dieser Fall ausgeschlossen. Es dürfen also keine Signaturen der Methode in einer Ober- oder Unterklasse der enthaltenden Klasse sein.
Um diese Überprüfung vorzunehmen, müssen zunächst sämtliche Ober- und Unterklassen der betroffenen Klasse gefunden werden. Für das Metamodell bedeutet dies, dass es Assoziationen zwischen Klassen zur Verfügung stellen muss, mit welcher sich Sub- und Superklassen zueinander in Beziehung setzen lassen.

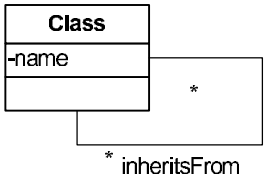

Abbildung 18: Metamodellausschnitt des DMM um Ober- und Unterklassen zu finden

Das DMM bietet für das Traversieren der Klassenhierarchie die *inherits-From*-Relation, mit welcher sich zwischen zwei Klassen eine Spezialisierung modellieren lässt. Den Modellausschnitt des DMM zeigt Abbildung 18. Durch die von GReQL unterstützten Pfadausdrücke lässt sich die Anfrage leicht formulieren.
Im Beispiel aus Abbildung 16 erhält man alle Subklassen der Klasse `Customer` durch die in Abbildung 19 dargestellte GReQL-Anfrage und alle Superklassen durch die in Abbildung 20 dargestellte Anfrage.

A 20: Ober- und Unterklassen auf die gleiche Methodensignatur überprüfen

Für jede Ober- und Unterklasse muss nun geprüft werden, ob sie eine Methodensignatur enthält, welche der Signatur der zu verschiebenden Methode gleicht.

```
FROM   super,sub : V{class}
WITH   super.name = 'Customer'
AND    sub (-->{inheritsFrom}+) super
REPORT sub END
```

Abbildung 19: GReQL-Anfrage, um alle Subklassen zu finden

```
FROM   super,sub : V{class}
WITH   sub.name = 'Customer'
AND    sub (-->{inheritsFrom}+) super
REPORT super END
```

Abbildung 20: GReQL-Anfrage, um alle Superklassen zu finden

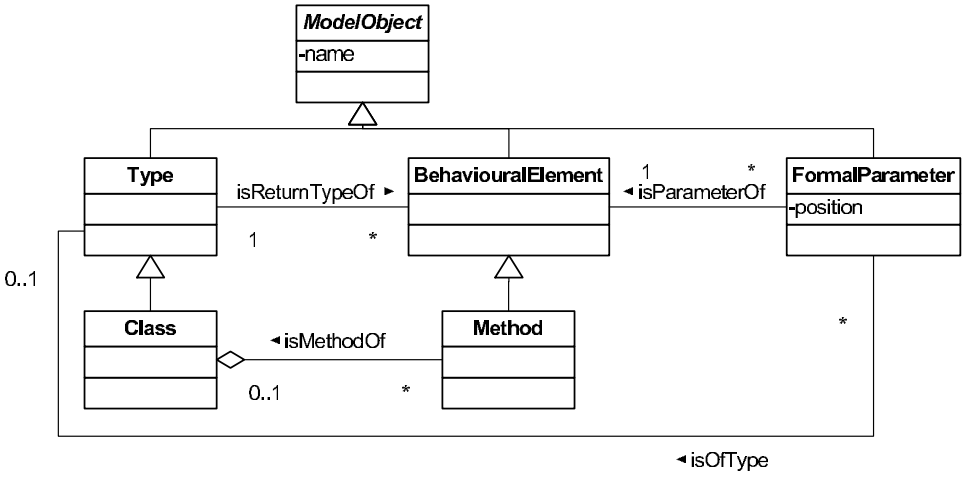

Abbildung 21: Metamodellausschnitt, um Methoden einer Klasse zu finden und auf Signaturgleichheit zu prüfen

Das Metamodell muss also Möglichkeiten bieten, um alle Methoden einer Klasse bestimmen zu können und für jede dieser Methoden den Methodennamen sowie die Typen der Ein- und Ausgabeparameter zu liefern. Für die Bestimmung der zu einer Klasse zugehörigen Methoden bietet das DMM die *isMethodOf*-Relation, welche einer Klasse ihre Methoden zuordnet. Für jede dieser Methoden muss anschließend geprüft werden, ob sie die gleiche Signatur wie die zu verschiebende Methode aufweist, also ob der Name sowie die Typen und Reihenfolgen der Ein- und Ausgabeparameter übereinstimmen. Ist dies der Fall, wird das Refactoring aus den zuvor genannten Gründen abgebrochen. Abbildung 21 zeigt den nötigen Modellausschnitt, um Methoden zu finden und auf Signaturgleichheit zu testen. Der Methodenname ist dabei durch das *name*-Attribut der Klasse ModelObject gegeben, den Typ des Rückgabewerts erhält man durch die DMM-Relation *isReturnTypeOf* und die Parameter der Methode über die Relation *isParameterOf*. Mit diesen drei Angaben kann man nun die zu verschiebende Methode auf Signaturgleichheit mit allen anderen Methoden der jeweiligen Ober- und Unterklassen prüfen. Da die Anordnung der Parameter einer Methode im DMM nicht durch den Graphen selbst festgelegt ist, sondern lediglich durch das Attribut *position* festgestellt werden kann, werden mit der GReQL-Anfrage in Abbildung 22 zunächst die Signaturdaten bestimmt. Das heißt, die GReQL-Anfrage liefert die Signaturelemente aller Methoden von Ober- und Unterklassen, die den gleichen Namen wie die zu verschiebende Methode besitzen. Die eigentliche Überprüfung auf Signaturgleichheit müsste dann anhand dieser Daten durch das Refactoring-Werkzeug selbst vorgenommen werden.

```
FROM cl: V{Class}, m: V{Method}, fp: V{FormalParameter}
WITH cl.name = 'Customertype'
 AND m.name = 'calculateDiscount'
 AND ( cl     <->{inheritsFrom}*
              <--{isMethodOf}
       m )
REPORT  m,
       m (-->[isMethodOf}),
       fp.position,
       fp (-->{isOfType}),                                    10
       m (<--{isReturnTypeOf} END
```

Abbildung 22: GReQL-Anfrage die Signaturdaten aller Methoden gleichen Namens der Ober- und Unterklassen zu erhalten

Werden von der zu verschiebenden Methode keine Methoden einer Superklasse überschrieben beziehungsweise wird die Methode selbst nicht in einer Subklasse überschrieben, kann das Refactoring durchgeführt werden. Es gilt dann zunächst alle von der Verschiebung betroffenen Elemente in der Aus-

gangsklasse (die Klasse `Customer` in Abbildung 16) zu identifizieren.

A 30: Benutzte Attribute finden

Wie bereits in der Einleitung geschrieben, müssen alle von der zu verschiebenden Methode benutzten Attribute gefunden werden, damit diese auch in der Zielklasse für die Methode erreichbar sind.
Das Metamodell muss also eine Möglichkeit bieten, die Attributzugriffe nach Methoden zu differenzieren und zusätzlich noch eine Unterscheidung ermöglichen, ob es sich tatsächlich um ein Attribut oder lediglich um eine lokal definierte Variable handelt, die nur innerhalb einer Methode sichtbar ist und genutzt wird.

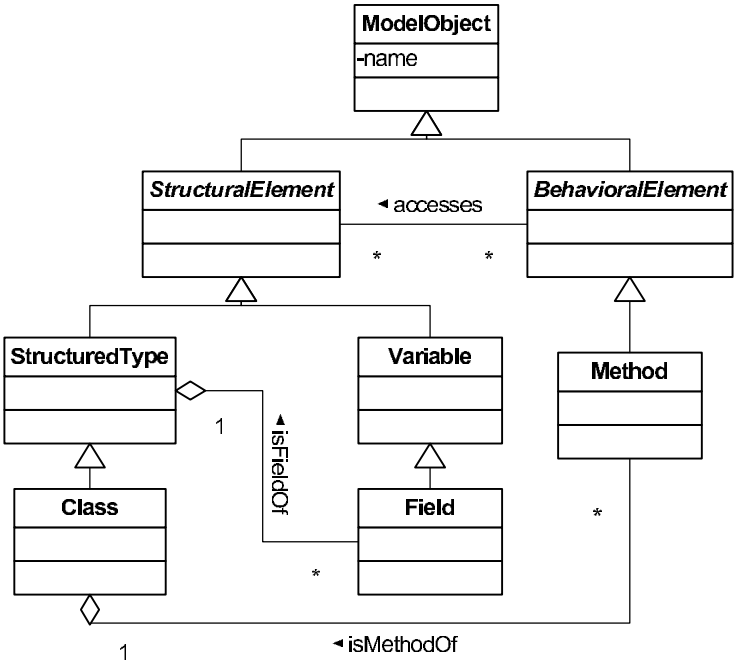

Abbildung 23: Erforderlicher Metamodellausschnitt, um von einer Methode benutzte Attribute zu finden

Um diejenigen Attribute zu finden, welche von der zu verschiebenden Methode benutzt werden, bietet das DMM die *accesses*-Relation. Diese Relation modelliert den Zugriff eines *BehaviouralElement*, was der zu verschiebenden Methode entspricht, auf ein *StructuralElement*, wovon Field eine Subklasse darstellt (vgl. auch Abbildung 23).
Nun muss noch unterschieden werden, ob es sich bei dem *StructuralElement* auch tatsächlich um ein Attribut oder lediglich um eine innerhalb der Methode deklarierte lokale Variable handelt. Das DMM besitzt dazu für

normale Variablen den Typ *Variable* und speziell für Attribute eine Unterklasse von *Variable*, nämlich *Field*, die über eine *isFieldOf*-Relation mit einem *StructuredType* (zum Beispiel einer Klasse) verbunden ist. Die Klasse der Methode und damit auch den StructuredType erhält man über die *isMethodOf*-Relation der zu verschiebenden Methode. Abbildung 23 zeigt die benötigten Elemente um diejenigen Attribute zu finden, welche von der Methode benutzt werden.
Um also alle von der Methode benutzten Attribute zu finden, müssen intuitiv gesprochen zunächst über die *accesses*-Relation alle Variablen gefunden und aus diesen dann diejenigen herausgesucht werden, die über eine *isFieldOf*-Relation mit der Klasse verbunden sind, die auch die Methode enthält. Die entsprechende Anfrage in GReQL, um für das Beispiel in Abbildung 16 alle von `calculateDiscount` benutzten Attribute zu finden, ist in Abbildung 24 dargestellt.

FROM f: V{Field}, m: V{Method}, c: V{Class}
WITH c.name = `'Customer'`
AND m.name = `'calculateDiscount'`
AND f (<−−{accesses}) m (−−>{isMethodOf}) c (<−−{isFieldOf}) f
REPORT f END

Abbildung 24: GReQL-Anfrage zum Finden der benutzte Attribute

Dabei werden aus der Menge aller Knoten vom Typ Class und aus allen Knoten vom Typ Method diejenigen herausgefiltert, die als Klassennamen `Customer` und als Methodennamen `calculateDiscount` haben (`c.name = 'Customer' AND m.name = 'calculateDiscount'`). Durch den Pfadausdruck werden genau diejenigen Attribute bestimmt, welche von den in der Klasse `c` enthaltenen Methoden benutzt werden. Durch die UND-Verknüpfung der drei Ausdrücke erhält man somit genau diejenigen Attribute, welche von der Methode `calculateDiscount` benutzt werden.

A 40: Verschiebbarkeit der benutzten Attribute untersuchen

Beim Verschieben von Methoden besteht die Möglichkeit, dass die zu verschiebende Methode Attribute exklusiv benutzt, das heißt, sie werden von keiner anderen Methode verwendet. In diesem Fall sollten diese Attribute gleich mitverschoben werden.
Um dies herauszufinden, verfolgt man für jedes Attribut die *accesses*-Relation. Ist das Attribut mit keinem anderen BehaviouralElement außer der zu verschiebenden Methode verbunden, sollte es gleich mit in die neue Klasse verschoben werden. Der Metamodellausschnitt ist hierfür der gleiche, wie im Abschnitt Attribute finden (vgl. Abbildung 23). Für das Beispiel in Abbildung 16 würde die GReQL-Anfrage in Abbildung 25 die exklusiv genutzten

Attributen der Methode `calculateDiscount` in der Klasse `Customer` liefern:

```
FROM f: V{Field}, m: V{Method}, c: V{Class}
WITH c.name = Customer
AND m.name = 'calculateDiscount'
AND f (<--{accesses}) m (-->{isMethodOf}) c (<--{isFieldOf}) f
AND inDegree{accesses}(f) = 1
REPORT f END
```

Abbildung 25: GReQL-Anfrage, um alle exklusiv benutzten Attribute zu finden

Bei der Anfrage handelt es sich prinzipiell um die gleiche, wie im Abschnitt 'Benutzte Attribute finden' auf Seite 26. Es wird aber zusätzlich noch geprüft, ob es sich bei den benutzten Attributen um exklusiv von der Methode `movingMethod` benutzte Attribute handelt. Dies lässt sich leicht über den *Indegree*, also die Anzahl der eingehenden Accesses-Kanten herausfinden, (`inDegree{accesses}(f) = 1`).

A 50: Benutzte Methoden finden

Genau wie die benutzten Attribute müssen auch benutzte Methoden gefunden werden. Denn auch diese müssen für die verschobene Methode von der Zielklasse aus erreichbar sein.
Ähnlich der *accesses*-Relation für die Suche nach den benutzten Attributen muss es im Metamodell eine Relation geben, mit der sich Methodenaufrufe darstellen lassen. Im DMM sorgt dafür die *invokes*-Relation, welche zwei *Method*-Objekte miteinander verbindet. Den nötigen Metamodellausschnitt dafür zeigt Abbildung 26.
Um am Beispiel in Abbildung 16 all diejenigen Methoden zu finden, die von `calculateDiscount` aufgerufen werden, ist die GReQL-Anfrage aus Abbildung 27 notwendig.

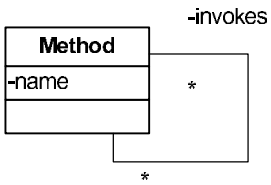

Abbildung 26: Erforderlicher Metamodellausschnitt, um die von einer Methode aufgerufenen Methoden zu finden

```
FROM caller,callee: V{Method}
WITH caller.name = 'calculateDiscount'
AND caller (-->{invokes}) callee
REPORT callee END
```

Abbildung 27: GReQL-Anfrage um benutzte Methoden zu finden

A 60: Verschiebbarkeit der benutzten Methoden untersuchen

Genau wie bei den Attributen kann es auch bei Methoden passieren, dass die zu verschiebende Methode eine andere exklusiv benutzt. Ist dies der Fall, kann die benutzte Methode gleich mit in die Zielklasse verschoben werden. Martin Fowler beschreibt in [Fow05] leider nicht, ob dies nur für die Methoden der Ausgangklasse gilt oder für alle exklusiv genutzten Methoden. Dieser Abschnitt geht davon aus, das alle exklusiv genutzten Methoden, also auch diejenigen aus einer anderen Klasse, mitverschoben werden. Um die exklusive Benutzung festzustellen genügt das Metamodell aus Abbildung 26. Es muss also geprüft werden, ob die benutzte Methode auch noch von anderen Methoden aufgerufen wird beziehungsweise selbst andere Methoden aufruft. Die Abbildungen 28 und 29 veranschaulichen die möglichen Fälle, wobei dort jeweils die Methode A verschoben werden soll. Die dargestellten Graphen könnten dabei aus einer real existierenden Software anhand des Metamodells in Abbildung 26 entstanden sein.

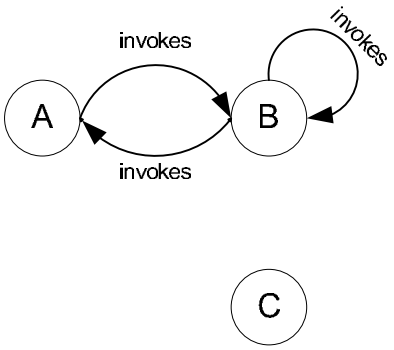

Abbildung 28: Exklusive Nutzung von B durch A

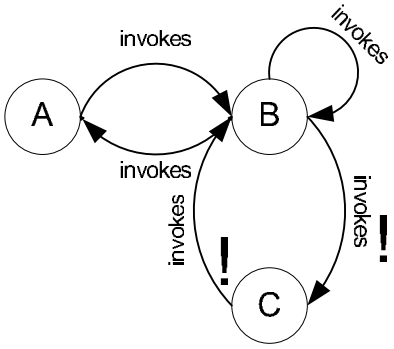

Abbildung 29: Nicht-exklusive Nutzung von B durch A

In Abbildung 28 bestehen nur gegenseitige Abhängigkeiten zwischen der zu verschiebenden Methode A und der Methode B. Es liegt somit der Fall der exklusiven Nutzung vor, wobei dies auch bei Wegfall einiger Kanten noch gegeben wäre. In diesem Fall kann das Refactoring *Methode verschieben* auch auf die Methode B angewandt werden.
Aus dem Graph in Abbildung 29 lässt sich die zusätzliche Nutzung der Methode C feststellen. Die Methode A nutzt die Methode B also nicht exklusiv

und wird damit nicht mitverschoben.
Die in Abbildung 30 dargestellte GReQL-Anfrage liefert die Namen aller Methoden, die von der Methode `calculateDiscount` exklusiv genutzt werden. Diese Anfrage erkennt keine Zyklen in den Aufrufen. In der Abbildung 29 könnte zum Beispiel die Methode A, B und C gemeinsam verschoben werden, da C keine weitere Methode (die Nutzung von A wäre unproblematisch) benutzt.

```
FROM caller, callee: V{Method}
WITH caller.name = calculateDiscount
 AND caller (-->{invokes}) callee
 AND NOT EXISTS otherMethod: V{Method} @
        callee (<->{invokes}) otherMethod
        AND NOT otherMethod isIn SET(caller,callee)
REPORT callee END
```

Abbildung 30: GReQL-Anfrage um alle von `calculateDiscount` exklusiv benutzten Methoden zu finden

A 70: Methodenname prüfen

Bevor die neue Methode nun in der Zielklasse erstellt werden kann, muss zunächst geprüft werden, ob der Methodenname bereits vergeben ist. Diese Überprüfung betrifft nicht nur die Zieklasse selbst, sondern muss auch für alle ihre Ober- und Unterklassen durchgeführt werden. Ansonsten könnte es zu Konflikten kommen, wenn die Methode in Ihrer neuen Klasse eine Methode einer Oberklasse überschreibt oder selbst in einer Unterklasse überschrieben wird. Um diese Überprüfung vorzunehmen ist es notwendig, namentlich alle Methoden der Klasse und ihrer Ober- und Unterklassen zu finden. Im DMM besteht die Möglichkeit, über die *isMethod*-Relation an alle Methoden einer Klasse und deren Namen zu gelangen. Die Ober- und Unterklassen erhält man über die *inheritsFrom*-Relation. Den nötigen Metamodellausschnitt des DMM zeigt Abbildung 31.
Ist der Methodenname bereits vergeben, muss ein anderer gewählt werden. Daher genügt die Ermittlung der Anzahl der Methoden mit gleichem Namen. Ist diese größer als 0, muss ein anderer Name gewählt werden. Im Codebeispiel in Abbildung 16 auf Seite 19 soll die Methode `calculateDiscount` verschoben werden. Die GReQL-Anfrage, um die Anzahl der Methoden zu bestimmen, welche den gleichen Namen besitzen, ist in Abbildung 32 dargestellt.
Ist der Methodenname bereits vorhanden, kann entweder durch Interaktion mit dem Anwender des Refactoring ein neuer gewählt werden oder das Refactoring muss abgebrochen werden.

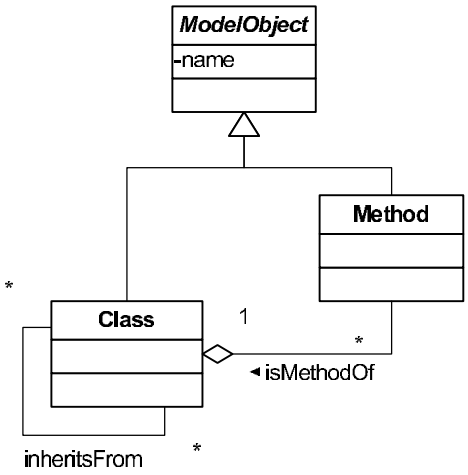

Abbildung 31: Metamodellausschnitt um Methoden auf Namensgleichheit zu prüfen

```
FROM cl: V{Class}, m: V{Method}
WITH cl.name = 'Customertype'
 AND m.name = 'calculateDiscount'
 AND ( cl      <->{inheritsFrom}*
               <--{isMethodOf}
       m )
REPORT cnt(m) END
```

Abbildung 32: GReQL-Anfrage um die Zielklasse auf gleiche Methodennamen zu prüfen

Eine andere Möglichkeit wäre die komplette Signatur auf Gleichheit zu testen und somit das Erstellen von sogenannten überladenen Methoden[5] zuzulassen. In dieser Arbeit wird diese Möglichkeit jedoch nicht berücksichtigt, da überladene Methoden nicht von allen Programmiersprachen unterstützt werden. Falls dies doch einmal Unterstützung finden sollte, kann dies analog zu dem Vorgehen und mit dem Metamodellausschnitt aus der Aktivität 'A20: Ober- und Unterklassen auf gleiche Methodensignatur überprüfen' auf Seite 22 geschehen.

A 80: Methode erstellen

Ist der Name noch nicht vorhanden kann die Methode in die neue Klasse verschoben werden. Dazu wird zunächst ein neues Objekt vom Typ Method erstellt und mit dem noch nicht vorhandenen Namen versehen. Es wird hier explizit ein neuer Knoten erstellt, um die eventuell vorhandenen *accesses*-Relationen zu der alten Methode zu erhalten. Dies ist notwendig, da die Methodenaufrufe später nicht alle auf die neue Klasse ausgerichtet werden können, sondern vielmehr eine delegierende Methode der alten Klasse benutzen, wofür dann der noch vorhandene Knoten benutzt wird.
Für das Refactoring *Methode verschieben* wurde bisher noch keines der im DMM aufgeführten Attribute benötigt (vgl. Abbildung 6 auf Seite 8). Sollten diese aber in dem zusammengefassten Gesamtmodell Verwendung finden, sind sie an dieser Stelle bereits zu berücksichtigen. Denn sämtliche Attributwerte der alten Methode müssen auf die neue Methode übertragen werden.
Neben den Attributen müssen auch alle Relationen der Methode auf das neu erstellte Methodengerüst in der neuen Klasse umgebogen, kopiert oder neu erstellt werden. Dies sind alle Relationen aus den bisherigen Metamodellausschnitten in diesem Unterkapitel. Der dazu benötigte Metamodellauschnitt ist daher auch eine erste Zusammenfassung aller bisherigen Ausschnitte zu einem gemeinsamen Metamodell. Dieses gemeinsame Gesamtmodell wird in den nächsten drei Abschnitten noch ergänzt und ist in Abbildung 40 dargestellt. Die anzupassenden Relationen lassen sich jedoch schon aus den einzelnen Teilmodellen in den Abbildungen 21, 23 und 26 ablesen und setzen sich aus allen Relationen zusammen, die mit der Klasse Method selbst oder einer ihrer Oberklassen assoziiert sind. Diese Relationen sind im Einzelnen:

- *invokes*
- *accesses*
- *isParameterOf*
- *isReturntypeOf*

[5] Als überladene Methoden werden Methoden mit gleichem Namen aber unterschiedlicher Signatur bezeichnet

- *isMethodOf*

Bei der Neuausrichtung der Relationen ist darauf zu achten, dass lediglich die *invokes*-Relationen der Ausgangs und der Zielklasse auf die neu erstellte Methode verweisen. Alle anderen Klassen rufen weiterhin die Methode der Ausgangsklasse auf, die in der Aktivität A110 zu einer delegierenden Methode umfunktioniert wird. Die *isParameterOf*- und die *isReturnTypeOf*-Relationen werden kopiert, erhalten allerdings die neue Methode in der Zielklasse als Assoziationsende und nicht die Methode in der Ausgangsklasse. Eine *isMethodOf*-Relation zwischen der neuen Methode und ihrer neuen Zielklasse muss neu erstellt werden.
Nach der Bestimmung der neuen Ziele der Relationen muss auch noch die Erreichbarkeit der in der Ausgangsklasse verbliebenen Methoden und Attribute angepasst werden. Denn die Methode muss in ihrer neuen Klasse Zugriff auf alle von ihr benutzten Elemente der Ausgangsklasse haben, was in den nächsten beiden Abschnitten erläutert ist.

A 90: Benutzte Attribute der Ausgangsklasse behandeln

Durch die Aktivität 'A30: Benutzte Attribute finden' wurden bereits all diejenigen Attribute gefunden, welche von der verschobenen Methode benutzt werden. Nun gilt es zu entscheiden, wie mit den einzelnen Attributen zu verfahren ist, da sie für die Methode auch aus der neuen Zielklasse heraus erreichbar sein müssen. Im folgenden wird zunächst das Vorgehen für ein Attribut skizziert und danach der Bezug zum DMM hergestellt. Diese Schritte müssen jedoch für alle benutzten Attribute durchgeführt werden.
Durch die Aktivität 'A40: Verschiebbarkeit der benutzten Attribute untersuchen' wurden bereits alle verschiebbaren Attribute behandelt. Die verbleibenden Attribute werden somit neben der verschobenen Methode auch von anderen Methoden der Ausgangsklasse benutzt. Um zu entscheiden, wie mit diesen Attributen zu verfahren ist, müssen diese zunächst auf einige Eigenschaften hin untersucht werden, was im Folgenden für ein Attribut exemplarisch beschrieben ist.

Eine dieser Eigenschaften ist die Sichtbarkeit. Ist ein Attribut *privat (engl. private)*, so ist es nur für die enthaltende Klasse selbst sichtbar. Ist es hingegen *öffentlich (engl. public)*, können auch andere Klassen darauf zugreifen. Da die Methode, welche das Attribut benutzt, in eine neue Klasse verschoben wird, muss dieses für die Methode erreichbar sein. Das Attribut einfach als öffentlich zu deklarieren ist kein guter Stil, da hierdurch das Geheimnisprinzip[6] verletzt wird. Der Zugriff sollte besser über eigens dafür vorgesehene Zugriffsmethoden erfolgen.

[6]Das Geheimnisprinzip besagt, dass der direkte Zugriff auf die interne Datenstruktur eines Objekts verhindert wird.

Existieren bereits Zugriffsmethoden, sind diese lediglich als *öffentlich* zu deklarieren. Sind keine Zugriffsmethoden vorhanden, wie das im Beispiel auf Seite 19 etwa für das Attribut `baseDiscount` der Fall ist, muss das Attribut zunächst als öffentlich deklariert werden. Im Beispiel würde die entsprechende Zeile dann folgendermaßen lauten:

```
public static double baseDiscount = 2;
```

Im Anschluss daran wird auf das Attribut das Refactoring *Attribut kapseln* (vgl. Kapitel 3.3.4 auf Seite 16) angewandt. Dies erstellt zum einen die öffentlichen Zugriffsmethoden und kapselt auch direkt das Attribut wieder, um das Geheimnisprinzip nicht zu verletzen. Aus dem obigen Beispiel würde das Refactoring *Attribut kapseln* dann den in Abbildung 33 dargestellten Code erzeugen.

```
private static double baseDiscount = 2;

public static double get_baseDiscount ()
{ return baseDiscount; }

public static void set_baseDiscount (double var)
{ baseDiscount = var; }
```

Abbildung 33: Code nach Anwendung des Refactoring 'Attribut kapseln'

An dieser Stelle ist man nun in der Lage, das Attribut mittels der Zugriffsmethoden zu manipulieren.

Eine andere Unterscheidung betrifft die Art des Attributs, da zwischen Objektattributen und Klassenattributen unterschieden werden muss. Handelt es sich um ein Klassenattribut, kann dieses einfach über die Zugriffsmethoden erreicht werden. Um die Klassenvariable `baseDiscount` aus dem Beispiel aus einer anderen Klasse als `Customer` zu überschreiben, genügt:

```
...
Customer.set_baseDiscount(4);
...
```

Im Falle eines Objektattributs muss die Signatur der Methode erweitert werden, so dass sie zukünftig eine Referenz auf das Ausgangsobjekt enthält. Über diese Referenz hat die Methode dann Zugriff auf alle öffentlichen Teile des Ausgangsobjekts. Wenn die Methode `calculateDiscount` aus dem Beispiel auf Seite 19 in die Klasse `Customertype` verschoben wurde, muss eine Objektreferenz auf das ursprüngliche `Customer`-Objekt mitgegeben werden, damit `calculateDiscount` auch weiterhin auf die Variablen `baseDiscount`,

`discount` und `sales` zugreifen kann, sofern diese nicht in die neue Klasse mitverschoben wurden. Die Methode hätte in der neuen Zielklasse dann die Form wie in Abbildung 34.

```
public class Customertype
{
        ...

public double calculateDiscount(Customer c)
{
        return Customer.get_baseDiscount + c.discount + (c.sales / 1000000);
}

        ...                                                              10
}
```

Abbildung 34: Code von `calculateDiscount` in der neuen Zielklasse

Eine letzte Unterscheidung bei Objektvariablen könnte noch bei der Art des Attributzugriffs erfolgen. Wird ein Attribut lediglich gelesen, müsste keine Objektreferenz im Methodenaufruf mitgegeben werden. Das gelesene Attribut als Parameter im Methodenaufruf würde ausreichen, was im übrigen auch für Klassenattribute gilt. Im Hinblick auf das DMM geschieht diese Unterscheidung jedoch zunächst nicht, da es im DMM keine Möglichkeit gibt zwischen lesendem und schreibendem Attributzugriff zu differenzieren. Falls das DMM aufgrund anderer Umstände später um diese Möglichkeit erweitert wird, kann dies für dieses Refactoring dann noch mitberücksichtigt werden. Abbildung 35 zeigt das Vorgehen in Form eines Aktivitätsdiagramm. Eine Schwierigkeit in der Umsetzung des zuvor beschriebenen Vorgehens liegt in der nicht festgelegten Syntax für Zugriffsmethoden. So werden dem Programmierer bei der Namensvergabe für diese Methoden freie Wahlmöglichkeiten eingeräumt. Diese lassen sich dadurch nur begrenzt, allenfalls in Form von Mustererkennung, von einem Parser erkennen. Für das weitere Vorgehen wird deshalb empfohlen, für die Wahl der eventuell vorhandenen Zugriffsmethoden mit dem Benutzer des Refactoringtools in Interaktion zu treten, damit dieser die Methoden manuell auswählen kann.

Das Metamodell muss die Möglichkeit bieten, die Sichtbarkeit von Attributen zu modifizieren und nach Klassen- und Objektattributen zu differenzieren. Für die Sichtbarkeit bietet das DMM das *visibility*-Attribut. Die Unterscheidung zwischen Klassen- und Objektattributen kann mit dem Standard-DMM nicht getroffen werden. Aus diesem Grund wird das DMM-Element *Field*, mit welchem die Attribute modelliert werden, um ein Attribut *isStatic* ergänzt. Diese Ergänzung des DMM ist in Abbildung 36 veranschaulicht.

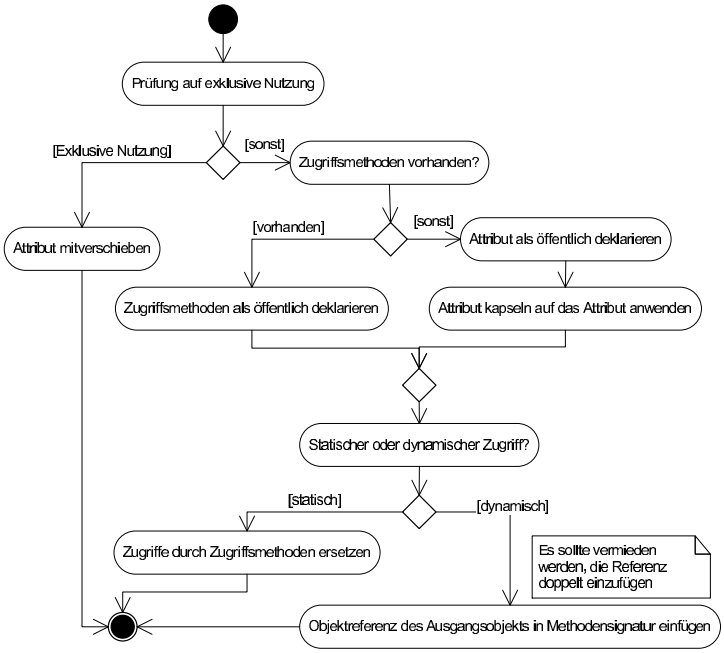

Abbildung 35: Aktivitätsdiagramm für die Behandlung benutzter Attribute

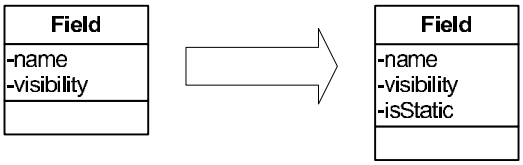

Abbildung 36: Links das Original-DMM; Rechts mit *isStatic*-Erweiterung

Die Sichtbarkeit eines Attributs zu ändern geht im Modell einfach durch Setzen der entsprechenden *visibility*-Eigenschaft des Modellelements. Das Erstellen der Zugriffsmethoden geschieht einfach durch Anwenden des Refactoring *Attribut kapseln*. Ob es sich dann bei einem Attribut um ein Klassen- oder Objektattribut handelt, kann dank der Erweiterung des DMM auch direkt abgelesen werden. Wie mit den Attributen weiter zu verfahren ist und wie das Verfahren anhand des Modells durchgeführt wird, ist im Folgenden beschrieben.

Klassenattribut Handelt es sich bei dem Attribut um ein Klasseattribut, müssen alle Zugriffe auf das Attribut angepasst werden. Es müssen alle direkten Attributzugriffe durch den Aufruf der Zugriffsmethoden ersetzt werden. Das bedeutet konkret für das Modell, dass alle *accesses*-Relationen zwischen der Methode und dem entsprechenden Attribut gefunden werden müssen. Allen diesen direkten *accesses*-Relationen zwischen Methode und Attribut müssen *invokes*-Relationen zwischen Methode und Zugriffmethode zwischengeschaltet werden. Die ursprüngliche *accesses*-Relation besteht danach nur noch zwischen Attribut und Zugriffsmethode. Abbildung 37 zeigt oben die Situation vorher, wo die Methode `calculateDiscount` direkt auf `baseDiscount` zugreift und darunter den Zugriff über die entsprechende Zugriffsmethode.

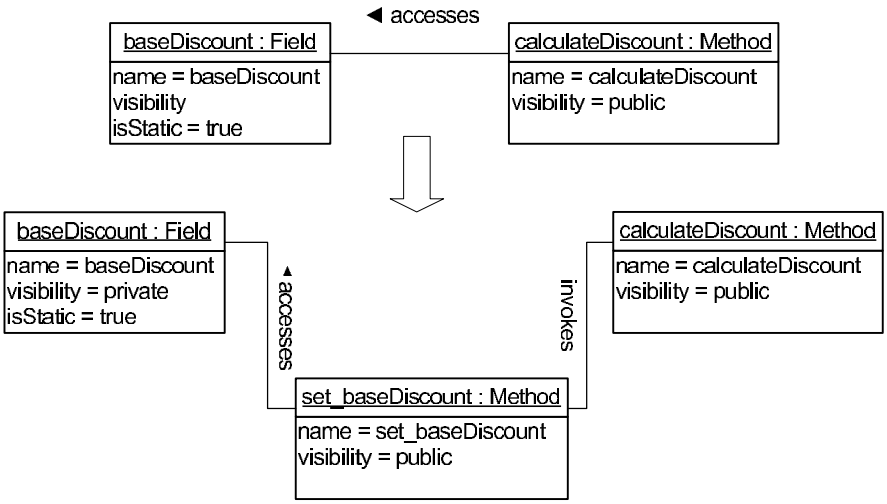

Abbildung 37: Direkter Zugriff auf Klassenvariable oben und Zugriff über die Zugriffsmethoden unten

Objektattribut Liegt ein Objektattribut vor, muss eine Objektreferenz in die Methodensignatur eingefügt werden, damit die verschobene Methode auch Zugriff auf die verwendeten Attribute bekommt.

Um einen weiteren Parameter in die Methodensignatur zu ergänzen, benötigt man zunächst Informationen über die bereits vorhandenen Parameter. Es ist einerseits nötig, die Einfügeposition zu bestimmen und andererseits muss ausgeschlossen werden, dass bereits eine Objektreferenz übergeben wurde. Dies passiert genau dann, wenn die Referenz bereits durch ein zuvor behandeltes Attribut übergeben wurde. Dies kann man umgehen, indem eine neue Objektreferenz mit einem wiedererkennbaren Namensmuster eingefügt wird.

Mit der Aktivität 'A70: Methodennamen prüfen' wurde bereits überprüft, dass keine Methode mit gleichem Namen in der Klasse selbst, beziehungsweise in einer ihrer Ober- und Unterklassen vorhanden ist. Der Parameter kann also direkt eingefügt werden und es kann keine Methode mit gleicher Signatur in einer Ober- oder Unterklasse vorhanden sein.

Das DMM bietet für Parameter einer Methode den Typ FormalParameter, welcher ein Attribut *Position* unterstützt, womit sich die Einfügeposition eines neuen Parameters bestimmen lässt. Neben der Einfügeposition wird noch der Typ des Parameters benötigt. Da es sich dabei um den Typ der Ausgangsklasse handelt, ist dieser bereits bekannt.
Ist der Parameter in die Methodensignatur eingefügt, müssen die Attributzugriffe wie schon bei Klassenattributen angepasst werden. Dazu werden in der verschobenen Methode allen direkten Zugriffe über die *accesses*-Relation Aufrufe der Zugriffsmethoden mittels *invokes*-Relationen vorgeschaltet.

Nach dieser Aktivität sind alle notwendigen Attribute der Ausgangsklasse für die Methode in ihrer neuen Umgebung erreichbar. Den für diese Aktivität nötigen Metamodellausschnitt des DMM und der Erweiterung zeigt Abbildung 38. In diesem Ausschnitt ist schon eine Erweiterung vorweg genommen, da auch Methoden ein Attribut *isStatic* besitzen, welches für die Behandlung der benutzten Methoden der Ausgangsklasse benötigt wird.

A 100: Benutzte Methoden der Ausgangsklasse behandeln

Ähnlich wie mit den benutzten Attributen der Ausgangsklasse muss auch mit den benutzten Methoden verfahren werden. Die benutzten und nicht mitverschobenen Methoden wurden bereits durch die Aktivitäten 'A50: Benutzte Methoden finden' und 'A60: Verschiebbarkeit der benutzten Methoden untersuchen' ab Seite 27 bestimmt. Auch hier muss sich für die jeweilige Methode zunächst um die Sichtbarkeit gekümmert werden. Wie in Abbildung 38 dargestellt, ist das *visibility*-Attribut auch für den Typ *Method* durch Vererbung vorhanden, mit dem man die aktuell betrachtete Methode einfach auf *öffentlich* setzt.

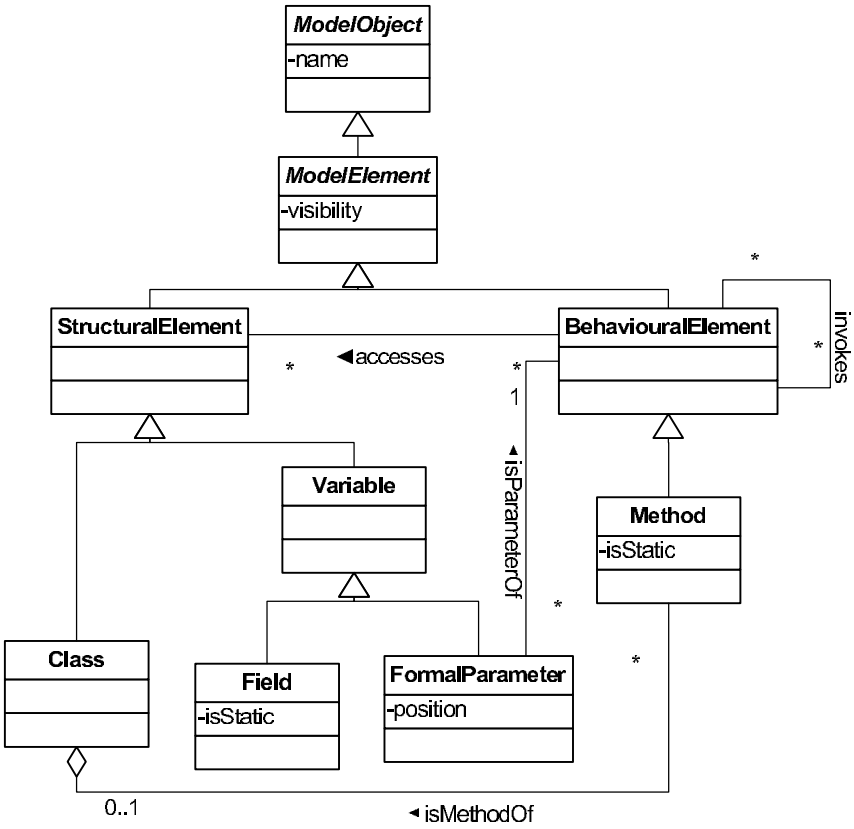

Abbildung 38: Erforderlicher Metamodellausschnitt um die benutzten Attribute richtig behandeln zu können

An dieser Stelle sollte nun eine Unterscheidung zwischen Klassenmethoden und Objektmethoden getroffen werden. Wie schon bei Attribut, kann mit den Mitteln des DMM nicht unterschieden werden, ob es sich um eine Objekt- oder Klassenmethode handelt. Aus diesem Grund wird die Metaklasse *Method* um ein Attribut *isStatic* erweitert. Diese Erweiterung ist in Abbildung 38 bereits berücksichtigt.
Da alle benutzten Methoden bereits als öffentlich deklariert wurden, können Klassenmethoden einfach unter Benutzung des Klassennamens der Ausgangsklasse aufgerufen werden. Im Falle von Objektmethoden muss, wie bereits bei Objektattributen, eine Referenz auf das Ausgangsobjekt in die Methodensignatur eingefügt werden, falls dies nicht durch benutzte Attribute bereits geschehen ist. Das Vorgehen ist dabei analog zu dem bei Attributen. Es wird ein weiterer *FormalParameter* erzeugt und der Signatur hinzugefügt. Aufgrund der Ähnlichkeit zu 'Benutzte Attribute der Ausgangsklasse behandeln' kann der gleiche Metamodellausschnitt verwendet werden (vgl. Abbildung 38). Auch hier müssen noch alle *invokes*-Relationen von den ur-

spünglichen Zielmethoden auf die Objekt-Referenz angepasst werden.

A 110: Delegierende Methode einführen

Ist die Methode in der neuen Klasse funktionsfähig, muss sie für die ursprüngliche Klasse noch erreichbar gemacht werden. Wenn bereits ein Attribut in der Ausgangsklasse besteht, kann dieses genutzt werden. Ansonsten muss man ein neues Attribut erstellen, um den Zugriff auf die Methode zu erhalten.
Die ursprünglichen Methode sollte nun in eine delegierende Methode umgewandelt werden. Falls die Signatur unverändert geblieben ist, also keine Objektreferenz hinzugefügt wurde, und wenn keine andere Klasse außer der Ausgangsklasse die Methode benutzt, ist es sinnvoll die delegierende Methode löschen. Dabei ist darauf zu achten, dass alle Referenzen auf die Methode angepasst werden. Die in Abbildung 39 dargestellte GReQL-Anfrage bestimmt die Anzahl der Methoden, welche die Methode `calculateDiscount` in der Ausgangsklasse neben den Methoden der Ausgangklasse selbst noch benutzen.

```
cnt(   FROM class, sourceClass: V{Class}, myMethod,method: V{Method}
       WITH myMethod.name = 'calculateDiscount'
        AND  sourceClass.name = 'Customer'
        AND  class.name <> 'Customer'
        AND  sourceClass (<--{isMethodOf}) myMethod
             (<--{invokes}) method (-->{isMethodOf}) class
       REPORT method END)
```

Abbildung 39: GReQL-Anfrage um die Anzahl benutzenden Methoden zu ermitteln

Martin Fowler beschreibt diesen Schritt in [Fow05] noch allgemeiner, so dass die delegierende Methode auch gelöscht werden kann, falls andere Klassen diese referenzieren, indem einfach alle Referenzen auf die neue Zielklasse geändert werden. Dies wird hier jedoch ausgeschlossen, da dann für alle diese Klassen sichergestellt sein muss, dass sie die neue Zielklasse der Methode erreichen können.

Zusammenfassung

Ziel dieses Abschitts war die Zerlegung des Refactoring *Methode verschieben* in kleinere Aktivitäten und die anschließende Beschreibung dieser Aktivitäten. Weiterhin sollten aus den einzelnen Aktivitäten Metamodelle zur modellbasierten Durchführung dieser Aktivitäten herausgearbeitet werden.

Aus diesen einzelnen Metamodellen lässt sich ein Gesamtmodell zusammensetzen, mit welchem das Refactoring *Methode verschieben* durchgeführt werden kann. Dieses Gesamtmodell ist in Abbildung 40 dargestellt, wobei die dunkleren Flächen Abweichungen zum Original-DMM hervorheben.

4.2 Attribut verschieben

Ein weiteres, häufig anzuwendendes Refactoring neben dem Verschieben von Methoden, ist das Verschieben von Attributen. Auch dieses Refactoring wird in den folgenden Unterabschnitten in einzelne Aktivitäten zerlegt. Wie bereits bei der Zerlegung von *Methode verschieben* werden für die Aktivitäten von *Attribut verschieben* Anforderungen an ein Metamodell formuliert und das *Dagstuhl Middle Metamodel* auf die Erfüllung dieser Anforderungen hin geprüft. Zur Veranschaulichung wird an nötigen Stellen auf das Codebeispiel aus Abbildung 16 auf Seite 19 zurückgegriffen.

Die Voraussetzungen, um ein Attribut in eine andere Klasse zu verschieben, sind nicht so vielschichtig wie jene, um eine Methode zu verschieben, da es sich bei Attributen nicht um aktive Elemente handelt, welche andere Elemente benutzen. Es müssen also lediglich diejenigen Elemente behandelt werden, die das Attribut verwenden.
Das Aktivitätsdiagramm in Abbildung 41 zeigt die Aufteilung des Refactoring *Attribut verschieben* in Teilaktivitäten.
Hierbei ist zunächst festzustellen, ob es sich um ein privates Attribut handelt oder nicht, also ob das Attribut auch von außerhalb der enthaltenden Klasse benutzt wird (A200). Ist dies der Fall, sollte zunächst das Refactoring *Attribut kapseln* (vgl. Kapitel 4.3 auf Seite 52) auf das Attribut angewendet werden. Danach ist sichergestellt, dass alle Zugriffe auf das Attribut ausschließlich über die Zugriffsmethoden erfolgen.

Als weitere Voraussetzung ist nun noch zu prüfen, ob der Name des zu verschiebenden Attributs bereits in der Zielklasse oder einer ihrer Ober- oder Unterklassen existiert, da dies sonst zu Konflikten führen kann (A210). Ist der Name bereits vorhanden, wird ein anderer gewählt und es kann mit dem eigentlichen Refactoring begonnen werden, indem zunächst das Feld in der Zielklasse erstellt wird (A220).
Im nächsten Schritt müssen Namen für die *get*- und *set*-Methoden gewählt werden (I110), welche auch automatisch vergeben werden können. Dafür muss die Zielklasse mit ihren Ober- und Unterklassen geprüft werden, ob die Namen bereits vergeben sind (A70). Diese Aktivität wurde für das Refactoring *Methode verschieben* bereits benötigt und kann an dieser Stelle wiederverwendet werden. Sind alle Namenskonflikte gelöst, werden die *get*- und *set*-Methoden in der Zielklasse erstellt (A230).
Da die noch vorhandenen Methoden weiterhin Zugriff auf das Attribut benö-

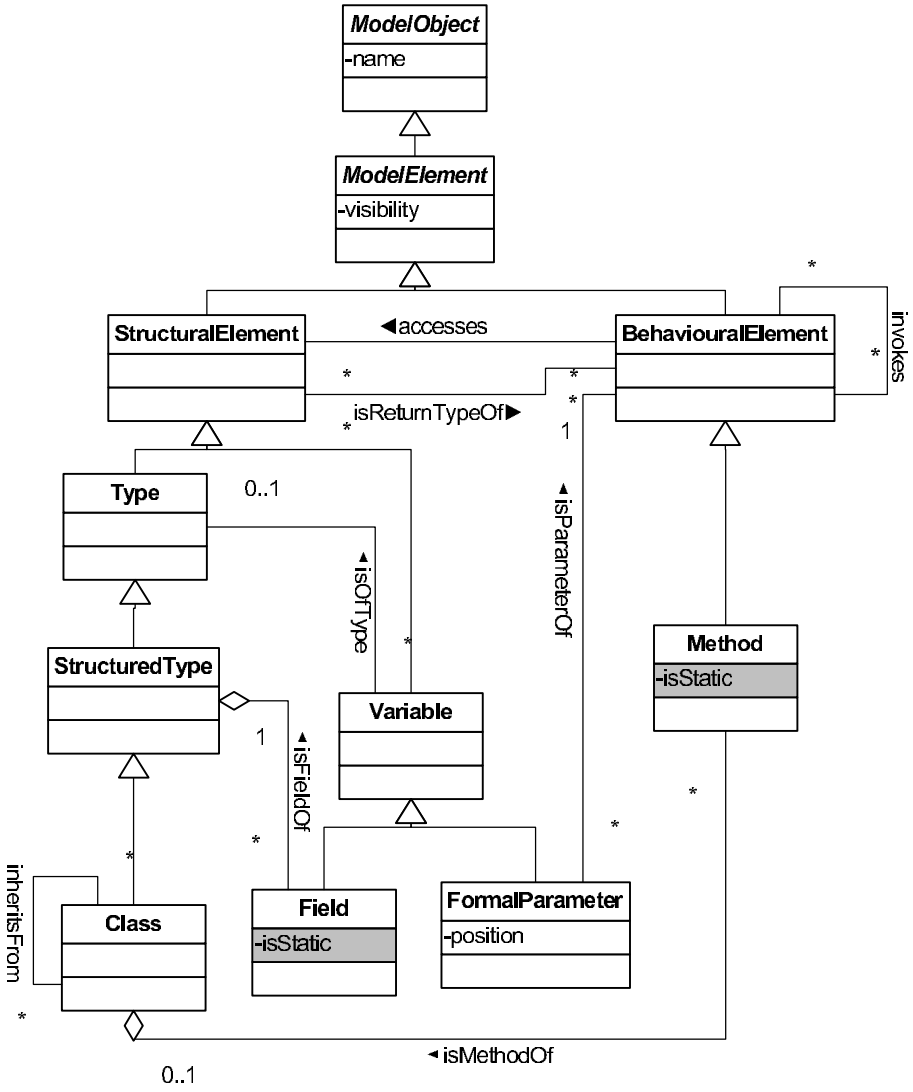

Abbildung 40: Metamodell für das Refactoring *Methode verschieben*

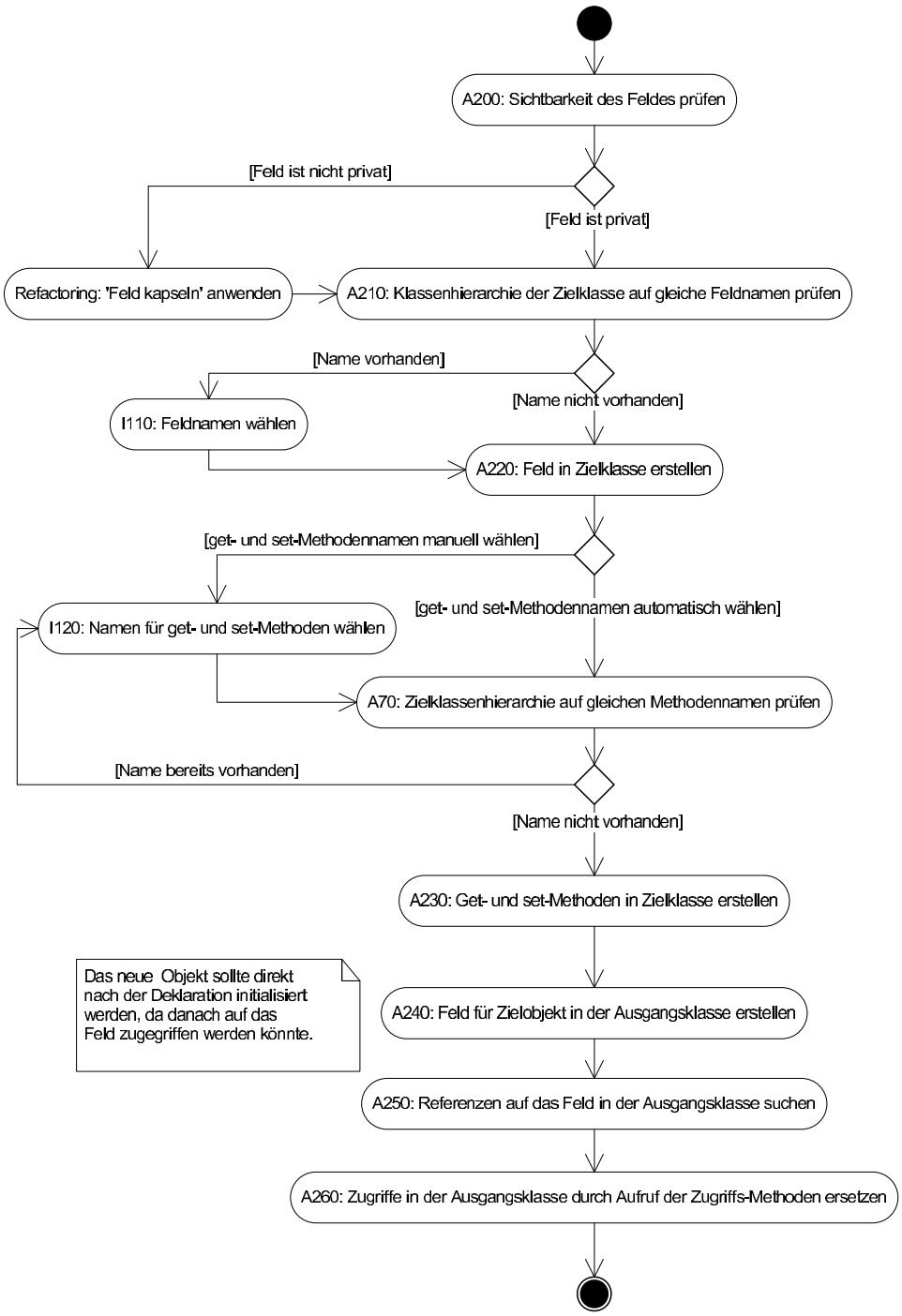

Abbildung 41: Aktivitätsdiagramm *Attribut verschieben*

tigen, muss die Erreichbarkeit der Zielklasse aus der Ausgangsklasse sichergestellt werden (A240).
Kann das Zielobjekt aus der Ausgangsklasse erreicht werden, müssen in der Ausgangsklasse alle Referenzen auf das Feld gesucht (A250) und durch den Aufruf der Zugriffsmethoden ersetzt (A260) werden. Eine detaillierte Beschreibung der einzelnen Aktivitäten erfolgt in den folgenden Unterabschnitten.

A 200: Sichtbarkeit des Feldes prüfen

Wie bereits im Aktivitätsdiagramm in Abbildung 41 zu erkennen ist, muss sich zunächst mit der Sichtbarkeit des Attributs befasst werden. Denn falls auch Objekte anderer Klassen als der Ausgangsklasse auf das Feld zugreifen, muss sichergestellt sein, dass diese Objekte das Feld auch nach dem Verschieben noch erreichen können.

Da es nur schwer möglich und sinnvoll ist, in allen zugreifenden Klassen ein Objekt mit der Zielklasse des Feldes zu erstellen, sollte zunächst das Refactoring *Attribut kapseln* angewandt werden. Dies stellt sicher, dass nun alle externen Zugriffe auf das Feld über Zugriffsmethoden erfolgen und sich direkte Feldzugriffe somit auf die Ausgangsklasse selbst beschränken.

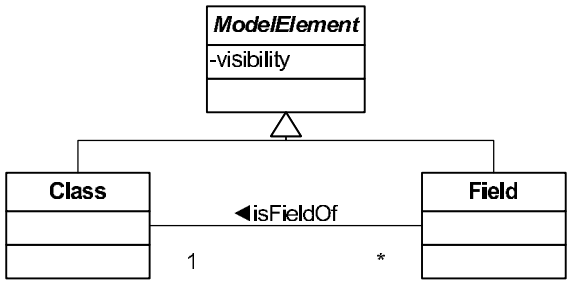

Abbildung 42: Nötiger Modellausschnitt, um die Sichbarkeit eines Feldes zu ermitteln

Um die Überprüfung vorzunehmen, ob es sich bei einem Feld um eine *öffentliches* Feld handelt, muss im Metamodell ein Attribut für die Sichtbarkeit von Feldern gegeben sein.
Im DMM ist hierfür ein Attribut *visibility* vorgesehen, welches genau diesen Zweck erfüllt. Das *visibility*-Attribut steht jedem ModelElement zur Verfügung und damit auch für Felder einer Klasse, die eine Unterklasse von ModelElement darstellen.

Abbildung 42 zeigt den nötigen Modellausschnitt, um die Sichtbarkeit von Attributen zu prüfen. Die GReQL-Anfrage, um die Sichtbarkeit des Attributs

`discount` aus dem Codebeispiel auf Seite 19 zu ermitteln, ist in Abbildung 43 dargestellt.

```
FROM  f : V{Field}, c : V{Class}
WITH f.name = 'discount'
AND   c.name = 'Customer'
REPORT f.visibility END
```

Abbildung 43: GReQL-Anfrage, um die Sichtbarkeit des Attributs `myAttribute` zu prüfen

A 210: Klassenhierarchie der Zielklasse auf gleiche Feldnamen prüfen

Im nächsten Schritt muss sichergestellt werden, dass der Name des zu verschiebenden Feldes weder in der Zielklasse noch deren Ober- und Unterklassen vergeben sein darf, da dies zu Konflikten führt.
Das Metamodell muss also Möglichkeiten bieten, die Klassenhierarchie der Zielklasse zu ermitteln und für jede dieser Klassen zu prüfen, ob der Feldname dort bereits vergeben wurde.
Wie bereits in der Aktivität 'A10: Ober- und Unterklassen finden' auf Seite 22 erwähnt, ist die Suche nach Ober- und Unterklassen mit dem DMM möglich. Da das DMM auch für jedes Modellobjekt ein Namensattribut bereitstellt, ist auch der Namensvergleich kein Problem. Den dafür notwendigen Modellausschnitt des DMM zeigt Abbildung 44.

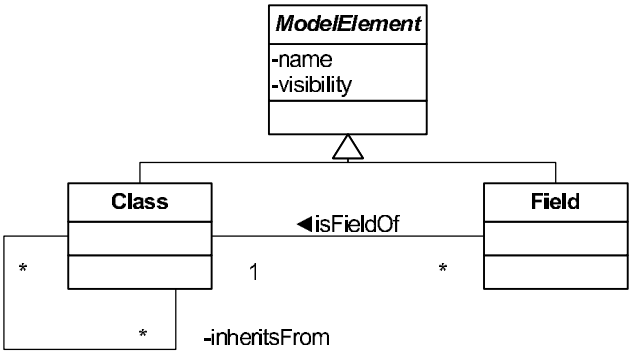

Abbildung 44: Metamodellausschnitt, um die Zielklassenhierarchie auf gleiche Feldnamen zu prüfen

Die GReQL-Anfrage, um die gesamte Klassenhierarchie der Klasse `Customer` zu durchlaufen und jede Klasse auf das Enthaltensein eines Feldes mit dem Namen `discount` zu prüfen, ist in Abbildung 45 dargestellt.

```
FROM   f : V{Field}, c : V{Class} s : V{Class}
WITH f.name = 'discount'
AND   c.name = 'Customer'
AND   c (<->{inheritsFrom}* s (<--{isFieldOf}) f
REPORT cnt(s) END
```

Abbildung 45: GReQL-Anfrage, um die Klassenhierarchie der Zielklasse auf gleiche Feldnamen zu prüfen

A 220: Feld in Zielklasse erstellen

Ist sichergestellt, dass der Feldname noch nicht verwendet wird, kann das Feld in der Zielklasse erstellt werden. Dazu ist die *isFieldOf*-Relation zwischen dem Feld und der Ausgangsklasse zu löschen und eine neue zwischen dem Feld und der neuen Klasse zu erstellen. Dadurch bleibt der Typ des Feldes direkt erhalten. Handelt es sich bei dem Feld um eine Klassenvariable, so bleibt durch das *Umbiegen* der Relation auch diese direkt erhalten. Den benötigten Modellausschnitt zeigt Abbildung 46, dabei wurde das Attribut *isStatic* bei der Klasse *Field* schon mit eingeführt, da dies beachtet werden muss, wenn in der nächsten Aktivität Zugriffsmethoden erstellt werden. Die dazu nötige Erweiterung des DMM wurde bereits in der Aktivität 'A90 : Benutzte Attribute der Ausgangsklasse behandeln' auf Seite 32 erläutert und wird hier wieder benutzt.

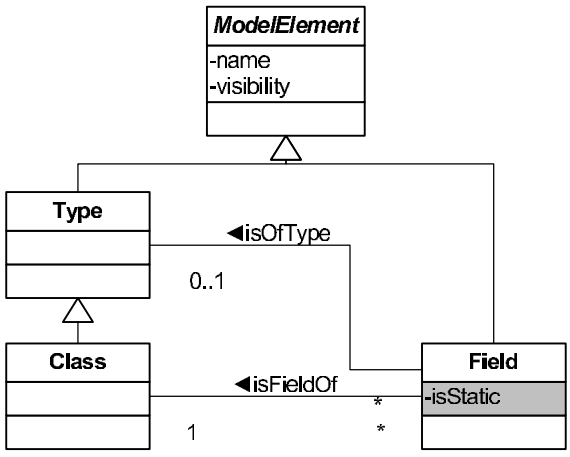

Abbildung 46: Modellausschnitt, um ein Feld in einer Klasse zu erstellen

A70: Zielklassenhierarchie auf gleichen Methodennamen prüfen

Bevor für das Feld in der Zielklasse auch Zugriffsmethoden erstellt werden können, muss auch für diese geprüft werden, ob in der Zielklasse nicht be-

reits Methoden mit gleichem Namen vorhanden sind. Die Namen für die Zugriffsmethoden können entweder manuell gewählt oder automatisch generiert werden.
Eine Aktivität, um die Zielklassenhierarchie auf bereits vorhandene Methoden mit den gewählten Namen zu prüfen, wurde bereits im Kapitel 4.1 unter A70 beschrieben.

A 230: *Get-* und *set-*Methoden in Zielklasse erstellen

Nachdem die Namen für die Zugriffsmethoden eindeutig sind, können die beiden Methoden in der Zielklasse erstellt werden, indem man zwei neue *Method*-Objekte erzeugt.
Handelt es sich bei dem zu verschiebenden Feld um ein statisches Feld, also eine Klassenvariable, müssen die Zugriffsmethoden ebenfalls als *static* gekennzeichnet werden. Auch hier wird die gleiche Erweiterung des DMM benutzt wie bereits im Kapitel 4.1.
Diese neuen Objekte benötigen außerdem noch Assoziationen zu ihrer enthaltenden Klasse sowie zu ihren Parametern und Rückgabewerten. Bei der get-Methode entspricht der Rückgabewert dem Typ des Feldes. Bei der set-Methode ist der übergebene Parameter im einfachsten Fall auch der Typ des Feldes. Andere Fälle, bei denen der oder die Parameter nicht dem Typ des Feldes entsprechen, können anhand eines modellbasierten Refactorings nur schlecht berücksichtigt werden. Diese müssen vom Anwender des Refactoring manuell erstellt beziehungsweise nachbearbeitet werden.

Den Modellausschnitt, um die beiden Methoden in der Zielklasse zu erstellen, zeigt Abbildung 47. Die notwendige GReQL-Anfrage zur Ermittlung des Typs des Feldes `discount` aus dem Beispiel auf Seite 19 zeigt Abbildung 48.

A 240: Feld für Zielobjekt in der Ausgangsklasse erstellen

Damit das Feld in der neuen Zielklasse auch weiterhin von der Ausgangsklasse erreichbar ist, muss in der Ausgangsklasse ein eigenes Feld für das Zielobjekt vorhanden sein. Fowler beschreibt dies in [Fow05] folgendermaßen:

> „Ein vorhandenes Feld oder eine vorhandene Methode kann Ihnen das Ziel liefern. Wenn das nicht der Fall ist, prüfen Sie, ob Sie leicht eine Methode erstellen können, die dies leistet. Schlägt dies fehl, so müssen Sie ein neues Feld in der Ausgangsklasse erstellen, das das Ziel speichern kann.“

Wird dieser Schritt von einem Programmierer manuell vorgenommen, ist dies sicher der geeignete Weg, um an das Zielobjekt zu gelangen. Dies mit einem Refactoringtool automatisch durchzuführen ist jedoch nur schwerlich möglich. Eine Methode zu generieren, welche das Zielobjekt liefert, wäre zwar

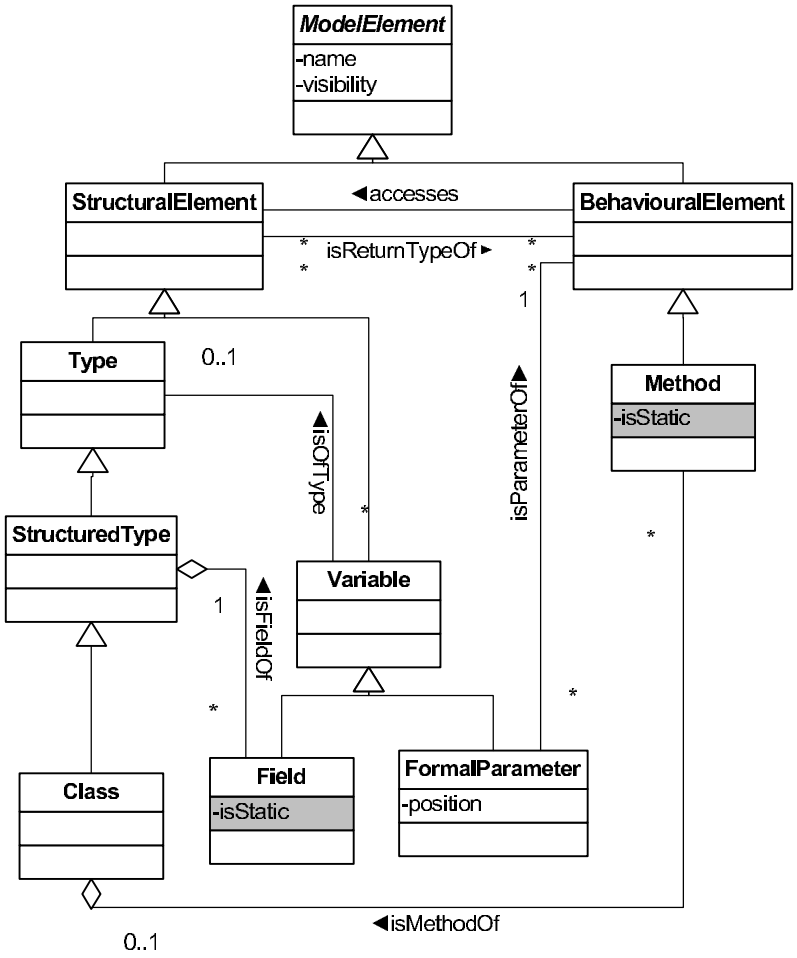

Abbildung 47: Metamodellausschnitt zur Ermittlung von Zugriffsmethoden

---

FROM f : V{Field}, c : V{Class}
WITH f.name = `'discount'`
AND c.name = `'Customer'`
REPORT c (<−−{isFieldOf} f (−−>{isOfType}) END

Abbildung 48: GReQL-Anfrage, um den Typ eines Feldes zu ermitteln

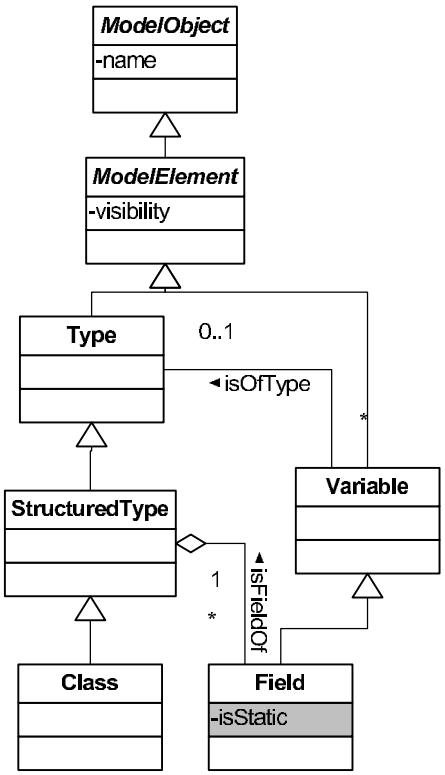

Abbildung 49: Metamodellausschnitt, um das Feld in der Ausgangsklasse zu erstellen

denkbar, aber wie Fowler bereits erwähnt, führt dies nicht immer zum Ziel. Bei der Frage nach dem Warum, lässt er den Leser jedoch im Unklaren.
Es lässt sich allerdings sagen, dass nicht sicherzustellen ist, ob eine vorhandene Methode das richtige Zielobjekt liefert, beziehungsweise ob das Zielobjekt auch die gewünschten Daten enthält. So ist es möglich, dass das Zielobjekt zwischenzeitlich von einer anderen Methode gelöscht und neu erstellt wurde und die Daten somit verloren sind.

Das beschriebene Problem tritt auch bei der Verwendung eines bereits vorhandenen Feldes für das Zielobjekt oder einer vorhandenen Methode auf. Um ein bereits vorhandenes Feld für das Zielobjekt zu benutzen oder eine Methode welches dieses liefert, müsste für alle Methoden, die dieses Objekt manipulieren sichergestellt sein, dass der Inhalt des neuen Feldes nicht verloren geht.
Es gibt ein weiteres Problem bei der Benutzung eines vorhandenen Feldes. Angenommen, im Beispiel auf Seite 19 soll das Feld `discount` in die Klasse `Customertype` verschoben werden. Man könnte hier eigentlich das bereits vorhandene Attribut `type` zu verwenden, um weiterhin von der Ausgangs-

klasse auf das verschobene Feld zuzugreifen. Das sich hier ergebende Problem ist, dass der erste Zugriff auf das verschobene Attribut erfolgt bevor das type-Objekt überhaupt initialisiert ist. Da mit dem DMM keine Aussagen über zeitliche Abläufe von Aufrufen und Zugriffen gemacht werden können, lässt sich dieses Problem damit im Vorhinein auch nicht feststellen.

Es bleibt für das modellbasierte Refactoring also lediglich die Variante, ein neues Feld für das Zielobjekt zu erstellen. Da es wie bereits vorher beschrieben im DMM keine Möglichkeit gibt, den ersten Zugriff auf das verschobene Feld zu ermitteln, sollte das neu erstellte Feld für das Zielobjekt auch gleich bei der Deklaration initialisiert werden. Den Metamodellausschnitt zu Erstellung des Feldes in der Ausgangsklasse zeigt Abbildung 49. Würde im Beispiel auf Seite 16 also das Feld `discount` in die Klasse `Customertype` verschoben, würde die Klasse `Customer` und ihr Konstruktor nach diesem Schritt aussehen wie in Abbildung 50.

```
public class Customer
{
        private Customertype newCustomertype = new Customertype();
        private Customertype type;
        private double sales;
        private static double baseDiscount = 2;
        //some other attributes like a name and a unique number etc.

        public Customer(Customertype type)
        {                                                              10
                newCustomertype.set_discount(1.5);
                this.type = type;
                this.sales = 0;
        }
        //.......... some other code ..........
}
```

Abbildung 50: In der Klasse `Customer` wird ein neues Feld für das Zielobjekt erstellt

A 250: Alle Referenzen auf das Feld in der Ausgangsklasse suchen

Zuletzt sind noch die Referenzen auf das Feld anzupassen. Dazu müssen diese zunächst gefunden werden. Hierzu bietet das DMM die bereits zuvor genutzte *accesses*-Relation. Mit der GReQL-Anfrage in Abbildung 51 erhält man für das Codebeispiel auf Seite 19 alle Methoden, die auf das Feld `discount` zugreifen.

```
FROM   m : V{Method}, f : V{Field}, c : V{Class}
WITH f.name = 'discount'
AND   c.name = 'customer'
AND   c (<--{isFieldOf)} f (<--{accesses}) m
REPORT m END
```

Abbildung 51: GReQL-Anfrage, um alle Referenzen auf ein Feld zu finden

A 260: Zugriffe durch Aufruf der Zugriffs-Methoden ersetzen

Sind alle Zugriffe auf das verschobene Feld bestimmt, müssen diese durch den Aufruf der Zugriffsmethoden ersetzt werden. Die einzigen beiden Methoden, welche direkt auf das Feld zugreifen, sind die zuvor erstellten Zugriffsmethoden selbst.

Zum Ersetzen müssen in der Ausgangsklasse die *accesses*-Relationen auf das Feld durch *invokes*-Relationen auf die entsprechende Methode der Zielklasse ersetzt werden.

Dieser Schritt ist im DMM nicht durchzuführen, da es dort keine Möglichkeit gibt, zwischen lesendem und schreibendem Zugriff zu unterscheiden. Somit kann auch nicht bestimmt werden, ob die zu erstellende *invokes*-Relation nun die *get*- oder die *set*-Methode als Ziel bekommt.
Aus diesem Grund wird das DMM hier erneut erweitert und zwei neue Unterklassen der *accesses*-Relation gebildet. Eine Klasse mit der Bezeichnung *Uses* und eine weitere mit der Bezeichnung *Sets*. Mithilfe dieser neu eingeführten Klassen kann dann mit dem DMM entschieden werden, um welche Art von Zugriff es sich handelt, da man es bei lesendem Zugriff mit einer *uses*-Relation zu tun hat und bei schreibenden Zugriff mit einer *sets*-Relation. Der für diese Aktivität und gleichzeitig auch für das gesamte Refactoring notwendige Metamodellausschnitt ist in Abbildung 52 zu sehen. Dabei sind die grau hinterlegenden Elemente Erweiterungen des DMM.

Zu beachten ist an dieser Stelle, dass sich dadurch auch die GReQL-Anfrage aus A250 geringfügig ändert, da sich die Referenzen nun durch *gets*- beziehungsweise *uses*-Relationen unterscheiden. Da die lesenden *gets*-Aufrufe auf durch *invokes*-Relationen auf die get-Methode und die schreibenden *uses*-Aufrufe durch *invokes*-Kanten auf die set-Methode ersetzt werden, wird für jede Referenz eine eigene GReQL-Anfrage benutzt. Diese beiden Anfragen sind in den Abbildungen 53 und 54 gezeigt und ersetzen die einzelne Anfrage aus Abbildung 51.

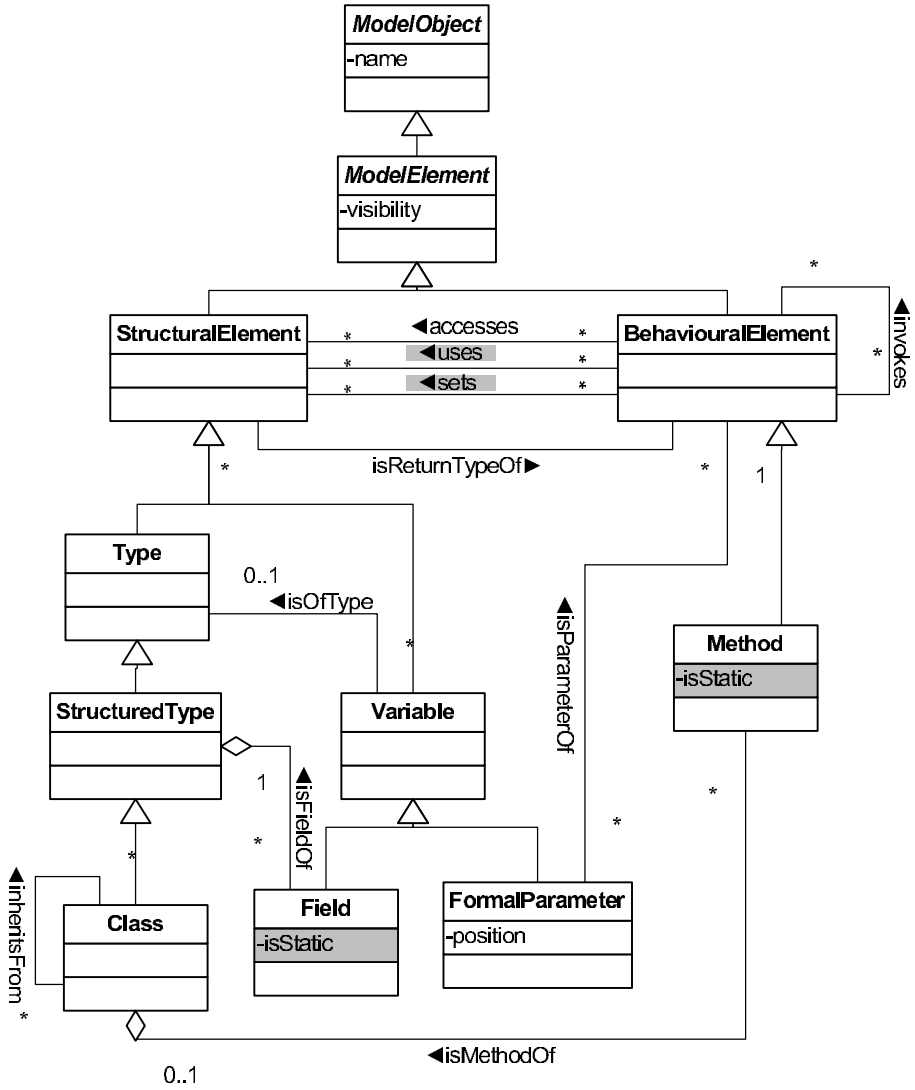

Abbildung 52: Metamodell für das Refactoring *Attribut verschieben*

FROM m : V{Method}, f : V{Field}, c : V{Class}
WITH f.name = 'discount'
AND c (<--{isFieldOf)} f (<--{uses}) m
REPORT m END

Abbildung 53: GReQL-Anfrage um alle lesenden Zugriffe auf ein Feld zu finden

```
FROM   m : V{Method}, f : V{Field}, c : V{Class}
WITH f.name = 'discount'
AND   c (<--{isFieldOf)} f (<--{sets}) m
REPORT m END
```

Abbildung 54: GReQL-Anfrage um alle schreibenden Zugriffe auf ein Feld zu finden

Zusammenfassung

In diesem Kapitel wurde das Refactoring *Attribut verschieben* in kleinere Aktivitäten zerlegt und diese detailliert erklärt. Wie schon bei dem Refactoring *Methode verschieben* in Kapitel 4.1 wurden für jede Aktivität Anforderungen an ein Metamodell formuliert und das Dagstuhl Middle Metamodell auf die Erfüllung dieser Anforderungen überprüft. Aus diesen einzelnen Metamodellen kann schließlich ein Gesamtmodell erstellt werden, welches in Abbildung 52 zu sehen ist. Die Metamodelle der beiden Refactorings *Methode verschieben* (S. 41) und *Attribut verschieben* S. 51 sind bis auf die Erweiterung um *uses-* und *sets*-Relationen bei *Attribut verschieben* (vgl. A260 auf Seite 50) gleich, so dass sie bei Bedarf leicht zu einem gemeinsamen Metamodell verschmolzen werden können.

4.3 Eigenes Attribut kapseln und Attribut kapseln

Der folgende Abschnitt beschreibt die Zerlegung der beiden Refactorings *Attribut kapseln* und *eigenes Attibut kapseln* in einzelne Aktivitäten. Zur Durchführung der einzelnen Aktivitäten werden für diese beiden Refactorings Anforderungen an ein Metamodell beschrieben. Schließlich wird dafür die Eignung des *Dagstuhl Middle Metamodel* geprüft. Auf bereits in anderen Refactorings beschriebene Aktivitäten wird an den entsprechenden Stellen verwiesen.

Die beiden Refactorings *Eigenes Attribut kapseln* und *Attribut kapseln* sind sich sehr ähnlich. Im Falle von *Attribut kapseln* ist jedoch zusätzlich darauf zu achten, dass man es hier mit einem öffentlichen Attribut zu tun hat und entsprechend nicht nur in der Klasse selbst nach Referenzen auf das Attribut suchen muss, sondern Zugriffe prinzipiell von überall her kommen können. Für das Refactoring auf Modellbasis wirken sich die Unterschiede der beiden Refactorings nicht aus, da es auf dieser Ebene gleich ist, von welcher Klasse aus via *accesses*-Kanten auf ein Attribut zugegriffen wird. Weiterhin muss das Attribut am Ende des Refactorings *Attribut kapseln* noch als privat deklariert werden, was im Falle von *Eigenes Attribut kapseln* bereits vorher der Fall war.

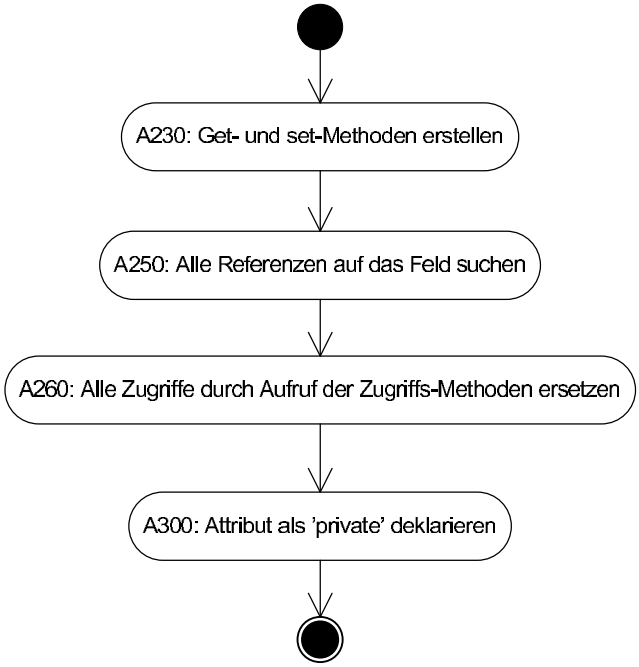

Abbildung 55: Aktivitätsdiagramm *(Eigenes) Attribut kapseln*

Die für das Refactoring *Kapsele Attribut* notwendigen Aktivitäten sind im Aktivitätsdiagramm in Abbildung 55 dargestellt und werden in den folgenden Unterabschnitten beschrieben, sofern das nicht bereits in früheren Refactorings geschehen ist.

A230: Get- und set-Methoden erstellen

Um ein Attribut zu kapseln, müssen im ersten Schritt Zugriffsmethoden dafür erstellt werden. Diese Aktivität wurde bereits für das Refactoring *Attribut verschieben* in A230 benötigt und kann an dieser Stelle wiederverwendet werden. Es kann natürlich sein, dass bereits Zugriffsmethoden vorhanden waren, diese aber nicht von allen Methoden benutzt werden. Mit dem Dagstuhl Middle Metamodel und aufgrund der nicht einheitlichen Syntax für Zugriffsmethoden in den verschiedenen Programmiersprachen, wohl auch mit anderen Modellen, lässt sich das Vorhandensein solcher Methoden nicht erkennen. Diese Fälle müssten vom Anwender des Refactoring selbst berücksichtigt werden, da ansonsten zu den ursprünglichen Zugriffsmethoden zwei weitere erstellt werden, an die die alten Zugriffsmethoden dann delegieren.

A250: Alle Referenzen auf das Feld suchen

Im zweiten Schritt muss herausgefunden werden, welche anderen Elemente alle auf das zu kapselnde Feld zugreifen. Die Suche nach solchen Referenzen wurde bereits im Refactoring *Attribut verschieben* in A250 beschrieben.

A260: Alle Zugriffe durch Aufruf der Zugriffs-Methoden ersetzen

sind alle Referenzen gefunden, müssen diese wie schon bei *Attribut verschieben* alle durch Aufrufe der entsprechenden Zugriffsmethoden ersetzt werden. Dies führt auch an dieser Stelle zu den im Kapitel 4.2 unter A260 beschriebenen Problemen, dass sich mit dem DMM nicht entscheiden lässt, ob es sich um einen lesenden oder schreibenden Zugriff handelt. Aus diesem Grund muss auch an dieser Stelle auf die Erweiterung des DMM zurückgegriffen werden, wie sie bereits in der Aktivität A260 gemacht wurden, so dass die *accesses*-Relation in eine *uses*- und eine *sets*-Relation spezialisiert wird.

A300: Attribut als 'private' deklarieren

Im letzten Schritt muss die Sichtbarkeit des Attributs noch als 'private' deklariert werden, um einen zukünftigen direkten Zugriff darauf zu verhindern.

Zusammenfassung

Die beiden Refactorings *Attribut kapseln* und *eigenes Attribut kapseln* besitzen nur eine Aktivität, die nicht aus anderen Refactorings wiederverwendet werden kann. Alle anderen Aktivitäten wurden bereits in dem Refactoring *Attribut verschieben* benötigt. Aus diesem Grund kann für die beiden in diesem Kapitel beschriebenen Refactorings auch das gleiche Metamodell wie für das Refactoring *Attribut verschieben* benutzt werden. Dieses Metamodell zeigt Abbildung 52 auf Seite 51.

4.4 Zusammenfassung

In den vorangegangenen Unterkapiteln wurden die drei Refactorings *Methode verschieben*, *Attribut verschieben* und *(Eigenes) Attribut kapseln* in einzelne Aktivitäten zerlegt. Für diese Aktivitäten wurden Anforderungen an ein Metamodell definiert und aus dem Dagstuhl Middle Metamodell Metamodellausschnitte präsentiert, welche diese Anforderungen erfüllen. Am Ende eines jeden Refactorings wurde aus den einzelnen Metamodellausschnitten ein Gesamtmodell entwickelt, mit welchem sich das jeweilige Refactoring unter Beachtung der beschriebenen Randbedingungen durchführen lässt. Dabei hat sich gezeigt, dass sich das DMM sehr gut zum modellbasierten Refactoring eignet.
Ein Gesamtmodell für alle beschriebenen Refactorings wird an dieser Stelle noch nicht präsentiert, dies erfolgt im Kapitel 6.2 Würdigung des *Dagstuhl Middle Metamodel* auf Seite 85, da dort auch die Metamodelle für die Erkennung von Bad Smells einfließen können, die im nächsten Kapitel vorgestellt werden.

5 Erkennung von Bad Smells auf Modellebene

Als *Bad Smells* (dt. „Schlechte Gerüche“) bezeichnet man Stellen in einer Software, die auf einen schlechten Entwurf hindeuten. Eine solche Stelle enthält dabei keineswegs einen echten Programmfehler oder eine ineffiziente Implementation, sondern sie zeichnet sich lediglich durch bestimmte Eigenschaften aus, die ein Refactoring der Programmstelle nahelegen, um die Lesbarkeit und Erweiterbarkeit zu optimieren.
Die Bad Smells liefern damit Anstoßpunkte für die Durchführung von Refactorings. Beispiele für solche Gerüche können lange Methoden oder duplizierter Code sein. Eine Liste von Bad Smells und die darauf anzuwendenden Refactorings findet sich in [Fow05].

Die Wahl der Metapher der schlechten Gerüche gibt bereits einen Hinweis auf die Eigenschaft der Entwurfsmängel. Denn wie auch Gerüche stellt schlechtes Design eine sehr vage und subjektive Eigenschaft von Software dar. So kann in einem Fall eine Methode mit 10 Zeilen Code schon zu lang sein, während in einem anderen Fall eine Methode mit 100 Codezeilen durchaus notwendig und sinnvoll ist. Aus diesem Grund weist Fowler in [Fow05] völlig zurecht darauf hin, dass

> *„... nach unseren Erfahrungen kein System von Metriken die informierte menschliche Intuition erreicht.“*

Es wird im Rahmen dieser Arbeit dennoch untersucht, inwiefern sich das *Dagstuhl Middle Metamodel* dazu eignet, Bad Smells auf Modellbasis zu finden. Dazu folgt im nächsten Unterabschnitt zunächst eine Beschreibung der Bad Smells und die dadurch entstehenden Probleme nach Martin Fowler [Fow05]. Danach werden, wie bereits bei den Refactorings im vorausgehenden Kapitel, Anforderungen an ein Metamodell definiert. Sollte sich ein Bad Smell mit dem DMM beziehungsweise einer Erweiterung des DMM nicht aufspüren lassen, wird an entsprechender Stelle darauf hingewiesen, sowie kurz auf die Gründe dafür eingegangen.
Kann ein Bad Smell anhand eines Modellausschnitts des DMM erkannt werden, wird dieser Modellausschnitt präsentiert. Beispielhaft wird dann eine GReQL-Anfrage angegeben, um den Bad Smell aus einer konkreten Instanz dieses Modellausschnitts zu ermitteln.

BS 1 Duplizierter Code [Duplicated Code]

Einer der am häufigsten anzutreffenden Bad Smells ist duplizierter Code [Fow05], also Code, der an mehreren Stellen einer Software in gleicher oder sehr ähnlicher Weise anzutreffen ist. Dies kann unter anderem dadurch entstehen, wenn Software später um ähnliche Klassen oder Subklassen erweitert wird und diese durch Copy-Paste-Techniken eingeführt werden. Das Problem

an dupliziertem Code liegt in der schwierigen konsistenten Wartung. Denn wenn an einer Stelle der Code geändert werden muss, passiert es häufig, dass er auch in allen anderen, durch Copy-Paste-Verfahren entstandenen Klassen, geändert werden muss, diese aber oft nur schwer zu finden und nachzuvollziehen sind.

Das Problem mit dupliziertem Code besteht nicht erst seit Einführung der Objektorientierung, sondern vielmehr bereits seit dem Beginn der Softwareentwicklung. Aus diesem Grund wurden für die Erkennung von dupliziertem Code bereits die verschiedensten Verfahren auf den unterschiedlichsten Sichten auf Software entwickelt. In [Bak97] finden sich Algorithmen zur Duplikaterkennung auf Zeichenebene, während Ira D. Baxter in [BYM+98] Verfahren zur Klonerkennung auf Basis der abstrakten Syntaxbäume vorstellt.
Einen wiederum anderen Ansatz stellt Daniel Ratiu in [RDGM04] vor, denn hier wird zusätzlich auf Versionsinformationen zurückgegriffen. Also ob die Gleichheit zweier Codeabschnitte im Laufe der Zeit zu- oder abnimmt.

Aufgrund der verschiedenartigen Möglichkeiten, sich diesem Bad Smell zu nähern, fällt es an dieser Stelle schwer, Anforderungen an ein Metamodell zu formulieren, um ihn entdecken zu können. Aus diesem Grund werden hier nur ansatzweise Möglichkeiten aufgezeigt, diesen Bad Smell auf Basis des Dagstuhl Middle Metamodel zu finden.

Zwei Probleme gibt es bei der Erkennung von Klonen mit dem DMM. Einerseits ist es im DMM nicht möglich, nur bestimmte Codeabschnitte zu untersuchen, denn man kann nur die im DMM auch vorhandenen Modellelemente verwenden, also nur komplette Klassen oder ganze Methoden. Andererseits sind im DMM keine Kontrollflüsse modelliert und es lässt sich bei zwei Methodenaufrufen nicht feststellen, welcher der beiden zuerst erfolgt.
Es bleibt aber die Möglichkeit, eine rudimentäre Klonerkennung durchzuführen, indem man überprüft, ob die *invokes*-Kanten zweier Methoden die gleichen Ziele haben. Das heißt, zwei verschiedene Methoden rufen beide die gleichen anderen Methoden auf. Hat man eine solche Stelle gefunden, muss dies nicht bedeuten, dass man ein Duplikat vorliegen hat, man kann die weitere Duplikatsuche auf einer feingranulareren Ebene jedoch auf solche Fundstellen beschränken und muss somit weniger Fälle berücksichtigen. Dies gilt wie bereits oben erwähnt nur, wenn man die Suche nach Duplikaten auf Methoden oder Klasse beschränkt und nicht einzelne Codeabschnitte betrachten möchte.

Der Metamodellausschnitt des DMM, um eine solche Duplikaterkennung durchführen zu können, ist in Abbildung 56 dargestellt.
Die in Abbildung 57 gezeigte GReQL-Anfrage liefert als Ergebnis Methoden, welche die gleichen anderen Methoden aufrufen. Die GReQL-Anfragen, um

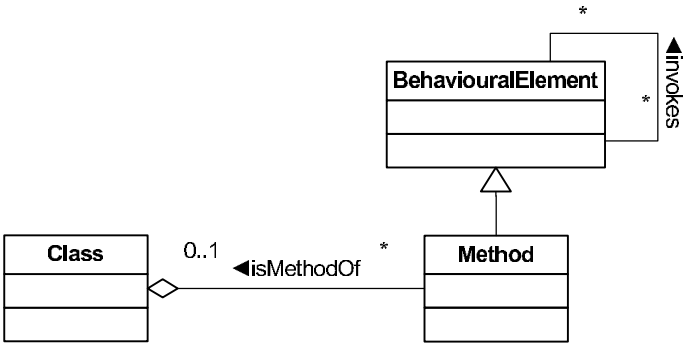

Abbildung 56: Metamodellausschnitt, um Methoden mit gleichen ausgehenden *invokes*-Kanten zu finden

```
FROM m1,m2 : V{Method}, c1,c2: V{Class}
WITH  c1 <> c2
AND   m1 (-->{isMethodOf}) c1
AND   m2 (-->{isMethodOf}) c2
AND   m1 (-->{invokes}) = m2 (-->{invokes})
REPORT c1,m1,c2,m2 END
```

Abbildung 57: GReQL-Anfrage, um Methoden mit gleichen ausgehenden *invokes*-Kanten zu finden

ganze duplizierte Klassen zu finden, können in ähnlicher Weise erstellt werden.

Mit dem DMM lassen sich also nur Vermutungen für duplizierten Code anstellen und dies auch nur auf Methoden- oder Klassenebene.

BS 2 Lange Methoden [Long Method]

Eine zu lange Methode ist oft sehr schwer verständlich und sollte in mehrere kleinere Methoden zerlegt werden. Dies aus dem Grund, da kurze Methoden besser wiederverwendet oder gewartet werden können. Das Problem dabei ist meist die Entscheidung, wann eine Methode zu lang ist. So kann es vorkommen, dass ein Methode mit über 50 Codezeilen durchaus ihre Berechtigung hat und nicht zu lang ist, wobei auf der anderen Seite eine Methode existieren kann, bei der 10 Zeilen bereits aufgeteilt werden sollten. So ist es nach Martin Folwer nicht die Anzahl der Codezeilen, die eine Methode zu einer langen Methode machen, sondern vielmehr die Semantik [Fow05].

Mit dem DMM lässt sich über die Länge beziehungsweise Überlänge einer Methode kaum eine Aussage treffen. Man kann hier allenfalls Vermutungen anstellen. Diese Vermutungen können einerseits auf die Anzahl der Assoziationen der Methode getroffen werden oder andererseits über das triviale Maß der *Lines of Code* (LOC). Wobei die LOC kein besonders geeignetes Maß darstellen, um lange Methoden zu finden, da es keine absolute Obergrenze dafür gibt.

Um die LOC einer Methode festzustellen, muss entweder die Länge dieser Methode im Metamodell gespeichert werden können oder aber die Zeilennummer des Methodenanfangs und des -endes. Das DMM geht dabei den zweiten Weg, da es jedes ModelObject mit einem SourceObject in Relation setzt und dabei die Start- und die Endzeile eines Elements speichert. Bei der Anzahl der Relationen einer Methode zählt nur die Anzahl der aufgerufenen Methoden, da nur diese Einfluss auf die Semantik einer Methode haben. Den erforderlichen Metamodellausschnitt um die beiden gewünschten Anforderungen zu erfüllen, zeigt Abbildung 58.

Um anhand des Metamodells aus Abbildung 58 die erforderlichen Informationen zu beschaffen, sind die beiden GReQL-Anfragen aus Abbildung 59 und 60 nötig. Die Anfrage in Abbildung 59 ermittelt alle Methoden, deren LOC über der im Parameter `minLOC` angegebenen Zahl liegt. Bei der Anfrage in Abbildung 60 wird die Anzahl der ausgehenden *invokes*-Kanten mit dem Parameter `minDegree` verglichen. Ist die Anzahl der Kanten größer als der Parameter, wird die Methode mit in die Ausgabe übernommen und eine Überlänge vermutet. Absolute Zahlen für die maximale Anzahl an LOC

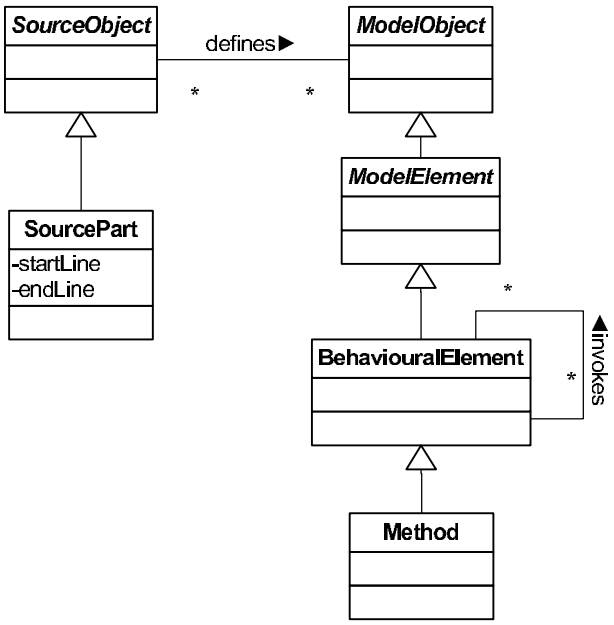

Abbildung 58: Metamodellausschnitt zur Ermittlung *langer Methoden*

oder für *invokes*-Kanten zu bestimmen ist nicht allgemeingültig möglich und sollte für jede Software individuell angepasst werden. Insbesondere hängen diese Zahlen von der verwendeten Programmiersprache ab.

USING minLOC
FROM m : V{Method} s : V{SourcePart}
WITH m (<−−{defines} s
AND (s.endLine − s.startLine) > minLOC
REPORT m END

Abbildung 59: GReQL-Anfrage, um Methoden mit mehr Zeilen als in `minLOC` angegeben zu finden

---

USING minDegree
FROM m : V{Method}
WITH outdegree{invokes}(m) > minDegree
REPORT m END

Abbildung 60: GReQL-Anfrage, um Methoden mit mehr *invokes*-Relationen als in `minDegree` angegeben zu finden

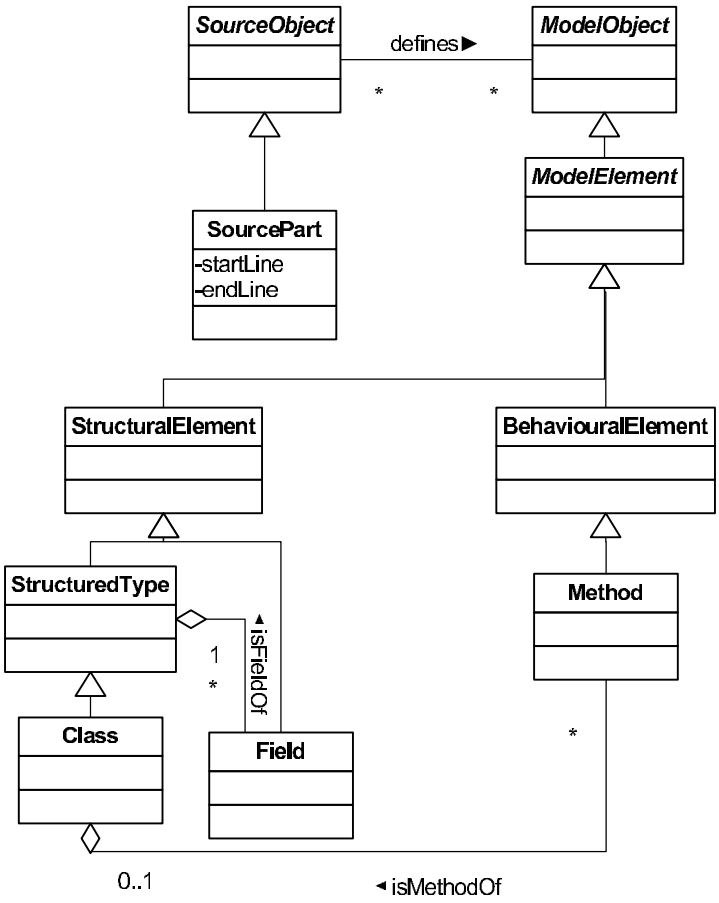

Abbildung 61: Metamodellausschnitt zur Ermittlung *großer Klassen*

BS 3 Große Klassen [Large Class]

Eine große Klasse zeichnet sich dadurch aus, dass sie für zu viele Probleme zuständig ist und häufig ist es sinnvoll, solch eine zu große Klasse in mehrere kleinere Klassen aufzusplitten. Ein Anzeichen für eine zu große Klasse kann eine hohe Anzahl Attribute oder Methoden sein. Diese sollten mit dem Metamodell zu bestimmen sein. Ein weiterer Punkt kann, wie auch schon bei *langen Methoden*, die Anzahl der Codezeilen einer Klasse sein, weshalb dies durch das Metamodell überprüfbar sein sollte.

Letztlich sind dies nur sehr *schwache* Merkmale und die Entscheidung, ob es sich um eine *zu großen Klasse* handelt, lässt sich besser auf einer feingranulareren Ebene, als auf Middle Model-Niveau treffen. Den Metamodellausschnitt, um die zuvor beschriebenen Anforderungen zu erfüllen, zeigt Abbildung 61.
Um Klassen mit einer großen Anzahl an Attributen oder Methoden zu fin-

den, kann die GReQL-Anfrage aus Abbildung 62 verwendet werden. Die einzustellenden Parameter `maxMeth` und `maxAttr` müssen auch hier für jede Software und die zugrundeliegende Programmiersprache individuell gewählt werden.

```
USING maxAttr, maxMeth
FROM c : v{ Class}
WITH  cnt(c (<−−{isMethodOf})) > maxMeth
OR    cnt(c (<−−{isFieldOf})) > maxAttr
REPORT c END
```

Abbildung 62: GReQL-Anfrage, um Klassen mit vielen Attributen oder Methoden zu finden

Abbildung 63 zeigt die GReQL-Anfrage, um alle Klassen zu ermitteln deren LOC größer sind als der im Parameter `minLOC` angegebene Wert.

```
USING minLOC
FROM c : V{Class} s : V{SourcePart}
WITH c (<−−{defines} s
AND (s.endLine − s.startLine) > minLOC
REPORT c END
```

Abbildung 63: GReQL-Anfrage, um Klassen mit mehr als im Parameter `minLOC` angegebenen Zeilen zu finden

BS 4 Lange Parameterlisten [Long Paramter List]

Eine lange Parameterliste bedeutet, dass einer Methode eine große Anzahl an Parametern übergeben werden müssen. Solche langen Parameterlisten sind meist nur schwer verständlich und sollten daher vermieden werden. Dies ist durch mehrere Maßnahmen möglich. So zum Beispiel durch das Einführen eines Parameterobjektes, welches die gewünschten Parameter enthält oder dadurch, dass die Methode in die Lage versetzt wird, sich die benötigten Daten selbst zu besorgen.

Meist entstehen lange Parameterlisten nicht bei der ersten Programmierung einer Methode, sondern wachsen vielmehr in der Wartungsphase, wenn einer Methode zusätzliche Funktionalität hinzugefügt wird.

Um eine lange Parameterliste erkennen zu können, muss das Metamodell Möglichkeiten bieten, jeder Methode ihre Parameter eindeutig zuzuordnen.

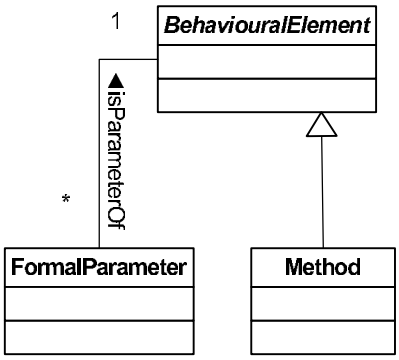

Abbildung 64: Metamodellausschnitt, um lange Parameterlisten zu erkennen

Im DMM ist hierfür die *isParameterOf*-Relation vorgesehen, die jeder Methode ihre Parameter zuordnet. Der erforderliche Modellausschnitt des DMM ist in Abbildung 64 gezeigt.
Um nun aus einer Modellinstanz des Metamodells aus Abbildung 64 Methoden mit langen Parameterlisten zu ermitteln, kann die GReQL-Anfrage aus Abbildung 65 benutzt werden. Diese Anfrage ist parametrisierbar, das heißt, in dem Parameter `maxPara` kann eine frei wählbare Anzahl von Parametern angegeben werden, so dass die Anfrage alle Methoden mit mehr Parametern liefert.

```
USING maxPara
FROM m : V{Method}
WITH indegree{isParameterOf}(m) > maxPara
REPORT m END
```

Abbildung 65: GReQL-Anfrage, um alle Methoden mit mehr Parametern als in `maxPara` angegeben zu ermitteln

Die Funktion `indegree` liefert dabei die Anzahl der eingehenden Kanten vom Typ *isParameterOf*. Lange Parameterlisten können mit dem DMM erkannt werden, allerdings muss der Anwender durch `maxPara` selbst bestimmen, ab wann eine Parameterliste als *lang* gilt. Nach Fowler gewinnt man bereits etwas, wenn man zwei oder mehr Parameter durch ein Parameterobjekt ersetzt, da der Aufruf der Methode verständlicher wird.

BS 5 Divergierende Änderungen [Divergent Change]

Von divergierenden Änderungen wird dann gesprochen, wenn eine Klasse **häufig geändert** wird und bei bestimmten Änderungen immer die **gleichen Methoden und Attribute dieser Klasse** betroffen sind und auch geändert werden müssen. Ist dies der Fall, sollten diese Methoden und At-

tribute nach Möglichkeit in eine eigene Klasse verschoben werden [Fow05].

Um den Bad Smell *Divergierende Änderungen* auf Modellbasis festzustellen, müsste das Metamodell die Möglichkeit der Versionsverwaltung bieten, wie dies spezielle Systeme wie CVS[7] oder das neuere SVN[8] leisten. Da dies im Dagstuhl Middle Metamodel nicht vorgesehen ist, kann dieser Code-Smell damit auch nicht erkannt werden.

BS 6 Schrotkugeln herausoperieren [Shotgun Surgery]

Schrotkugeln herausoperieren bezeichnet ein ähnliches Problem wie schon divergierende Änderungen. Hierbei sind allerdings die **Methoden mehrerer Klassen** betroffen, wenn lediglich **eine Änderung** durchgeführt wird. Das Problem daran ist, dass die Software bei einer Änderung an vielen Stellen angepasst werden muss. Diese Stellen sind oft nur schwer aufzufinden und können somit leicht vergessen oder übersehen werden. Deshalb sollte versucht werden, alle diese Stellen in einer Klasse zusammenzufassen.

Aus den gleichen Gründen wie bereits der Bad Smell *Divergierende Änderungen* kann auch *Schrotkugeln herausoperieren* nicht mit dem DMM erkannt werden.

BS 7 Neid [Feature Envy]

Ist eine Klasse zu sehr an den Elementen einer anderen Klasse interessiert, leidet diese Klasse unter dem Bad Smell *Neid*. Meist sind das Ziel dieses Interesses die Daten einer anderen Klasse und man sollte überlegen, ob die betreffenden Methoden nicht besser in der Klasse aufgehoben sind, von der auch häufig Daten benötigt werden.
Zurecht weißt Fowler darauf hin (vgl. [Fow05]), dass es einige Entwurfsmuster, wie beispielsweise *Strategie* und *Besucher* gibt (vgl. [Gam96]), welche diesen Bad Smell erzeugen. Dies geschieht jedoch, um den Bad Smell divergierende Änderungen zu bekämpfen, und laut Fowler ist es fundamental, Verhalten so zu verschieben, dass Änderungen nur eine Stelle betreffen. Das heißt, wenn man durch das Verschieben von Verhalten eine Software so gestalten kann, dass Änderungen immer nur eine Stelle betreffen, ist dies in jedem Fall vorzuziehen und andere Code Smells wie in diesem Fall *Neid* können dafür in Kauf genommen werden.

Um feststellen zu können, ob Methoden einer Klasse unter Neid leiden, muss es im Metamodell Möglichkeiten geben, um die Anzahl der Methodenaufrufe

[7] Concurrent Versions System (vgl. http://de.wikipedia.org/wiki/CVS)
[8] Subversion (vgl. http://de.wikipedia.org/wiki/Subversion_(Software))

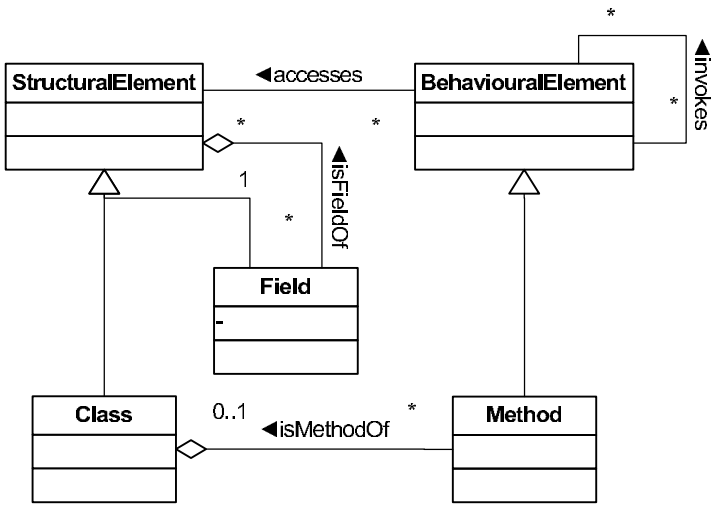

Abbildung 66: Metamodellausschnitt, um den Bad Smell Neid zu erkennen

(bei Zugriffsmethoden) oder die Anzahl der direkten Variablenzugriffe zu ermitteln.
Das DMM bietet hierfür die beiden Relationen *invokes* und *accesses*. Der erforderliche Metamodellausschnitt für die Erkennung dieses Code Smells ist in Abbildung 66 dargestellt.
GReQL-Anfragen, die anhand des in Abbildung 66 gezeigten Metamodells den Bad Smells *Neid* ermitteln können, sind in den beiden Abbildungen 67 und 68 dargestellt.

```
USING maxInvokes
FROM m: V{Method}, c: V{Class}
WITH cnt(     m (-->{invokes}-->{isMethodOf}) c
        AND NOT m (-->{isMethodOf} c) ) > maxInvokes
REPORT m, c END
```

Abbildung 67: GReQL-Anfrage, um Methoden die unter Neid leiden zu ermitteln

Die GReQL-Anfrage aus Abbildung 67 ermittelt für Methoden die Anzahl (Funktion `cnt`) an Methodenaufrufen außerhalb der eigenen Klasse und vergleicht diese mit einem anzugebenden Parameter (`maxInvokes`). Die Ausgabe der Anfrage enthält dann diejenigen Methoden, die externe Methoden öfter als in `maxInvokes` angegeben aufrufen, sowie die Klassen der jeweiligen externen Methoden.
Die in Abbildung 68 dargestellte GReQL-Anfrage ermittelt in ähnlicher Weise die Anzahl an direkten externen Attributzugriffen. Falls diese Anzahl größer als die in `maxAccesses` angegebene Anzahl ist, werden die Methoden und

```
USING maxAccesses
FROM m: V{Method}, c: V{Class}, f:{Field}
WITH cnt(     c (<--{isMethodOf}) m (-->{accesses}) f
        AND NOT f (-->{isFieldOf}) c ) > maxAccesses
REPORT m, f(-->{isFieldOf}) END
```

Abbildung 68: Weitere GReQL-Anfrage, um Methoden die unter Neid leiden zu ermitteln

die Klassen der externen Felder durch die Anfrage ausgegeben.

BS 8 Datenklumpen [Data Clumps]

Als Datenklumpen bezeichnet man solche Daten, die häufig gemeinsam gebraucht werden, zum Beispiel als Parameter in der Signatur einer Methode oder als Felder in einigen Klassen. Um den Code lesbarer zu gestalten, sollten solche Datenklumpen zu einer Klasse zusammengefasst werden. Dies kann bereits im Kleinen geschehen, da man nach Fowler schon *gewonnen* hat, wenn man zwei oder mehr Attribute durch ein neues Objekt ersetzen kann.

Das Metamodell muss zur Suche nach Datenklumpen einerseits die Möglichkeit bieten, die Namen und Typen von Attributen einer Klasse mit denen einer anderen Klasse zu vergleichen. Andererseits muss die Möglichkeit bestehen, die Signaturen von Methoden zu vergleichen.
Im DMM sind diese Informationen durch den in Abbildung 69 dargestellten Metamodellausschnitt ermittelbar.
Die GReQL-Anfragen in Abbildung 70 vergleicht auf Basis des Metamodells beispielsweise die Parameter von unterschiedlichen Methoden auf Typgleichheit. Sind die Typen und die Anzahl der Parameter von Methoden gleich, werden die Methoden, die zugehörigen Klassen und die Parameter durch die GReQL-Anfrage ausgegeben. Vorher ist noch der Parameter `minPara` zu setzen, da es nicht sinnvoll ist, null- oder einstellige Parameterlisten in den Vergleich einzubeziehen. Ansonsten würden alle einstelligen Parameterlisten, in denen beispielsweise nur ein String übergeben wird, als Datenklumpen erkannt werden.

BS 9 Neigung zu elementaren Typen [Primitive Obsession]

Häufig handelt es sich bei der Bearbeitung von Code um Klassen, die aus vielen kleinen elementaren Datentypen (wie z.B. integer, character o.ä.) zusammengesetzt sind. Es hat sich jedoch herausgestellt, dass selbst für kleinere Aufgaben (wie z.B. Geldklassen) eine eigene Klasse oft sinnvoll ist [Fow05].

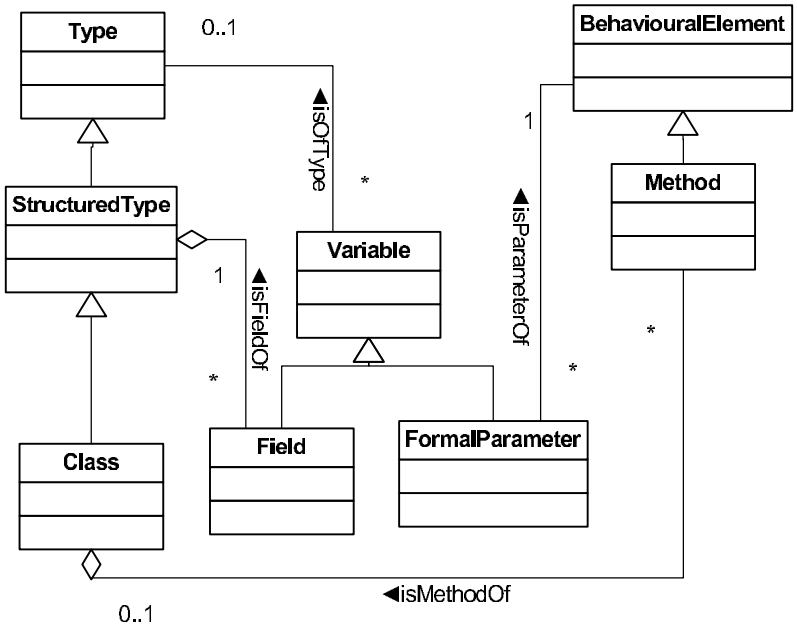

Abbildung 69: Metamodellausschnitt zum Finden von Datenklumpen

```
USING minPara
FROM m1,m2: V{Method}
WITH   m1 <> m2
AND    m1 (<--{isParameterOf} <--{isOfType})
       =
       m2(<--{isParameterOf} <--{isOfType})
AND cnt(m1 (<--{isParameterOf})) > minPara

REPORT   m1 (-->{isMethodOf}),
       m1,                                                    10
       m1(<--{isParameterOf}) END
```

Abbildung 70: GReQL-Anfrage, um gleiche Parameterlisten zu erkennen

Um diesen Bad Smell zu finden, müssten im Metamodell Unterscheidungen zwischen elementaren und anderen Datentypen möglich sein. Diese Unterscheidung wird im DMM nicht getroffen, und somit kann dieser Bad Smell auf Modellbasis mit dem DMM als Metamodell folglich nicht gefunden werden. Außerdem ist die Nutzung in den unterschiedlichen Sprachen, was elementare Typen angeht höchst unterschiedlich. So werden in C# beispielsweise keine dieser Typen mehr benutzt. Aus diesem Grund ist auch eine Erweiterung des DMM an dieser Stelle nicht sinnvoll.

BS 10 `Switch`-Befehle [Switch Statements]

Von Verzweigungen durch `switch`-Befehle rät Martin Fowler gänzlich ab (vgl. [Fow05] S. 76), da solche Verzweigungen häufig an mehreren Stellen im Programm redundant gemacht werden müssen. Er rät vielmehr zur Verwendung des Polymorphismus[9], falls ein switch-Befehl aufgrund eines Typenschlüssels eine Fallunterscheidung trifft.

Dieser Code-Smell lässt sich mit dem DMM nicht finden, da es sich beim DMM um ein statisches Modell handelt, in welchem keine Kontrollstrukturen mehr zu erkennen sind.

BS 11 Parallele Vererbungshierarchien [Parallel Inheritance Hierarchies]

Parallele Vererbungshierarchien sind ein Spezialfall des Bad Smells *Schrotkugeln herausoperieren*. Sie zeichnen sich dadurch aus, dass bei der Bildung einer Unterklasse einer Klasse A gleichzeitig eine Unterklasse einer Klasse B gebildet werden muss. Zu erkennen ist dieser Code-Smell einerseits bei der Durchführung von Änderungen, wenn das oben beschriebene Problem der Unterklassenbildung auftaucht und andererseits nach Fowler daran, dass die Präfixe der Klassennamen in beiden Hierarchien gleich sind.

Um diesen Bad Smell bei der Durchführung von Änderungen auf Basis eines Metamodells zu erkennen, wird, wie bereits bei den beiden Code Smells *Divergierende Änderungen* (vgl. S. 62) und *Schrotkugeln herausoperieren* (vgl. S. 63), eine Versionsverwaltung benötigt. Diese ist nicht im Dagstuhl Middle Metamodel vorhanden.
Die Erkennung auf Basis der Präfixe von Klassennamen kann jedoch zum Teil auf Modellbasis erfolgen. Dazu muss das Metamodell lediglich die Möglichkeit bieten, die Vererbungshierarchie von Klassen zu durchlaufen und

[9] Zitat aus [Bal00]: „Polymorphismus ermöglicht es, den gleichen Namen für gleichartige Operationen zu verwenden, die auf Objekten verschiedener Klassen auszuführen sind. Der Sender muss nur wissen, dass ein Empfängerobjekt das gewünschte Verhalten besitzt. Er muss nicht wissen, zu welcher Klasse das Objekt gehört.“

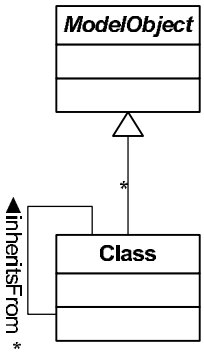

Abbildung 71: Metamodellausschnitt, um die Verebungshierarchie einer Klasse ermitteln zu können

deren Namen auszugeben. Diese Möglichkeit ist durch den Metamodellausschnitt des DMM in Abbildung 71 gegeben. Die eigentliche Erkennung auf namensgleiche Präfixe bei den Klassen kann jedoch nicht direkt mit einer GReQL-Anfrage durchgeführt werden, da in GReQL die Möglichkeiten der Mustererkennung auf Zeichenkettenbasis sehr eingeschränkt sind.
Mit der GReQL-Anfrage in Abbildung 72 werden für eine im Parameter `topClass` anzugebende Klasse alle direkten und indirekten Unterklassen ausgegeben. Die eigentlich Suche nach gleichen Präfixen in dieser Hierarchie erfolgt dann durch andere Werkzeuge.

```
USING topClass
FROM c: V{Class}
REPORT c (<--{inheritsFrom}*) END
```

Abbildung 72: GReQL-Anfrage, um die Klassenhierarchie einer Oberklasse `topClass` auszugeben

BS 12 Faule Klasse [Lazy Class]

Als *Faule Klasse* werden solche bezeichnet, die im Laufe der Zeit ihre Bedeutung verloren haben und kaum oder gar nicht mehr genutzt werden. Diese nutzlosen Elemente sollten in die bestehende Struktur integriert werden.

Um *faule Klassen* bestimmen zu können, gibt es keine eindeutige Metrik. So könnte die Anzahl der benutzten Methoden einer Klasse dazu herangezogen werden oder aber die Anzahl ihrer Konstruktoraufrufe. Um *faule Klassen* auf Modellbasis bestimmen zu können, muss es in einer Modellinstanz möglich sein, die Anzahl der Aufrufe von Methoden beziehungsweise des Konstruktors einer Klasse zu bestimmen. Im DMM ist diese Anzahl durch die eingehenden

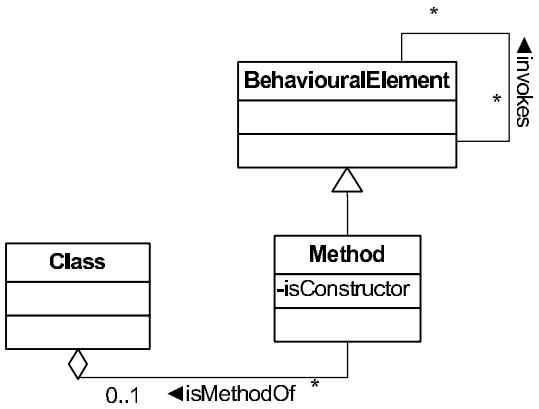

Abbildung 73: Metamodellausschnitt um *faule Klassen* zu ermitteln

invokes-Kanten einer Methode zu ermitteln. Ob es sich bei dem Aufruf um einen Konstruktoraufruf handelt, kann durch das Attribut `isConstructor` ermittelt werden. Den erforderlichen Metamodellausschnitt dazu zeigt Abbildung 73.
Da keine Studien gefunden wurden, welche Methodenaufrufe genau berücksichtigt werden sollten und ab wann eine Klasse als *faule Klasse* gilt, wird hier nur eine GReQL-Anfrage angegeben, die Klassen ermittelt, deren Konstruktor weniger als im Parameter `minInvokes` eingestellt aufgerufen wird. Diese Anfrage ist in Abbildung 74 gezeigt.

```
USING minInvokes
FROM m : V{Method}, c : V{Class}
WITH  m (-->{isMethodOf}) c
AND   m.isConstructor = true
AND   indegree{invokes}(m) < minInvokes
REPORT c END
```

Abbildung 74: GReQL-Anfrage, um wenig genutzte Konstruktoren zu ermitteln

Durch das in Abbildung 73 dargestellte Metamodell sollten sich jedoch auch die anderen Metriken, sofern vorhanden, ermitteln lassen. Anzumerken ist noch, dass nicht die echte Anzahl an Aufrufen zur Laufzeit ermittelt werden kann. Vielmehr handelt es sich um die Anzahl der benutzenden Methoden eines Konstruktors. Es kann also nur ausgesagt werden, dass beispielsweise ein Objekt der Klasse A nur von einer Methode der Klasse B erzeugt wird. Es kann hingegen nicht festgestellt werden, wie oft dies zur Laufzeit auch tatsächlich geschieht. Der Bad Smell *faule Klasse* ist also zur Laufzeit besser

zu bestimmen als anhand eines statischen Modells wie dem DMM.

BS 13 Spekulative Allgemeinheit [Speculative Generality]

Spekulative Allgemeinheit entsteht dadurch, dass im Voraus versucht wird, alle möglichen (und unmöglichen) Fälle zu berücksichtigen. Stellt man irgendwann fest, dass einige dieser Fälle niemals auftreten, sollten diese aus Gründen der Übersichtlichkeit aus der Software entfernt werden. Ansonsten leidet darunter die Verständlichkeit des Codes, da dieser unnötig aufgebläht ist.

Um diesen Bad Smell zu erkennen, müssen Laufzeitinformationen benutzt werden, da erst zur Laufzeit mit Sicherheit entschieden werden kann, ob bestimmte Teile der Software tatsächlich benötigt werden. Bei den wenigen Fällen, wo dies eventuell am statischen Modell entschieden werden könnte, kann es zusätzlich noch zu Problemen durch die Nutzung sogenannter *Testfälle*[10] kommen. Denn es sieht im statischen Modell so aus, als würden bestimmte Teile der Software benutzt, obwohl es sich dabei nur um Tests handelt.

BS 14 Temporäre Felder [Temporary Field]

Temporäre Felder sind Attribute einer Klasse, die nur selten benutzt werden. Dies trägt nicht zur Lesbarkeit des Codes bei, da normalerweise erwartet werden kann, dass ein Objekt alle seine Attribute auch verwendet. Die Entstehung temporärer Felder ist häufig darauf zurückzuführen, dass ein Programmierer für einen bestimmten Algorithmus vermeiden möchte, lange Parameterlisten zu verwenden.

Aus den gleichen Gründen wie bei der *spekulative Allgemeinheit* lassen sich auch temporäre Felder nur schwer bestimmen. Zum einen können einige Testfälle auf diese Felder zugreifen, zum anderen lässt sich die tatsächliche Anzahl der Attributzugriffe nur zur Laufzeit der Software ermitteln. Ein weiteres Problem stellen die Zugriffsmethoden dar, welche im DMM nicht explizit als solche kenntlich gemacht werden können, da die meisten Programmiersprachen sie wie gewöhnliche Methoden behandeln. Es lässt sich deshalb auch nicht ermitteln, wie oft die Zugriffsmethoden aufgerufen werden.

BS 15 Nachrichtenketten [Message Chains]

Als *Nachrichtenketten* bezeichnet man das Traversieren einer Objektstruktur. Das heißt, ein Client fragt Objekt A nach Objekt B, danach fragt er Objekt B nach Objekt C und anschließend Objekt C nach Objekt D. Ein

[10] Als Testfälle bezeichnet man solche Abschnitte des Codes, welche nur zum Testen der Software benutzt werden

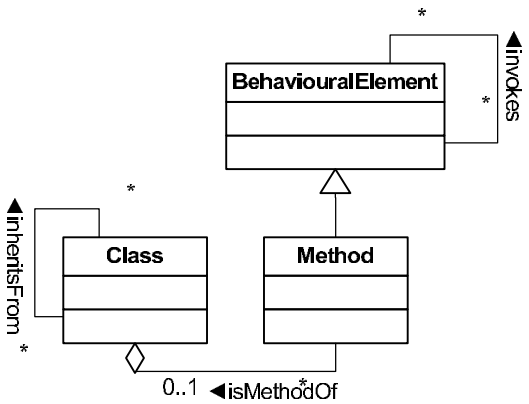

Abbildung 75: Metamodellausschnitt, um Vermittlerobjekte zu ermitteln

Problem ergibt sich, wenn sich beispielsweise die Beziehung zwischen Objekt B und Objekt C ändert. Dann müsste der Client geändert werden, da er nicht mehr über den ursprünglichen Weg an das Objekt C herankommen kann.

Auch dieser Code-Smell lässt sich mit dem DMM nicht erkennen, da für die *invokes*-Relationen keine zeitliche Abfolge modelliert werden kann. Es kann also nicht bestimmt werden, ob ein Objekt D zuerst über eine Nachrichtenkette geliefert werden muss, bevor eine Methode von diesem aufgerufen wird.

BS 16 Vermittler [Middle Man]

Vermittler entstehen dann, wenn eine Klasse zu viele Delegationsmethoden besitzt und damit die Anzahl der Indirektionen zu groß ist. Ist dies der Fall, sollten diese Delegationen entfernt werden und der Aufrufende sich direkt an das betreffende Objekt wenden.

Ein überflüssiger Vermittler ist beispielsweise dann wahrscheinlich, wenn die Mehrzahl an Methoden einer Klasse jeweils nur eine ausgehende *invokes*-Kante besitzen. Dieser Sachverhalt lässt vermuten, dass es sich um ein Delegationsobjekt handelt.
Um diese Situation modellbasiert festzustellen, muss es im Metamodell Möglichkeiten geben, die ausgehenden *invokes*-Kanten einer Methode zu zählen. Ferner muss die Anzahl dieser Methoden mit nur einer ausgehenden invokes-Kante in ein Verhältnis mit der Gesamtzahl der Methoden gebracht werden. Ist dieses Verhältnis größer als ein anzugebender Schwellwert, kann ein Vermittlerobjekt vermutet werden.
Den nötigen Metamodellausschnitt zeigt Abbildung 75. Um aus einer Instanz dieses Metamodellausschnitts ein Vermittlerobjekte zu suchen, kann die GReQL-Anfrage aus Abbildung 76 verwendet werden.

```
USING delegates
FROM c : V{Class}
WITH (cnt(
        FROM m : V{Method}
        WITH m (−−>{isMethodOf} c
        AND outdegree{invokes}(m) = 1
        ) / cnt(
                c (<−−{isMethodOf})
                ) > delegates
REPORT c END
```

Abbildung 76: GReQL-Anfrage, um Vermittlerobjekte zu finden

Dabei gibt der innere FWR-Ausdruck in Abbildung 76 diejenigen Methoden aus, welche nur eine ausgehende *invokes*-Kante besitzen. Die Anzahl dieser Methoden einer Klasse wird ins Verhältnis zur Gesamtzahl der Methoden dieser Klasse gesetzt. Liegt dieses Verhältnis über dem Parameter `delegates`, so wird vermutet, dass es sich um ein Vermittlerobjekt handelt und dieses in die Ausgabemenge übernommen.

Am Besten lassen sich Vermittlerobjekte jedoch am Quellcode selbst erkennen, dies ist jedoch auf Modellbasis nur eingeschränkt möglich.

BS 17 Unangebrachte Intimität [Inappropriate Intimacy]

Wenn sich zwei Klassen sehr intensiv miteinander befassen und gegenseitig häufig Attribute manipulieren oder Methoden aufrufen, bezeichnet man dieses Verhalten als *unangebrachte Intimität*.
Da Klassen nach dem Kapselungsprinzip streng getrennt sein sollten, ist es besser, diese enge Bindung aufzulösen. Dies geschieht beispielsweise, indem die Gemeinsamkeiten in einer zusätzlichen Klasse gekapselt werden oder eine Klasse zur Delegation zwischengeschaltet wird.

Die *Unangebrachte Intimität* kann durch eine hohe Anzahl gegenseitiger Methodenaufrufe festgestellt werden. Für das Metamodell bedeutet dies wiederum, dass Methodenaufrufe und Variablenzugriffe zwischen Klassen im Modell enthalten sein sollten. Der in Abbildung 77 gezeigte DMM-Ausschnitt bietet genau diese Möglichkeiten.
Um mit Hilfe dieses Metamodellausschnitts häufige gegenseitige Methodenaufrufe oder Attributzugriffe zweier Klassen zu finden, kann die GReQL-Anfrage aus Abbildung 78 zum Einsatz kommen. Diese zählt die Anzahl der durch Methodenaufrufe vorhandenen Pfade, beziehungsweise die durch Attributzugriffe vorhandenen Pfade zwischen zwei unterschiedlichen Klassenknoten. Diese Anzahl wird gegen einen mit `minInvokes` anzugebenden

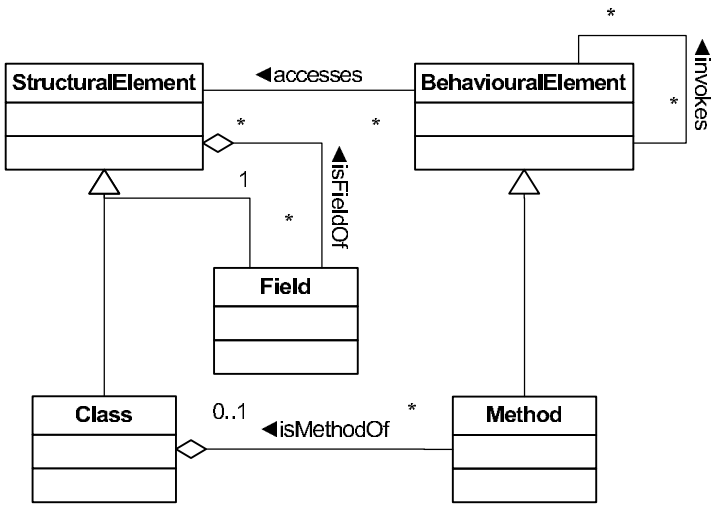

Abbildung 77: Metamodellausschnitt, um *Unangebrachte Intimität* zwischen zwei Klassen zu erkennen

Schwellwert verglichen. Wird dieser Schwellwert überschritten, wird den beiden Klassen eine *unangebrachte Intimität* unterstellt.

```
USING minInvokes
FROM c1,c2: V{Class}
WITH cnt(
        c1 <> c2
        AND c1 (<--{isMethod}
                (-->{invokes}-->{isMethodOf}) |
                (-->{accesses}-->{isFieldOf})
        ) c2) > minInvokes
REPORT c1, c2 END
```

Abbildung 78: GReQL-Anfrage, um *unangebrachte Intimität* zwischen zwei Klassen festzustellen

BS 18 Alternative Klassen mit verschiedenen Schnittstellen [Alternative Classes with Different Interfaces]

Wenn Klassen offensichtlich das gleiche tun, aber unterschiedliche Schnittstellen beziehungsweise Methodensignaturen dazu verwenden, sollte man die Schnittstellen zwecks besserer Lesbarkeit einander angleichen.

Um diesen Bad Smell modellbasiert erkennen zu können, müsste die Semantik von Methoden im Modell vorhanden sein. Dies ist weder beim DMM noch

in anderen Middle Metamodellen der Fall. Daher kann dieser Code Smell mit dem DMM auch nicht erkannt werden.

BS 19 Unvollständige Bibliotheksklasse [Incomplete Library Class]

Unvollständige Bibliotheksklassen findet man recht häufig, da die Ersteller einer solchen Klasse niemals wissen können, welche Wünsche die Anwender der Bibliothek haben. Außerdem kann es vorkommen, dass Bibliotheksklassen aufgrund neuer Entwicklungen unvollständig sind. Da diese in den seltensten Fällen angepasst werden können oder sollten, muss man den gewünschten Code beispielsweise durch die beiden Refactoring *Fremde Methode einführen* oder *Lokale Erweiterung einführen* an anderer geeigneter Stelle selbst bereitstellen.

Dieser Bad Smell ist nur vom Programmierer selbst festzustellen, da die Vollständigkeit einer Bibliothek nur bei der Entwicklung selbst überprüft werden und eine Aussage darüber auch nur für diesen Zeitraum getroffen werden kann.

BS 20 Datenklassen [Data Class]

Als Datenklasse bezeichnet man solche Klassen, die außer ihren Attributen und eventuell zugehörigen `get-` und `set`-Methoden keinen bedeutsamen Inhalt haben. Häufig werden solche Klassen zu stark von anderen Klassen manipuliert und es sollte zusätzliches Verhalten in diese Klassen verschoben werden. Das Ziel sollte sein, der Klasse mehr Verantwortung über ihre eigenen Daten zu geben.

Eine Datenklasse zeichnet sich durch viele Attribute und nur wenige Methoden aus. Um Datenklassen zu erkennen, muss das Metamodell Elemente für Methoden, Attribute und deren jeweilige Zugriffe bieten. Den nötigen DMM-Ausschnitt zeigt Abbildung 79. Wie bereits erwähnt, können Zugriffsmethoden allerdings nicht automatisch als solche erkannt werden, weil sie, wie beispielsweise in Java, wie *normale* Methoden behandelt werden, dies aber für andere Sprachen anders sein kann.
Um dennoch Datenklassen zu bestimmen, könnte man sich einer Heuristik wie in der GReQL-Anfrage in Abbildung 80 dargestellt bedienen.
In dieser wird unterstellt, dass jedes Attribut einer Datenklasse jeweils eine `get-` und eine `set`-Methode besitzt. Es wird dann für jede Klasse geprüft, wie viele Methoden, außer den vermuteten Zugriffsmethoden, sie enthält und diese Anzahl gegen einen im Parameter `minMethods` einzustellenden Wert geprüft. Enthält eine Klasse weniger Methoden als im Parameter gefordert, wird sie als vermutliche Datenklasse in die Ergebnismenge übernommen.

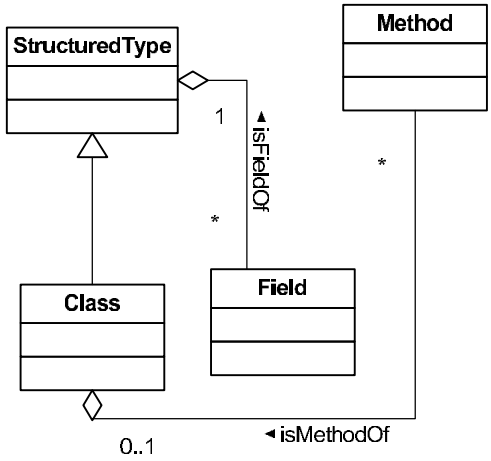

Abbildung 79: Metamodellausschnitt zur Erkennung von *Datenklassen*

```
USING minMethods
FROM c : V{Class}
WITH  cnt(c (<--{isMethodOf}))
      – (2 * cnt(c (<--{isFieldOf})))
      < minMethods
REPORT c END
```

Abbildung 80: GReQL-Anfrage, um Datenklassen zu finden

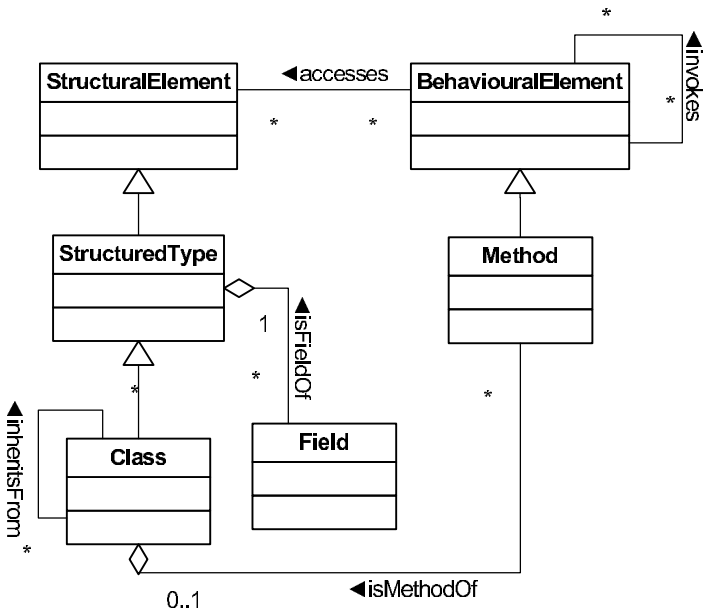

Abbildung 81: Metamodellausschnitt, um *ausgeschlagenes Erbe* von Unterklassen zu erkennen

BS 21 Ausgeschlagenes Erbe [Refused Bequest]

Beim *Ausgeschlagenen Erbe* verwenden die Unterklasssen Methoden ihrer Oberklasse nur teilweise oder gar nicht. Diese nicht oder selten benutzten Methoden sollten dann entweder entfernt oder in der Klassenhierarchie weiter nach unten befördert werden, sofern die Oberklasse selbst diese Methoden nicht benötigt.

Um ein *Ausgeschlagenes Erbe* auf Modellbasis festzustellen, müssen mit dem Metamodell einerseits Methodenaufrufe modelliert werden können, und anderseits muss damit die Klassenhierarchie abzubilden sein. Beides stellt im DMM kein Problem dar und Abbildung 81 zeigt den notwendigen Metamodellausschnitt, um nicht benutzte Methoden beziehungsweise Attribute der Oberklasse zu finden.
In Abbildung 82 ist eine GReQL-Anfrage gezeigt, die als Ergebnis die nicht benutzten Methoden einer Oberklasse und diejenigen Unterklassen, welche die Methoden nicht benutzen, ausgibt. Abbildung 83 zeigt eine ähnliche GReQL-Anfrage, durch diese werden allerdings nicht benutzte Attribute der Oberklasse ausgegeben.

BS 22 Kommentare [Comments]

Nach Fowlers Auffassung sind Kommentare zwar nichts Schlechtes, aber sie deuten häufig auf andere Bad Smells hin. Nach seiner Auffassung ist gu-

```
FROM msuper: V{Method}, sub,super: V{Class}
WITH  sub <> super
AND sub (<--{inheritsFrom}*) super
AND NOT super ( <--{isMethodOf} msuper
                <--{invokes}
                -->{isMethodOf} sub
REPORT sub, msuper END
```

Abbildung 82: GReQL-Anfrage, um von Unterklassen nicht benutzte Methoden zu finden

```
FROM fsuper: V{Field}, sub,super: V{Class}
WITH  sub <> super
AND sub (<--{inheritsFrom}*) super
AND NOT super ( <--{isFieldOf} fsuper
                <--{accesses}
                -->{isMethodOf} sub
REPORT sub, fsuper END
```

Abbildung 83: GReQL-Anfrage, um von Unterklassen nicht benutzte Attribute zu finden

ter Code selbsterklärend und könnte ebensogut auf Kommentare verzichten. Man sollte also überall dort, wo Kommentare stehen, überprüfen, ob nicht gerade der Kommentar einen Bad Smell überdecken soll (Fowler bezeichnet Kommentare in [Fow05] als Deodorant für Bad Smells).

Um diesen Bad Smell zu erkennen, müssten Kommentare zunächst einmal in das Modell integriert werden. Dies ist im DMM zwar möglich, jedoch lässt sich die Semantik der Kommentare im Metamodell nicht ausdrücken, da die Kommentare im Modell nur ihre Position im Quelltext speichern können. Aus diesem Grund lässt sich der Bad Smell Kommentare nicht mit dem DMM finden, da damit nichts über den Inhalt eines Kommentars ausgesagt werden kann.

Zusammenfassung

Die letzten Abschnitte haben gezeigt, dass sich einige Bad Smells in einer Instanz des DMM erkennen lassen, wieder andere nur mit Einschränkung und einige überhaupt nicht. Eine zusammenfassende Auflistung der Bad Smells ist in den Abbildungen 84 und 85 zu finden.

Die Gründe, warum einige der Bad Smells nicht erkannt werden können, sind

Erkennbar	
BS 1	Duplizierter Code
BS 2	Lange Methode
BS 3	Große Klasse
BS 4	Lange Parameterlisten
BS 7	Neid
BS 8	Datenklumpen
BS 11	Parallele Vererbungshierarchien
BS 12	Faule Klasse
BS 16	Vermittler
BS 17	Unangebrachte Intimität
BS 20	Datenklassen
BS 21	Ausgeschlagenes Erbe

Abbildung 84: Mit dem Dagstuhl Middle Metamodel erkennbare Bad Smells

Nicht erkennbar	
BS 5	Divergierende Änderungen
BS 6	Schrotkugeln herausoperieren
BS 9	Neigung zu elementaren Datentypen
BS 10	`Switch`-Befehle
BS 13	Spekulative Allgemeinheit
BS 14	Temporäre Felder
BS 15	Nachrichtenketten
BS 18	Alternative Klassen mit verschiedenen Schnittstellen
BS 19	Unvollständige Bibliotheksklasse
BS 22	Kommentare

Abbildung 85: Mit dem Dagstuhl Middle Metamodel nicht erkennbare Bad Smells

verschiedener Natur. So liegt es bei *Divergierenden Änderungen* (BS5) und *Schrotkugeln herausoperieren* (BS6) an den fehlenden Versionsinformationen. Die Erkennung von *`Switch`-Befehle* (BS10), *Nachrichtenketten* (BS15) und *Alternative Klassen mit verschiedenen Schnittstellen* (BS18) scheitert an den im DMM fehlenden Kontrollflussinformationen. *Spekulative Allgemeinheit* (BS13) und *Temporäre Felder* (BS14) lassen sich nur zur Laufzeit bestimmen und der Bad Smell *Neigung zu elementaren Datentypen* (BS9) ist nicht sprachunabhängig auf Middle Model Niveau zu erkennen, da nicht alle Programmiersprachen elementare Datentypen nutzen. Zuletzt lassen sich *Unvollständige Bibliotheksklasse* BS(19) und *Kommentare* (BS22) generell nicht oder schwierig automatisch bestimmen, da die Bedeutung von Kommentaren oder die Vollständigkeit einer Bibliothek letztlich nur durch einen Programmierer bestimmt werden kann.

Für diejenigen Bad Smells, die sich mit dem DMM erkennen lassen, wurden nötige Metamodellausschnitte präsentiert. Diese Ausschnitte sind in Abbildung 86 zu einem Gesamtmodell zusammengefasst, mit welchem dann alle erkennbaren Bad Smells gefunden werden können.
Generell lässt sich feststellen, dass die Erkennung von Bad Smells auf Middle Model Niveau nur sehr ungenau möglich ist und allenfalls grobe Hinweise liefert. Für eine genauere Analyse einer Software sollte die Erkennung der Bad Smell auf feingranularerer Ebene, zum Beispiel direkt auf dem Quellcode, durchgeführt werden.

6 Grenzen des Sprachunabhängigen Refactoring

Die Kapitel 4 und 5 zeigten, dass die Durchführung von Refactorings sowie die Erkennung von Bad Smells auf Modellbasis nicht unproblematisch ist.

So gilt es bezüglich der unterstützten Sprachen, bestimmte Kontextbedingungen einzuhalten und Annahmen zu treffen, um die Refactorings auf Modellebene fehlerfrei durchführen zu können. Diese Kontextbedingungen und Sprachannahmen sind im Kapitel 6.1 erläutert.

Weiterhin wurde in den vorangegangenen Untersuchungen der Refactorings und der Bad Smells festgestellt, dass Teile davon mit dem Dagstuhl Middle Metamodel nur eingeschränkt, nur mit Änderungen am DMM oder gar nicht möglich sind. Diese Einschränkungen und Änderungen werden im Kapitel 6.2 noch einmal zentral zusammengefasst und es wird ein Gesamtmodell präsentiert, welches alle für diese Arbeit relevanten Aspekte enthält.

Zuletzt weist das Kapitel 6.3 noch auf allgemeine Schwierigkeiten im Zu-

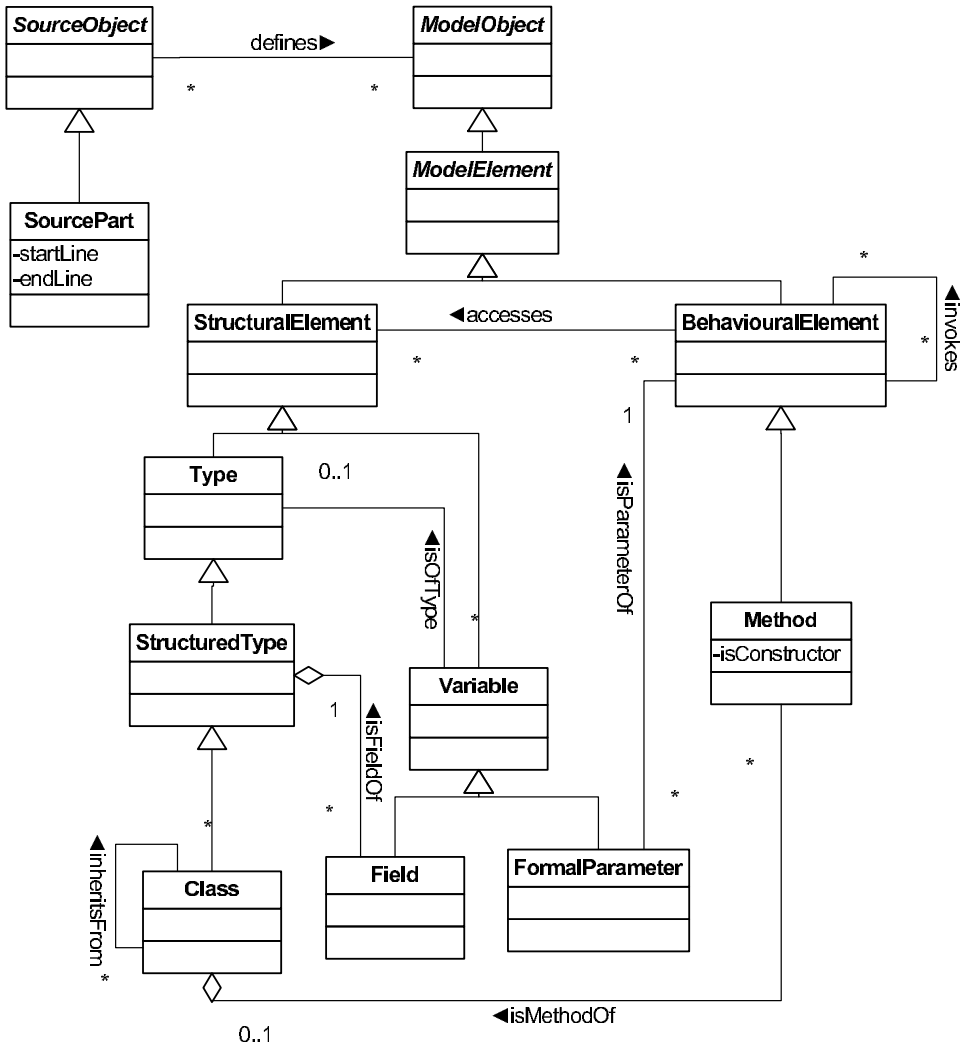

Abbildung 86: Gesamtmodell zur Erkennung von Bad Smells

sammenhang mit dem modellbasierten Refactoring und der Bad Smell-Erkennung hin.

6.1 Kontextbedingungen und zentrale Annahmen zu unterstützten Sprachen

Um Refactorings sprachübergreifend auf Modellbasis durchzuführen, ist ein Rahmen zu schaffen, in welchem sich die unterstützten Sprachen bewegen müssen. So hat sich bereits in der ersten Aktivität (A10) des Refactorings *Methode verschieben* gezeigt, dass eine Verabredung über die Unterstützung von Polymporphismus getroffen werden muss, da dies nicht von allen Programmiersprachen unterstützt wird. Es ist dann zu entscheiden, ob Polymorphismus zugelassen wird oder nicht.
Solche Annahmen und Verabredungen müssen auch an anderen Stellen getroffen werden und sind in diesem Unterkapitel noch einmal zusammenfassend dargestellt.

Arten unterstützter Programmiersprachen

Eine erste wichtige Annahme, welche für das sprachunabhängige Refactoring zu treffen ist, gilt der Art der unterstützten Programmiersprachen. Im Rahmen dieser Arbeit wurden die Refactorings im Hinblick auf objektorientierte Sprachen wie Java oder C# betrachtet. Es wäre aber auch denkbar, die Aktivitäten vor dem Hintergrund eines anderen Sprachenparadigmas wie beispielsweise SML[11] als funktionale Sprache zu betrachten.

Überladen von Methoden

Als Überladen von Methoden wird eine Technik bezeichnet, bei der innerhalb einer Klasse mehrere Methoden gleichen Namens, jedoch mit unterschiedlicher Signatur existieren. Welche der Methoden tatsächlich aufgerufen wird, entschiedet sich anhand der übergebenen Parameter. Meist werden beim Überladen von Methoden nur solche mit unterschiedlichen Parametertypen zugelassen und nicht solche mit gleichen Parametertypen aber unterschiedlichem Rückgabetyp.
Es gilt für das modellbasierte Refactoring zu entscheiden, ob man das Überladen von Methoden generell zulässt oder nicht.

Zudem sind auch Zwischenlösungen denkbar, wie sie beim des Refactoring *Method verschieben* im Kapitel 4.1 genutzt werden. So wird das Refactoring in der Aktivität A 20: 'Ober- und Unterklassen auf gleiche Methodensignatur überprüfen' nicht abgebrochen, wenn Methoden gleichen Namens, aber unterschiedlicher Signatur in der Klasse selbst beziehungsweise in einer ihrer Ober- oder Unterklassen vorhanden sind.

[11] Standard Meta Language

Hingegen muss in der Aktivität A 70: 'Methodenname prüfen' auf Seite 29 ein neuer Name gewählt werden, falls ein gleicher Name bereits in der Zielklasse vorhanden ist. Dieses Vorgehen ist sinnvoll, da dadurch überladene Methoden zugelassen werden, sofern sie bereits vorher vorhanden waren, aber es wird beim Verschieben in die neue Zielklasse keine Situation mit überladenen Methoden erzeugt.

Allgemein ist es ebenfalls möglich, das Überladen generell zuzulassen, wenn man sicher ist, das die Refactorings nur auf Modelle solcher Sprachen angewendet werden, die diese Technik auch unterstützen. Beispiele für Sprachen, die das Überladen von Methoden unterstützen, sind Java und C#, wohingegen in Oberon das Überladen ausgeschlossen ist.

Mehrfachvererbung

Mehrfachvererbung ist eine zweite Eigenschaft, für die im Voraus geklärt werden muss, ob sie für das modellbasierte Refactoring zugelassen wird oder nicht. Wird die Mehrfachvererbung zugelassen, gestalten sich einige Aktivitäten sehr viel komplizierter. Da Methoden dadurch mehr als eine Oberklasse besitzen können, muss bei der Durchführung der Refactorings verstärkt auf Namenskonflikte geachtet werden.
Weil die Mehrfachvererbung auch für die Programmiersprachen selbst Probleme wie Namenskonflikte oder unübersichtliche Klassen mit sich bringt, wird sie von den meisten Sprachen (z.B. Java oder C#) gar nicht erst unterstützt. Für diese Arbeit wurde Mehrfachvererbung nicht zugelassen.

Für das DMM als Metamodell würde die Mehrfachvererbung keine Hürde darstellen, da sie mit dem DMM modellierbar ist.

Properties

Die Benutzung von Zugriffsmethoden für das Auslesen oder Manipulieren von Feldern einer Klasse hat sich in der Programmierung bewährt, da hierdurch keine unkontrollierten Änderungen an den Datenfeldern einer Klasse mehr möglich sind. Wenn beispielsweise in ein Datenfeld `discount` ein neuer Rabatt eingetragen werden soll, kann innerhalb der Zugriffsmethode direkt geprüft werden, ob es sich um einen gültigen Rabatt oder ungültigen handelt. Die Klasse, welche das Datenfeld enthält, kann somit selbst die Validierung der Daten vornehmen.
In Java können Zugriffsmethoden als normale Methoden erstellt werden, die vom Programmierer meist nach einem bestimmten Muster benannt werden können, aber nicht müssen. Ein Beispiel in Java für Zugriffsmethoden auf ein Datenfeld `discount` zeigt Abbildung 87.
In anderen Sprachen gibt es für solche Zugriffsmethoden spezielle Konstrukte, die Properties genannt werden. Beispiele für solche Sprachen sind C#,

```
public class Customer
{
        private double discount;
                //.......... some other code ..........

                public double get_Discount()
        {
                return this.discount;
        }
                                                                        10
                public void set_Discount(double input)
        {
                //.......... code to validate input ..........
                this.discount = input;
        }

                //.......... some other code ..........
}
```

Abbildung 87: Java-Zugriffsmethoden für ein Datenfeld `discount`

Visual Basic oder Delphi.
Der Vorteil solcher Properties gegenüber dem von Sun gewählten Weg für Java liegt einerseits in der verbesserten Lesbarkeit des Codes, da die Zugriffsmethoden in Form von Properties direkt als solche identifiziert werden können. Andererseits sind Properties von einem Client aus genauso einfach wie öffentliche Datenfelder zu benutzen, was die Benutzung erleichtert (vgl. hierzu auch [Arc01]. Properties sind somit eine Mischung zwischen Methode und Datenfeld (vgl. Abbildung 88).

Das vorherige Javabeispiel aus Abbildung 87 ist in C# unter Verwendung von Properties in Abbildung 89 zu sehen.
Für das Refactoring im Allgemeinen und speziell für das sprachunabhängige Refactoring wäre es wünschenswert, Zugriffsmethoden als solche von anderen Methoden zu unterscheiden (vgl. auch den entsprechenden Abschnitt auf Seite 90. Properties stellen dabei eine erste Möglichkeit dar, diese Unterscheidung treffen zu können.
Im DMM könnten solche Properties als *ExecutiveValue* modelliert werden (vgl. Abbildung 6 auf Seite 8).

Schnittstellen

Im Abschnitt über Mehrfachvererbung auf Seite 82 wurde bereits über deren Nachteil gesprochen. Da es genügend Situationen gibt, in denen es sinnvoll ist, Eigenschaften von mehr als einer Oberklasse zu implementieren, bieten

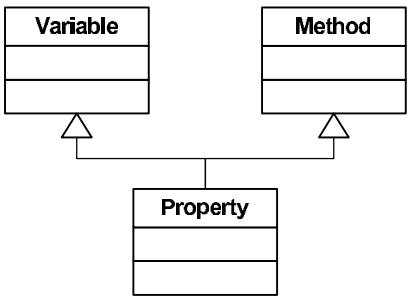

Abbildung 88: Properties haben sowohl Methoden als auch Variablencharakter

```
public class Customer
{
        private double discount;

        //.......... some other code ..........

        public double Discount
        {
                get
                {                                                    10
                        return this.discount;
                }

                set
                {
                        //.......... code to validate input ..........
                        this.discount = value;
                }
        }
}                                                                    20
```

Abbildung 89: C#-Propertis für ein Datenfeld `discount`

einige Programmiersprachen das Konzept der Schnittstellen (engl. *interfaces*) an. In diesen Schnittstellen sind lediglich Definitionen von Methoden oder Konstanten enthalten. Implementiert eine Klasse eine oder mehrere Schnittstellen, müssen alle in den Schnittstellen definierten Methoden in der Klasse implementiert werden (vgl. hierzu auch [AG96]).

Für sprachunabhängiges Refactoring muss entschieden werden, ob Schnittstellen zugelassen werden oder nicht. In dieser Arbeit wurden sie nicht berücksichtigt, da das Dagstuhl Middle Metamodel keine Möglichkeit bietet, diese zu modellieren.
Die Erweiterung des DMM, um auch Schnittstellen zuzulassen, erweist sich als nicht trivial, denn die Erweiterung der Klasse *Class* um ein Attribut *isInterface* oder die Spezialisierung dieser Klasse zu einer Klasse Interface ist problematisch. Dadurch wäre es in einem konkreten Modell dann möglich, dass ein Interface Felder besitzt, da diese in der Klasse Class erlaubt sind. Das Einführen einer Schnittstellenklasse an einem höheren Punkt der ModelObject-Hierarchie ist auch nicht möglich, da dadurch die Assoziation *isMethodOf* zu Methoden verloren geht (vgl. dazu das Metamodell in Abbildung 90 auf Seite 88).
Sollten Schnittstellen für das sprachunabhängige Refactoring zugelassen werden, müssten im DMM also erst Elemente geschaffen werden, mit denen sich die Schnittstellen auch modellieren lassen.

Delegates

Delegates sind eine in der Microsoft-Welt angesiedelte Möglichkeit für Rückrufe und (asynchrone) Ereignisbehandlung. Prinzipiell stellen *Delgates* eine Erweiterung der Funktionszeiger aus C++ dar. Im Rahmen dieser Arbeit wurde nicht untersucht, ob der Einsatz von *Delegates* Auswirkungen auf das sprachunabhängige Refactoring hat und sie wurden daher ausgeschlossen. Näheres zu Delegates findet sich in [Arc01], [Sun] und [Mic].

Zusammenfassung

Dieses Unterkapitel sollte die für das modellbasierte Refactoring zu treffenden Sprachannahmen zusammenfassen. Diese Annahmen sind für die ausgewählten Refactorings zu treffen, erheben aber allgemein keinen Anspruch auf Vollständigkeit, denn es ist möglich, dass durch andere Refactorings weitere solcher Annahmen hinzukommen.

6.2 Würdigung des *Dagstuhl Middle Metamodel*

In den Kapiteln 4.1 bis 4.3 wurden die Refactorings in kleinere Aktivitäten zerlegt und Anforderungen an ein Metamodell definiert, um diese Aktivitäten durchzuführen. Dabei wurde das Dagstuhl Middle Metamodel auf die

Erfüllung dieser Anforderungen hin geprüft und es hat sich gezeigt, dass sich das DMM in großen Teilen sehr gut eignet, um die Refactorings auf Modellebene durchzuführen.

Im Kapitel 5 wurden weitere Anforderungen formuliert, um auch die Erkennung von Bad Smells auf Modellbasis zu ermöglichen. Auch für diese wurden, sofern möglich, Ausschnitte aus dem DMM präsentiert, welche die Anforderungen erfüllen. Dabei hat sich gezeigt, dass das DMM für die Erkennung der Bad Smells nicht in dem Maße geeignet ist, wie für die untersuchten Refactorings. So konnten nicht alle Bad Smells mit dem DMM oder einer Erweiterung erkannt werden (vgl. Abbildung 85 auf Seite 78).

An einigen Stellen, wo sich das DMM als unzureichend erwies, konnte es entsprechend erweitert werden. Diese Erweiterungen und Veränderungen werden im Unterabschnitt 6.2.1 noch einmal kurz erläutert und darauf basierend ein Gesamtmodell präsentiert, welches eine Vereinigung aller in dieser Arbeit beschriebenen Teilmodelle darstellt. Abschnitt 6.2.2 erläutert gesondert die bisher stillschweigend durchgeführten Berichtigungen am DMM und der Abschnitt 6.2.3 listet die noch fehlenden Eigenschaften des DMM auf, um das sprachunabhängige Refactoring für eine noch breitere Palette von Anwendungsfällen zu ermöglichen.

6.2.1 Änderungen des DMM

Der folgende Abschnitt fasst die für die Refactorings und die Bad Smells notwendigen Änderungen am DMM noch einmal zusammen.

Erweiterung der Klassen *Field* und *Method*

Bei der Zerlegung des Refactorings *Methode verschieben* in kleinere Aktivitäten stellte sich heraus, dass ein Unterschied in der Behandlung statischer Variablen gegenüber Objektvariablen existiert. Da das DMM diese Unterscheidung nicht zuließ, wurde es erweitert, so dass die Klassen *Field* und *Method* ein zusätzliches Attribut *isStatic* erhielten. Diese Erweiterung des DMM wurde ebenso bei dem Refactoring *Attribut verschieben* benutzt. Im Gesamtmodell in Abbildung 90 ist die Erweiterung in den Klassen *Field* und *Method* grau hinterlegt.

Spezialisierung der *accesses*-Relation

Sowohl für das Refactoring *Attribut verschieben* als auch bei *Attribut kapseln* war es notwendig, zwischen lesendem und schreibendem Zugriff zu Unterscheiden, was mit dem Standard-DMM nicht möglich war. In diesem sind lediglich *accesses*-Relationen enthalten, weshalb das DMM auch an dieser

Stelle erweitert wurde. Deshalb wurden zwei Spezialisierungen der *accesses*-Relation gebildet, wobei die *uses*-Relation einen lesenden und die *sets*-Relation einen schreibenden Zugriff modelliert. Auch diese Erweiterungen sind im Gesamtmodell in Abbildung 90 grau hinterlegt.

6.2.2 Berichtigungen am DMM

Im Original-DMM, welches in den Abbildungen 4, 5 und 6 gezeigt ist, fehlen einige Elemente, welche für diese Arbeit an den entsprechenden Metamodellen einfach als vorhanden angesehen wurden.
Es fehlen in Abbildung 6 auf Seite 8 die Leserichtungen für die beiden Relationen *invokes* und *inheritsFrom*.
Weiterhin besitzen die beiden Klassen *Field* und *Method* ein Attribut `visibility`, welches aber überflüssig ist, da beide Klassen dieses Attribut bereits von der Klassen *ModelElement* erben.
Als letztes fehlt in der Abbildung 6 die Relation *isReturnTypeOf* zwischen den beiden Klassen *StructuralElement* und *BehaviouralElement*.

6.2.3 Fehlende Erweiterungen des DMM

Folgende Aspekte sind im DMM nicht vorhanden und sollten noch integriert werden, da sie allgemein für den Umgang mit Quellcode erforderlich sind.

Schnittstellen

Wie bereits im Abschnitt 6.1 auf Seite 83 beschrieben, gibt es im DMM keine Möglichkeit zwischen normalen Klassen und Schnittstellen zu unterscheiden. Eine Möglichkeit, um diese Unterscheidung mit dem DMM treffen zu können, wäre die Erweiterung der Klasse *Class* um ein Attribut `isInterface`. Dann ist jedoch darauf zu achten, dass eine so gewählte Erweiterung zumindest auf Modellebene zulassen würde, dass eine Schnittstelle Felder besitzt (vgl. Abbildung 6 auf Seite 8), was wiederum in den Sprachen, die *Interfaces* unterstützen, nicht erlaubt ist.

Ausnahmebehandlung

Ein anderer Punkt, der mit dem DMM nicht modelliert werden kann, sind Ausnahmen (engl. *exceptions*). In modernen Programmiersprachen sind Ausnahmen ein effektives Mittel, um Laufzeitfehler abzufangen und in sinnvoller Art und Weise zu behandeln. Im DMM besteht keine Möglichkeit solche Ausnahmen zu modellieren. Wirft eine Methode allerdings eine Ausnahme, so müsste zum Beispiel beim Verschieben dieser Methode in eine neue Zielklasse entschieden werden, welche Klasse die Ausnahme behandelt.

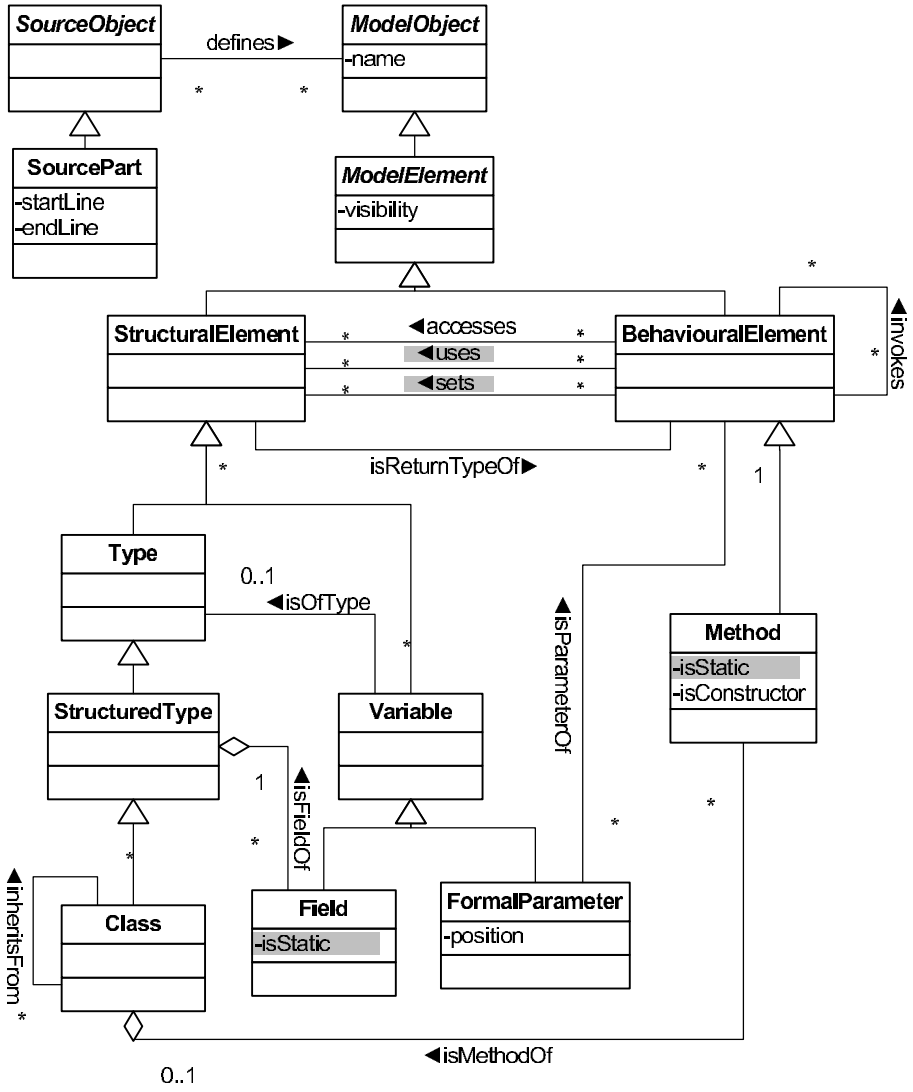

Abbildung 90: Gesamtmodellausschnitt des DMM mit allen für diese Arbeit relevanten Aspekten

Versionsinformationen in der Wartungsphase

Allgemein lässt sich sagen, dass viele Refactorings erst durch nachträgliche Änderungen in der Wartungsphase einer Software notwendig werden. Da das DMM immer nur einen aktuellen Schnappschuss einer Software liefert, kann damit nicht auf frühere Versionen zurückgegriffen werden.
Einige Bad Smells, wie *Divergierende Änderung*, *Schrotkuglen herausoperieren*, lassen sich nur durch solche historischen Informationen feststellen. Auch für die Erkennung von dupliziertem Code kann das Hinzuziehen von historischen Informationen nützlich sein, da dadurch eventuell zu erkennen ist, ob es sich tatsächlich um duplizierten Code handelt. So gibt es für die Duplikaterkennung Verfahren, Codestücke auf Gemeinsamkeiten zu vergleichen und in früheren Versionen festzustellen, ob diese Gemeinsamkeiten in der Anzahl zu- oder abnehmen (vgl. hierzu auch [RDGM04]).

Im DMM können auf frühere Versionen der Software zunächst nur anhand mehrerer statischer Gesamtmodelle zugegriffen werden.

Kontrollflüsse

Für die Erkennung einiger Bad Smells wie *`Switch`-Befehle* (BS10), *Nachrichtenketten* (BS15) und *Alternative Klassen mit verschiedenen Schnittstellen* (BS18) ist es notwendig, die Kontrollflüsse zu kennen. Diese lassen sich mit dem Standard DMM nicht modellieren. Daher sollte das DMM erweitert werden, damit auch diese Bad Smells modellbasiert erkannt werden können.

6.3 Allgemeine Schwierigkeiten für sprachunabhängiges Refactoring

Außer den sprachspezifischen Problem (vgl. Abschnitt 6.1) und den fehlenden Modellierungskonzepten im DMM (vgl. Abschnitt 6.2) existieren weitere generelle Schwierigkeiten im Umgang mit sprachunabhängigem Refactoring. Diese Probleme sind in den nächsten Abschnitten kurz erläutert.

Mangelnde Spezifikation der Refactorings

Allgemein lässt sich zu den Refactorings sagen, dass sie in [Fow05] nur sehr unscharf definiert sind. Dies betrifft sowohl den Zeitpunkt der Anwendung als auch deren Durchführung. Zum Beispiel heißt es bei dem Refactoring *Methode verschieben* zu eventuell vorhandenem Polymorphismus:

> „Gibt es weitere Deklarationen, so kann es sein, dass Sie die Verschiebung nicht durchführen können, es sei denn, Polymorphismus kann auch mit der anderen Klasse ausgedrückt werden.“

Diese Formulierung ist sehr unscharf und überlässt viele Entscheidungen dem Anwender des Refactoring. Bei der Durchführung wird also die größte Verantwortung dem Anwender selbst übertragen und es gibt keine feste Anleitung, *wann* genau *was* zu tun ist. Somit hängt die Qualität des Ergebnisses eines konkreten Refactoring sehr stark am Erfahrungsschatz des Anwenders und kann nur schwerlich automatisiert werden. Wünschenswert wäre für das automatisierte beziehungsweise werkzeuggestützte Refactoring eine genauere Spezifikation der durchzuführenden Schritte, sofern das überhaupt möglich ist.

Import-/Using-Anweisungen

Ein weiteres Problem für das sprachunabhängige Refactoring ergibt sich durch die Verwendung von *import-* (Java) oder *using*-Anweisungen (C#). So kann es passieren, das beim Verschieben eines Attributs aus einer Klasse in eine andere Klasse, die neue Zielklasse den Typ des Attributs nicht kennt, weil die Zielklasse keinen Zugriff auf das entsprechende Paket besitzt.

Es ist im DMM zwar prinzipiell möglich, Pakete zu modellieren, aber das parsen solcher Informationen ist sehr schwierig, da man aus dem Quellcode selbst nicht erkennen kann, welcher Typ aus welchem Paket entstammt. Dies wird in der Regel erst durch den Compiler festgestellt.

Uneinheitliche Syntax für Zugriffsmethoden

Ein bereits im Abschnitt 6.1 bei den Properties beschriebenes Problem liegt in der nicht einheitlich geregelten Syntax von Zugriffsmethoden. So ist es dadurch schwierig, den Code tatsächlich durch ein Tool zu einer besseren Lesbarkeit zu führen.

Wenn beispielsweise ein Attribut gekapselt werden soll, ist es für ein Werkzeug, welches diese Kapselung automatisch durchführen soll, nicht möglich festzustellen, ob für das Attribut bereits Zugriffsmethoden vorhanden sind. Dies kann nur durch Interaktion mit dem Anwender des Tools geklärt werden, ansonsten müssen generell Zugriffsmethoden erstellt werden, auch wenn dies nicht zur besseren Lesbarkeit und Wartbarkeit führt.
Allgemein wäre es für die Durchführung von automatisiertem Refactoring wünschenswert, wenn Zugriffsmethoden in Programmiersprachen deutlicher gekennzeichnet werden könnten, beispielsweise durch ein Schlüsselwort.

Messbarkeit von lesbarem Code

Ein weiteres grundsätzliches Problem beim werkzeuggestützten Refactoring liegt im Sinn des Refactoring selbst verborgen. Refactoring soll dazu dienen den Code lesbarer und somit leichter wartbar zu machen. Leider gibt es kaum

Möglichkeiten, diese Lesbarkeit in Form von Metriken oder anderen Mitteln zu bestimmen. So dass nach der Durchführung eines Refactoring nur das menschliche Auge entscheiden kann, ob der Code nun lesbarer ist oder nicht (was wiederum auch selbst ein höchst subjektives Empfinden sein kann).

Große Refactorings

Eine andere Schwierigkeit betrifft große Refactorings, also Refactorings bei denen beispielsweise mehrere Methoden und Attribute einer Klasse gleichzeitig verschoben werden sollen. Die Durchführung solcher größeren Refactoring ist oft sinnvoll, da es meist nicht nur eine Methode ist, die verschoben werden soll, sondern diese benutzt wiederum bestimmte Attribute, die von sonst keiner anderen Methode benutzt werden (exklusive Nutzung vgl. dazu auch die Aktivität A40 im Abschnitt 4.1).

Das Problem liegt hier in der Schwierigkeit, bestimmte Refactorings parallel durchzuführen. Wenn beispielsweise zwei Attribute und eine Methode, die diese beiden Attribute benutzt, in eine andere Klasse verschoben werden sollen, wird beim Verschieben des ersten Attributs ein Objekt für die Zielklasse in der Ausgangsklasse erzeugt. Dieses Objekt wird eigentlich gar nicht benötigt (falls nur die eine Methode auf das Attribut zugreift), da die benutzende Methode zu einem späteren Zeitpunkt in die neue Klasse verschoben wird.

Entwurfsmuster

Ein Problem mit der Erkennung der Bad Smells sind bestimmte Entwurfsmuster. So gibt es Entwurfsmuster, die bei der Erkennung von Bad Smells genau als solche erkannt werden. Beispiele hierfür sind das Besuchermuster oder das Strategiemuster (vgl. [Gam96]), die selbst wiederum dazu dienen, andere Bad Smells wie *divergierende Änderungen* und *Schrotkugeln herausoperieren* bekämpfen sollen.

Laufzeitinformationen

Ein weiteres Problem beim Refactoring besteht darin, dass sich bestimmte Bad Smells, wie *Spekulative Allgemeinheit* (BS13) und *Temporäre Felder* (BS14), nur mit Hilfe von Laufzeitinformationen ermitteln lassen. Da es nicht möglich ist, Laufzeitinformationen modellbasiert zu ermitteln, werden sich diese beiden Bad Smells nicht modellbasiert finden lassen.

6.3.1 Zusammenfassung

Dieses Kapitel sollte zentrale Sprachannahmen zusammenfassen und auf zu erfüllende Kontextbedingungen eingehen. Des Weiteren wurden hier alle Erweiterungen und Berichtigungen des DMM gesammelt, dazu ein für diese

Arbeit geeignetes Gesamtmodell vorgestellt und auf allgemeine Probleme im Umgang mit dem Refactoring hingewiesen.

7 Abschließende Betrachtungen

Ziel der vorliegenden Diplomarbeit war die Untersuchung der Eignung des *Dagstuhl Middle Metamodell* zur Durchführung von sprachunabhängigem Refactoring.

Dazu wurden in Kapitel 2 zunächst die benötigten Grundlagen, das *Dagstuhl Middle Metamodell* und die Anfrage-Sprache GReQL, vorgestellt.

In Kapitel 3 erfolgte aufgrund mehrerer Gesichtspunkte eine Auswahl von vier Refactorings, welche als Referenz für die weiteren Untersuchungen dienen sollten. Die Kriterien zur Auswahl der Refactorings waren dabei die Häufigkeit der Anwendung, die Abhängigkeit zwischen den Refactorings und die Ähnlichkeit bezüglich der durchzuführenden Aktivitäten, da sich im weiteren Verlauf gezeigt hat, das einige Aktivitäten und damit auch einige Metamodelle wiederverwendet werden konnten.

Die vier Refactorings wurden im Kapitel 4 in kleinere Aktivitäten zerlegt, für welche dann jeweils Anforderungen an ein Metamodell zur Durchführung dieser Aktivitäten definiert wurden. An dieser Stelle wurden die Metamodelle als Ausschnitte des DMM (zum Teil auch mit Erweiterungen des DMM) präsentiert, die diese Anforderungen erfüllen konnten. Hier hat sich gezeigt, dass die Aktivitäten mit diversen Einschränkungen oder Erweiterungen alle mit dem DMM durchzuführen sind.

Im Kapitel 5 wurden anschließend, genau wie in Kapitel 4, Anforderungen für die Erkennung von Bad Smells formuliert und das DMM auf die Erfüllung dieser Anforderungen hin untersucht. Hier wurden Metamodellausschnitte präsentiert und beispielhaft GReQL-Anfragen gezeigt, mit welchen sich die einzelnen Bad Smells erkennen lassen. Hierbei zeigte sich, dass im Gegensatz zu den Refactorings die Erkennung der Bad Smells zum Teil nicht mit Ausschnitten aus dem DMM und ebenso nicht mit kleineren Erweiterungen des DMM möglich war. Das heißt, um einige Bad Smells zu finden sind feingranularere Ebenen zur Suche erforderlich.

Im Kapitel 6 finden sich zusammenfassend noch einmal alle Probleme, Erweiterungen, Grenzen des modellbasierten Refactoring und der Bad Smell-Erkennung. Dabei werden im Unterkapitel 6.1 Kontextbedingungen und zentrale Sprachannahmen erläutert, die zu treffen sind, bevor ein Werkzeug zum sprachunabhängigen Refactoring entwickelt wird.

Das Unterkapitel 6.2 fasst alle für diese Arbeit durchgeführten Erweiterungen, Berichtigungen und fehlenden Aspekte des *Dagstuhl Middle Metamodel* zusammen, bevor im Unterkapitel 6.3 noch auf allgemeine Probleme mit der Sprachunabhängigkeit eingegangen wird.

Im Laufe dieser Arbeit hat sich gezeigt, dass die Schwierigkeiten mit sprachunabhängigem Refactoring sehr vielschichtig sind. So gibt es in bestimmten Programmiersprachen Konstrukte wie Properties (vgl. Seite 82), für die Elemente im Metamodell erstellt werden müssen, die in anderen Sprachen nicht existieren. Diese sprachspezifischen Probleme sind wahrscheinlich durch die Konstruktion entsprechender Parser und Unparser gut zu lösen, wobei vor allem der Unparser ein nicht zu unterschätzendes Projekt darstellt, denn dieser muss dem Refactoringgedanken folgen, besser lesbaren Code zu erzeugen.
Zusätzlich mangelt es, wie bereits erwähnt, sehr an einer eindeutigeren Spezifikation der Refactorings. So kann mit den Anleitungen von Martin Fowler in [Fow05] nicht nur schwer gesagt werden, *wann* ein Refactoring durchzuführen ist, sondern die Frage nach dem *wie* wird ebensowenig geklärt, wie die zu erwartenden Ergebnisse nach der Durchführung eines Refactoring.
Ob es jemals ein Werkzeug für sprachunabhängiges Refactoring geben wird, wird die Zukunft zeigen. Hier sollte gezeigt werden, dass sich das *Dagstuhl Middle Metamodel* in großen Teilen zur Durchführung der Refactorings auf Modellbasis eignet und dass einige der Aktivitäten innerhalb der einzelnen Refactorings in anderen wiederverwendet werden können.

Literatur

[AG96] Arnold, Ken und James Gosling: *Die Programmiersprache Java.* Addison-Wesley, Bonn, 1996.

[Arc01] Archer, Tom: *Inside C#.* Microsoft Press, Unterschleißheim, 2001.

[Bak97] Baker, Brenda S.: *Parameterized Duplication in Strings: Algorithms and an Application to Software Maintenance.* SIAM J. Comput., 26(5):1343–1362, 1997.

[Bal00] Balzert, Helmut: *Lehrbuch der Softwaretechnik Bd.1.* Spektrum Akademischer Verlag, Heidelberg, 2000.

[Bec00] Beck, Kent: *Extreme programming explained: embrace change.* Addison-Wesley, Boston et al., 2000.

[BYM+98] Baxter, Ira D., Andrew Yahin, Leonardo M. De Moura, Marcelo Sant'Anna und Lorraine Bier: *Clone Detection Using Abstract Syntax Trees.* In: *ICSM*, Seiten 368–377, 1998.

[CEK+00] Czeranski, Jorg, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plodereder, Daniel Simon, Yan Zhang, Jean-Francois Girard und Martin Wurthner: *Data Exchange in Bauhaus.* In: *Working Conference on Reverse Engineering*, Seiten 293–295, 2000.

[EKW97] Ebert, J., M. Kamp und A. Winter: *A generic system to support multi-level understanding of heterogeneous software*, 1997. Fachbericht Informatik 6/97, Universität Koblenz-Landau, Institut für Informatik.

[Fow05] Fowler, Martin: *Refactoring.* Addison-Wesley, München, 2005.

[Gam96] Gamma, Erich: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software.* Addison-Wesley, Bonn, 1996.

[Hin05] Hinterwäller, Bodo: *Metamodell-basierte Spezifikation von Refactorings.* Diplomarbeit, Institut für Softwaretechnik, Universität Koblenz-Landau, Abteilung Koblenz, 2005.

[KK01] Kamp, Manfred und Bernt Kullbach: *GReQL – Eine Anfragesprache für das GUPRO–Repository – Sprachbeschreibung (Version 1.3).* Projektbericht 8/01, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, 2001.

[Kos98] Koschke, Rainer: *An Intermediate Representation for Integrating Reverse Engineering Analysis.* In: *Proceedings of the Working Conference on Reverse Engineering*, Seiten 241–250, 1998.

[Let01] Lethbridge, Timothy C.: *Requirements and Proposal for a Software Information Exchange Format (SIEF) Standard.* Technischer Bericht, 2001. `http://www.site.uottawa.ca/~tcl/papers/sief/standardProposal.html`.

[LTP04] Lethbridge, Timothy, Sander Tichelaar und Erhard Plödereder: *The Dagstuhl Middle Metamodel: A Schema For Reverse Engineering.* Electr. Notes Theor. Comput. Sci., 94:7–18, 2004.

[Mic] Microsoft: *The Truth About Delegates.* `http://msdn.microsoft.com/vjsharp/productinfo/visualj/visualj6/technical/articles/general/truth/default.aspx`.

[RDGM04] Ratiu, Daniel, Stephane Ducasse, Tudor Girba und Radu Marinescu: *Using History Information to Improve Design Flaws Detection.* Band 00, Seite 223, Los Alamitos, CA, USA, 2004. IEEE Computer Society.

[Sun] Sun: *About Microsoft's Delegates.* `http://java.sun.com/docs/white/delegates.html`.

[TDD00] Tichelaar, Sander, Stéphane Ducasse und Serge Demeyer: *FAMIX and XMI.* In: *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, Seite 296, Washington, DC, USA, 2000. IEEE Computer Society.

[Tic01] Tichelaar, Sander: *Modeling Object-Oriented Software for Reverse Engineering and Refactoring.* Doktorarbeit, Universität Bern, Dezember 2001.