Modellbasierte Integration von Joinpoint Queries für die aspektorientierte Sprache ObjectTeams/Java

Diplomarbeit

Andreas Mertgen
Mat.-Nr. 205804
Berlin, 17. Oktober 2007

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

Diplomarbeitsbetreuer: Dipl.-Inf. Marco Mosconi
Gutachter: Prof. Dr. Ing. Stefan Jähnichen und Dr. Ing. Stephan Herrmann

Eidesstattliche Erklärung

Die selbstständige und eigenhändige Anfertigung versichere ich an Eides Statt.

_____

Berlin, den / Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Listings

1 Einleitung

1.1 Zielsetzung

In der modernen Softwareentwicklung existieren verschiedene Ansätze die Konzepte der Objektorientierung, die mittlerweile als Standard etabliert sind, zu erweitern. Zwei dieser Ansätze, die Modellgetriebene Softwareentwicklung (MDSD) und die Aspektorientierte Softwareentwicklung (AOSD), sollen in dieser Arbeit konkrete Verwendung finden.

Der Ansatz der Modellgetriebenen Softwareentwicklung verfolgt das Ziel, Software durchgängig in Modellen zu beschreiben. Die Arbeit mit Modellen bietet die Möglichkeit, funktionale und technische Anforderungen voneinander trennen. Die funktionalen Anforderungen der Software können auf eine höhere Abstraktionsebene verlagert und weitgehend unabhängig von Technik bzw. Plattform dargestellt werden. Modelle lassen sich grafisch visualisieren und sind somit besonders für Nicht-Entwickler zugänglicher und verständlicher. Mittels automatisierter Transformationen können große Teile der Implementierung direkt aus den Modellen generiert werden.

Die Aspektorientierte Softwareentwicklung verfeinert die objektorientierten Modularisierungskonzepte mit dem Ziel, Programme übersichtlicher und flexibler strukturieren zu können. Die Auslagerung von Aspekten ermöglicht es, auch solche Programmanforderungen zu modularisieren, welche sonst für eine Kapselung in der objektorientierten Klassenstruktur eher ungeeignet sind.

Mit ObjectTeams/Java (OT/J) existiert ein Programmiermodell, das mehrere Konzepte der Aspektorientierung unterstützt (rollenbasiertes Design, Kollaborationen von Objekten in Teams, Adaptierung bestehenden Programmverhaltens). Daneben stehen aber noch Anforderungen aus, die in anderen aspektorientierten Sprachen (z.B. AspectJ) bereits vorhanden sind. Ein wichtiges Konzept ist dabei die Identifizierung von Joinpoints und deren Auswahl in Pointcuts, die mit Advices verknüpft werden können. Zu diesem Zweck ist für OT/J eine Joinpoint Abfragesprache (Joinpoint Query Language) in der Entwicklung.

Die Vorteile der MDSD sollen auch für die Entwicklung von OT/J-Programmen nutzbar gemacht werden. Langfristig ist geplant, im Object Teams Development Tooling Werkzeuge für die modellbasierte Entwicklung zu integrieren. Dies betrifft auch die Modellierung der Querysprache, welche in dieser Arbeit behandelt werden soll. Konkret stellen sich dabei vor allem die Fragen, ob sich eine angemessene grafische Abbildung für Queries finden lässt, und, ob eine

Modellierungsnotation sowohl eine sinnvolle Abstraktion für den Anwender bieten kann, als zugleich auch für den modellgetriebenen Entwicklungsprozess mittels automatisierter Transformationen geeignet ist.

Ziel dieser Arbeit ist es also, die Entwicklung und Anwendbarkeit der Joinpoint-Querysprache für OT/J konzeptionell und durch Implementierung eines grafischen Modellierungswerkzeuges zu unterstützen. Das Werkzeug soll die modellgetriebene Entwicklung von Queries ermöglichen, seine Implementierung soll wiederum selbst mit modellgetriebenen Techniken realisiert werden. Konkret gilt es in dieser Arbeit, die aspektorientierten Konzepte und Mechanismen der Querysprache zu diskutieren, eine grafische Notation zur Element-Selektion zu entwickeln, diese in einem Editor zur Erstellung von Query-Modellen umzusetzen, für die Modelle wiederum eine Transformationsvorschrift zu entwickeln, und diese dann in einem Generator zu implementieren.

1.2 Struktur

Kapitel 2 vermittelt die Grundlagen der Aspektorientierung und beschreibt das Programmiermodell von Object Teams. In Kapitel 3 folgt eine konkrete Erläuterung der Joinpoint-Querysprache. Die Kapitel 4 und 5 vertiefen die Joinpoint-Thematik und diskutieren verschiedene Probleme in Bezug auf ihre Behandlung in Object Teams. Eine grafische Notation zur Modellierung von Queries wird in Kapitel 6 vorgestellt. Kapitel 7 behandelt deren Implementierung für eine Editor-Anwendung, in welcher Query-Modelle erstellt werden können. In Kapitel 8 werden Transformationen entwickelt, mit denen aus solchen Modellen Implementierungs-Code generiert werden kann. Mit einem Fazit in Kapitel 9 findet die Arbeit ihren Abschluss.

2 Aspektorientierte Softwareentwicklung

Die Konzepte und Werkzeuge der Softwareentwicklung unterliegen einer stetigen Evolution, um mit den Anforderungen moderner Systeme und Entwicklungsprozesse Schritt zu halten. Eine Großzahl der dominierenden Programmiersprachen, wie z.B. Java oder C++, arbeiten heute mit den Konzepten der Objekt-Orientierten-Programmierung (OOP), welche mittlerweile als Quasi-Standard angesehen werden kann. Die OOP verfolgt zwei primäre Ziele: zum einen möchte man Programme modularisieren, um sie übersichtlich, zuverlässig und leicht wartbar zu halten, zum anderen möchte man einzelne Programmbausteine flexibel kombinieren und nach Möglichkeit mehrfach verwenden können. Insbesondere Wartbarkeit und Wiederverwendbarkeit sind bei komplexen Projekten heutzutage allein schon aus Aufwands- und damit Kostengründen von zentraler Bedeutung. Zur Verwirklichung dieser Ziele nutzt die OOP verschiedene Programmiertechniken; zu den wichtigsten zählen: Abbildung und Abstraktion von Objekten auf eine Klassenstruktur, Vererbung, Polymorphie und die Kapselung von Daten. Diese Konzepte erfüllen viele der an sie gestellten Anforderungen. Dennoch existieren einige Probleme, welche mit diesen Mitteln nur unzureichend angegangen werden können.

2.1 Crosscutting Concerns

Bei der Strukturierung eines Programms werden die Anforderungen in verschiedenen Modulen gekapselt. Während es für die Kernfunktionen meist noch leicht möglich ist, diese in separaten Bereichen voneinander zu trennen, gibt es auch Anforderungen, die sich nicht separat auflösen lassen. Diese werden ≪crosscuting concerns≫ genannt, da sich diese Belange ≪quer≫ zum Programm durch verschiedene Module schneiden, d.h. dass sich eine Funktionalität über mehrere Module erstreckt oder dass mehrere Funktionalitäten in einem Modul vermischt sind. Eine klare Abgrenzung dieser Bereiche nach den bewährten Prinzipien der Klassifizierung und Kapselung ist an dieser Stelle nicht mehr möglich. Klassische Beispiele für querschneidende Anforderungen sind z.B. Logging, Tracing, Fehlerbehandlung und Serialisierung.

Gegenüber den Crosscuting Concerns steht das Prinzip der ≪Separation of Concerns≫, der Wunsch verschiedene Belange eines Programms getrennt voneinander definieren zu können, selbst wenn sie sich in einigen Bereichen überlappen. Aus dieser Motivation ist die Aspekt-Orientierte-Programmierung (AOP) entstanden, mit dem Ziel nun auch Crosscutting Concerns sauber modularisieren zu können.

2.2 Begriffe

Die allgemein verwendeten Begriffe und Konzepte der AOP sollen im Folgenden kurz erläutert werden. Die Bezeichnungen orientieren sich dabei an der *AOSD Ontologie* [vdBCC05] und der Sprache AspectJ, die zu den am weitest verbreiteten AO-Sprachen gehört und als deren klassischer Vertreter herangezogen wird. Im Vergleich dazu werden danach die Konzepte der Sprache ObjectTeams/Java (OT/J) vorgestellt. Sowohl AspectJ als auch OT/J sind für die Sprache Java implementiert, die Konzepte sind aber prinzipiell sprachunabhängig und finden auch in anderen Sprachen Verwendung.

Aspekt Als Oberbegriff für eine Programmeinheit zur Modularisierung eines ≪Concern≫ wurde die Bezeichnung ≪Aspekt≫ geprägt. Es existieren keine maßgeblichen, eindeutigen Definitionen für diese Begriffe. Als Aspekt werden aber vor allem übergreifenden Anforderungen bezeichnet, die zwar eine logische Einheit bilden, sich aber nicht modular in eine Klassenhierarchie integrieren lassen. In der AOP können Aspekte mittels neuer Sprachkonstrukte ähnlich den Klassen trotz ihrer querschneidenden Funktionalität als eigenständiges Modul definiert werden. Anstatt der direkten Einbindung in die Klassenstruktur des Basisprogramms definiert ein Aspekt nun neben dem eigentlichen Aspekt-Code auch wo und unter welchen Bedingungen dieser Code ausgeführt werden soll.

Weben In einem Aspekt sind die Anweisungen, die sonst an mehrere Stellen über das Programm verteilt wären, an zentraler Stelle definiert. Die Anpassungen des Quellcodes zur Einbindung der gewünschten Funktionalität müssen aber trotzdem erfolgen. Dies wird in der AOP durch die Technik des ≪Webens≫ realisiert. Der Aspekt-Code wird dabei vom Compiler an definierten Stellen in der Kernanwendung eingewoben. Das Verhalten der ursprünglichen Klassen kann dadurch im Nachhinein verändert bzw. erweitert werden. Man kann Ebene und Zeitpunkt des Webens unterscheiden. Normalerweise bearbeiten AO-Konzepte den Quellcode bzw. Bytecode auf der Klassenebene, entsprechend der zugrunde liegenden Klassenstruktur des Basisprogramms. Es existieren aber auch Ansätze direkt auf Objektebene zu arbeiten, d.h. gezielt Objektinstanzen zu manipulieren. Wird der Aspekt-Code statisch in den Quellcode eingewoben, so findet der Webevorgang bereits zur Compilezeit statt. Vorraussetzung dafür ist, dass der Quellcode der Basisanwendung zur Verfügung steht. Der Webevorgang kann aber auch zur Lade- oder Laufzeit durchgeführt werden, was ggf. die Möglichkeit eröffnet, den Aspekt dynamisch aktivieren bzw. deaktivieren zu können.

Joinpoint Die Stellen, an denen Aspektcode eingewoben werden kann, werden ≪Joinpoints≫ genannt. Ein Joinpoint ist nichts anderes als ein definierter Punkt im Programmablauf, wie z.B. der Aufruf einer bestimmten Methode oder der Zugriff auf eine Objektvariable. Mitunter werden Joinpoints dabei explizit von ≪Joinpoint Shadows≫ unterschieden. Ein Punkt in einem statischen Programm wird zunächst nur als Joinpoint Shadow bezeichnet, erst wenn dieser Programmpunkt zur Laufzeit tatsächlich erreicht und ausgeführt wird, wird daraus ein Joinpoint. In dieser Arbeit wird diese Unterscheidung nicht weiter verwendet, im Zusammenhang

mit einer Querysprache für Joinpoints sind vor allem die statischen Joinpoint Shadows relevant, eine explizite Abgrenzung zu erreichten Joinpoints ist in diesem Kontext nicht notwendig oder wird gesondert erwähnt.

Pointcut Ein Pointcut definiert eine Menge von Joinpoints. Pointcuts können dabei allgemein als Prädikate verstanden werden, die aus allen vorhanden Jointpoints eines Programms eine Teilmenge anhand ihrer logischen Aussagen auswählen. Der Pointcut dient als Auslöser für den an ihn gebundenen Aspekt-Code. Wird ein Joinpoint der Joinpoint-Menge im Programmablauf erreicht, so wird der entsprechende (an diese Stelle eingewobene) Aspekt-Code ausgeführt. Das Signal, dass ein bestimmter Joinpoint erreicht wurde, der in der Folge die Aspekt-Ausführung auslöst, wird ≪Trigger≫ genannt. Die Joinpoints müssen nicht unbedingt explizit angegeben werden, sie können auch anhand semantischer oder syntaktischer Muster identifiziert werden. Der Vorgang, Joinpoints anhand von Selektionskriterien als Menge auszuwählen, wird ≪Quantifizierung≫ genannt. Spezielle Befehle zur Definition von Joinpoints bzw. Joinpoint-Mustern (die auch den Gebrauch von Wildcards erlauben) sind wesentlicher Bestandteil der aspektorientierten Spracherweiterungen zur Einbindung von Quantifizierung. Die Joinpoint-Menge eines Pointcuts kann dabei prinzipiell aus einer Zusammenstellung von Joinpoints jedweder Art bestehen. Sinnvollerweise wird in einem Pointcut aber meist eine homogene Menge von Joinpoints gleicher Art gebündelt, z.B. Aufrufe auf einer bestimmten Methode. Je spezieller die Beschränkung der Joinpoint-Menge dabei ist, desto konkreter kann der Programm-Kontext, in dem ein solcher Joinpoint ausgeführt wird, bestimmt werden. Der Zugriff auf Kontextinformationen, wie z.B. das aktuell behandelte Objekt oder die Werte von Methodenparametern, ist je nach Anforderung notwendig für die Umsetzung eines Aspekts.

Advice Ein Advice ist ein Stück Aspekt-Code, der an ausgewählten Joinpoints ausgeführt werden soll. Die Advice-Beschreibung definiert die Art und Weise der Bindung des Codes an das Basisprogramm. Es existieren drei Möglichkeiten den zusätzlichen Code einzubinden: er kann vor dem Joinpoint ausgeführt werden (before-Advice), er kann hinter dem Joinpoint ausgeführt werden (after-Advice) einzufügen, oder er kann anstelle eines Joinpoints ausgeführt werden (around-Advice). In letzterem Fall wird die ursprüngliche Anweisung bei dem betreffenden Joinpoint vollständig durch neuen Code ersetzt, sie kann aber ggf. innerhalb des Aspekt-Codes erneut aufgerufen werden (in AspectJ der Befehl `proceed`).

Inter-type Deklaration Eine spezielle Form von Aspekten stellen die inter-type Deklarationen dar. Mit ihnen kann eine bestehende Deklaration nachträglich erweitert werden. Einer Klasse können also beispielsweise neue Methoden oder Eigenschaften hinzugefügt werden.

Die aufgeführten Begrifflichkeiten sowie ihre Umsetzung variiert im Umfeld der AOP, im Kern ist es aber stets die Komposition von Aspekt-Code und seine Bindung an definierte Joinpoints, die einen vollständigen Aspekt ausmacht. Vorteilhaft bei der Verwendung von Aspekten ist wie bereits erläutert die Möglichkeit Crosscutting Concerns zu modularisieren. Darüber hinaus sind Aspekte äußerst flexibel, sie können einfach und ohne Änderungen an der Basisanwendung eingefügt bzw. wieder entfernt werden. Nachteilig ist dagegen die erschwerte Nachvollziehbarkeit

des Programmablaufs. Insbesondere durch die Möglichkeit der Quantifizierung können bisweilen auch unbeabsichtigte und unerwünschte Effekte auftreten, deren Ursache nur schwer zu lokalisieren ist. Der eingewebte Code und die zusätzlichen Mechanismen führen zudem zu einer höheren Komplexität im Programmablauf und wirken sich eventuell nachteilig auf Performance aus.

2.3 Object Teams

Mit Object Teams wurde ein Programmiermodell entwickelt, dass zum Ziel hat, die bestehenden Probleme bei der Modularisierung von Software mit der Verwendung aspektorientierter Konzepte besser bewältigen zu können. Object Teams greift dabei einige der im vorigen Kapitel vorgestellten Ansätze auf und vereint diese mit neuen Ideen. Die Sprache kommt dabei mit nur wenigen Erweiterungen gegenüber der Basissprache Java aus, was den Einstieg und das Verständnis der Konzepte erleichtern soll. Im Folgenden werden die wichtigsten Eigenschaften und Konzepte von Object Teams vorgestellt.

Die Modularisierung objektorientierter Programme wird anhand der strukturellen Eigenschaften eines Systems vorgenommen. Klassen abstrahieren die wesentlichen Bestandteile realer Objekte und stehen über Vererbungs- und Assoziationsrelationen miteinander in Beziehung. Neben der Struktur ist aber auch die Zusammenarbeit der Objekte ein entscheidendes Kriterium für die Abläufe im System. Ein Objekt steht immer in einer oder mehreren Kollaborationsbeziehungen zu anderen Objekten. In jeder Kollaboration nehmen Objekte eine bestimmte Rolle an, die je nach Zweck und Aufgabenbereich der Kollaboration unterschiedliche Formen annehmen kann.

Die Klassenhierarchie bietet keine Konstrukte, um diese Kollaborationen bei der Modularisierung in gesonderter Weise zu berücksichtigen. In Object Teams wurde daher das Konzept von Rollen und Teams eingeführt, mit dem Kollaborationen in die Modularisierung einbezogen werden können.

2.3.1 Teams und Rollen

Ein Team stellt den Kontext einer Kollaboration dar, Objekte nehmen innerhalb eines Teams eine bestimmte Rolle an. Teams bilden spezielle Container-Klassen, deren Kollaborationsbeziehungen in Form von inneren Klassen, den Rollen, eingebunden sind. Im aspektorientierten Sinne ist das Konzept der Rollen und Teams vergleichbar mit den Aspekten. Das Modul wird separat von der Basisanwendung aufgeschrieben. Die Rollen können an Basisklassen gebunden werden und adaptieren deren Funktionalität mittels aspektorientierter Webetechnik. Ein solches Team kann im Übrigen gesondert aktiviert und deaktiviert werden (mittels der Team-Methoden `activate` bzw. `deactivate`), die Einbindung eines Aspekts lässt sich in Object Teams also auch zur Laufzeit steuern.

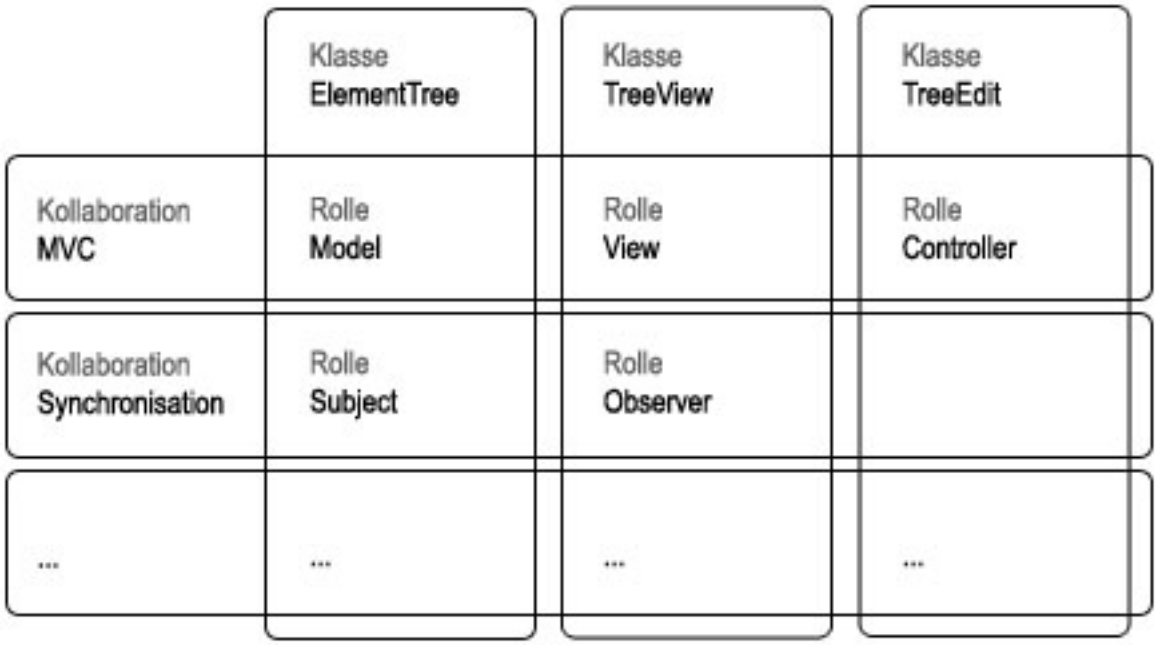

Abbildung 2.1: Klassen übernehmen verschiedene Rollen bei der Zusammenarbeit

Zur Veranschaulichung des Rollen-Teams-Konzept betrachten wir die Implementierung des Observer-Patterns in Object Teams [Her07, Mos03]. Die Kollaborationsbeziehung zwischen Subject und Observer ist zunächst als abstraktes Team implementiert. Das Team `ObserverPattern` kapselt die für die Zusammenarbeit notwendige Funktionalität, es übernimmt die Aufgabe des Konnektors. Innerhalb des Teams existieren die Rollen `Subject` und `Observer`. Ein Subject besitzt eine Liste seiner Observer. Mittels der Methoden `addObserver` und `removeObserver` auf Subject-Seite, bzw. `start` und `stop` auf Observer-Seite kann ein Objekt als Observer an- und abgemeldet werden. Die Methode `changeOp` benachrichtigt die Observer, dass eine Zustandsänderung am Subject stattgefunden hat. Die Methode `update`, die danach den Aktualisierungsmechanismus in Gang setzt, bleibt zunächst abstrakt (und damit auch seine Rolle und das Team). Erst bei der Bindung an eine bestehende Basisanwendung werden Klassen an die Rollen gebunden und übernehmen deren Funktionalität. An dieser Stelle werden die Methoden gebunden und es kann definiert werden, was im Falle der `update`-Methode konkret geschehen soll.

```
1 public abstract team class ObserverPattern {
2
3   protected abstract class Subject {
4         private LinkedList observers = new LinkedList();
5
6         void addObserver    (Observer o) {observers.add(o);}
7         void removeObserver (Observer o) {observers.remove(o);}
8
9         public void changeOp() {
```

```
10            // für alle Observer o der Liste: o.update(this)
11        }
12
13    protected abstract class Observer {
14            abstract void update(Subject s);
15            public void start (Subject s) {s.addObserver(this);}
16            public void stop  (Subject s) {s.removeObserver(this);}
17    }
18 }
```

Listing 2.1: Observer-Pattern in Object Teams

Ein Beispiel für die Anwendung des Observer-Patterns ist die Team-Klasse `ObserveElementTree`. Ein `ElementTree` sei dabei die Implementierung einer Baumstruktur, die als Container für Elemente dient. Die Klasse `TreeView` repräsentiert die grafische Darstellung für einen solchen Baum. Die TreeView soll stetig über Änderungen am Modell informiert werden und ihre Ansicht entsprechend aktualisieren, sie übernimmt die Rolle des Observers. Der `ElementTree` ist Gegenstand der Bearbeitung, er nimmt die Rolle des Subjects an. Mit dem Schlüsselwort `playedBy` werden die Rollen an ihre Basisklassen gebunden. Die Definition der Joinpoints wird mittels sogenannter *callin* und *callout* Bindungen vorgenommen.

```
1  public team class ObserveElementTree extends ObserverPattern {
2
3      public class Observer playedBy TreeView {
4          // callout binding
5          update   -> updateView;
6          // callin binding
7          start    <- after  createElement;
8          stop     <- before deleteElement;
9      }
10
11     public class Subject playedBy ElementTree {
12         changeOp <- after editElement;
13     }
14 }
```

Listing 2.2: Beispiel für die Anwendung des Observer-Patterns

callout Eine callout-Bindung dient der Weiterleitung von Aufrufen an Originalmethoden der Basisklasse. Abstrakte Rollen-Methoden, wie die Methode `update`, werden somit an klassenspezifische Methoden gebunden. Die Pfeilrichtung symbolisiert den Kontrollfluß, die Rolle ruft eine Methode der Basis auf.

callin Eine callin-Bindung entspricht im aspektorientierten Sinne dem Prinzip der Advices. Hier löst die Basis einen Methodenaufruf an der Rolle auf. Umgesetzt wird dieser Aufruf durch Code-Weaving an den entsprechenden Joinpoints. Die Abfolge wird über die Modifikatoren `before`, `after` und `replace` bestimmt (analog zu before, after und around-Advices). Bei der Verwendung von `replace` kann der Zugriff auf die Original-Methode mit einem sogenannten *base-call* der Form `base.<Name der Rollenmethode>` erfolgen.

2.3.2 Parameter Mapping

Sowohl bei der callout als auch bei der callin-Bindung werden eventuell vorhandene Parameter an die gebundenen Methoden weitergereicht. Lassen die Signaturen der Rollenmethode und ihrer korrespondierenden Basismethode eine direkte Abbildung zu (bestenfalls sind die Signaturen gleich), so kann das Mapping implizit ohne weitere Angaben stattfinden. Andernfalls können Parameter und Rückgabe mit den Schlüsselwörtern `with` und `result` auch explizit zugeordnet werden, wobei es ebenfalls möglich ist, Modifikationen an Typ oder Wert vorzunehmen.

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

Listing 2.3: Beispiel für Parameter Mapping aus [Her07]

2.3.3 Lifting und Lowering

Rollen existieren nur innerhalb ihrer Teams. Die Implementierungen von einer gebundenen Rolle und ihrer Basis sind unabhängig voneinander. Die Basis-Objekte haben keine Kenntnis von ihren Rollen und vice versa. Dennoch sind Basis und Rolle durch ihre Bindung aneinander gekoppelt, je nach Programmkontext muss ein Objekt den Typ der Basis oder den Typ der Rolle annehmen. Zur Realisierung dieser Bindung existiert in Object Teams der sogenannte Translationspolymorphismus [HHM04]. Die Typüberführung findet in zwei Richtungen statt und wird automatisch von der Laufzeitumgebung gesteuert. Beim Lifting wird eine Instanz der Basisklasse in eine Instanz der Rollenklasse überführt, dafür muss ggf. eine neue Rolleninstanz angelegt werden. Lifting ist notwendig zur Umsetzung der callin-Bindungen. Das Gegenstück zum Lifting ist das Lowering. Beim Lowering wird von der Rolle zurück zur Basis navigiert, wie es bei callout-Bindungen der Fall ist. Die Umsetzung des Lowerings ist einfach, da jede Rolleninstanz eine Referenz auf ihre Basisinstanz besitzt. Aus konzeptionellen Gründen gestattet Object Teams der Rolle allerdings keinen direkten Zugriff auf ihre Basis.

2.3.4 Selektion von Joinpoints

Im Unterschied zu AspectJ existieren in Object Teams keine Pointcuts. Der Begriff und das Konstrukt ≪Pointcut≫ wird in [Her06b] in Frage gestellt. Die bisher gezeigten Möglichkeiten der Aspektbindung von Object Teams in den callin-Bindungen sind allein durch Method-Call-Interception realisiert, d.h. die Ausführung einer Methode dient als Joinpoint. Es handelt sich dabei grundsätzlich um Execution-Joinpoints, also die Methodenausführung, im Gegensatz zu Call-Joinpoints, dem Methodenaufruf. Die Unterscheidung ist besonders im Hinblick auf den Kontext wichtig, da ein Call-Joinpoint im Kontext des Aufrufers, eine Execution-Joinpoint dagegen im Kontext des Aufgerufenen liegt. Weiterhin gibt es Unterschiede etwa bei rekursiven Aufrufen oder dynamischen super-Aufrufen, die aber in diesem Rahmen nicht weiter vertieft werden sollen. Die Behandlung von Joinpoints, die über eine einfache Method-Call-Interception hinausgehen, ist in OT/J in drei Teile gegliedert: Event Filtering (Guards), spezielle Pointcut-Klassen und Quantifizierung (Joinpoint Queries). Die Teile können sowohl einzeln verwendet als auch miteinander kombiniert werden.

Guard Predicates

Zur Filterung von Joinpoints auf Basis dynamischer Eigenschaften dienen Guard Predicates, mit denen zustandsabhängige Ausdrücke formuliert und ausgewertet werden [HHMW05]. Joinpoints werden in diesem Zusammenhang als Ereignisse aufgefasst, die im Programmverlauf stattfinden. Der Aspekt-Code ist die Aktion, die in Folge eines solchen Ereignisses ausgeführt werden soll. Mithilfe von Wächterprädikaten werden Bedingungen angegeben, mit welchen die Ereignisse anhand ihres aktuellen Kontextes gefiltert werden.

```
1 public class ARole playedBy ABase {
2   // ...
3   void absCalc(int x) <- replace calc(int y) when (x < 0);
4 }
```

Listing 2.4: Ein Guard-Ausdruck formuliert Bedingungen für den Kontext, in diesem Fall für den Parameter x.

Wird die Guard-Bedingung nach `false` ausgewertet, so wird der Ereignis-Trigger zurückgewiesen und das Verhalten der Basis bleibt unverändert. Erst im Falle der Auswertung nach `true` wird der Trigger akzeptiert und die Bindung wird aktiv (in gleicher Weise, wie eine Bindung ohne Guard). Die Einbeziehung von Guards als Sprachfeature soll verhindern, dass diese Bedingungen sonst in unübersichtliche, geschachtelte Konditional-Anweisungen ausgelagert werden müssen. Zudem sind Guards auf verschiedenen Ebenen anwendbar. Ein Guard kann dediziert eine Rollenmethode bzw. deren Bindung einschränken, er kann aber auch für eine Rolle selbst (gilt dann für alle Rollenmethoden) oder ein Team (gilt dann für alle Rollen des Teams) formuliert werden.

Pointcut-Klassen

Pointcut-Klassen in Object Teams werden dazu verwendet, Trigger miteinander zu kombinieren [Her06b]. Dies gilt sowohl für Trigger einfacher Methodenaufrufe an einer Basisklasse, wie für Trigger über einer statisch und/oder dynamisch eingegrenzten Joinpoint-Menge. Auf diese Weise können neue Trigger generiert werden, die in komplexer Weise von Zustand und Ablauf des Programms abhängig sind. Die Bezeichnung ≪Pointcut≫ soll hier explizit darauf hinweisen, dass das Zusammenspiel mehrerer Ereignisse für die Auslösung eines solchen Triggers verantwortlich ist. Mithilfe dieses Konstrukts ist es u.a. möglich Ereignisse zu beschreiben, die vom Kontrollfluss abhängig sind. Aspektbindungen, die ohne kombinierte Trigger auskommen, können dagegen ohne Verwendung der Pointcut-Klassen mit den aufgeführten Mitteln der Method-Call-Interception, Guard Predicates und Joinpoint Queries realisiert werden. Das Pointcut-Klassen-Konstrukt und dessen Syntax ist bisher nur in konzeptioneller Form vorhanden. Ein Beispiel dazu wird in Listing 5.14 in Kapitel 5 zu sehen sein.

Joinpoint Queries

Zur Quantifizierung werden Joinpoint Queries herangezogen, die es ermöglichen, Mengen von Programmelementen anhand verschiedener Selektionskriterien zu definieren. Für eine callin-Bindung kann also anstelle einer einzelnen Methode auch eine Query angegeben werden, die eine Menge von Methoden selektiert. Auf diese Weise ist es möglich, Elemente anhand bestimmter Merkmale auszuwählen, ohne alle einzeln aufführen zu müssen. Nähere Erläuterungen und Details der Querysprache sind Inhalt des nächsten Kapitels.

3 Die OT/J Joinpoint Querysprache

Mit der Querysprache wird das Konzept der Quantifizierung in Object Teams integriert. Queries dienen der Selektion von Programmelementen. Sie werden durch Angabe von Scope, Kind und Constraints spezifiziert. Der *Scope* definiert die Menge der Elemente, welche die Query für ihre Ergebnissuche heranzieht. *Kind* bestimmt, welche Art von Elementen gesucht wird. Ein (optionaler) *Constraint* bestimmt, welche Bedingungen ein Element dabei zu erfüllen hat.

Betrachten wir zur Veranschaulichung ein einfaches Beispiel. Der folgende Ausdruck soll uns alle öffentlichen Methoden eines Programms liefern:

```
allClasses.methods[isPublic]
```

Der Scope wird duch `allClasses.methods` auf die Menge aller Methoden aus allen Klassen festgelegt. Die Angabe `methods` bestimmt dabei die Art der Rückgabe, in diesem Fall werden Methoden verlangt. Zusätzlich wird nun mit dem Constraint `isPublic` ein Kriterium angegeben, welches eine Methode zu erfüllen hat, um in die Ergebnismenge aufgenommen zu werden.

Queries in Object Teams werden bereits zur Compilezeit ausgewertet und sind damit losgelöst von dynamischen Eigenschaften eines Programms. Die zur Selektion verwendeten Kriterien basieren auf statischen Eigenschaften.

Die OT/J-Joinpoint-Querysprache ist Gegenstand der aktuellen Entwicklung und daher noch nicht in allen Einzelheiten definiert. Im Folgenden werden die wesentlichen Bereiche der Querysprache auf ihrem aktuellen Stand nach [HH07] ausgeführt.

3.1 Das Query-Metamodell

Zum besseren Verständnis der Queries und ihrer Funktionsweise wird an dieser Stelle das Metamodell von Object Teams zur Selektion von Programmelementen gezeigt (siehe Abbildung 3.1). Die dargestellte Form bildet eine Synthese auf Grundlage verschiedener Informationen aus [HH07, Her06b, Sel06]. Das Metamodell gewährt (statische) Informationen über die Struktur eines Programms. Diese Form der Introspektion ermöglicht es der Querysprache, Selektionskriterien für die diversen Elemente wie Klassen oder Methoden eines Programms zu spezifizieren.

Im Fokus der Betrachtungen stehen dabei die Methoden, da sie vor allen anderen für die Bindung von Aspekten eine Rolle spielen.

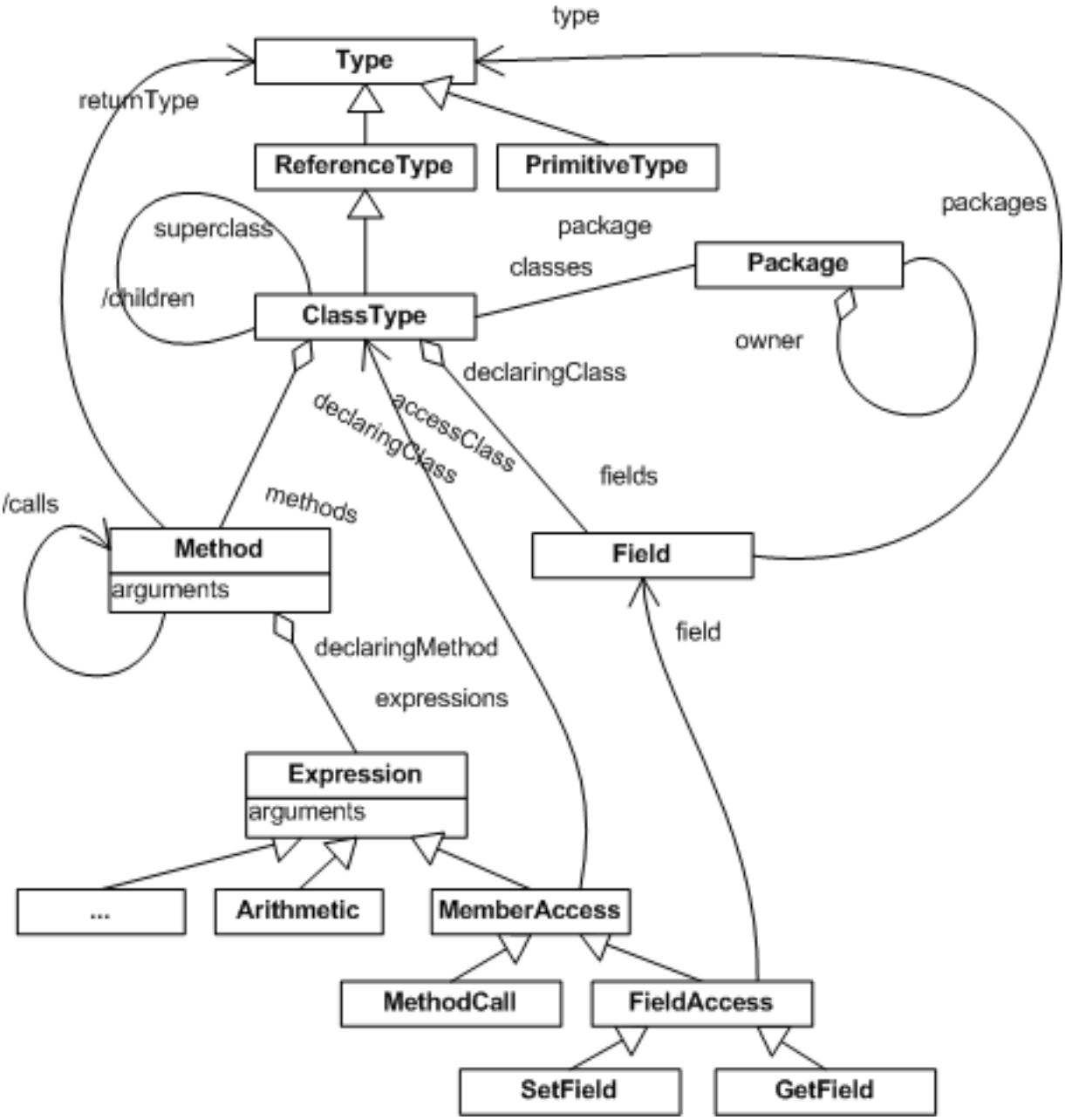

Abbildung 3.1: Ein Ausschnitt wichtiger Bestandteile des Query-Metamodells

Package repräsentiert Pakete. Von einem Paket kann zu den enthaltenen Paketen und Klassen navigiert werden.

ClassType repräsentiert Klassen. Eine Klasse enthält Methoden, Felder und Informationen zur Vererbungshierarchie.

Method repräsentiert Methoden. In einer Methode sind Signatur-Informationen über Rückgabetyp, Parameter sowie Zugriffs- und Polymorphiemodifizierer verfügbar. Eine Methode besteht aus einer Sequenz von Expressions, über diese kann die Methode u.a. ermitteln, welche anderen Methoden von ihr aufgerufen werden können.

Expression repräsentiert einzelne Anweisungen, z.B. eine arithmetische Operation oder einen MethodCall. Die verschiedenen Arten von Joinpoints (siehe 3.7) werden in Expressions gekapselt. Die Argumente einer Expression beinhalten die Elemente, die zur Ausführung herangezogen werden (z.B. das Feld, auf welches eine Access-Expression zugreift).

3.2 Struktur

Eine Query stellt einen einzelnen, funktionalen Ausdruck dar. Sie wird frei von Seiteneffekten ausgeführt, da zur Auswertung allein lesende Zugriffe auf den Elementen notwendig sind. Queries werden durch das Schlüsselwort `query` gekennzeichnet. Die Signatur besteht aus einem Bezeichner, einem Rückgabetyp und einer Liste von Parametern. Als Rückgabe können die Typen `int`, `boolean` und `String` sowie alle Typen des Query-Metamodells verwendet werden, zudem Sets und Listen dieser Metamodell-Typen.

Queries werden in zwei Ausprägungen verwendet, den internen Queries und den Toplevel Queries. Die Unterscheidung bezieht sich allein darauf, ob die Ergebnismenge zur Bindung herangezogen wird. Interne Queries sind Ausdrücke, die in anderen Queries Verwendung finden. Sie haben keinerlei Einschränkung bezüglich des Ergebnisses und dienen u.a. der übersichtlicheren Strukturierung. Toplevel Queries selektieren Mengen von Joinpoints für die Verwendung in callin-Bindungen. Sie heißen ≪toplevel≫, da sie im Gegensatz zu internen Queries an dieser Stelle in einem OT/J-Programm zu Tage treten. Nur Queries mit einer für die Joinpoint-Interception tauglichen Rückgabe können in einer callin-Bindung verwendet werden, dies bedeutet letztlich eine Rückgabe vom Typ `Expression` bzw. `Set<Expression>`. Da, wie bereits erläutert, bei der Bindung nicht der Call- sondern der Execution-Joinpoint ausschlaggebend ist, spricht alles dafür, an dieser Stelle ebenfalls ein `Method` bzw. `Set<Method>` zu akzeptieren, wobei in diesem Fall eben genau die Ausführung einer Methode als Joinpoint dient.

```
1 query Set<void MethodCall(int x)> intSetterCalls(ReferenceType aClass)
2 {...};
```

Listing 3.1: Query-Signatur

Die Deklaration der Query in Listing 3.1 stellt sicher, dass in dem Ergebnis-Set nur Joinpoints von Aufrufen an Methoden mit Rückgabe `void` und einem Parameter vom Typ `int` enthalten sind.

Zur Spezifizierung der Joinpoints, welche durch die Query selektiert werden, können Basistypen oder Typen der Anwendung als Parameter angeben werden. Im Aufruf der Query in Listing

3.2 bei einer callin-Bindung wird z.B. die Klasse `MyBase` als Parameter spezifiziert.

```
1 myOp <- replace query intSetterCalls(MyBase);
```

Listing 3.2: Query mit Parameter

3.3 Syntax

Queries werden als formale Ausdrücke definiert. Die verwendeten Konstrukte (siehe auch Abbildung 3.2) finden ihre Entsprechungen in den Funktionen höherer Ordnung und anderen Bereichen der funktionalen Programmierung

3.3.1 Vordefinierte Ausdrücke

Es existieren die folgenden vordefinierten Mengen, die als Einstiegspunkt für Navigation und Selektion einer Query dienen können. Eine Menge bezieht sich dabei stets auf den Kontext der aktuellen Basis-Anwendung.

`Set<Package> allPackages` - alle Pakete der Anwendung.
`Set<ClassType> allClasses` - alle Klassen (aller Pakete) der Anwendung.
`Set<Method> allMethods` - alle Methoden (aller Klassen) der Anwendung.

3.3.2 For-Ausdruck

```
for (Type element : collection) expression
```

Ein `for`-Ausdruck iteriert über eine Menge von Elementen. Der Rumpf des Ausdrucks ist ebenfalls als Query formuliert und wird auf jedes Element der Menge angewendet. Die Funktionsweise entspricht der Funktion höherer Ordnung «map». Als Ergebnis wird eine Menge zurückgeliefert, in der jedes Element der Eingangs-Menge mittels des Rumpf-Ausdrucks in ein Element der Ergebnis-Menge überführt wurde.

3.3.3 If-Ausdruck

```
if (condition_expression) then_expression else else_expression
```

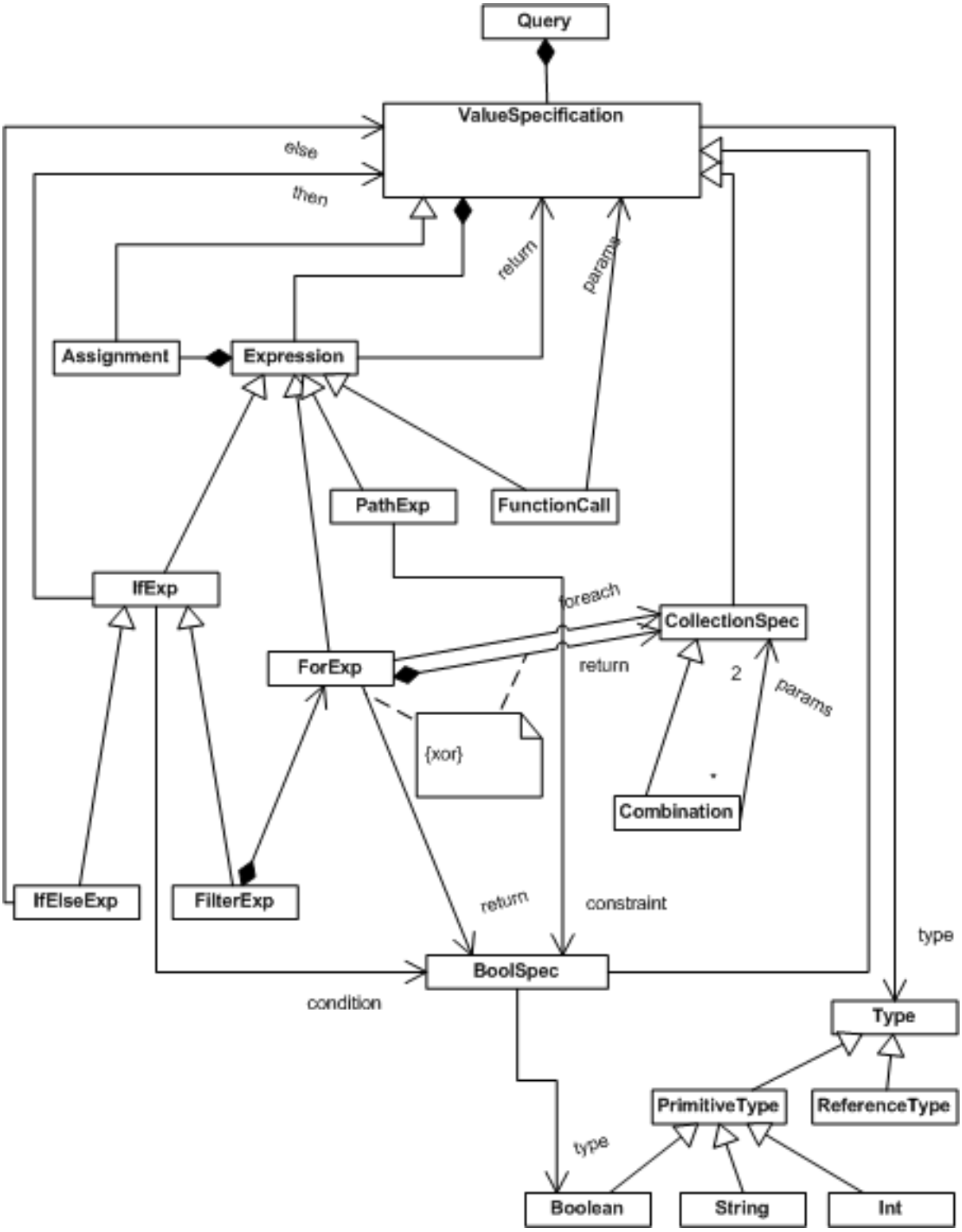

Abbildung 3.2: Metamodell der Querysyntax

Der `if`-Ausdruck entspricht in seiner Funktionsweise der üblichen Konditional-Anweisung, für das Ergebnis in `then`- und `else`-Zweig muss allerdings wieder ein Ausdruck stehen. Da es sich nicht um imperative Anweisungsblöcke handelt muss der `else`-Zweig zwingend vorhanden sein, ansonsten wäre das Ergebnis undefiniert. Eine Ausnahme bildet die Kombination von `for` und `if`, in diesem Fall kann der `else`-Zweig ausgelassen werden. Nur die Elemente der Eingangs-Menge, welche die Bedingung erfüllen, werden dann in die Ergebnis-Menge übernommen. Die Funktionsweise dieser Kombination entspricht der Funktion höherer Ordnung «filter».

3.3.4 Lokale Variablen

```
VarType var1 = init_expression; expression_using_var1
```

Lokale Variablen können nach dem Prinzip der «let»-Konstruktion der funktionalen Sprachen definiert werden. Die Zuweisung erfolgt mittels eines Ausdrucks (kann also auch durch eine interne Query definiert werden). Das Ergebnis wird nur einmal zugewiesen und bleibt konstant. Eine erneute Zuweisung oder anderweitige Veränderung ist nicht möglich.

3.4 Pfadsyntax

Für eine Teilmenge der beschriebenen Konstrukte existiert auch eine kürzere Schreibweise in Form einer Pfadsyntax, mit der weniger komplexe Ausdrücke knapp und übersichtlich formuliert werden können.

Die intuitive Schreibweise für eine Zugriffspfad besteht aus der Aneinanderreihung von Segmenten. Als Ausgangsknoten muss dabei immer eine bekannte Variable aus dem aktuellen Scope verwendet werden (wie z.B. die vordefinierten Ausdrücke `allPackages`, `allClasses` und `allMethods`). Ein Segment kann für ein einzelnes Element oder eine Menge von Elementen stehen.

```
prefix.feature
```

Im Falle das `prefix` einer Menge entspricht findet die Auswertung in Form eines `for`-Ausdrucks statt:

```
for (PrefixType p : prefix) p.feature
```

Für eine Menge von Elementen kann auch durch Angabe eines Constraints eine bedingte Selektion formuliert werden.

```
prefix[constraint]
```

Die Auswertung entspricht der ≪filter≫-Anweisung:

```
for (PrefixType p : prefix) if (constraint) p
```

Sind zwei Segemente mittels `..` voneinander getrennt (`prefix..feature`), so wird das Segment vor dem Punkt beliebig oft angewandt (also z.B. auch `prefix.prefix.feature`).

Desweiteren können Constraints zur Formulierung von quantorenlogischen Ausdrücken verwendet werden. Ein Fragezeichen markiert einen Existenzquantor, ein Ausrufezeichen einen Allquantor. Die Auswertung erzeugt in diesem Fall keine Collection, sondern liefert einen Wert vom Typ boolean.

`prefix?[constraint]`
steht für die Aussage: $\exists p \in prefix \bullet constraint(p)$

`prefix![constraint]`
steht für die Aussage: $\forall p \in prefix \bullet constraint(p)$

Zur Veranschaulichung betrachten wir das Beispiel eines Query-Ausdrucks, der uns alle Klassen, die wenigstens eine öffentliche Methode besitzen, liefern soll:

```
for (ClassType c: allClasses)
  for (Method m: c.methods)
    if (m.isPublic)
      c
```

Der selbe Ausdruck in Pfadsyntax:

```
allClasses[methods?[isPublic]]
```

Die Schreibweise ist deutlich kürzer, die Leserichtung verfolgt dabei intuitiv den Selektions-Pfad. Die Constraints sind durch die umschließenden eckigen Klammern deutlich gekennzeichnet. Auch eine Verschachtelung, wie in diesem Fall, ist vergleichsweise einfach darstellbar.

3.5 Query Introduction

Queries können auch als Erweiterung der vordefinierten Typen des Metamodells formuliert werden, eine Form der Inter-type declaration. Auf diese Weise können eigene Kurzschreibweisen in die Pfadsyntax integriert werden.

```
1 query Set<Method> ClassType.getters() {
2   methods[isGetter]
3 }
```

Listing 3.3: Query Introduction für Getter-Methoden

In einer Query kann dann `myclass.getters` anstatt `myclass.methods[isGetter]` benutzt werden.

Der Mechanismus der Query Introduction eröffnet die Möglichkeit, dass jeder Entwickler maßgeschneiderte Erweiterungen für seine individuellen Anforderungen einbinden kann. Alternativ könnte man die Erweiterungen natürlich von vornherein in die Sprache integrieren. Allerdings läuft man dabei sehr schnell Gefahr, die Sprache mit einer großen Anzahl von Features zu überladen. Will man sich dagegen auf wenige Features beschränken, so droht andererseits die Gefahr, dass diese zu unflexibel sind, um die Anforderungen der Entwickler zu erfüllen. Die Erweiterungen mittels Query Introduction sind dagegen individuell verwendbar. Sie können in Bibliotheken gesammelt und gegliedert werden. Ein Grundstock solcher Bibliotheken kann die Sprache um allgemein nützliche Erweiterungen ergänzen, womit ein etwaiger Mangel an Sprachfeatures verhindert wird. Das Bibliothekenkonzept bleibt dabei stets flexibel genug, um die Erweiterungen je nach Anforderung anzupassen bzw. zu ersetzen.

3.6 Vergleichbare Abfragesprachen

Es existieren selbstverständlich bereits andere Sprachen, deren Zweck und Aufbau der vorgestellten Querysprache ähneln. An erster Stelle ist die Object Constraint Language (OCL) [W3C06] zu nennen. Sie ist Bestandteil der UML und ist etablierter Standard bei der textuellen Spezifikation von Aussagen zu Programmelementen. Die OCL ist allerdings sehr mächtig und bietet zahlreiche Features, die für die Zwecke der Querysprache irrelevant sind. Nachteilig ist vor allem, dass die OCL keine Pfadsyntax besitzt, welche die für unsere Zwecke bevorzugten aussagekrätigen Kurzschreibweisen erlaubt. Die Sprache XQuery [W3C07] mit dem zugrunde liegenden XPath, welche zur Spezifikation von Abfragen zu XML-Dokumenten dient, erfüllt diese Anforderung und ermöglicht auch die funktionale Programmierung. Allerdings ist die Syntax vergleichsweise kompliziert. Insbesondere das Prinzip der Navigation zu den Elementknoten entlang verschiedener Achsen (`Achse::Knotentest[Prädikat]`) wurde in ähnlicher Form in der OT/J-Querysprache integriert, mit dem Unterschied, das XPath für beliebige XML-Dokumente vorgesehen ist, OT/J-Queries dagegen direkt auf dem Query-Metamodell aufbauen. Bei der Navigation exitieren noch einige Konzepte, die bei Bedarf ebenfalls in die Querysprache übernommen werden könnten. So kann in XPath beispielweise die Reihenfolge der Elemente einer Menge berücksichtigt werden. Dies könnte in der Querysprache mit dem Ausdruck `prefix.feature[position]` formuliert werden, wobei `position` die Stelle des jeweiligen Elements beziffert. Weiterhin existieren in XPath verschiedene Typen von Achsen zur Navigation im Elementbaum, welche in Queries zumindest teilweise für die Navigation in der Vererbungshierarchie der Klassen nützlich sein könnten (siehe Abbildung 3.3). Für die Realisierung könnten Query Introductions verwendet werden.

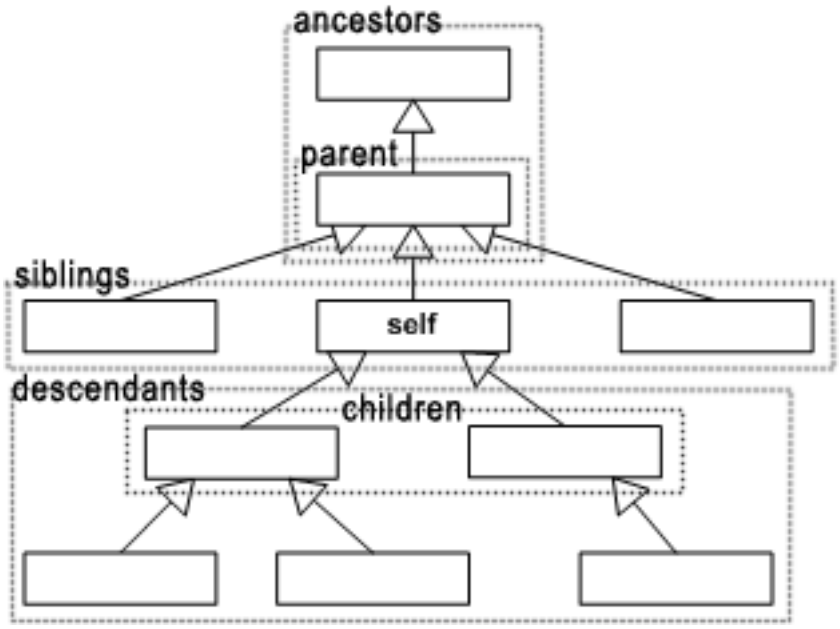

Abbildung 3.3: Navigationsachsen für die Vererbungshierarchie von Klassen

3.7 Joinpoint Klassifizierung

Die Joinpoints eines Programms können in horizontaler wie vertikaler Dimension betrachtet werden. In der horizontalen Dimension schneiden die Joinpoints eines Aspekts über strukturelle Module hinweg (crosscutting concerns), in der vertikalen Dimension liegen sie auf verschiedenen Granularitätsebenen der Programmeinheiten verteilt. So können Joinpoints auf der Ebene atomarer Anweisungen definiert werden oder aber auch auf der Ebene von in komplexer Art miteinander verbundenen Einheiten.

3.7.1 Atomare Joinpoints

Atomare Joinpoints sind Joinpoints, die elementaren Sprachkonstrukten entsprechen [Sel06]. Zu den relevantesten zählen Methodenaufrufe, Feldzugriffe und die Erzeugung von Objekten. Wie bereits erwähnt ist es bei der Quantifizierung häufig sinnvoll, Mengen von gleichartigen oder ähnlichen Joinpoints zusammenzufassen. Für die verschiedenen Arten von Joinpoints bietet sich daher eine Klassifzierung in sogenannten Multihierarchien an, die verschiedene Graduierungen (vom Allgemeinen zum Speziellen) der Differenzierung erlauben.

In [Her06b, Sel06] werden bereits einige Multihierarchien gezeigt.

Die zwei Arten von Feldzugriffen (Abbildung 3.4: `get` - lesender Zugriff, `set` - schreibender Zugriff) werden hier beispielsweise unter dem Typ `fieldAccess` vereint. Diese Joinpoint-Art

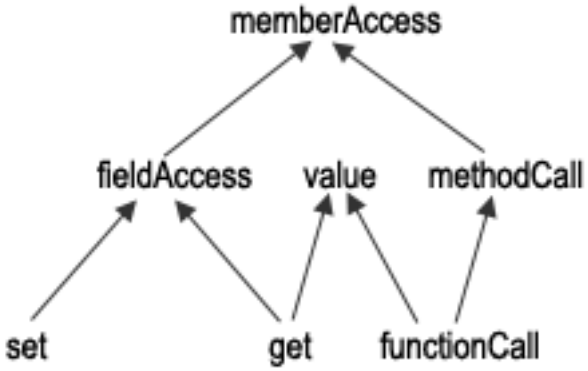

Abbildung 3.4: Eine Multihierarchie verschiedener Joinpoints

ist wiederum Bestandteil des Typs `memberAccess`, worin sowohl Feldzugriffe als auch Methodenaufrufe zusammengefasst sind. Jeder Joinpoint-Typ hat damit eine (oder mehrere) Entsprechungen auf der Ebene direkt identifizierbarer Joinpoints.

Eine Anweisung der Form `obj.field = value` bildet also einen Joinpoint der Art ≪set≫. Eine naheliegende Möglichkeit ist es, die Joinpoints auf Ebene der Expressions zu erfassen, wie sie im Query-Metamodell vorhanden sind. Ein Joinpoint der Art ≪set≫ wird dann mit einer Expression vom Typ `setField` gekapselt. Die Expression bietet auch den passenden Rahmen, um die mit dem Joinpoint bzw. seiner Signatur verbundenen Informationen verfügbar zu machen. Im Falle des schreibenden Feldzugriffs spielen beispielweise die referenzierte Klasse und der Name des Feldes ein Rolle.

Anmerkung: In [Sel06] wird gesagt, dass abstrakte Bezeichner wie z.B. `getField` und `setField` in Joinpoint-Queries verwendet werden können, um einen Joinpoint zu beschreiben. Es wird jedoch nicht konkret angegeben, wie dies in der Syntax dargestellt wird. Sofern die Jointpoint-Bezeichner auf Ebene der Expressions angewendet werden möchte ich folgende Syntax vorschlagen:

```
1 query boolean Expression.isSetField() {
2   (this instanceof SetField)
3   //alternativ: this.isOfType(SetField)
4 }
```

Listing 3.4: Prüfung auf set-Joinpoint mittels SetField-Expression

Die Joinpoint-Signatur ist an dieser Stelle noch nicht berücksicht. Die involvierten Typen müssen u.U. konkret für die jeweilige Art eines Joinpoints offengelegt werden.

```
1 query boolean Expression.isSetField_TypeAndField(ClassType t, Field f) {
2   if (this instanceof SetField)
3     SetField exp = (SetField) this;
4     exp.accessClass.equals(t) && exp.field.equals(f)
```

```
5     else
6       false
7   }
```

Listing 3.5: SetField-Expression mit Berücksichtigung der Signatur

Der Typvergleich wurde hier exemplarisch mit `equals` dargestellt, ist aber letztlich (wie auch die übrige Syntax) von der Implementierung des Metamodells abhängig. Der Name der Query wurde gegenüber dem vorigen Beispiel verändert, da bisher nicht vorgesehen ist, dass die OT-Querysprache Overloading unterstützt.

Die Multihierarchie der Joinpoints lässt sich nun mittels der Vererbungshierarchie der Expressions abbilden. Der Expression-Typ `SetField` erbt also von `MemberAccess` usw. Da in den Multihierarchien ggf. auch Mehrfachvererbung verwendet wird, müssen in diesen Fällen zur Realisierung auch Java-Interfaces in die Vererbungshierarchie integriert werden.

3.7.2 Weitere Joinpoints

Neben den atomaren Joinpoints gibt es noch weitere Joinpoint-Arten, die nicht auf ein einzelnes bzw. eine fixe Sequenz von Hochsprachkonstrukten abgebildet werden. Beispiele dafür wären Schleifenkonstrukte oder Joinpoints, die sich etwa auf Ressourcennutzung wie Zeit oder Speicher beziehen. Die Implementierung (ggf. sogar die Definition) dieser Joinpoint-Arten ist im Allgemeinen aufwendiger als die der bisher genannten Joinoints. Die Menge der atomaren Joinpoints ist für unsere Zwecke aber vollkommen ausreichend, weswegen die Betrachtung weiterer, spezieller Joinpoint-Arten im Rahmen dieser Arbeit nicht weiter verfolgt wird.

Die Klassifizierung von Joinpoints der atomaren Ebene ist dank ihrer vergleichsweise einfach zu bestimmenden Struktur von Vorteil bei ihrer Verwendung in aspektorientierten Programmen. In der Praxis, insbesondere bei der Zusammenarbeit von Rollen und Teams in OT/J, wird allerdings meist auf Ebene von Methoden gearbeitet, da diese in objektorientierten Programmen den Hauptteil der Kommunikation und Zustandsveränderung tragen. Eine Joinpoint-Klassifizierung auf Ebene der Methoden wird in Kapitel 4 betrachtet. In diesem Zusammenhang werden auch die mit der Quantifizierung auftretenden Probleme bezüglich Bindung, Kontext und Korrektheit diskutiert.

4 Quantifizierung

4.1 Korrektheit von Pointcuts

Die korrekte Identifizierung von relevanten Joinpoints für bestimmte Anforderungen ist ein entscheidendes Kriterium für den erfolgreichen Einsatz aspektorientierter Konzepte. In der Praxis wird hierbei meist auf Ebene von Methoden gearbeitet, insbesondere bei der Zusammenarbeit von Rollen und Teams in OT/J, welche daher auch im Fokus dieser Betrachtungen liegen.

In vielen Fällen ist es nicht praktikabel, alle in Frage kommenden Joinpoints einer gewünschten Joinpoint-Menge explizit anzugeben. Stattdessen nutzt man das Prinzip der Quantifizierung, um Mengen von Joinpoints anhand spezifischer Kriterien auszuwählen [FF00]. Die Beeinflussung des Programms folgt dabei im Allgemeinen der Struktur: *Wenn im Programm Bedingung B erfüllt wird, dann führe Aktion A durch*, wobei B das Selektionskriterium einer Joinpoint-Menge ist und A der für diese Stelle vorgesehene Advice. Als Kriterien kommen statische (z.B. Signatur einer Methode) wie dynamische Informationen (z.B. Kontext der aktuellen Routine) in Frage. Bei den statischen Kriterien stellt der Einsatz von Wildcards wie dem Stern-Symbol (*), mit dem eine Joinpoint-Menge implizit über ein syntaktisches Muster erfasst wird, ein weit verbreitetes Mittel dar. Ein typischen Beispiel hierzu wäre ein `public void set*`, in dem alle Setter-Methoden inbegriffen sind.

Problematisch an dieser Vorgehensweise ist, dass das zugrunde liegende Programm einer bestimmten Form entsprechen muss [KGBM06], damit die Quantifizierung auch zu den gewünschten Ergebnissen führt. Das eigentlich Ziel des public void `set*`-Pointcuts ist schließlich, alle Methoden zu erfassen, die schreibenden Zugriff auf ein Feld einer Klasseninstanz nehmen. Die Eigenschaft, dass solche Methoden mit der Bezeichnung `set` am Anfang benannt sind, ist abgeleitet aus den de facto vorherrschenden Konventionen in der Programmierung. Sowohl Code professioneller Entwickler als auch automatisch generierter Code genügt im Allgemeinen den gängigen Regeln, welche Struktur und Benennung von Getter- und Setter-Methoden u.ä. vorgeben. An dieser Stelle wird versucht semantische Information per syntaktischer Konvention zu vermitteln. Durch die vorherrschenden Design-Richtlinien wird eine Annahme über den Aufbau des Programms getroffen.

Diese auf Konvention beruhende Verbindung ist aber fragil und stößt sehr bald an ihre Grenzen. Zum einen sind nicht alle inhaltlichen Konzepte eins zu eins auf die Syntax abbildbar. Allenfalls rudimentäre Informationen wie lesender bzw. schreibender Zugriff mittels `get` bzw. `set` sind auf diese Weise direkt im Programmcode darstellbar. Zum anderen verlangt die Vorgehensweise die

vollständige Einhaltung vorgegebener Konventionen, andernfalls werden fehlerhafte Ergebnisse produziert. Und selbst bei korrekter Umsetzung der Konvention können dennoch Probleme entstehen. So müsste bei unserem `public void set*`-Beispiel beachtet werden, ob nicht auch `put*`-Methoden mit schreibendem Feldzugriff existieren. Diese Methoden würden, da sie nicht dem Muster entsprechen, durch das Raster fallen. Dem gegenüber stünden Methoden mit beispielsweise einer Bezeichnung der Form `setup*`, die zwar dem Muster entsprechen, aber inhaltlich eigentlich nicht in die Joinpoint-Menge mit aufgenommen werden sollen.

Das Problem falsch identifizierter Joinpoints hängt auch mit der Robustheit von Pointcuts zusammen. Die meisten Programme unterliegen einem fortlaufenden Entwicklungsprozess. Eine einmal korrekt definierte Joinpoint-Menge hat keinen dauerhaften Bestand und kann nach einer Änderung des Basisprogramms plötzlich unvollständig oder fehlerhaft sein. Das Problem der mangelnden Robustheit von Pointcuts gegenüber Änderungen wird auch *Fragile Pointcut Problem* [KGBM06, KS04] genannt. Wie im Beispiel gesehen hat die Problematik zwei Komponenten: zum einen kann es wie bei `put*` zu einer falsch-negativ Identifizierung kommen, d.h. ein relevanter Joinpoint wird übersehen (*beta-Fehler*). Zum anderen kann es wie bei `setup*` zu einer falsch-positiv Identifizierung kommen, d.h. ein nicht relevanter Joinpoint wird fälschlicherweise in der Ergebnis-Menge aufgenommen (*alpha-Fehler*).

In der Literatur werden mehrere Lösungsansätze für dieses Problem vorgeschlagen. Viele laufen darauf hinaus, relevante Informationen zur korrekten Klassifizierung von Joinpoints bereits im Modell der Anwendung unterzubringen. Es wird (mehr oder weniger maschinell unterstützt) bereits auf die spätere Zuordnung von Joinpoints getrimmt. Dabei wird das Modell mit entsprechenden Informationen angereichert, auf die in der Folge zurückgegriffen werden kann, z.B. durch klassifizierende Annotationen im Quell-Code [KM05] oder durch Constraints, die Joinpoints und ihre Verknüpfungen untereinander weiterführend validieren [KGBM06]. Mithilfe dieser Strategien lässt sich das Problem syntaktischer Zuordnung vermeiden. Nachteil bei dieser Vorgehensweise ist aber, dass sie je nach Detaillierungsgrad sehr aufwändig und kompliziert ist. Fehler sind dabei nach wie vor nicht ausgeschlossen, die Fehlerquelle wird lediglich von der Code- auf die Design-Ebene verlagert.

4.2 Query-Unterstützung in ObjectTeams

Für die OT/J-Querysprache soll hier ein einfacherer Ansatz vorgestellt werden, der grundlegende semantische Informationen berücksichtigen kann. Dies wird mit bestehenden Mitteln umgesetzt, ohne dass zusätzliche Werkzeuge oder Techniken zum Einsatz kommen müssen. An dieser Stelle genügen die vorhandenen Erweiterungsmöglichkeiten der Querysprache mittels Query Introduction. Die dafür notwendigen semantischen Informationen können wie beim Reverse Engineering direkt aus dem Code entnommen werden, ohne dass eine gesonderte manuelle Behandlung notwendig ist.

Ähnlich, wie Joinpoints in Multihierarchien gegliedert werden können, lassen sich auch Methoden gliedern, um einen Überblick über ihre Art, Eigenschaften und Aufgaben zu geben. In

der UML-Modellierung können diese Informationen mittels Stereotypen an Methoden geheftet werden, womit sie entsprechend klassifiziert werden. Es existieren diverse alternative Taxonomien zur Klassifizierung, die je nach Einsatzgebiet in Bezeichnung und Definition varieren. Ein Vorschlag für die in der Praxis verwendeten Stereotypen wird in [DCM06] gemacht. Die hier genannten Stereotypen *Accessor* und *Mutator* werden im Umfeld der UML häufig unter den Namen als *query* und *modifier* geführt. Zwar stehen beide Varianten inhaltlich für die selbe Aussage, beim Reverse Engineering wird aber eine eher technische Sichtweise zur Definition herangezogen, während die Dinge in der UML eher konzeptionell betrachtet werden. Die exakte Definition ist eine Frage des Standpunkts. Auch in den folgenden Erläuterungen werden leichte Änderungen der ursprünglichen Definitionen vorgenommen, um sie für unser Einsatzgebiet anzupassen.

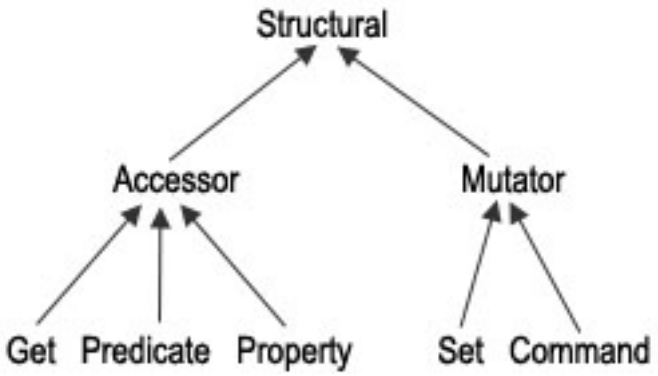

Abbildung 4.1: Klassifizierung struktureller Methoden

Betrachten wir zunächst die strukturellen Methoden (Abbildung 4.1), deren Vertreter auch bei der Definition von Pointcuts oft eine wichtige Rolle spielen. Strukturelle Methoden nehmen Zugriff auf die Eigenschaften eines Objekts und beschreiben oder verändern seinen Zustand. Sie unterteilen sich in Accessor- und Mutator-Methoden.

Accessor-Methoden sind Methoden mit nur lesendem Zugriff, die Informationen über ein Objekt liefern. Zu ihnen zählen *Get*-, *Predicate*-, und *Property*-Methoden. Die *Get*-Methoden sind einfache, meist atomare Zugriffs-Funktionen, die Felder eines Objekts auslesen und zurückliefern, z.B. die `getRadius()`-Methode in einer Klasse `Circle` (siehe Abbildung 4.2). *Predicate*-Methoden liefern einen boolschen Wert resultierend aus einer Bedingung, für deren Auswertung direkt oder indirekt Felder des Objekts verwendet werden. Dies wäre z.B. die Methode `isUnitCircle()`, die anhand des Radius prüft, ob der Kreis die Vorraussetzung für einen Einheitskreis erfüllt. Eine *Property*-Methode ist eine Methode, die Informationen über das Objekt liefern, welche aus Feldern des Objekts abgeleitet werden. Praktisch alle `toString()`-Methoden oder in unserem Beispiel die Methode `getCircumference()` zur Berechnung des Umfangs sind Vertreter dieser Art.

Für die Zustandsänderung von Objekten sind *Mutator*-Methoden zuständig. Sie unterteilen sich in *Set*- und *Command*-Methoden und liefern führgewöhnlich kein Ergebnis zurück. *Set*-Methoden ändern direkt Eigenschaften des Objekts, d.h. nehmen schreibenden Zugriff auf die Datenfelder. Den Wert erhalten sie typischerweise als Parameter. Ein Beispiel wäre die Metho-

de `setRadius(double radius)`. *Command*-Methoden sind für komplexere Zustandsänderungen verantwortlich. Sie verändern indirekt durch den Aufruf weiterer Setter eines oder mehrere Felder, wobei eventuell komplexe Berechnungen vorausgehen. Die Methode `setCircumference (double circumference)` wäre eine *Command*-Methode, da sie für einen vorgegebenen Kreisumfang den Radius berechnen und dann durch Aufruf von `setRadius(..)` verändern muss.

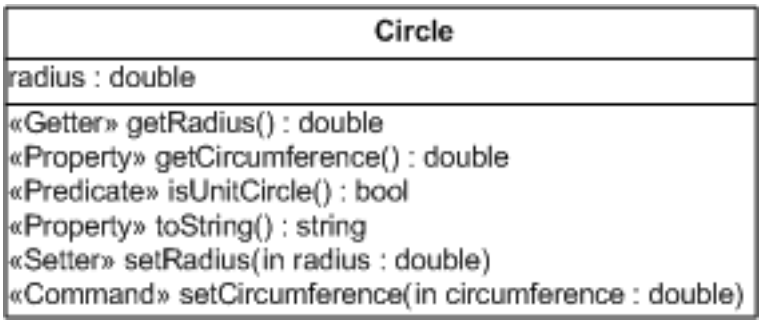

Abbildung 4.2: Ausschnitt einer Klasse zur Repräsentation eines Kreises

Ob eine Unterscheidung bis auf die letzte Ebene für die Zwecke der Aspektorientierung sinnvoll ist, sei dahingestellt. Die Ebene der *Accessor*- und *Mutator*-Methoden ist jedoch ohne Zweifel nützlich. Das Beispiel zeigt bereits deutlich, welche Fülle von Methoden, die nicht unbedingt im Namensraum von `get*` bzw. `set*` zu finden sind, sich dahinter verbergen kann.

Es sei an dieser Stelle daran erinnert, dass die Methoden anhand ihrer statischen Eigenschaften identifiziert werden, nicht nach ihrem tatsächlichen dynamischen Verhalten. D.h. sie können die Kriterien prinzipiell erfüllen, es ist aber nicht gesagt, dass sie es zur Laufzeit unbedingt tun. Z.B. ruft `setCircumference(..)` normalerweise die Methode `setRadius(..)` auf, bricht aber in einigen Fällen vielleicht schon vorher ab, weil eine Bedingung (wie Umfang ≥ 0) nicht erfüllt wurde oder eine Exception auftritt. Allein der mögliche Aufruf einer *Setter*-Methode macht `setCircumference(..)` daher schon zu einer *Command*-Methode. Das tatsächlich zur Laufzeit stattfindende Verhalten kann dagegen nur mittels komplexer Pointcuts wie *cflow* etc. ermittelt werden.

Bleiben wir bei unserem `set*`-Beispiel. Das eigentliche Ziel ist hier die Erfassung der (*Mutator*-)*Set*-Methoden. In der OT/J-Querysprache werden schreibende Feldzugriffe über den Joinpoint-Typ `setField` identifiziert [Sel06]. Mittels Query Introduction lässt sich so für Methoden eine `isSetter`-Prüfung implementieren.

```
1 query boolean Method.isSetter() {
2     this.expressions?[isSetField]
3 }
```

Listing 4.1: Der Existenzquantor ? soll Auskunft geben, ob in der Menge der Expressions wenigstens ein Schreibzugriff auf ein Feld stattfindet.

Analog zu `isSetter()` lassen sich `isCommand()` und dementsprechend `isMutator()` implementieren.

```
query boolean Method.isCommand() {
  this.calls?[isSetter() && isOfClass(this.declaringClass)]
}

query boolean Method.isOfClass(ReferenceType c) {
  this.declaringClass = c
}

query boolean Method.isMutator() {
  this.isSetter() || this.isCommand()
}
```

Listing 4.2: Die weiteren Mutator-Methoden

Bei `isCommand()` wird geprüft, ob in der Menge der von `m` aufgerufenen Methoden wenigstens eine vorhanden ist, die als Setter-Methode fungiert. Bei der Implementierung dieser Auswertung ist darauf zu achten, dass keine Endlosrekursion entstehen darf, da `this.calls` auch `this`, also den Aufrufer selbst enthalten kann. Die Methode `isMutator()` ist nun nichts weiter als eine boolsche Verknüpfung der unteren Hierarchie-Ebene. Es sei allerdings darauf hingewiesen, dass in dieser Version die Zugriffe auf Klassenebene geprüft werden, ein Zugriff auf der eigenen Klasse bedeutet aber nicht unbedingt einen Zugriff am eigenen Objekt. Letzteres ließe sich durch die Einführung eines Sprachfeatures wie `receiverIsThis()` prüfen, womit die Objektreferenz `this` zum Vergleich herangezogen wird.

```
query boolean Method.isCommand() {
  this.calls?[isSetter() && receiverIsThis()]
}
```

Listing 4.3: Command auf Objekt-Ebene

Neben der strukturellen Gliederung lassen sich auch klasseninterne und klassenübergreifende Methoden unterscheiden, d.h. die Kollaboration von Klassen. Entscheidend ist hier, ob die Zugriffe in der eigenen Klasse stattfinden oder auf Objekten anderer Klassen. Eine Kollaboration kann z.B. über die folgende `isCollaborator()`-Funktion identifiziert werden.

```
query boolean Method.isReclusive() {
  this.calls![isOfClass(this.declaringClass)]
}

query boolean Method.isCollaborator() {
  !(this.isReclusive())
}
```

Listing 4.4: Falls alle Aufrufe einer Methode innerhalb der eigenen Klasse stattfinden, dann arbeitet die Methode selbstbezogen, andernfalls ist sie ein Kollaborateur.

Die Methoden können somit gemäß ihrer strukturellen und kollaborativen Eigenschaften unterschieden werden. Weiterhin wäre denkbar, eine Verknüpfung dieser Informationen zu erstellen, also z.B. nicht nur die unabhängige Klassifizierung *Collaborative*, *Command* vorzunehmen, sondern ggf. auch *Collaborative-Command* in Verbindung zu sehen. In letzterem Fall werden die Eigenschaften Collaborative und Command in Kombination betrachtet, d.h. die *Collaborative-Command*-Eigenschaft soll auf einen Setter-Aufruf an einem fremden Objekt hinweisen.

Als weiteres Beispiel für eine mögliche Klassifizierung sind noch die kreationellen Methoden zu nennen, welche die Erzeugung bzw. Vernichtung von Objekten behandeln. Sie ermöglichen u.a. die Unterscheidung zwischen *New-*, *Copy-* und *Factory*-Konstruktoren.

Mit den hier vorgestellten Mitteln lassen sich natürlich nicht alle denkbaren semantischen Informationen extrahieren. Aber sie bieten eine einfache Lösung für häufig auftretende Anforderungen an Pointcuts. Ein großer Teil der in der Praxis auf Basis semantischer Information gesuchten Methoden lässt sich mit diesen Mitteln komplett oder wenigstens unterstützend beschreiben.

4.3 Beispiel: Persistenz von Datenobjekten

Ein klassisches Problem, das für den Einsatz aspektorientierter Techniken aufgeführt wird, ist die Persistenz-Verwaltung von Objekten, die als Repräsentation von Daten aus einer Datenbank dienen (das Beispiel wird hier nur ausschnittweise wiedergegeben und ist vollständig in [LRC+06] nachzulesen).

```
1  public class PersistentRoot {
2    protected boolean isDeleted = false;
3
4    public void delete(){
5      this.isDeleted = true;
6    }
7    public boolean isDeleted() {
8      return this.isDeleted;
9    }
10 }
```

Listing 4.5: Die Basisklasse für Datenobjekte

Wird ein Datensatz in der Datenbank gelöscht, so wird auch das geerbte Flag in den Objekten der `PersistentRoot`-Subklassen gesetzt. Eventuell existieren aber noch Referenzen auf ein gelöschtes Objekt, so dass es nicht sofort vom Garbagecollector entsorgt wird. Für den Fall, dass dennoch versucht wird, auf strukturrelevante Bereiche des Objekts zuzugreifen soll ein Pointcut dazu dienen, diese Aufrufe abzufangen.

```
1  pointcut trapAccessOnDeletedObjects( PersistentRoot obj ):
2    this( obj ) &&
3    (execution(public * PersistentRoot+.get*(..)) ||
4     execution(public * PersistentRoot+.set*(..)) ||
5     execution(public String PersistentRoot+.toString ())
6  );
```

Listing 4.6: AspectJ-Pointcut für alle Versuche auf die transiente Repräsentation eines persistenten Datenobjektes zuzugreifen.

In diesem Beispiel werden die typischen syntaktischen Pointcuts wie `get*` und `set*` verwendet, um zu erreichen, das jegliche strukturellen Zugriffe abgefangen werden. Mit den für die OT/J-Query-Sprache eingeführten Mitteln ließe sich der gesamte Pointcut jetzt einfach durch einen Aufruf von `isStructural()` abdecken.

```
1  query boolean Method.isStructural() {
2    this.isAccessor() || this.isMutator()
3  }
4
5  query Set<Method> trapAccessOnDeletedObjects(ReferenceType c) {
6    c.descendants.methods[isPublic() && isStructural()]
7  }
8
9  // Anwendung
10 someOp() <- replace query trapAccessOnDeletedObjects(PersistentRoot);
```

Listing 4.7: Umsetzung von trapAccessOnDeletedObjects in OT/J

5 Anwendung von Rollen, Teams und Queries

Lag in den vorangegangenen Kapitel der Fokus primär auf der Querysprache selbst, so wollen wir im Folgenden betrachten, wie die Queries sich in den Kontext von Rollen und Teams einbetten. Insbesondere die Verknüpfung von Queries und Rollenmethoden in callin-Bindungen stellt uns, wie wir sehen werden, vor einige Probleme, welche für eine erfolgreiche Integration der Querysprache in Object Teams zu lösen sind.

5.1 Software für eine Einzelhandelsfiliale

Im Folgenden ziehen wir als Beispiel das einfache Verwaltungssystem der Filiale eines Einzelhändlers heran. Folgende Situation: ein Einzelhändler hat die Filiale eines in Konkurs gegangenen Konkurrenten samt Einrichtung und Software übernommen.

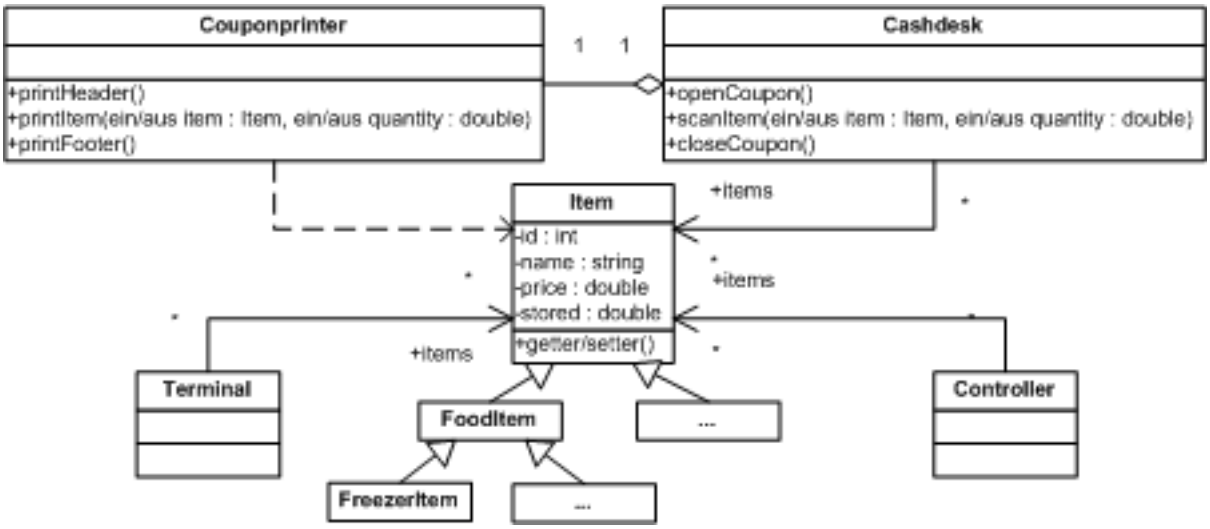

Abbildung 5.1: Eine einfache Verwaltungssoftware

Das vorhandene System leistet rudimentäre Dienste. Es existiert eine Datenbank, in der alle Waren (repräsentiert durch die Klasse `Item`) gespeichert sind. Die Daten können entweder direkt am Computerterminal (`Terminal`) oder mit mobilen Lesegeräten in Art eines Handhelds

(`Controller`) direkt an den Regalen bearbeitet werden. An den Kassen (`Cashdesk`) werden die Waren anhand des Barcodes identifiziert. An jede Kasse ist ein Bondrucker (`Couponprinter`) angeschlossen, der die registrierten Waren, Menge und Preis auf den Kassenbon druckt.

Der Ablauf zur Bearbeitung eines Kunden, der mit seinem Einkauf an die Kasse kommt, sieht dabei wie folgt aus: der Kassierer lässt einen neuen Bon erstellen (`openCoupon()`). Danach werden die Artikel einzeln registriert, jeder registrierte Artikel wird dabei direkt in einer Zeile auf den Bon geschrieben (`scanItem(Item item, double quantity)`). Nachdem alle Artikel bearbeitet sind wird die Summe vermerkt und der Bon abgeschlossen (`closeCoupon()`).

```
1  public class Cashdesk {
2          private Couponprinter printer;
3          private double sum;
4
5          public Cashdesk(Couponprinter printer) {
6                  this.printer = printer;
7          }
8
9          public void openCoupon() {
10                 sum = 0;
11                 printer.printHeader();
12         }
13
14         public void scanItem(Item item, double quantity) {
15                 sum += item.getPrice() * quantity;
16                 item.setStored(item.getStored() - quantity);
17                 printer.print(item, quantity);
18         }
19
20         public void closeCoupon() {
21                 printer.printFooter(sum);
22         }
23
24 }
25
26 public class Couponprinter {
27
28   public void printHeader() {
29           System.out.println(new java.util.Date());
30   }
31
32   public void print(Item item, double quantity) {
33     System.out.println(item.getName() + " "
34         + quantity + "* " + item.getPrice());
35   }
36
```

```
37    public void printFooter(double sum) {
38            System.out.println("Summe: " + sum);
39            System.out.println("Vielen Dank für Ihren Einkauf");
40    }
41 }
42
43 public class Item {
44          private double price;
45          private String name;
46          private double stored;
47          private int id;
48
49          ... //getter/setter-Implementation
50 }
```

Listing 5.1: Die Klassen zur Repräsentation der Kasse, des Bondruckers und der einzelnen Waren

5.2 Ein einfacher Aspekt

Nach der Übernahme soll die Filiale modernisiert werden, wovon auch die Software betroffen ist. Die existierende Software ist sehr einfach gehalten, tut aber ihren Dienst. Um die Kosten so gering wie möglich zu halten soll das vorhandene System weiter benutzt werden. Erweiterungen bzw. Anpassungen sollen am bestehenden System vorgenommen werden. Die Übernahme der Filiale erfolgt im laufenden Betrieb. Aus Sicherheitsgründen und um Umstellungsschwierigkeiten zu vermeiden sollen Veränderungen so erfolgen, dass nach Möglichkeit die bestehende Kernanwendung nicht angefasst wird.

Als erste Maßnahme wollen wir nach der Übernahme neue Kunden mit attraktiven Rabatt-Angeboten in den Laden locken. Zur großen Neueröffnung soll vorübergehend für eine Woche 20% Nachlass auf alle Artikel gewährt werden. Unsere Kassensoftware ist aber nicht in der Lage, einen Preisnachlass zu berücksichtigen. Wir müssten alle Preise direkt in der Datenbank senken, um diesen Nachlass umzusetzen, was aber aufwendig wäre, insbesondere, da wir später ja wieder zu den Normalpreisen zurückkehren wollen. Wir wollen daher die Flexibilität der Software in diesem Punkt verbessern.

Ein einfacher Aspekt löst das Problem. Mit dem Team `TDiscount` gestalten wir einen Aspekt, der jedem `Item` die Rolle `DiscountItem` zuweist. Bei jedem Abruf des Preises durch die Methode `getPrice()` wird nun statt dem ursprünglichen Preis ein Preis samt Abschlag um einen bestimmten Prozentsatz zurückgeliefert, den wir bei der Erstellung des Teams festlegen. Durch Aktivierung bzw. Deaktivierung des Teams können wir den Rabatt für unsere Kassensoftware praktisch auf Knopfdruck an- bzw. ausschalten.

```
1 public team class TDiscount {
```

```
2           private double discount;
3
4           public TDiscount(double discount) {
5                   this.discount = discount;
6           }
7
8           public class DiscountItem playedBy Item {
9
10                  callin double getDiscountPrice() {
11                          return (base.getDiscountPrice() * discount);
12                  }
13
14                  getDiscountPrice <- replace getPrice;
15
16          }
17  }
18
19  // Programmaufruf für Rabatt von 20%
20    ...
21    TDiscount td = new TDiscount(0.80);
22    td.activate();
23    ...
```

Listing 5.2: Teamklasse und Aufruf zur Umsetzung eines generellen Preisnachlasses. Jeder Aufruf von getPrice() wird durch getDiscountPrice() ersetzt.

5.3 Quantifizierung vs. Bindung

Bis zu diesem Punkt erreichen wir unser Ziel mit Standard-Mitteln. Als nächstes wollen wir die Kontrollstrukturen verbessern. Da wir mehrfach Fehler bei den Eintragungen der Waren in der Datenbank gefunden haben, die zu überflüssigen Bestellungen und falsch bemessenen Verkaufspreisen geführt haben, möchten wir in der Folge alle Änderungen protokollieren. Konkret heißt das, wir wollen alle Setter-Aufrufe an der Klasse `Item` in einer Log-Datei vermerken. An dieser Stelle ziehen wir Queries zur Quantifizierung über den Setter-Methoden heran (siehe Listing 5.3).

```
1   public team class TLogger {
2
3           public class LogItem playedBy ? {
4
5                   void writelog() {
6                           print(new java.util.Date());
7                   }
8
```

```
9                   writelog <- after query Item.allSetters;
10
11          }
12  }
13
14  query Set<Method> ReferenceType.allSetters {
15      this.methods[isSetter]
16  }
```

Listing 5.3: Teamklasse zur Umsetzung eines Loggers für alle schreibenden Aufrufe der Klasse Item

Problem: Durch die fehlende Bindung bei `playedBy ?` kann nicht mittels callout auf den Basistyp zugegriffen werden. Unser Log könnte keine konkreten Informationen aus der Klasse `Item` beziehen. Um dies zu umgehen müssten wir hier auf die Quantifizierung verzichten und jede `set`-Methode explizit binden, was allerdings zum Nachteil hätte, dass dabei alle zusätzlichen `set`-Methoden der Unterklassen von `Item` unter den Tisch fielen oder extra in eigenen Rollen ausprogrammiert werden müssten. Eine einfachere Lösung wäre die konkrete Festlegung eines Scopes durch explizite Angabe einer Basisklasse (siehe Listing 5.4) - eine Variante, die dem `this(BaseClass)` in AspectJ entspräche.

```
1   public team class TLogger {
2
3           public class LogItem playedBy Item {
4
5                   void writelog() {
6                           print(new java.util.Date() + "␣" + toString());
7                           // ... etc.
8                   }
9
10                  writelog <- after query base.getClass().allSetters;
11
12                  abstract String toString();
13                  toString -> toString;
14          }
15  }
```

Listing 5.4: Logger mit Zugriff auf Basisklasse

In dieser Variante muss allerdings durch die Syntax zugesichert werden, dass alleine Queries aus dem Scope der Basisklasse ausgeführt werden. Ein schwieriges Unterfangen, solange die Queries in ihrem Ergebnis grundsätzlich nicht eingeschränkt sind. Ein Aufruf von `someQuery` (siehe Listing 5.5) anstelle von `allSetters` würde zu einem Problem führen, denn das globale Set `allMethods`, was wir dort erhalten, enthält auch Methoden, die außerhalb des Scopes der Klasse `Item` liegen. Für die Query muss also gefordert werden, dass sie intern immer mit `this` arbeitet, wie etwa bei `safeQuery`. Die Angabe `playedBy Item` für die Rolle ist dabei für

die Festlegung dieses Scopes im Grunde schon hinreichend. Diese Information könnte für die Queries implizit verwendet werden, ohne dass sie nochmal explizit angegeben werden muss. Das Ergebnis müsste dazu automatisch für den jeweiligen Scope gefiltert werden.

```
1 query Set<Method> ReferenceType.someQuery {
2     allMethods
3 }
4
5 query Set<Method> ReferenceType.safeQuery {
6     this.someQuery[isOfClass(this.ClassType)]
7 }
```

Listing 5.5: someQuery würde zu einer unzulässigen Bindung führen. Ein Filter wie bei safeQuery, der eventuell auch automatisch vom System vorgenommen wird, vermeidet das Problem.

Entscheidend dabei ist, dass in der Bindung keine Aufrufe angenommen werden, welche den Scope der jeweiligen Rolle verletzen (wir könnten an dieser Stelle von ≪gutmütigen≫ Queries sprechen).

Eine andere Möglichkeit wäre, Queries, deren Ergebnisse den Scope verletzen, zu verbieten und dem Programmierer die Verantwortung für die korrekte Einschränkung zu überlassen. Dies führt aber im Grunde zu unnötiger Redundanz, da der Programmierer in der `playedBy`-Beziehung bereits den gewünschten Scope festgelegt hat. Es wäre ausreichend, wenn die Entwicklungsumgebung einen Hinweis liefert (z.B. Warnung bei Eclipse), sofern sie die vom Programmierer definierte Query auf den Scope bezogen einschränken muss.

5.4 Matching vs. Mapping

Bei der Selektion über einer Joinpoint-Menge spielen zwei wichtige Konzepte eine Rolle, die in der Anwendung sehr ähnlich aussehen, aber prinzipiell verschieden sind. Da ist zum einen das Zusichern eines bestimmten Typs bzw. einer Methoden-Signatur (Matching), zum anderen die Bereitstellung von Kontext-Informationen (Mapping). Für sowohl Matching als auch Mapping existieren in AspectJ die Schlüsselworte `this`, `target` und `args`. Zum Matching kann weiterhin auch eine konkrete Signatur angegeben werden (ggf. mit Wildcards).

```
1 pointcut match(): target(Item) && call(public void set*(..));
2
3 pointcut matchnmap(Item t): target(t) &&
4                             call(public void set*(..));
```

Listing 5.6: AspectJ-Pointcut einmal ohne und einmal mit Bindung

Das Zusichern bzw. Selektieren eines bestimmten Typs bei `this` und `target` findet wie vergleichsweise bei `instanceOf` zur Laufzeit statt. Die OT/J-Querysprache wird dagegen zur Compilezeit ausgewertet. Die beanspruchten Typen werden hier über `ReferenceType`-Parameter festgelegt. Listing 5.7 zeigt einen AspectJ-Pointcut mit Angabe einer Methoden-Signatur, Listing 5.8 zeigt dessen Umsetzung in der Querysprache.

```
1 pointcut itemDoubleSetter(): execution(public void Item.set*(double));
```

Listing 5.7: AspectJ-Pointcut für alle double-Setter der Klasse Item

```
1 someOp() <- after query myDoubleSetters(Item);
2
3 query Set<Method> myDoubleSetters(ReferenceType c) {
4    c.methods[isSetter() && isPublic() && returntype == void
5              && arguments[0].type == double && arguments.size == 1]
6 }
```

Listing 5.8: Umsetzung der Joinpoint-Menge in einer Query mit indirekter Angabe der Signatur

Die Schreibweise in 5.8 ist wenig komfortabel und hat den Nachteil, dass die Informationen nicht für eine eventuelle Parameter-Bindung bereit stehen. Vorraussetzung für das Mapping von Parametern ist ein erfolgreiches Matching. An dieser Stelle kann eine intuitivere Schreibweise angewandt werden (siehe Listing 5.9).

```
1 query Set<void MethodCall(double)> myDoubleSetters(ReferenceType c) {
2     c.methods[isSetter() && isPublic()]
3 }
```

Listing 5.9: Alternative Schreibweise zu 5.8 mit direkter Angabe der Signatur

Die Schreibweise in 5.9 enthält nun alle Signatur-Informationen zur Methode in der Query-Signatur. Das ist zum einen übersichtlicher und hat zum anderen den Vorteil, dass es hier eine zentrale Stelle für eine eventuelle Parameter-Bindung gibt. Ein Parameter-Mapping könnte dann wie bei den einfachen callin-Bindungen vorgenommen werden (siehe Listing 5.10).

```
1 someOp(double value) <- after query Set<void MethodCall(double arg0)>
2                         myDoubleSetters(Item) with {value <- arg0};
3
4 query Set<void MethodCall(double arg0)>
5     myDoubleSetters(ReferenceType c)
6 {
7   for (Method m: c.methods[isSetter() && isPublic()])
8     m {arg0 <- m.arguments[0]}
9 }
```

Listing 5.10: OT/J-Query mit Parameter-Mapping

Was an dieser Variante fehlt, ist eine explizite Bindung des Zielobjekts (`target` in AspectJ), oder ggf. auch mehrerer beteiligter Objekte bzw. Variablen. Sie ließe sich nach dem gleichen Prinzip einfügen, sofern die Querysprache eine Zuordnung auf der Objektebene zulässt.

```
1  someOp(double value, Item i) <-
2          after query Set<void MethodCall(double arg0)>
3            myDoubleSetters(Item item) with {value <- arg0, i <- item};
4
5  query Set<void MethodCall(double arg0)> myDoubleSetters(Object o)
6  {
7    ClassType c = o.getClass();
8    for (Method m: c.methods[isSetter() && isPublic()])
9      m {arg0 <- m.arguments[0], o <- target(m)}
10 }
```

Listing 5.11: OT/J-Query mit Parameter- und Target-Mapping

In Listing 5.11 wollen wir alle Setter-Methoden von `Item` finden. Gleichzeitig mit Parameter `arg0` soll nun auch die Referenz des jeweiligen `Item`-Objekts zurückgegeben werden. Bei den Query-Parametern wird zusätzlich zu dem Klassentyp `Item` noch die Objektreferenz `item` angegeben, damit diese an den Parameter `Item i` in `someOp` weitergereicht werden kann. Voraussetzung dafür ist aber, dass die Query uns in der Situation das konkrete Objekt liefern kann. Zu diesem Zwecke wurde `target(m)` aufgerufen, welches uns die Objektreferenz zu der Methode `m` liefern soll. Die Syntax für diesen Befehl dient hier lediglich zur Veranschaulichung des Problems. Es bleibt zu überlegen, ob und in welcher Weise Objektreferenzen in der Querysprache gebunden werden können.

5.5 Filter- vs. Ereignis-basierte Pointcuts

Zurück zu unserem Beispiel: es existieren zwei Schnittstellen, an denen unsere Filialmitarbeiter auf die Datenbank zugreifen können. Zum einen direkt am Computerterminal, zum anderen mit den mobilen Lesegeräten. Bei der Auswertung unserer Log-Datei haben wir festgestellt, dass die meisten fehlerhaften Eingaben geschehen, wenn die Mitarbeiter Daten über die Hand-Lesegeräte eingeben. Nach Rücksprache mit den Mitarbeitern wird deutlich, dass bei der Handhabung der Lesegeräte nicht immer klar ist, an welchen Stellen tatsächlich Daten verändert werden. Aus diesem Grund wollen wir die Mitarbeiter an den entsprechenden Stellen mit einem Hinweis darauf aufmerksam machen. Vor dem Aufruf von Methoden, in welchen Werte verändert werden können, soll nun ein Warnhinweis gezeigt werden.

```
1  public team class TWarning {
2
3          protected class WarningItem playedBy ? {
4
5                  void showWarning(Item changeItem) {
6                          // ...
7                  }
8
9                  showWarning(Item changeItem) <- before
10                   query controllerItemChange(Item item)
11                   with {changeItem <- item};
12         }
13 }
14
15 query Set<Method> controllerItemChange(ClassType t) {
16   Controller.methods[calls?[isOfClass(t) && isSetter()]]
17 }
```

Listing 5.12: Filter-basierte Warnung vor allen Methoden der Klasse Controller, die Setter-Methoden der Klasse Item aufrufen.

Nach Einführung der Maßnahme müssen wir leider feststellen, dass es trotz der Hinweise zu fehlerhaften Eingaben kommt. Wir wollen daher die Möglichkeit Datenänderungen am Lesegerät vorzunehmen vorübergehend blockieren, bis wir eine Schulung der Mitarbeiter durchgeführt haben. Zur Umsetzung wollen wir jeden schreibenden Aufruf verhindern, der von einem Lesegerät ausgeht. Wir machen diese Aufrufe einfach wirkungslos, in dem wir sie durch leere Methoden überschreiben. Neben der statischen Filterung müssen wir hier auch dynamische Aspekte miteinbeziehen. Auf diese Weise können alle im Kontrollfluss auftretenden Aufrufe abgefangen werden.

```
1  public aspect ABlockController {
2
3          pointcut setter(): call(* Item.set*(..));
4
5          pointcut controller(): call(* Controller.*(..));
6
7          void around(): setter() && cflow(controller()){
8                  // do nothing
9          }
10 }
```

Listing 5.13: Ereignis-basierte Umsetzung von Block in AspectJ

```
1  public team class TBlockController {
2
```

```
3           protected class BlockItem playedBy ? {
4
5                   void donothing() {
6                           // no base call
7                   }
8
9                   donothing <- replace CFlowControllerSet.fire;
10          }
11  }
12
13  pointcut class CFlowControllerSet playedBy ? {
14    Boolean enabled = false;
15    callin void controllerOp() {
16      enabled = true;
17      base.controllerOp();
18      enabled = false;
19    }
20    void fire() {};
21    controllerOp <- replace query allMethods(Controller) when (!enabled);
22    fire <- replace query allSetters(Item) when (enabled);
23  }
```

Listing 5.14: Ereignis-basierte Umsetzung von Block in OT/J

Die Schwierigkeit bei diesem Pointcut ist die Erfassung des Kontrollflusses, für welche wir in Object Teams das Konzept der Pointcut-Klasse aufgreifen. Die Pointcut-Klasse ist wie eine Rollen-Klasse aufgebaut, allerdings ohne direkt an ein Team gebunden zu sein. Zur Umsetzung dekorieren wir die Klasse `Controller` mit einem Wrapper, der ein Flag bereit hält, welches immer dann gesetzt ist, wenn eine Methode aus `Controller` sich in der Ausführung befindet (das Flag darf dabei nur beim jeweils äußersten Aufruf im Kontrollfluss gesetzt werden, um zu verhindern, dass bei der Beendigung von Subroutinen der `enabled`-Status ggf. schon vorzeitig wieder auf `false` gesetzt wird). Die Methode `fire`, welche als Trigger des Pointcuts dient, wird nun an die Ausführung von Setter-Methoden der Klasse `Item` gebunden, allerdings nur in eben den Fällen, in denen das Flag anzeigt, dass wir uns im Kontrollfluss von `Controller` befinden. Somit erreichen wir genau jene Setter-Methoden, die im Rahmen der Controller-Aktivität ausgeführt werden.

Die Pointcut-Klasse ist hier nur konzeptionell dargestellt. Zwecks kürzerer Ausdrucksweise und besserer Lesbarkeit soll diese Funktionalität in näherer Zukunft ein eigenes Konstrukt erhalten.

6 Query Modellierung

Die Anforderung, Abfragen zu Software-Artefakten zu formulieren und auszuwerten, hat in der heutigen Software-Entwicklung an Bedeutung gewonnen. Es existieren mehrere textuelle Sprachen, wie z.B. die OCL, um für Programme Aussagen zu formulieren und zu validieren. Eine Anforderung neben anderen ist dabei, Elemente in einem Modell anhand bestimmter Kriterien zu spezifizieren. Insbesondere in der AOSD ist diese Spezifikation eine wichtige Vorraussetzung, um in einem Programm Joinpoint-Mengen zu identifizieren, an deren Stellen Aspekt-Code eingewoben werden kann. Aspektorientierte Programmiersprachen, wie z.B. AspectJ, bedienen sich dabei im Allgemeinen ebenfalls einer textbasierten Notation, um entsprechende Joinpoint-Mengen (≪pointcuts≫) zu definieren. Textuelle Beschreibungen haben den Vorteil, dass sie sehr mächtig und flexibel in Bezug auf die Darstellung eines Systems sind. Dem steht als Nachteil gegenüber, dass ein solcher Text sehr abstrakt und insbesondere bei komplexeren Sachverhalten nur schwer zugänglich ist. Zum Verständnis muss man mit den sprachspezifischen Operatoren und Schlüsselwörtern vertraut sein. Darüber hinaus muss man in der Lage sein, den Aufbau einer Struktur, sowie die darin vorkommenden Beziehungen und Abhängigkeiten korrekt aus dem Text heraus zu interpretieren. Aufgrund dieser Nachteile bedienen sich viele Modellierungssprachen (wie z.B. die UML) grafischer Elemente, da mit ihnen die Darstellung von Strukturen und Zusammenhängen häufig sehr viel übersichtlicher aufbereitet werden kann.

Es stellt sich nun die Frage, welche Formen einer grafische Notation für die Query- bzw. Pointcut-Modellierung geeignet sind? Es existieren beispielsweise grafische Repräsentationen für die Querysprache XQuery inklusive XPath (u.a. Visual XQuery und Visual XPath). Diese dienen aber primär der grafischen Darstellung der Baumstruktur der XML-Dokumente und beziehen sich dabei nur bedingt auf dessen Inhalt. Die Domäne der Programm-Modellierung, welche konkret in UML-Diagrammen abgebildet wird, kommt hier also nicht direkt zum Vorschein. Auch für die OCL existiert mit Visual OCL [Win05] eine grafische Repräsentierung. Der Fokus liegt allerdings auf der Darstellung von Constraints. Die Mächtigkeit dieser Darstellung wird im Rahmen der Query-Modellierung nur teilweise benötigt, sie ist für diesen Kontext vergleichsweise textlastig. Eine grafische Notation zur Spezifikation von Queries, die speziell auf das Gebiet der Pointcut-Modellierung zugeschnitten ist, wurde dagegen mit den ≪Join Point Designation Diagrams≫ (kurz JPDDs) entwickelt [SHU04a, SHU04b, SHU07]. Ihre Darstellung basiert direkt auf UML-Diagrammen. JPDDs dienen der Selektion von Programmelementen, wie z.B. Klassen, Methoden oder Nachrichten, die für die Beschreibung von Joinpoints eine Rolle spielen. Als Selektionskriterien werden Bezeichner (auch namensbasierte Muster), Struktur- und Kontextinformationen verwendet. Ein großer Vorteil der JPDDs ist, dass sie dank der Adaptierung bekannter UML-Diagramme sowohl gut in die Domäne der Programm-Modellierung passen, als auch intuitiv leicht verständlich sind.

Das Gebiet der aspektorientierten Modellierung geht natürlich über die reine Pointcut-Modellierung hinaus. Die Modellierung aspektorientierter Programme mittels objektorientierter Techniken stößt auf ähnliche Probleme, wie sie bei der Umsetzung von Aspekten in der objektorientierten Programmierung hervortreten. In der Vergangenheit wurden daher bereits mehrere Ansätze zur Modellierung von aspektorientierten Programmen entwickelt [CRS+05]. Im Umfeld der UML sind exemplarisch Theme/UML oder die ≪Aspect-Oriented Executable Modeling≫-Erweiterung zu nennen. Betrachten wir diese Technologien kurz, besonders im Hinblick auf die Pointcut-Modellierung

Theme/UML [CB05] ist eine aspektorientierte Erweiterung von UML (bisher UML 1.3, eine Erweiterung als UML 2.0-Profil ist aber geplant). Der Fokus wird dabei primär auf das Design der Anforderungen gelegt. Bei dem ≪Theme≫-Ansatz wird zunächst eine Dekomposition der Anforderungen vorgenommen, sodass jede Anforderung, also auch ein Crosscutting Concern, individuell in einem eigenen Modell entworfen wird. Die einzelnen Modelle werden dann nach entsprechenden Mustern miteinander komponiert. Überlappende Bereiche der einfachen Anforderungen werden aufeinander abgebildet (≪match≫), abhängige Bereiche der Crosscutting Concerns werden mithilfe von Template-Parametern dargestellt und aneinander gebunden (≪bind≫). Da Matching und Binding automatisiert von einem Werkzeug vorgenommen werden sollen, kann der Entwickler sich unabhängig von Design-Restriktionen (insbesondere die Einbindung der Crosscutting Concerns betreffend) auf die Modellierung der Anforderungen konzentrieren. Zur Verhaltensmodellierung der Crosscuting Concerns werden in Theme/UML Sequenzdiagramme verwendet. Aufgrund der Stärken der JPDDs in diesem Bereich wird aber bereits an einer Integration von Theme/UML mit JPDDs gearbeitet [JC06].

Aspect-Oriented Executable Modeling (AOEM) [FS07b, FS07a] ist ein Vertreter des eher technischen Ansatzes, die UML mittels eines umfassenden Profils für die Spezifikation von aspektorientierten Modellen zu erweitern. Der Fokus liegt hier auf der detaillierten Beschreibung eines ausführbaren Modells. Crosscutting Concerns werden mittels des AOEM-Profils als Aspekt (inklusive des kompletten Verhaltens) in Form einer Klasse modelliert. Die Bindung der Aspekte an die Basisanwendung wird in einem Pointcut-Modell in Form eines erweiterten Sequenz-Diagramms spezifiziert. Das Basis-Modell der Anwendung wird zusammen mit dem Aspekt-Modell und dem Pointcut-Modell von einem Model-Weaver zusammengeführt. Das Ergebnis ist ein ausführbares Modell (Executable UML). Auch in AOEM basiert das Modell zur Pointcut-Modellierung bislang auf Sequenzdiagrammen, teilt aber die wesentlichen Eigenschaften von JPDDs. Als einer der nächsten Entwicklungsschritte ist daher eine Integration mit JPDDs geplant [FS07b, FS07a].

Die breite Akzeptanz und Anwendung von JPDDs in diesem Bereich spricht deutlich für deren Vorzüge. Die Darstellung mittels adaptierter Struktur- und Verhaltensdiagramme aus der UML hat sich in punkto Integration, Verständlichkeit und Lesbarkeit gegenüber alternativen Ansätzen durchgesetzt. Die grafische Notation für die OT/J-Querysprache basiert daher ebenfalls auf den JPDDs und orientiert sich sehr stark an den darin verwendeten Konstrukten, weshalb im Folgenden ausführlicher auf die Darstellung von JPDDs eingegangen wird.

6.1 Join Point Designation Diagrams

JPDDs bauen auf den Diagrammen der UML auf. Die Verwendung bekannter Elemente aus der UML soll die Query-Darstellung möglichst leicht verständlich machen. Ein JPDD zur Selektion relevanter Elemente besteht dabei aus einem strukturellen Teil und einem verhaltensorientierten Teil, für deren Darstellung leicht abgewandelte Formen des UML-Klassendiagramms, des UML-Sequenzdiagramms bzw. des UML-Zustandsdiagramms herangezogen werden. Im Unterschied zu diesen Diagrammen dient eine JPDD aber nicht dazu, die Struktur oder das Verhalten von Programmfragmenten darzustellen, stattdessen wird in einem JPDD ein Muster mit diversen Selektionskriterien definiert, anhand dessen Elemente im Programm, die diesem Muster entsprechen, ausgewählt werden.

6.1.1 Struktur

Klassen An erster Stelle betrachten wir eine Klassendarstellung, anhand derer auch das grundlegende Prinzip der Selektion veranschaulicht ist (Abbildung 6.1).

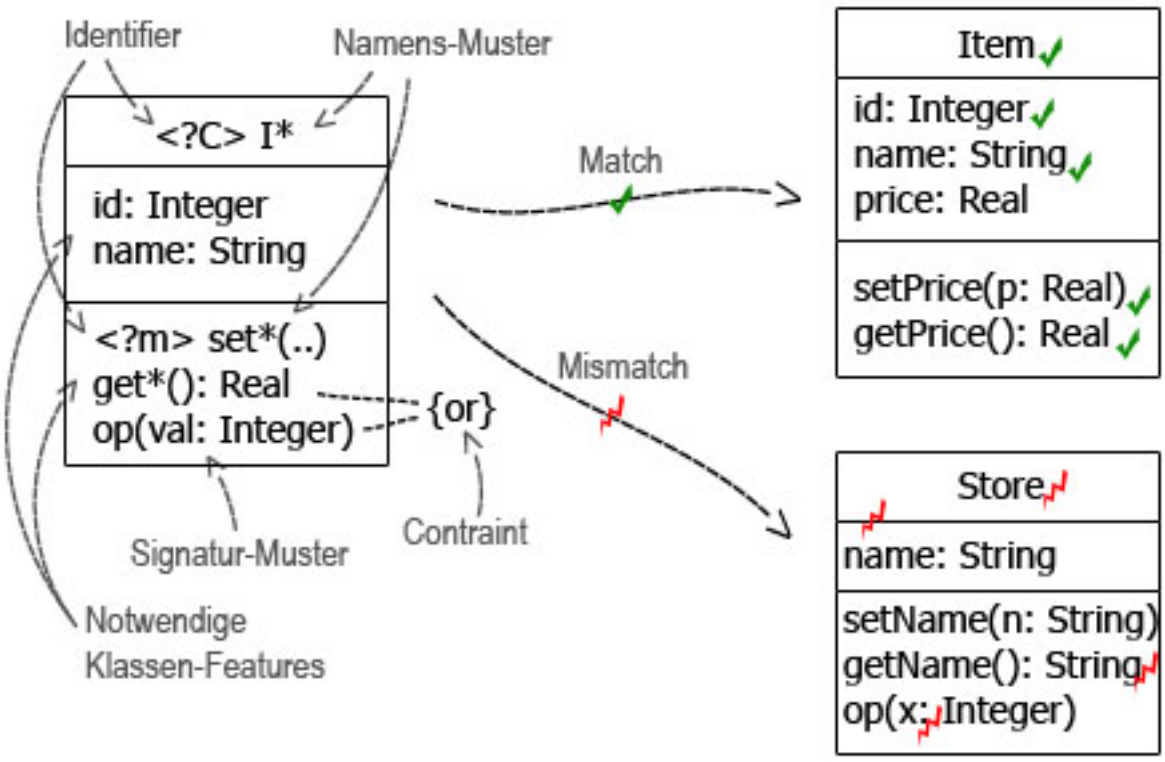

Abbildung 6.1: Ein JPDD (links) im positiven und negativen Abgleich mit Klassen (rechts)

Eine Klasse definiert verschiedene Auswahlkriterien für ihre Struktur. Im Beispiel wird gefordert, dass die Klasse, deren Name mit «I» beginnen muss, die Attribute `id` vom Typ `Integer`

und `name` vom Typ `String` besitzt. Im Methoden-Teil wird wenigstens eine Setter-Methode und eine Getter-Methode oder alternativ eine Methode `op` verlangt. Für die Bezeichner können Namensmuster angegeben werden, wobei Wildcards wie etwa ein ≪*≫ (steht für einen beliebigen Text, ggf. auch leer) verwendet werden können. Die Klasse steht als Selektionsmuster gegenüber allen existierenden Klassen im System, sie werden mit diesem Muster abgeglichen. Nur wenn eine Klasse alle Auswahlkriterien erfüllt ist der Vergleich positiv. Implizit repräsentieren alle Elemente des Diagramms so jeweils eine Menge passender Gegenstücke (≪Matches≫) aus dem konkreten Programm. Damit eine solche Menge an anderer Stelle verwendet werden kann bzw. als Ergebnis einer Query zurückgeliefert werden kann, kann ein Element durch voranstellen eines Identifiers auch explizit referenzierbar gemacht werden. Ein Identifier steht in spitzen Klammern zusammen mit einem Fragezeichen und einem Bezeichner. Im Beispiel steht `<?C>` als Identifier für die Menge der passenden Klassen. In einer vollständigen JPDD wird der Identifier in einer Box rechts unterhalb des Diagramms zur Kenntlichmachung als Rückgabe eingefügt.

Attribute und Methoden Analog zu den Klassen können auch Felder und Operationen definiert werden. Neben einem Namen kann bei ihnen noch eine Signatur angegeben werden. Bei Attributen umfasst die Signatur den Typ, bei Methoden darüber hinaus noch eine Liste der Parameter. Bei Angabe der Signatur kann wiederum auf Wildcards zurückgegriffen werden. Für die Parameterliste einer Operation gibt es zusätzlich die Wildcard ≪..≫, die für eine unbestimmte Anzahl beliebiger Parameter steht.

Beziehungen Neben den Klassen selbst können auch Beziehungen der Klassen untereinander definiert werden (Abbildung 6.2). Das gilt für Assoziationen und Generalisierungen. Eine Besonderheit sind hier die indirekten Beziehungen, die mit einer von einem Doppelstrich unterbrochenen Linie gekennzeichnet sind. Eine indirekte Beziehung in einem Diagramm drückt aus, dass zwischen den beteiligten Klassen nicht unbedingt eine direkte Verbindung bestehen muss. Es muss allerdings grundsätzlich ein navigierbarer Pfad im Diagramm existieren, durch den die Klassen verbunden sind. Der Pfad darf dabei über beliebig viele Zwischenstationen führen. Es besteht allerdings auch die Möglichkeit die Anzahl bzw. maximale Anzahl von Zwischenstationen konkret anzugeben.

Abbildung 6.2: Direkte und indirekte Assoziation (links) bzw. Generalisierung (rechts)

Multiplizitäten Analog zur UML können in JPDDs zu Attributen und Assoziationen auch Multiplizitäten angegeben werden. Diese werden als konkrete Werte oder als Grenzbereiche angegeben. Eine Ausrufungszeichen markiert bei Grenzbereichen einen exakt zu erfüllenden Wert, ansonsten handelt es sich um einen Minimal- bzw. Maximalwert.

Kombination Prinzipiell gilt für ein JPDD, dass alle Elemente des Diagramms gemeinsam ein zusammenhängendes Muster bilden. Alle Selektionskriterien müssen vollständig erfüllt sein. Eine nur partielle Übereinstimmung führt nicht zu einer positiven Zuordnung. Logisch gesehen handelt es also standardmäßig um eine `AND`-Verknüpfung der einzelnen Selektionskriterien. An manchen Stellen wird es aber sinnvoll sein, dass wir Selektionskriterien in anderer Weise miteinander verknüpfen. Zu diesem Zweck können Constraints angegeben werden. Damit lassen sich auch logische Verknüpfungen wie `OR`, `XOR` und `NOT` realisieren. Bei der Verwendung von `NOT` sollte man dabei sehr genau auf die Konsequenzen achten. Eine Verneinung soll explizit formulieren, dass ein bestimmtes Element nicht existiert. Die Vergabe eines Identifiers für ein solches Element macht keinen Sinn. Eine zu allgemein gehaltene Verneinung kann dazu führen, dass man eine Menge wie ≪Alle Elemente außer den Elementen mit Eigenschaft x≫ erhält. Eine derart unspezifische Menge ist insbesondere für die Aspektbindung meistens nicht praktikabel.

6.1.2 Verhalten

Neben dem strukturellen Diagrammteil, der die statischen Eigenschaften eines Programms betrachtet, gibt es noch den verhaltensorientierten Diagrammteil zur Spezifikation der dynamischen Abläufe eines Programms. Der Aufbau des verhaltensorientierten Teils basiert auf dem Sequenzdiagramm (Message Invocation Model) oder dem Zustandsdiagramm (State Model).

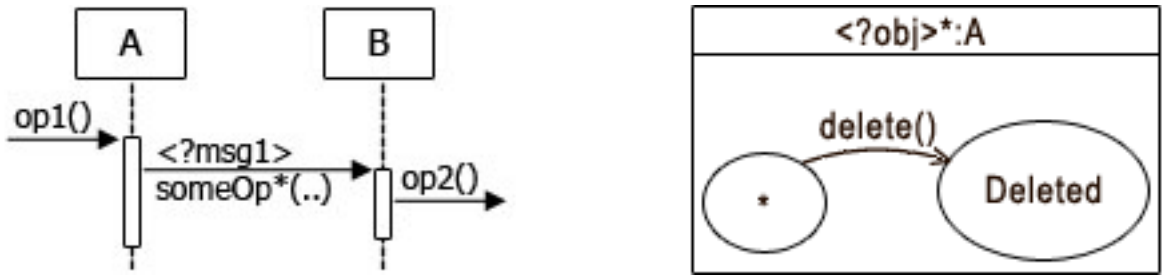

Abbildung 6.3: Beispiele für das Message Invocation Model (links) und das State Model (rechts)

Die Selektion von Elementen funktioniert nach dem selben Prinzip wie im Struktur-Teil. Als Kriterien des Message Invocation Modells dienen Interaktionen (und deren Reihefolge) zwischen Elementen. Der Kontrollfluß ist anhand von Lebenslinien, Ausführungspezifikationen und Nachrichten dargestellt. Die Nachrichten werden identifiziert anhand der Aktionen, mit denen sie verknüpft sind. Die Anordnung bestimmt die Sequenz, in der die Nachrichten vorkommen müssen und zeigt, welche Nachricht in welchem Kontrollfluß ausgelöst wird. Auch

hier ist es wieder möglich durch eine doppelt durchgestrichene Verbindung einen indirekten Nachrichtenfluß darzustellen. Im State Model wird in ähnlicher Weise verfahren. Anstatt der zeitlichen Abläufe stehen hier aber Zustände und Zustandsübergange im Fokus. Mit der verhaltensorientierten Darstellung können in JPDDs beispielsweise AspectJ-Pointcuts dargestellt werden, die Gebrauch von den Anweisungen `cflow`, `call` oder `execution` machen.

6.2 Object Teams Query Diagrams

In Anlehnung an die Benennung der JPDDs sollen die in Object Teams verwendeten Selektions-Diagramme nachfolgend als «Object Teams Query Diagrams» (OTQDs) bezeichnet werden. Wie auch bei anderen Programmiersprachen und ihren Modellierungswerkzeugen ist es kaum praktikabel, die gesamte Vielseitigkeit und Mächtigkeit der OT/J-Querysprache auf eine grafische Darstellung zu übertragen. Nicht alle Möglichkeiten und Details einer textuellen Beschreibung lassen sich sinnvoll grafisch abbilden. Bei sehr speziellen oder komplexen Queries wird man an einen Punkt kommen, wo die sich die grafischen Mittel erschöpfen und es günstiger ist, die Aussage mithilfe der textuellen Sprachkonstrukte zu formulieren. Es ist denkbar, diese Teile als Textbausteine gesondert in eine grafische Darstellung zu integrieren (im einfachsten Fall in Form von Kommentaren). Ziel der grafischen Notation ist es dennoch, möglichst viele Bereiche der Querysprache sinnvoll zu visualisieren und so dem Anwender ein einfaches und übersichtliches Werkzeug zur Query-Definition an die Hand zu geben. Entscheidend dafür ist, dass das Modell zur grafischen Notation einen sinnvollen Abstraktionsgrad erreicht, bei dem komplexe Informationen eine vergleichsweise einfache Repräsentierung finden.

6.2.1 Unterschiede zwischen OTQDs und JPDDs

Die JPDDs bieten eine sehr eingängliche und übersichtliche Art der Darstellung zur Visualisierung von Queries. Die grundlegenden Elemente der JPDDs sollen daher auch für OTQDs übernommen werden. Die konkreten Gegebenheiten von OT/J und seiner Querysprache lassen es aber nicht zu, JPDDs eins zu eins für die Query-Darstellung zu übernehmen. Ein grundlegender Unterschied im Anwendungsfeld von JPDDs gegenüber OTQDs betrifft die Einbeziehung von Kontextinformationen. JPDDs sind für eine Auswertung zur Laufzeit eines Programms vorgesehen und können daher auch auf dynamische Informationen und das Ablaufverhalten Bezug nehmen. Queries in OT/J werden hingegen zur Compilezeit ausgewertet, in diesem Kontext spielen allein statisch auswertbare Informationen eine Rolle. Aus diesem Grund ist für die Definition von OTQDs nur der strukturelle Anteil von Bedeutung. Des Weiteren sind Queries für den Einsatz in callin-Bindungen vorgesehen und dementsprechend konkret mit der Sprache OT/J verknüpft. Auch an einigen anderen Stellen, die noch gezeigt werden, ist es notwendig und/oder sinnvoll die JPDD-Darstellung für unsere Zwecke anzupassen.

Nachdem im vorigen Abschnitt bereits JPDDs und ihre wichtigsten Elemente erläutert wurden sollen in diesem Abschnitt die OTQDs vorgestellt werden. Da die beiden Diagramm-Formen

viele Gemeinsamkeiten haben wird im Folgenden nur noch auf die Teile eingegangen, in denen sich OTQDs von JPDDs unterscheiden.

Struktur Die generelle Darstellung der Struktur erfährt zunächst nur eine geringfügige Anpassung. Da wir uns im konkreten Umfeld der OT/J- bzw. Java-Programmiersprache bewegen, soll die Darstellung gleich die entsprechende Syntax widerspiegeln. Attribute und Operationen von Klassen werden also direkt wie in der Java-Deklaration angezeigt. Insbesondere bei Angabe von Metainformationen wie Zugriffs- oder Polymorphiemodifizierern verbessert dies die Übersichtlichkeit. Assoziationen werden in Form von unidirektionalen Referenzbeziehungen verwendet, entsprechen also den Attributen.

Des Weiteren werden in OTQDs auch Pakete explizit in die Struktur miteingebunden. Pakete, die auch als Container von Klassen dienen, können ebenfalls als Selektionskriterium herangezogen werden.

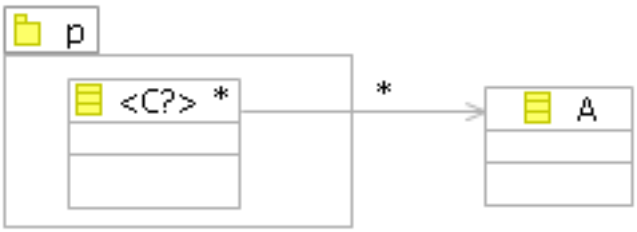

Abbildung 6.4: Query: «finde alle Klassen aus dem Paket p, welche die Klasse A referenzieren»

Zur Wertemenge bei der Selektion sei definiert, dass zwei verschiedene Elemente eines Query-Diagramms, trotz ggf. zutreffender Muster, niemals zur gleichen Zeit ein und dasselbe Objekt «matchen» können. Wir gehen davon aus, dass, wenn der Anwender zwei Diagramm-Elemente definiert, er auch wünscht, dass diese unterschiedlich gematcht werden, selbst wenn sich die Mengen der passender Repräsentations-Elemente überschneiden. Es soll sicher gestellt werden, dass ein Selektionsmuster eins zu eins auf konkrete Elemente abgebildet wird, ohne dass es dabei zu einer Mehrfachbelegung kommt. Im Ergebnis führt dies lediglich dazu, das die Menge positiver Matches kleiner wird. Das Diagramm ist so wesentlich intuitiver verständlich, gegenüber dem Fall, dass mehrere Diagramm-Elemente sowohl unterschiedliche als auch gleiche Repräsentationen finden.

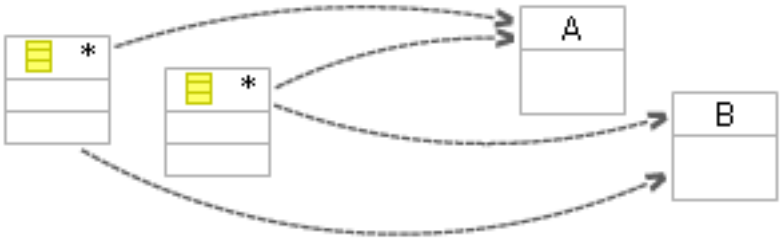

Abbildung 6.5: Mehrfachbelegung in der Ergebnismenge.

Die *-Selektoren (siehe Abbildung 6.5) finden beide sowohl in der Klasse A als auch B einen positiven Match. Die uneingeschränkte Ergebnismenge enthielte also folgende Tupel: {(A,A), (A,B), (B,A), (B,B)}. Das gleichzeitige Matchen eines Elements möchten wir aber per default ausnehmen, denn Elemente, die sich in der Diagrammstruktur unterscheiden, sollen sich auch in der repräsentierenden Abbildung unterscheiden. In der Ergebnismenge bleiben also nur die Tupel: {(A,B), (B,A)}.

Multiplizititäten Im Unterschied zu den JPDDs wird bei den OT/J-Queries auf eine Angabe der Multiplizitäten bei Beziehungen verzichtet. Dies hat sowohl technische als auch sachliche Gründe. Grundsätzlich ist der Sinn von Multiplizitäts-Angaben in einem Query-Diagramm ein wenig anders zu verstehen als in einem Klassendiagramm. Beispielsweise hat die Angabe 0..* an einer Referenzbeziehung als Selektionskriterium keinerlei Aussagekraft. Egal, ob eine Verbindung existiert oder nicht, dass Kriterium ist in jedem Fall erfüllt, damit ist die Assoziation überflüssig und kann komplett entfallen. Die Angabe einer konkreten Multiplizität wie z.B. 1..99 wirft dagegen ein Problem auf. Die Queries werden bereits zur Compilezeit ausgewertet, d.h. wir können nicht auf dynamische Laufzeit-Informationen zurückgreifen. Die Verwaltung einer solchen Menge von Referenzen wird aber meist in Datenstrukturen, wie z.B. einem Array, realisiert, die dynamisch anpassbar sind. Die notwendige Information zur Überprüfung eines Multiplizitäts-Kriteriums steht uns zur Compilezeit also nicht mit Sicherheit zur Verfügung. Die vollständige und korrekte Auswertung der Query könnte an dieser Stelle nicht gewährleistet werden. Wir beschränken uns daher bei einer Verbindung auf die Aussage, ob die Verbindung existiert oder nicht. Gleiches gilt für indirekte Referenzbeziehungen oder indirekte Generalisierungen. In den JPDDs kann hier ebenfalls eine Multiplizität zur Bestimmung der Ebenentiefe, über welche sich die Verbindung erstreckt, angegeben werden. Bei den bisher untersuchten Beispielen für praktische Anwendungen hat sich jedoch gezeigt, dass eine Unterscheidung zwischen einer direkten und einer indirekten Verbindung im Allgemeinen ausreichend ist. Im Sinne einer überschaubaren Abstraktion wurde daher auf die Detailangabe der Multipliztität verzichtet. Eine indirekte Verbindung in OT/J-Queries sagt nur aus, dass überhaupt ein navigierbarer Pfad existiert, ohne Einschränkung, über wie viele Ebenen dieser geht. Diese Art der Abbildung entspricht auch den zugrunde liegenden OT/J-Query-Sprachkonstrukten. Es sind sicherlich spezielle Beispiele konstruierbar, in denen eine solche Aussage nicht mehr ausreichend ist. Eine Verbindung mit konkret vorgegebener Tiefe ist in diesem Fall nach wie vor darstellbar, allerdings muss sie mithilfe mehrerer, direkter Verbindungen modelliert werden.

Methoden-Stereotypen Wie in Kapitel 3 erläutert können Queries auch als Erweiterung von Metaklassen formuliert werden (≪Query Introduction≫). Im Bereich der Multihierarchien von Joinpoints wurde eine Klassifizierung für Methoden vorgeschlagen, die mittels der Typerweiterungen realisiert werden können. Es ist wünschenswert, diese Möglichkeit in der grafischen Darstellung zu berücksichtigen. In JPDDs können dazu Meta-Attribute angegeben werden. In OTQDs soll das Prinzip der Stereotypen Verwendung finden. Für eine Methode kann eine beliebige Anzahl von Stereotypen angegeben werden. Die Stereotypen müssen im Namen einer definierten Typerweiterung entsprechen, die somit als zusätzliches Selektionskriterium bei der Auswahl der Methoden herangezogen wird. Die Angabe eines Stereotyps, wie z.B. ≪getter≫ (siehe Abbildung 6.8), wird dann in einem Constraint der Form `method[isGetter]` ausgedrückt;

`isGetter` muss entsprechend als Erweiterung definiert sein.

Dependency Ein weiterer Unterschied zu den JPDDs ist die Einführung von Dependencies. In JPDDs werden, wie bereits gesagt, sowohl statische als auch dynamische Kriterien berücksichtigt. Die OT/J-Queries dagegen befassen sich nur mit der statischen Struktur. Dennoch sollen einige wichtige Zusammenhänge bei der Kollaboration von Klassen und Methoden darstellbar sein. In der Aspektorientierung ist es häufig ein wichtiges Auswahlkriterium zu wissen, ob Elemente zusammenarbeiten, d.h. beispielsweise eine Methode eine andere Methode aufruft. Wir wollen daher für Klassen und Methoden ausdrücken können, ob sie in einer derartigen Nutzbeziehung zueinander stehen. Ob ein solcher Aufruf zur Laufzeit tatsächlich ausgeführt wird lässt sich an dieser Stelle nicht sagen, der Aufruf muss aber zumindest möglich sein. In der Querysprache wird zur Umsetzung dieser Anforderung Gebrauch von den UML-Dependencies gemacht. Eine Abhängigkeitsbeziehung sagt allgemein aus, dass Änderungen an der Spezifikation eines Elements sich auf die abhängigen Elemente auswirken, aber nicht umgekehrt. Anders ausgedrückt können wir sagen, dass ein Element in irgendeiner Form ein anderes Element benutzt. UML-Dependencies können ganz allgemein verwendet werden oder durch die Verwendung von Stereotypen in Gruppen eingeteilt werden. Es gibt eine große Zahl bereits vordefinierter Stereotype. Für die OT/J-Queries werden die Stereotype CALL, ACCESS und USE verwendet. Eine CALL-Beziehung besteht dann, wenn auf eine bestimmte Methode verwiesen wird, eine ACCESS-Beziehung bei Verweis auf ein bestimmtes Feld, eine USE-Beziehung bei Verweis auf eine bestimmte Klasse. Die Dependencies werden mittels Query Introduction als boolsche Aussage implementiert.

```
1  query boolean Method.use(ClassType t) {
2    this.expressions?[isMemberAccess_Type(t)]
3  }
4
5  query boolean Method.access(Field f) {
6    this.expressions?[isFieldAccess_Field(f)]
7  }
8
9  query boolean Method.call(Method m) {
10   (m elementof this.calls);
11 }
```

Listing 6.1: Erweiterung von Method für USE-, ACCESS- und CALL-Abhängigkeit. Die Erweiterung für Klassen erfolgt analog.

Bei Abhängigkeitsbeziehungen, die von Klassen ausgehen, müssen die jeweiligen Aussagen für mindestens eine Methode der Klasse erfüllt sein. Abhängigkeiten zu direkten Referenzbeziehungen funktionieren in gleicher Weise wie Abhängigkeiten auf Felder.

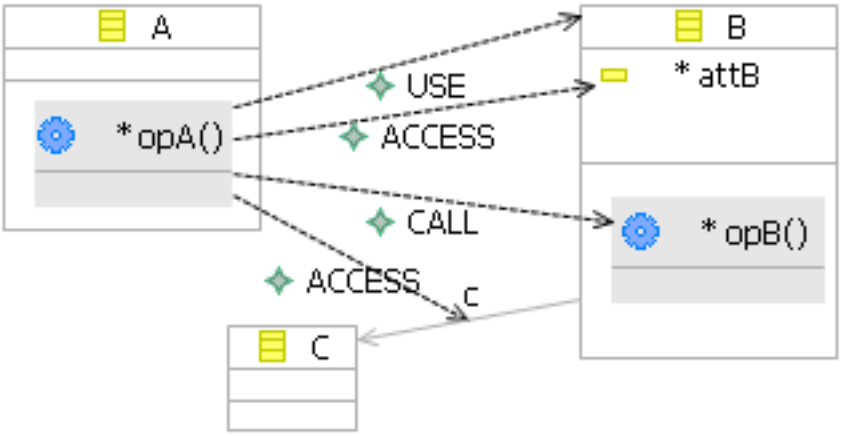

Abbildung 6.6: Abhängigkeitsbeziehungen

Die Abhängigkeitsbeziehungen in Abbildung 6.6 haben folgende Bedeutung:
Methode `opA` → Klasse `B`: *die Methode opA besitzt einen MemberAccess für die Klasse B.*
Methode `opA` → Attribut `attB`: *die Methode opA besitzt einen Zugriff für das Feld attB.*
Methode `opA` → Methode `opB`: *die Methode opA besitzt einen Aufruf für die Methode opB.*
Methode `opA` → Referenz `c`: *die Methode opA besitzt einen Zugriff auf das Feld c.*

Constraints Die Selektionskriterien einer Query können wie bei den JPDDs durch Angabe der Constraints NOT, OR und XOR logisch miteinander verknüpft werden. Das NOT entspricht dabei der klassischen einstelligen Negation aus der Logik. Die Junktoren OR und XOR dagegen werden für eine beliebige Anzahl von Parametern zugelassen:

- OR(A_0,..,A_n) ist wahr, wenn gilt ≪mindestens eine der Aussagen in $\{A_0,..,A_n\}$ ist wahr≫
- XOR(A_0,..,A_n) ist wahr, wenn gilt ≪genau eine der Aussagen in $\{A_0,..,A_n\}$ ist wahr≫

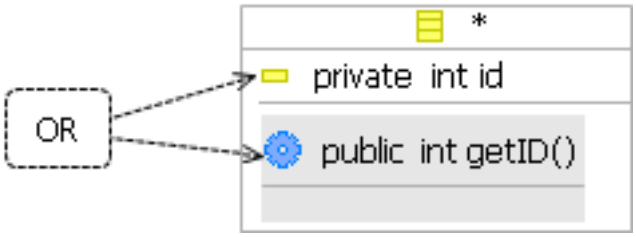

Abbildung 6.7: Beispiel für die Angabe eines Constraints: mindestens eines der beiden Elemente wird hier für die Klasse gefordert.

Eine Komposition von Constraints auf Constraints ist vorerst nicht vorgesehen. Leserlichkeit und Verständlichkeit des Diagramms würden durch eine Verschachtelung logischer Junktoren beeinträchtigt. Zudem erschwert die zusätzliche Komplexitätsebene die Abbildung der Constraints bei der Codegeneration, insbesondere im Hinblick auf die Qualität des Ergebnisses

(Performance, Leserlichkeit; siehe auch 8.2.4). Daher wird von vornherein eine Auflösung verschachtelter Bedingungen auf die Ebene der Knoten und Verbindungen verlangt.

Ausgabe und Identifier Eine deutliche Abweichung zu den JPDDs besteht letztlich in der Darstellung der Ein- und Ausgabeparameter. In JPDDs sind keine Eingabeparameter vorgesehen. Alle Informationen zur Spezifikation der Selektionskriterien werden direkt im Diagramm vorgegeben. Zur Ausgabe können wiederum mehrere der mittels von Identifiern spezifizierten Mengen angegeben werden. JPDDs sind in dieser Abstraktion nicht in allen Fällen direkt auf eine entsprechende Implementierung in einer Zielsprache umsetzbar, bedienen sich dafür aber verschiedener Workarounds [SHU07]. OTQDs dagegen repräsentieren eine konkrete Query, die direkt auf eine Funktion abgebildet werden kann. Eine solche Query soll flexibel gestaltbar sein, so dass sie auch durch die Angabe von Eingabeparametern spezifiziert werden kann. Sie darf dagegen nur ein Ergebnis eines bestimmten Typs, im Allgemeinen in Form einer Menge von spezifischen Elementen, zurückliefern. Es können weiterhin auch mehrere verschiedene Mengen selektiert werden, für die Rückgabe muss aber ein konkretes Return-Statement angegeben werden. Das Return-Statement kann dabei als beliebiger Ausdruck formuliert werden, im einfachsten Fall durch Angabe eines einzelnen Identifiers, ggf. aber auch als Verknüpfung mehrerer Ergebnismengen mithilfe von Mengenoperatoren, z.B. `<m1?> & <m2?>`.

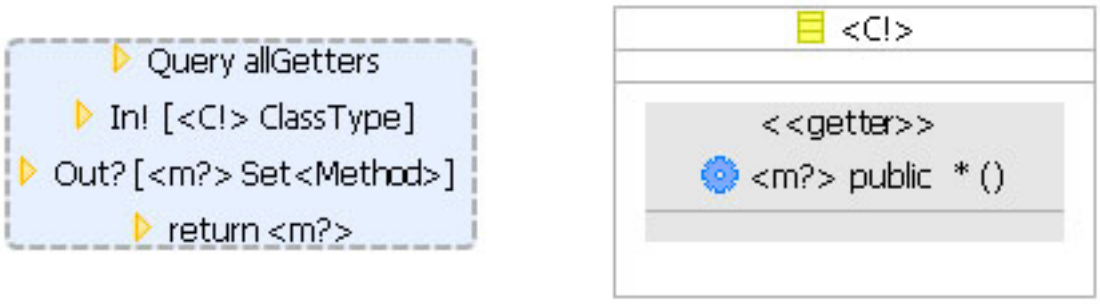

Abbildung 6.8: Im Basisknoten einer Query werden Name und Return-Anweisung der Query angegeben. In- und Out-Parameter können aus dem Diagramm entnommen werden. In! zeigt dabei die Identifier, die als Parameter verwendet werden. Out? zeigt die Identifier, die eine Zuweisung erhalten

In der Konsequenz bedeutet dies für die Darstellung auch, dass zwischen vorgegebenen und dynamisch zugewiesenen Identifiern unterschieden werden muss. Beide sollen nach wie vor in der Platzhalter-Form `<eineZuweisung?>` dargestellt werden. Zur Abgrenzung der fixen Vorgaben erhalten diese aber alternativ zu dem ≪?≫ nun ein ≪!≫ der Art `<eineVorgabe!>` (zur deutlicheren Unterscheidung wird hier das ≪?≫ bzw. ≪!≫ hinten angestellt).

Mit OTQDs sind wir nun in der Lage, ausgewählte Mengen von Elementen unseres Metamodells zu selektieren. Die grafische Notation in Anlehnung an das bekannte UML-Klassendiagramm bietet eine übersichtliche Darstellung der diversen Selektionskriterien. Insbesondere Elemente wie Methoden und Klassen lassen sich in dieser Form eingänglich darstellen. Die Granularität der Darstellung ist allerdings nach unten auf die Ebene der Methoden begrenzt. Grafische Elemente zur Auflösung nach den einzelnen Expression-Arten innerhalb der Methoden sind

vorerst nicht vorgesehen. Aufgrund der großen Anzahl und unterschiedlichen Beschaffenheit der einzelnen Expressionarten, und der damit einhergehenden Komplexität, eignen sich diese Informationen eher für die textuelle Query-Repräsentation.

6.3 UML-Mapping

Es ist augenscheinlich, dass die meisten Elemente eines JPDD bzw. eines OTQD sich ohne weiteres auf die UML-Elemente abbilden lassen. Elemente wie Klassen, Assoziationen, Constraints oder Abhängigkeiten haben äquivalente Entsprechungen in der UML. Dennoch gibt es einige problematische Aspekte, die bei einer solchen Abbildung zu berücksichtigen sind. Von Bedeutung sind dabei die semantischen Veränderungen bzw. Erweiterungen, die sich im Unterschied von der Darstellung eines Klassenmodells zur Darstellung musterbasierter Selektionskriterien ergeben. So verlangt die UML beispielsweise, dass die Namen für Klassen innerhalb eines Pakets eindeutig sind. Durch die Verwendung von Wildcards und Input-Parameteren wird diese Vorgabe eventuell verletzt, was aber inhaltlich gesehen im Grunde unproblematisch ist. Der entscheidende Unterschied liegt in der Verwendung von Identifiern, die in dieser Form in der UML nicht existieren. Stein, Hanenberg und Unland [SHU04b] bilden zur Einbindung die Identifier auf UML-Templates ab. Templates sind ≪potentielle≫ Modell-Elemente eines vorgebenen Typs, die erst durch Bindung eines Template-Parameters konkret instanziert werden können (das Konzept ist vergleichbar mit Generics in objektorientierten Sprachen wie Java oder C#). Der Selection-Parameter einer JPDD entspricht also dem Template-Parameter. Ein JPDD ist dann als ≪classifier template≫ oder ≪package template≫ o.ä. zu interpretieren. Das UML-Template, was im eigentlichen Sinne als Muster bzw. Vorlage zur Instanzierung dient, wird somit zu einem Selektions-Muster, dass eine Rückgabe liefert. Dies stellt eine deutliche Veränderung der ursprünglichen semantischen Aussage dar und ist geschuldet dem gravierenden Unterschied zwischen einer Strukturbeschreibung (UML) und einem Selektionsmuster (JPDD). In OTQDs wird zwar zwischen Ein- und Ausgabe unterschieden, das Mapping kann aber in gleicher Weise erfolgen. Die Parameter können dabei ebenfalls mittels Templates an die Modell-Elemente gebunden werden. Was die Eingabeparameter betrifft findet in diesem Fall auch keine semantische Umkehrung des Template-Prinzips statt.

7 Editor-Entwicklung

Für den Einsatz der OTQDs in der Praxis werden selbstverständlich neue Entwicklungs-Werkzeuge benötigt. Um dem modellgetriebenen Entwicklungsansatz gerecht zu werden benötigen wir zunächst einen Editor, mit dem OTQDs definiert werden können und danach einen Generator, der aus den OTQDs die zugehörige Implementierung generiert. Die Entwicklung des Editors wird in diesem Kapitel erläutert, die des Generators folgt im nächsten Kapitel. Sowohl Editor als auch Generator basieren auf dem Query-Metamodell. Für die konkrete Umsetzung der Werkzeuge wird, wie für die Sprache OT/J selbst, die Entwicklungsumgebung ≪Eclipse≫ verwendet. So können die Werkzeuge in das bestehende Object Teams Development Tooling (OTDT) integriert und direkt in den Entwicklungsprozess eingebunden werden.

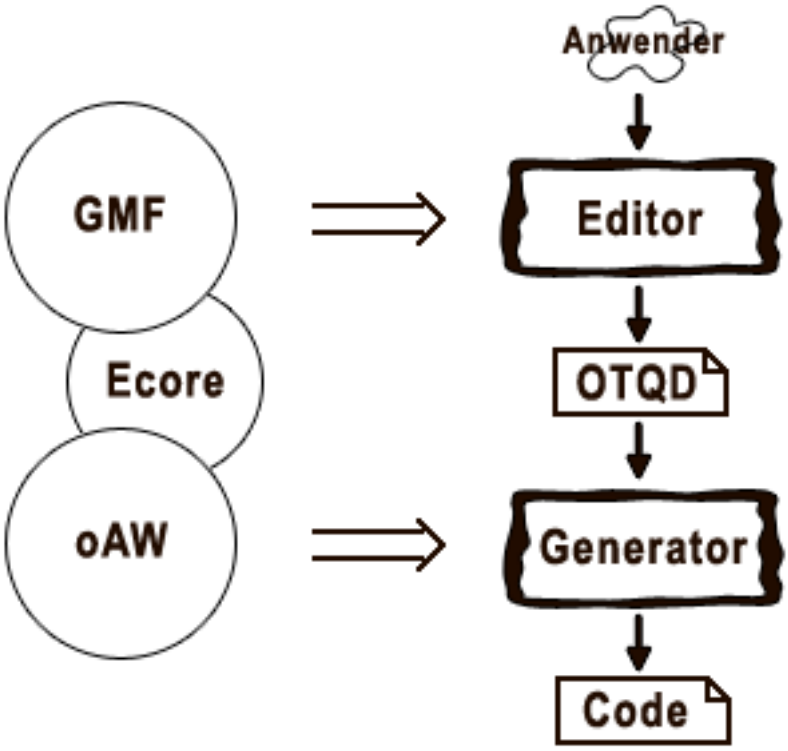

Abbildung 7.1: Entwicklungsprozess für OT/J-Queries: vom Anwender bis zur fertigen Implementierung. Der Editor wird mit dem Graphical Modeling Framework (GMF) realisiert, der Generator mit openArchitectureWare (oAW). Beide basieren auf dem Query-Metamodell im Ecore-Format.

7.1 Eclipse Frameworks

Aufgrund seiner Vielseitigkeit und des Open-Source-Konzeptes existieren zahlreiche Plugins für Eclipse, welche auch die modellgetriebene Entwicklung unterstützen. Im Folgenden werden die für den Editor relevanten Erweiterungen EMF, GEF und GMF (die einen Quasi-Standard darstellen) und ihre Anwendung näher vorgestellt. Die Frameworks sind allesamt Open-Source-Projekte und daher wie die Plattform Eclipse frei verfügbar.

7.1.1 Eclipse Modeling Framework

Das ≪Eclipse Modeling Framework≫ (EMF) [EF07a, SS05] dient der Erstellung und Bearbeitung von strukturierten Modellen und versucht damit, die modell-getriebene Software-Entwicklung in pragmatischer Weise zu unterstützen. Ziel ist es, mit möglichst geringem Einarbeitungs- und Entwicklungsaufwand Modelle bearbeiten und nutzen zu können. Modelle in EMF basieren auf dem Ecore-Metamodell, das zu großen Teilen der EMOF-Spezifikation (Essential Meta-Object-Facility) der OMG entspricht. Die Abweichungen sind nur gering und betreffen hauptsächlich die Bezeichnungen der diversen Elemente. EMF ist in der Lage EMOF zu lesen und zu schreiben.

Ein Ecore-Modell ist kompatibel zu diversen anderen Formaten und kann entsprechend aus diesen im- bzw. exportiert werden. Neben der Möglichkeit ein Modell direkt als Ecore zu definieren bestehen zur Zeit die Möglichkeiten, ein Modell als XML-Dokument, als annotierte Java Interfaces oder in Modellierungswerkzeugen wie Omondo oder Rational Rose zu spezifizieren. Somit reicht es aus, wenn ein Modell in einem dieser Formate vorhanden ist, und es muss kein weiterer Aufwand auf die Portierung verwendet werden.

EMF unterstützt den Entwickler mit der automatischen Implementierung seines Datenmodells, sie kann als Grundlage für Modell-Editoren und andere auf dem Datenmodell basierenden Anwendungen dienen. Alle für den Rahmen der Datenverwaltung relevanten Klassen und Interfaces werden anhand des Modells generiert. Um die reinen Modell-Informationen von den Generator-Anweisungen zu trennen, wird dazu aus der Ecore-Datei eine Genmodel-Datei generiert, die als Modell für den Generator dient. Alle für den Generierungsprozess relevanten Informationen sind hier enthalten und können auch manuell angepasst werden. Aus der Genmodel-Datei kann nun der Quellcode der Modellklassen generiert werden. Der Code besteht aus den entsprechenden Java-Interfaces des Modells, sowie einem Paket `impl` mit der Implementierung der Klassen und einem Paket `util` mit einigen Hilfsklassen.

Um Instanzen eines solchen Ecore-Modells anlegen zu können, unterstützt EMF den Entwickler auch mit der automatischen Generierung eines einfachen Editors. Im Paket `edit` werden Adapterklassen angelegt, welche die Bearbeitung der Modell-Elemente in der Eclipse-GUI möglich machen. Das Paket `editor` stellt schließlich das Plugin dar, mit dem der Anwender einen Editor zur Erstellung und Bearbeitung einer Modellinstanz bekommt. Die Darstellung der Modellelemente erfolgt in einem Baum, einer Liste oder Tabelle. Eine grafische Darstellung,

wie etwa in Form eines Klassen-Diagramms, existiert in einem EMF-Editor nicht und kann erst in Verbindung mit GEF bzw. GMF realisiert werden.

Die Nutzung von EMF bietet dem Entwickler viele Vorteile. Die verwendeten Modelle sind kompatibel zu einer Vielzahl von anderen Werkzeugen. Durch die automatische Code-Generierung für Implementierung und Editor wird dem Entwickler viel Arbeit abgenommen, zudem werden durch die Automatisierung mögliche Fehlerquellen reduziert. Dabei ist EMF außerordentlich flexibel. Sowohl das Generierungsmodell als auch die generierte Implementierung können individuell angepasst werden. Im Framework inbegriffen sind weiterhin Serialisierung (als XML-Datei) und Validierung (automatisch generierte Tests), ohne das der Entwickler sich darum kümmern muss. Zu guter Letzt gibt es weitere Eclipse-Projekte (wie GMF, das im Folgenden noch gezeigt wird), die das Ecore-Metamodell verwenden und mit EMF zusammenarbeiten.

7.1.2 Graphical Editing Framework

Das ≪Graphical Editing Framework≫ (GEF) [EF07b] ist ein Framework, dass Entwickler bei der Erstellung eines grafischen Editors unterstützt.

Die Architektur von GEF ist nach dem Model-View-Controller-Prinzip (MVC) aufgebaut. Die Modell-Daten (≪Model≫) und ihre für den Anwender sichtbare Repräsentation (≪View≫) werden logisch voneinander getrennt, ein Kontrolleinheit (≪Controller≫) sorgt für die Synchronisation und die Ereignisbehandlung. GEF realisiert dabei die Bereiche View und Controller. Die Spezifikation eines Modells ist dem Anwender überlassen. Es bietet sich an, dafür das Eclipse Modeling Framework zu verwenden, prinzipiell können aber auch beliebige andere Applikationen als semantisches Modell herangezogen werden. Die MVC-Architektur ermöglicht es auch, mehrere verschiedene Views bzw. verschiedene Controller für ein Modell zu erstellen.

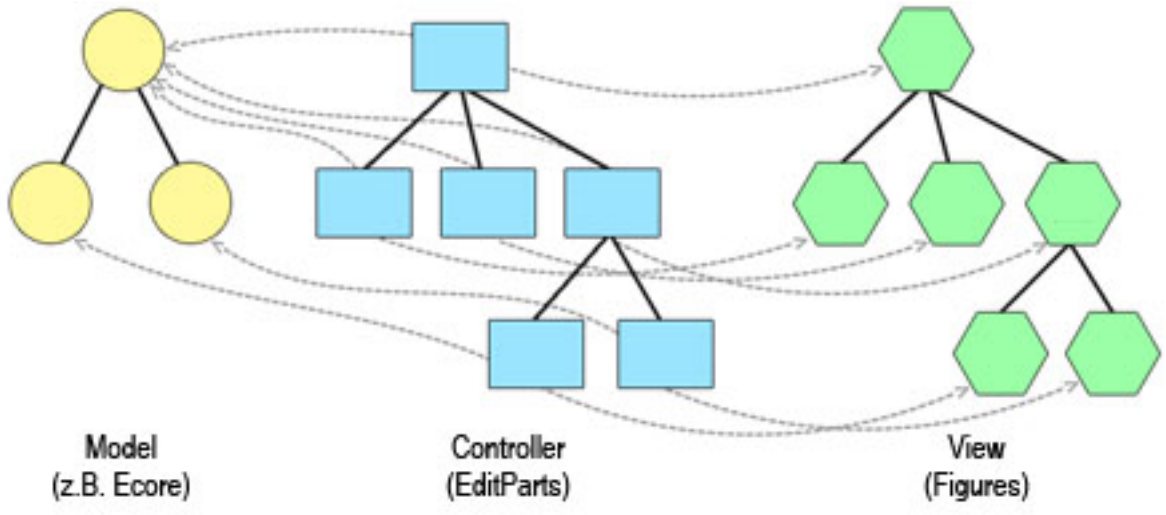

Abbildung 7.2: Model-View-Controller-Architektur

Für die View werden Figuren spezifiziert, mit welchen die Modell-Daten grafisch dargestellt werden. Zum Zeichnen und zum Layout der Grafiken wird das Plugin `org.eclipse.draw2d` verwendet. Der Controller wird mittels sogenannter EditParts realisiert. Die EditParts sorgen dafür, dass die Daten im Modell und ihre grafische Darstellung konsistent sind und bearbeiten die vom Anwender im Editor initiierten Aktionen zur Veränderung der Daten. Im Einzelnen wird auf diese Bereiche im folgenden Abschnitt zu GMF eingegangen.

Die Vorteile bei der Nutzung von GEF liegen in den vielen vorgefertigten Teilen des Editors. Das MVC-Konzept bietet eine gut strukturierte und anpassbare Architektur. Die Standard-Aufgaben, wie das Zeichnen von Figuren und das Bereitstellen diverser Werkzeuge für die Arbeit des Anwenders im Editor, sind in wesentlichen Teilen bereits realisiert und müssen nur noch auf das konkrete Modell zurechtgeschnitten werden.

7.1.3 Graphical Modeling Framework

In der Praxis werden EMF und GEF häufig miteinander kombiniert, da sich ihre Anwendungsbereiche in sinnvoller Weise ergänzen. Als logische Konsequenz folgte daher mit dem ≪Graphical Modeling Framework≫ (GMF) [EF07c] die Integration der Projekte EMF und GEF. Ziel des GMF-Projektes ist es, den zunächst immer noch recht anspruchsvollen Entwicklungsprozess EMF+GEF zu vereinfachen bzw. zu beschleunigen und dabei trotzdem die Funktionalität und Flexibilität der beiden Ursprung-Frameworks zu behalten. Im Sinne modellgetriebener Entwicklung reicht dabei die Vorgabe einiger weniger abstrakter Modelle aus, mit denen eine Editor-Anwendung spezifiziert wird. Integrierte Generatoren sorgen dann für automatische Implementierung. Auf diese Weise ist es möglich, auch ohne die Einzelheiten der EMF- und GEF-Implementierung zu kennen, einen voll funktionstüchtigen Editor zu erstellen. Die meisten Komponenten des Graphical Modeling Frameworks entstammen dabei trotzdem entweder dem Eclipse Modeling Framework oder dem Graphical Editing Framework. Eine manuelle Anpassung wie in diesen Frameworks ist daher nach wie vor möglich.

Auf die Struktur von GMF und einige konkrete Details wird im Zuge der Beschreibung der Implementierung des Query-Editors im folgenden Abschnitt näher eingegangen.

7.2 Implementierung des Query-Editors

7.2.1 Metamodell

Der Ausgangspunkt für die Umsetzung der OTQDs ist die Erstellung des Metamodells mit EMF. Abbildung 7.3 zeigt das Diagramm des OTQD-Metamodells, wie es in dem Framework selbst zur Darstellung und Bearbeitung verwendet wird (zwecks Übersichtlichkeit wurden allerdings einfache Superklassen wie `NamedElement`, `TypedElement` etc. entfernt).

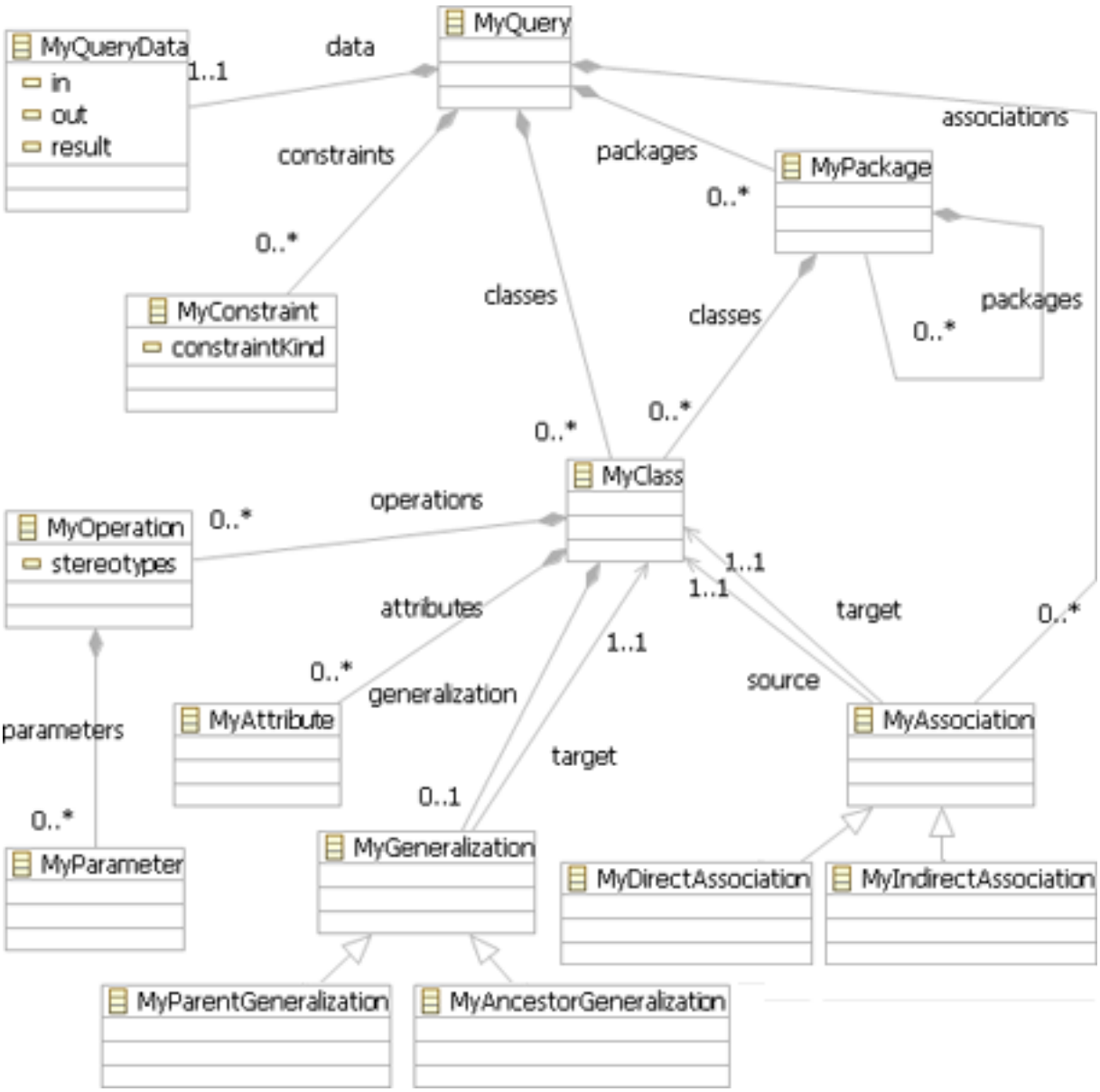

Abbildung 7.3: Ein Ausschnitt aus dem OTQD-Metamodell

Ein Metamodell sollte nach Möglichkeit unabhängig von der zur Umsetzung herangezogenen Plattform und deren technischen Beschränkungen erstellt werden, was in diesem Fall auch weitgehend beherzigt wurde. In Hinblick auf die nachfolgende Implementierung in GMF wurden allerdings einige Kompromisse zugelassen. Um eine deutliche Unterscheidung zu Schlüsselwörtern wie etwa «class» in Java und Ecore zu erreichen, wird allen Elementen der OTQD-Definition stets die Vorsilbe «my» vorangestellt. Weiterhin wurden die Daten der Query, wie z.B. ihr Name, in einen zusätzlichen Datenknoten ausgelagert, damit dieser später im Editor als eigenständiger Knoten (der Basisknoten der Query) zur Darstellung und Bearbeitung dient. Die eigentliche Query stellt die Wurzel eines Diagramms dar und hat daher keinen Knoten zur grafischen Repräsentation in dem selbigen.

7.2.2 Graph-, Tool- und Mapping-Definition

Zur Definition eines Editors für ein bestehendes Ecore-Modell sind in GMF zunächst drei Dateien vorgesehen: eine Graph-Defintion (gmfgraph), eine Tool-Definition (gmftool) und eine Mapping-Definition (gmfmap). Für alle drei stehen Wizards zur Verfügung, die den Anwender bei der Erstellung der Dateien unterstützen und den Entwicklungsprozess beschleunigen. Mit wenigen Mausklicks lässt sich so bereits eine Standard-Editor für das Modell erstellen, der als Grundlage für die weitere Arbeit dienen kann.

Die Graph-Definition (siehe Beispielabbildung 7.4) dient dazu, Figuren für das Diagramm zu erstellen. Die Figuren selbst haben an dieser Stelle noch keinen direkten Bezug zum Modell. Es geht allein um die grafische Repräsentation im Sinne von Form, Größe, Farbe, Aufteilung etc. Die Figuren werden in der Figure Gallery angelegt. Zur Definition von Knoten stehen diverse Standard-Formen wie Rechtecke, Ellipsen und Polygone zur Auswahl, die auch ineinander verschachtelt werden können. Eigene Formen können als «Custom Figure» eingebunden werden. Zur Textdarstellung können Labels in der Figur platziert werden. Für die konkrete Anordnung der Elemente einer Figur stehen diverse Layouts zur Verfügung. Im *FlowLayout* beispielsweise werden die Elemente fließend neben- bzw. untereinander angeordnet, beim *XYLayout* kann man dagegen konkrete Positionen vorgeben. Zur Definition von Link-Figuren werden Linien verwendet. Sowohl der Start- als auch der Zielpunkt der Verbindung kann dabei mit einer speziellen Dekoration versehen werden, so dass eine Linie z.B. mit Pfeilspitzen endet.

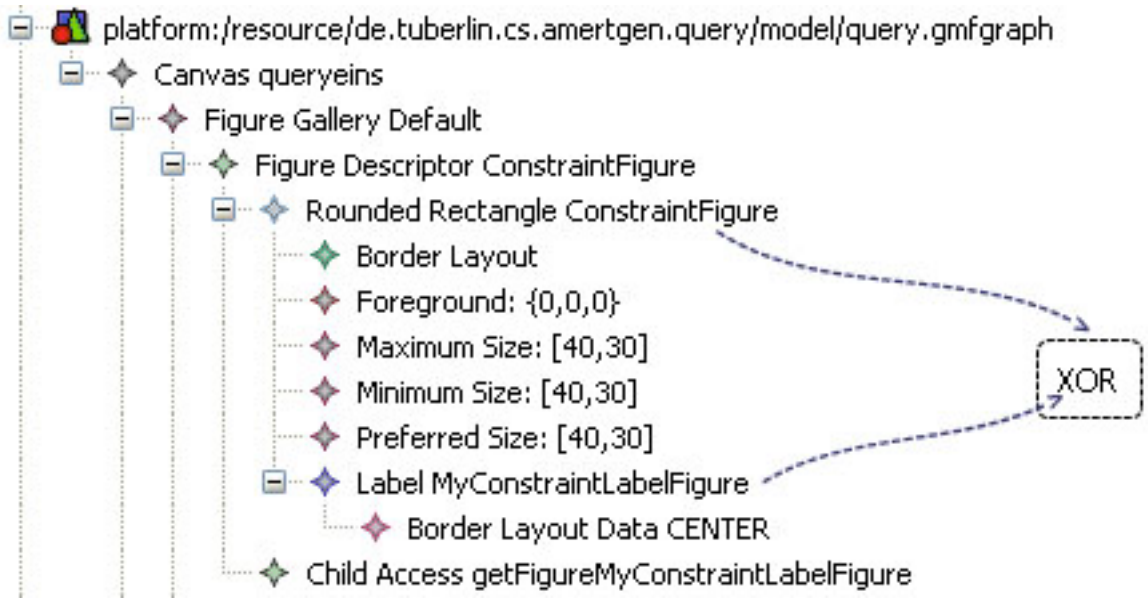

Abbildung 7.4: Eine Definition für eine ContraintFigure und deren sichtbares Ergebnis

Die Graph-Definition stellt nach außen hin Elemente für Knoten (Nodes), Verbindungen (Links), Unterteilungen (Compartments), und Textfelder (Diagram Labels) zur Verfügung. Die Figuren aus der Figure Gallery werden zur Definition dieser Diagrammelemente herangezogen. Dazu wird ein entsprechendes Objekt angelegt und mit einer Figur verknüpft, mit der es dargestellt

werden soll. Dieser Mechanismus dient der Kapselung, so kann eine Figur für mehrere Diagrammelemente verwendet werden. Des Weiteren kann einem Diagrammelement im Nachhinein ohne großen Aufwand eine andere Figur zugewiesen werden. Die Einzelteile einer Figur sind für das spätere Mapping durch ChildAccess-Methoden zugänglich. Einige Graph-Definitionen für gängige Figuren sind in GMF bereits vordefiniert und können als Ressource geladen werden (z.B. platform:/plugin/org.eclipse.gmf.graphdef/models/basic.gmfgraph).

Nach dem selben Prinzip wird die Tool-Definition erstellt (siehe Beispiel 7.5). Sie ist allerdings deutlich einfacher, da es hier nur wenige Möglichkeiten zur Einstellung gibt. In der Tool-Definition wird die Palette zusammengestellt, die später dem Anwender im Editor zur Verfügung stehen soll. Für jedes Element, das im Diagramm erstellt werden kann, muss hier ein «Creation Tool» angelegt werden. Die Creation-Tools können dabei in Gruppen geordnet werden. Die Verknüpfung mit einem entsprechenden Icon und einer aussagekräftigen Beschreibung ist alles, was an zusätzlicher Anpassung noch notwendig ist.

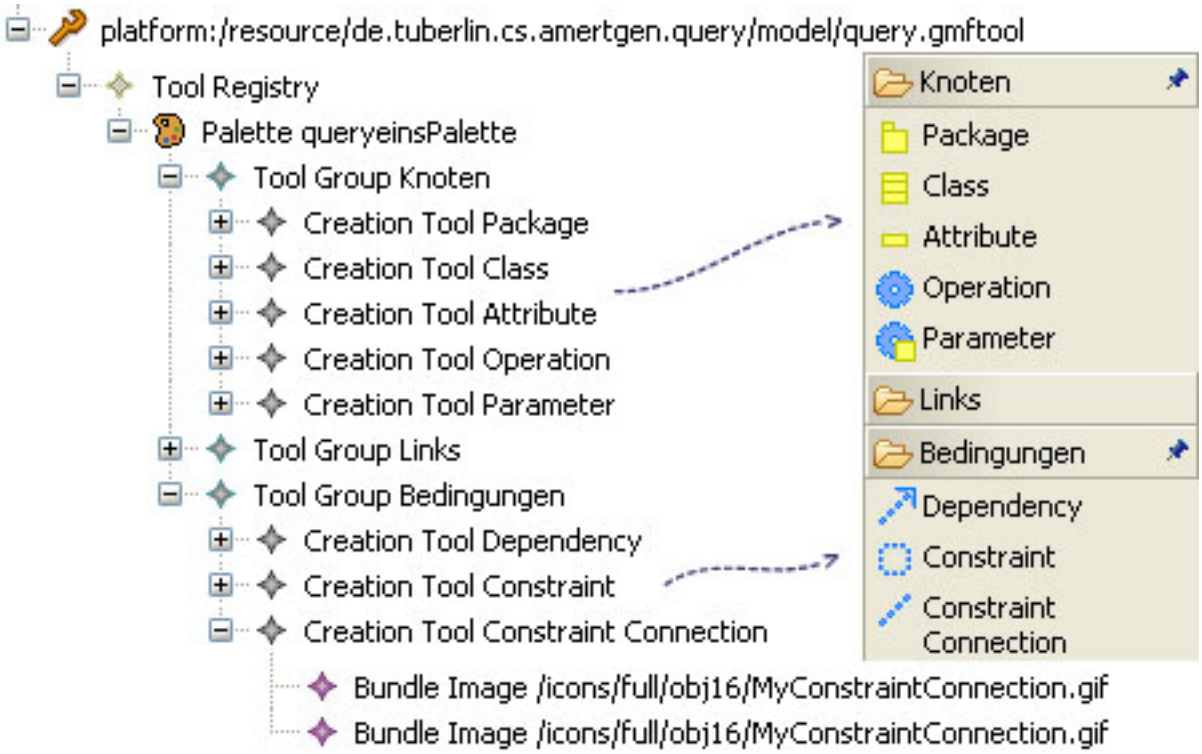

Abbildung 7.5: Eine Tool-Defintion und die fertige Palette

In der Mapping-Definition werden schließlich die einzelnen Komponenten der Graph- und Tool-Definition mit dem Ecore-Modell zusammengeführt und verknüpft. Aufgabe des Mappings ist es, für das Modell eine grafische Abbildung zu definieren. Die Zeichenfläche bildet dabei die Wurzel des Modells ab. Unter der Wurzel können Elemente für Knoten bzw. Verbindungen angelegt werden. Im Node- bzw. Link-Mapping wird jedem Diagrammelement ein entsprechendes Modellelement zugewiesen, das es repräsentieren soll. Aus der Graph-Definition wird eine Figur gewählt, mit der das Element gezeichnet wird. Die Verknüpfung mit einem Tool aus der Tool-Definition sorgt dafür, dass der Anwender ein solches Element später im Editor erzeugen

kann. In den Knoten können wiederum weitere Kinderknoten definiert werden. Auf diese Weise wird die Baumstruktur des Modells auf die Hierarchie der Diagrammelemente übertragen. In der Mapping-Definition werden auch die Einstellungen zur Textanzeige der Label und die Validierung vorgenommen, auf die an späterer Stelle noch ausführlicher eingegangen wird.

Sind alle Einstellungen in der Mapping-Definition vollständig kann daraus die gmfgen-Datei generiert werden. Die gmfgen-Datei enthält alle relevanten Informationen zur Generierung des Editors. Im Prinzip können alle Einstellung auch direkt hier vorgenommen werden, die anderen Dateien beschränken sich allerdings auf das für den Programmierer Wesentliche und sind daher deutlich übersichtlicher. Aus der gmfgen-Datei kann nun die Implementierung des Editor-Plugins generiert werden (das Paket `diagram`). Der Quellcode ist sehr umfangreich und umfasst ein Dutzend Pakete mit einer Vielzahl von Klassen. Um die Struktur und Funktionsweise des Editors zu veranschaulichen soll im Folgenden auf die wichtigsten Komponenten eingegangen werden. Das Hauptaugenmerk liegt dabei auf den Bereichen, an denen wir den Code ggf. noch für unsere Zwecke manuell anpassen wollen.

7.2.3 EditParts

Der Kern eines GMF-Editors sind die EditParts (`diagram.edit.parts`). EditParts fungieren als Controller und dienen als zentrale Schnittstelle, um die Modellelemente und ihre grafische Repräsentation zu verwalten. Zu jedem EditPart gehört ein Element im Diagramm, das kann zum Beispiel ein Knoten oder ein Label sein. Die Methode `getModel()` liefert das jeweilige Element, was bei grafischen EditParts vom Typ `View` ist. Ein EditPart dient mitunter als Container für weitere EditParts, die über die Methode `getChildren()` erreicht werden können. Das gesamte Diagramm liegt in Form einer Baumstruktur vor, wobei das Basis-Objekt der Diagramm-Oberfläche die Wurzel darstellt. Die EditParts werden in einer Factory erzeugt und dabei mit einem bestimmten Modellelement verknüpft. Mit der Methode `resolveSemanticElement()` kann auf dieses Element zugegriffen werden. Das Modellelement ist ein Objekt aus der Implementierung des zugrunde liegenden Ecore-Modells und hält somit die eigentlich relevanten Daten. Es ist durchaus möglich, dass mehrere EditParts mit dem selben Modellelement verknüpft sind, wenn sie jeweils unterschiedliche Bereiche dieses Elements repräsentieren.

Ein Beispiel aus der Query-Implementierung soll zur Veranschaulichung der EditParts und weiterer GMF-Elemente dienen. Dazu sehen wir die Darstellung einer Methode im Editor (Abbildung 7.6), die Struktur der dazu gehörenden Editparts (Abbildung 7.7) und die Modellinformationen in Form einer XML-Datei (Listing 7.1).

```
1 <operations name="foo" type="void" visibility="public">
2   <parameters name="x" type="int"/>
3   <parameters name="y" type="int"/>
4   <stereotypes>someStereotype</stereotypes>
5 </operations>
```

Listing 7.1: XML-Repräsentation einer Methode mit Parametern

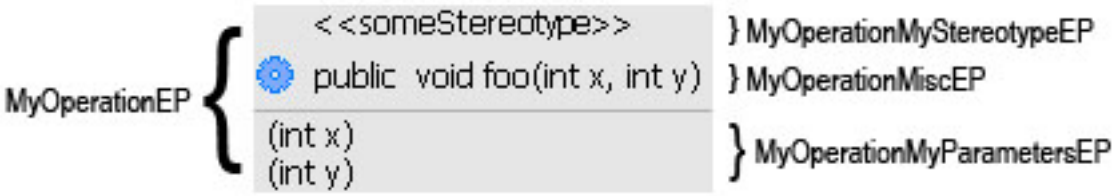

Abbildung 7.6: Eine Methode wird im Editor mit einem grauen Rechteck dargestellt

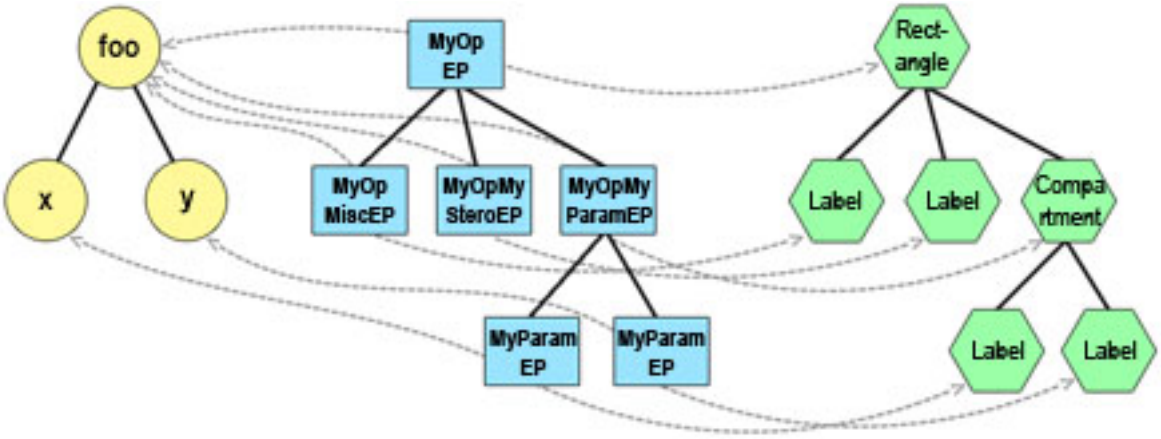

Abbildung 7.7: Ausschnitt aus der Editor-Struktur mit Modellelementen, EditParts und Figures

Eine Klassenmethode `foo` (als Instanz vom Typ `MyOperation`) wird im Diagramm als Knoten dargestellt. Der Knoten beinhaltet zwei Labels und ein Compartment. Das erste Label dient dazu, die verschiedenen Eigenschaften der Methode wie Name und Sichtbarkeit darzustellen. Das zweite Label darüber soll die zur Methode gehörenden Stereotypen darstellen. Das Compartment darunter dient zur Aufnahme von Parametern, die wiederum in einem eigenen Typ gekapselt sind. Für den Diagramm-Knoten der Methode wird nun ein `MyOperationEditPart` angelegt. Dieser EditPart hat wiederum drei Kinder, ein `MyOperationMiscEditPart` zur Repräsentation des ersten Labels (der Bezeichner wurde zwecks Leserlichkeit geändert, der generierte Bezeichner dieses EditParts basiert auf der Feature-Liste und lautet `MyOperationIdentifier-MyVisibilitySEditPart`), ein `MyOperationMyStereotypesEditPart` zur Repräsentation des zweiten Labels, und ein `MyOperationMyParametersEditPart` zur Repräsentation des Parameter-Compartments. Alle genannten EditParts haben als semantisches Element die Klassenmethode `foo` vom Typ `MyOperationImpl`. Wird ein EditPart im Diagramm bearbeitet, so werden die Änderungen an dieses Element weitergegeben. Während die beiden EditParts für die Label keine weiteren Kinder haben, so dient das `MyOperationMyParametersEditPart` lediglich als Container für die Aufnahme von Parametern. Die wiederum werden jeweils durch ein MyParameterEditPart repräsentiert und sind mit Elementen vom Typ `MyParameterImpl` verknüpft.

Die konkrete Darstellung des Textes, der in den Labeln steht, wird im GMF-Mapping eingestellt. Dort kann im Feld ≪Features≫ eine Liste von Eigenschaften des Modellelements ausgewählt werden, die zu einem String verknüpft werden sollen. Die Art der Darstellung

kann durch die Angabe eines *View-Patterns* individuell angepasst werden. Ein Beispiel für eine solches Muster wäre die Angabe ≪({0}, {1})≫ zur Parameterdarstellung im Label des `MyParameterEditParts`. Die geschweiften Klammern dienen dabei als Platzhalter für das Feature mit der jeweiligen Nummer (definiert über die Reihenfolge in der Liste). Im Beispiel entspräche 0 dem Typ und 1 dem Namen eines Parameters. Die übrigen Zeichen werden an den entsprechenden Positionen hinzugefügt. In gleicher Weise kann auch ein *Edit-Pattern* definiert werden, womit angegeben wird, wie eine Text-Eingabe in das Feld zu interpretieren ist. Die Muster zur Darstellung und Eingabe dürfen durchaus verschieden sein.

Etwas aufwendiger wird es, wenn man eine bedingte Formatierung verwenden möchte. Komplexere Ausdrücke, wie eine Fallunterscheidung und die Berücksichtigung von Kontext-Informationen, lassen sich im View-Pattern nicht abbilden. An dieser Stelle muss der generierte Quellcode angepasst werden. Für die Abhängigkeitsbeziehung `MyDependency` möchten wir beispielsweise sehen, ob es sich um eine *use*, *access* oder eine *call*-Beziehung handelt. Der zugehörige Link soll diese Information in einem Label anzeigen. Die Art der Beziehung ergibt sich automatisch aus dem Zielknoten, auf den der Link verweist. Ein Anpassung des Modells ist dabei weder erwünscht noch notwendig, wir wollen ausschließlich Einfluss auf die grafische Darstellung nehmen. Wenn der Anwender einen neuen Link des Typs `MyDependency` anlegt wird eine entsprechende Diagramm-Figur instanziiert und angezeigt. Um den gewünschten Effekt zu erzielen müssen wir bei der Erstellung der Figur nur den zugehörigen String initialisieren. Der Elternknoten des Label-EditParts ist die eigentliche Verbindung, anhand der wir das Zielobjekt ermitteln können.

```
1  protected IFigure createFigurePrim() {
2          WrapLabel label = new LabelDependencyFigure();
3          ConnectionNodeEditPart parent =
4              (ConnectionNodeEditPart) getParent();
5          INodeEditPart target =
6              (INodeEditPart) parent.getTarget();
7          if ((target instanceof MyClassEditPart) {
8                  label.setText("USE");
9          }
10         if ((target instanceof MyAttributeEditPart)
11         || (target instanceof MyDirectAssociationEditPart)) {
12                 label.setText("ACCESS");
13         }
14         if (target instanceof MyOperationEditPart) {
15                 label.setText("CALL");
16         }
17         return label;
18 }
```

Listing 7.2: Methode zur Erstellung der Figur in LabelDependencyEditPart

Alle vom Framework generierten Methoden sind übrigens mit dem Tag ≪generated≫ im Javadoc-Kommentar versehen. Wird eine Methode manuell verändert sollte dieses Tag in ≪generated

NOT» o.ä. geändert werden, damit der Code bei einem eventuellen Neugenerieren nicht überschrieben wird.

Zurück zum Beispiel aus Abbildung 7.6. In der Methode `foo` sehen wir, dass die Parameter `x` und `y` nicht nur im Compartment, sondern auch im `MyOperationMiscEditPart` angezeigt werden. Da Parameter aber im Modell als Kinder einer Methode angelegt sind liegen diese standardmäßig nicht im Scope dieses EditParts. Das GMF-Mapping bietet bisher auch leider keine Möglichkeit automatisch auf diese Knoten zuzugreifen. Die zur Darstellung notwendigen Anpassungen müssen also ebenfalls von Hand vorgenommen werden. Der Zugriff auf die Parameter erfolgt entlang der Baumstruktur des Modells. Über die Parent-Child-Hierarchie der EditParts bzw. der Modellelemente können wir von der Methode zu den Parametern gelangen. Für die konkrete Zusammenstellung des angezeigten Strings ist ein Parser zuständig, der entsprechend der View- bzw. Edit-Patterns konfiguriert ist. Der Parser wird im EditPart von der Methode `getLabelText()` aufgerufen. Durch Anpassung des Parsers und des generierten Strings können wir jetzt die Parameterdarstellung an dieser Stelle mit aufnehmen. Die Anpassung des Strings allein reicht aber noch nicht aus, wir müssen auch dafür sorgen, dass der String bei Änderungen an den Parametern aktualisiert wird.

7.2.4 Listener und Notification

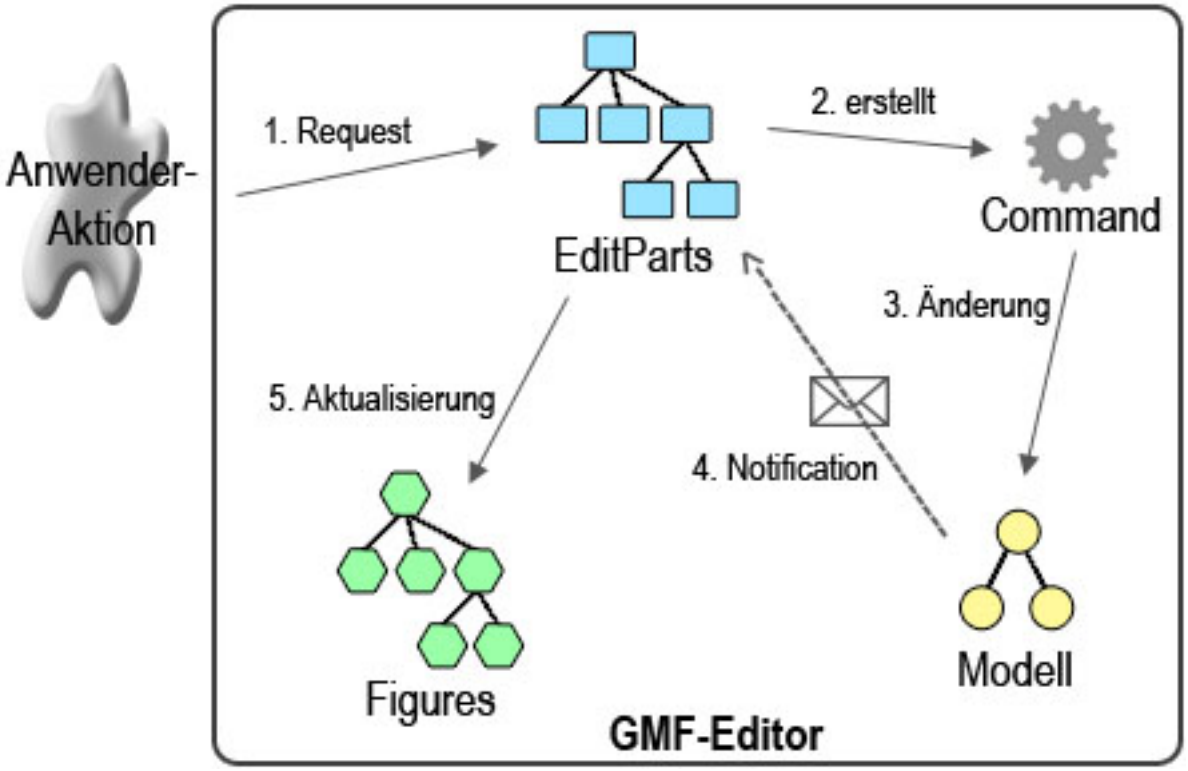

Abbildung 7.8: Notification-Mechanismus in GMF

Die MVC-Architektur in GMF sieht vor, dass alle Aktionen des Anwenders sofort bearbeitet werden und unverzüglich im Modell übernommen werden. Änderungen am Modell lösen dann eine Aktualisierung der grafischen Darstellung aus. Die Synchronisation findet also nur in eine Richtung statt, vom Modell zur Ansicht (Abbildung 7.8), was den Implementierungsaufwand reduziert.

Damit ein EditPart auf Änderungen am Modell reagieren kann exisitiert ein Notification-Mechanismus. Der Mechanismus wird über die Methoden `activate()` bzw. `deactivate()` an- bzw. ausgeschaltet. Es gibt zwei Gruppen von Listenern: einmal die *NotationalListeners*, welche Diagramm-Elemente wie EditParts, Views und Figures beobachten, zum zweiten die *SemanticListeners*, welche die zugrunde liegenden Modellelemente beobachten. Standardmäßig lauscht ein Editpart nur auf den direkt mit ihm verknüpften Elementen. Um zu erreichen, dass er darüber hinaus auf Ereignisse reagiert müssen wir manuell weitere Listener hinzufügen. In diesem Fall möchten wir, dass jedes `MyParameterEditPart` sich bei dem `MyOperationMiscEditPart` seiner Methode registriert, damit dieses über Änderungen der Parameter informiert wird. Dazu erweitern wir die Methode `addSemanticListener()` in `MyParameterEditPart`.

```
1  //Children of MyOperationEditPart
2  List editparts = getParent().getParent().getChildren();
3    for (int i = 0; i < editparts.size(); i++) {
4          Object obj = editparts.get(i);
5          if (obj instanceof MyOperationMiscEditPart) {
6            MyOperationMiscEditPart misc = (MyOperationMiscEditPart) obj;
7            addListenerFilter("SemanticParameter", misc,
8              this.resolveSemanticElement());
9      }
10   }
11 }
```

Listing 7.3: Erweiterung von addSemanticListeners() in MyParameterEditPart

Die Methode `addListenerFilter()` registriert den Listener bei einem `DiagramEventBroker`, der alle Listener eines Diagramms zentral verwaltet. Wird nun ein Parameter bearbeitet, so wird eine Nachricht an das entsprechende `MyOperationMiscEditPart` weitergeleitet, die dort in der Methode `handleNotificationEvent(..)` behandelt werden kann. Zur Aktualisierung der gesamten Darstellung dient die Methode `refreshVisuals()`, in diesem Fall ist aber bereits die Methode `refreshLabel()` ausreichend.

7.2.5 EditPolicies und Commands

Die Funktionalität, um das Modell zu bearbeiten, wird in sogenannten EditPolicies (`diagram.edit.policies`) implementiert. Dabei wird zwischen *CanonicalEditPolicies* und *SemanticEditPolicies* unterschieden. Erstere sind speziell mit der Synchronisierung von Modell und View

betraut, letztere sind für die Bereiche der Modellbearbeitung gedacht. Jeder EditPart-Klasse können eigene Policies zugeordnet werden, die in der Methode `createDefaultEditPolicies()` angelegt werden. Standardmäßig werden in GMF bereits alle notwendigen Policies für die grundsätzlichen Arbeitsschritte wie das Erstellen, Bearbeiten oder Löschen von Elementen generiert. Zur individuellen Anpassung können diese verändert oder neue Policies angelegt werden. Das Grundgerüst dazu wird von den Klassen `SemanticEditPolicy` bzw. `AbstractEditPolicy` vorgegeben. Aktionen des Anwenders werden in Form von Requests an die EditParts weitergeleitet. Der Request wird dort von den EditPolicies ausgewertet. Sofern die Auswertung erfolgreich ist wird ein der Aktion entsprechendes Command erstellt, das für die eigentliche Umsetzung der Änderungen am Modell sorgt. Die Kapselung der Anweisungen in einem Command-Objekt dient dazu, eine Undo-/-Redo-Funktion zu ermöglichen. Alle Command-Objekte landen auf einem Stack, von wo aus sie ausgeführt bzw. die Ausführung wieder rückgängig gemacht werden kann.

Als kurzes Beispiel zur Bearbeitung einer EditPolicy nehmen wir den EditPart für den Datenknoten einer Query. Da jede Query genau einen Knoten als Container für ihre grundlegenden Daten wie Name und Rückgabe halten soll, möchten wir verhindern, dass dieser Knoten vom Diagramm gelöscht werden kann. Zum Löschen kann der Benutzer im Diagramm für ein Element die Option ≪deleteFromModel≫ auswählen, wobei ein DestroyRequest generiert wird. In der EditPolicy wird nun u.a. die Methode `shouldProceed(DestroyRequest destroyRequest)` aufgerufen, deren Rückgabewert vom Typ `boolean` Auskunft gibt, ob das Löschen durchgeführt werden soll oder nicht. Im Allgemeinen ist dieser Wert `true`. In der Klasse `MyQueryDataItemSemanticEditPolicy` müssen wir nun die Methode überschreiben, so dass sie `false` zurückliefert. Wählen wir jetzt im Diagramm für das Element die Option ≪deleteFromModel≫ würde der Request zurückgewiesen und das Löschen wird nicht durchgeführt.

Da die Veränderung des generierten Codes nicht unbedingt erwünscht ist, gibt es noch eine andere Möglichkeit auf die EditPolicies Einfluss zu nehmen. EditHelper (`diagram.edit.helper`) können definiert werden, um vor, während oder nach einer Aktion zusätzliche Anweisungen auszuführen. Ein EditHelper implementiert dazu entsprechende *EditHelperAdvices* (Before-, Instead-, After-Advice), die an den jeweiligen Stellen aufgerufen werden. Die Advice-Bindings müssen dazu extra unter den Extension-Points der entsprechenden Elementtypen in der *plugin.xml* des Diagramms registriert werden. Dies sind also keine Advices im aspektorientierten Sinne, sondern vordefinierte Hooks, die für eine Adaptierung verwendet werden können.

7.2.6 Validierung

Ein wichtiger Arbeitsschritt bei der Erstellung von Modellen ist die Validierung. Die Anzahl nutzbarer und darstellbarer Element zur Definition eines Modells ist naturgemäß begrenzt. Im Allgemeinen dienen sie dazu, Struktur und Beziehungen von Modellelementen darzustellen. Die Anforderungen, die wir an das Modell stellen, sind aber manchmal so komplex oder so speziell (z.B. die Einhaltung bestimmter Randbedingungen), dass wir sie mit diesen Mitteln nicht direkt im Metamodell verankern könnten. Zu diesem Zweck bietet uns das GMF-Framework durch sogenannte *Audits* die Möglichkeit, zusätzliche Validierungs-Funktionen einzubinden [DG07].

Die Audits werden in der gmfmap-Datei angelegt. Zur Definition können verschiedene Sprachen herangezogen werden. Zur Auswahl stehen die Programmiersprache Java selbst, reguläre Ausdrücke und die OCL. Im Vergleich zu den regulären Ausdrücken, die nur nach `true` oder `false` ausgewertet werden können, ist die OCL wesentlich mächtiger. Die OCL ist Bestandteil der UML und dient zur Spezifikation von Aussagen, mit denen Bedingungen (Vor-, Nachbedingung, Invariante, etc.) definiert werden können, beispielsweise für Klassen und Methoden in einem Klassendiagramm. OCL ist eine reine Abfragesprache, d.h. sie produziert keine Seiteneffekte. OCL-Ausdrücke werden ausgewertet, ohne dass dabei in irgendeiner Form die zugrunde liegenden Daten verändert werden. Ein Constraint, man könnte auch von einer ≪Bedingung≫ oder ≪Zusicherung≫ sprechen, wird dabei immer im Rahmen eines konkreten Kontextes definiert. ≪Kontext≫ steht hier für ein beliebiges Element des zu spezifizierenden Systems. Im Falle eines Klassendiagramms z.B. für eine Klasse. Eine einfaches Beispiel für einen Constraint sieht folgendermaßen aus:

```
1 context MyNamedElement inv:
2 self.name.size() > 0
```

Listing 7.4: OCL-Ausdruck um sicher zu stellen, dass ein Name angegeben wurde

Als Kontext wird das Element `MyNamedElement` angegeben, was bedeutet, dass der folgende Ausdruck nur auf Objekte diesen Typs anzuwenden ist. Das Kürzel `inv` beschreibt die Art des Contraints und steht in diesem Fall für ≪Invariante≫. In der zweiten Zeile steht der eigentliche Constraint. Hier stellen wir die Forderung auf, dass das Attribut `name` des `MyNamedElement`-Objekts eine Länge größer als 0 haben muss. Alle Elemente, deren Name mindestens ein Zeichen lang ist, würden diese Anforderung also erfüllen. Die Angabe `self` als Selbstreferenz ist in diesem Fall nicht notwendig, kann aber besonders in komplexeren Ausdrücken die Lesbarkeit erhöhen.

In GMF werden die einzelnen Elemente eines Constraints separat angegeben. Das umschließende Konstrukt ist eine sogenannte *Audit Rule*, der Kontext wird als *Domain Element Target* ausgewählt. Am Inhalt der OCL-Aussagen ändert sich dadurch aber nichts. Darüber hinaus kann aber das mit dem Constraint verbundene Verhalten definiert werden, so z.B. ob es sich bei dieser Regel um einen Fehler, eine Warnung oder lediglich eine Information handelt sowie ein Hinweistext für die Anzeige. Die Validierung kann permanent im laufenden Betrieb stattfinden oder explizit durch eine Aktion vom Anwender gestartet werden. Im Folgenden wird die Anwendung von OCL-Ausdrücken exemplarisch für unser Query-Metamodell gezeigt. Die OCL ist eine sehr mächtige Sprache, mit der sich eine Vielzahl von Ausdrücken formulieren lässt. Die genannten Beispiele schöpfen nur einen Bruchteil dieser Möglichkeiten aus.

Zunächst möchten wir die einfache Anforderung realisieren, dass für jedes Objekt, dem ein Identifier zugewiesen werden kann, mindestens ein Name oder ein Identifier spezifiziert wurde. Der Anwender darf für den Namen ruhig einen ≪*≫ einsetzen, wir wollen lediglich sicherstellen, dass die Angabe nicht völlig vergessen wurde. Zu diesem Zweck legen wir den folgenden Constraint an (siehe Listing 7.5).

```
1  context MyIdentifier inv:
2  (name->notEmpty() and name.size() > 0)
3  or (identifier->notEmpty() and identifier.size() > 0)
```

Listing 7.5: Prüfung, ob Name oder Identifier vergeben wurden

Für die Attribute `name` und `identifier`, die beide vom Typ String sind, wird geprüft, ob sie eine Länge von mindestens einem Zeichen haben. Da ein String auch mit `null` belegt sein kann ist vorher zusätzlich die Abfrage `notEmpty()` notwendig. Die logische Oder-Verknüpfung gibt an, dass die Anforderung für mindestens eins der beiden Attribute erfüllt sein muss.

Im Modell werden die Identifier allein über einen String repräsentiert. Eine explizite Typzuweisung findet nicht statt, was seine Vor- und seine Nachteile mit sich bringt. Wir wollen verhindern, dass es bei der impliziten Typisierung zu Fehlern kommt. Dazu müssen wir sicherstellen, dass ein Identifier bei jedem Vorkommen im richtigen Kontext seines Typs steht. Z.B. darf ein Identifier, der für eine Klasse vergeben wurde, nicht an anderer Stelle noch mal für eine Methode verwendet werden. Für diese Prüfung müssen wir alle Instanzen des Typs `MyIdentifier` betrachten und untereinander vergleichen. Es gilt, dass für zwei Objekte entweder die Identifier (sofern vorhanden) verschieden oder die Typen gleich sein müssen. Dass ein Identifier für ein und denselben Typ nicht mehrfach zugewiesen wird ist an dieser Stelle noch nicht gesichert und wird in einem anderen Constraint geprüft.

```
1  context MyIdentifier inv:
2  MyIdentifier.allInstances()->forAll(other | ((other <> self)
3  and (self.identifier->notEmpty()) and (self.identifier.size()>0)
4  and (other.identifier->notEmpty()) and (other.identifier.size()>0))
5  implies ((other.identifier <> self.identifier)
6  or (other.oclIsTypeOf(MyClass) and self.oclIsTypeOf(MyClass))
7  or (other.oclIsTypeOf(MyOperation) and self.oclIsTypeOf(MyOperation))
8  or (other.oclIsTypeOf(MyPackage) and self.oclIsTypeOf(MyPackage))
9  or ((other.oclIsTypeOf(MyAttribute) or other.oclIsTypeOf(MyDirectAssociation))
10 and (self.oclIsTypeOf(MyAttribute) or self.oclIsTypeOf(MyDirectAssociation)))))
```

Listing 7.6: Prüfung, ob ein Identifier immer im richtigen Typkontext steht

Mittels der Funktion `allInstances()` wird eine Collection definiert, in der alle Instanzen eines Typs vorhanden sind. Die Funktion `forAll(..)` dient als Iterator, um für jedes Element aus der Menge eine Bedingung zu prüfen. Zur Prüfung auf einen bestimmten Typ eines Elements existiert in OCL 2.0 die Funktion `oclIsTypeOf(..)` (äquivalent zu `instanceof`). Leider existiert aber keine Funktion, um den Laufzeit-Typ eines Objekts zu extrahieren (die Funktion `oclType` ist in OCL 2.0 nicht mehr enthalten). Um zwei Elemente auf Typgleichheit zu prüfen kann also nicht auf einen Ausdruck wie `obj1.oclIsTypeOf(obj2. oclType())` zurückgegriffen werden. Stattdessen müssen alle Typen explizit geprüft werden, was zu zusätzlichen Schreibaufwand und verminderter Lesbarkeit führt.

Eine weitere Anforderung an das Klassenmodell ist eine zyklenfreie Vererbungshierarchie. Hier stoßen wir auf das klassische Problem der Identifizierung von Zyklen in einem Graph und damit an die Grenzen der OCL.

```
1 context MyClass inv:
2 self.myGeneralization.target.myGeneralization.target <> self
```

Listing 7.7: Prüfung, ob ein Vererbungszyklus besteht

Der Constraint kann zwar für einen explizit angegebenen Zyklus ausgewertet werden (und somit könnten alle Vererbungszyklen bis zu einem gewissen Grad manuell definiert werden). Eine rekursive Suche entlang der Verbindungen der Elemente, die alle Ebenen abdeckt, ist aber nicht möglich.

Bei komplexen Constraints wie diesem, die mit der OCL nur schwierig darstellbar sind, ist es alternativ möglich den Constraint in Java zu formulieren. Dazu stellen wir die Audit Rule in GMF auf die Sprache Java ein. Im Constraint-Body definieren wir nur den Namen einer Operation. Bei der anschließenden Code-Generierung wird dann in der Klasse `XXXValidationProvider` im Paket `diagram.providers` eine statische Klasse `Java Audits` mit einem Skelett für diese Operation angelegt. An dieser Stelle muss der Constraint jetzt konkret implementiert werden. Um Zyklen in der Vererbungshierarchie aufzudecken, definieren wir eine Schleife, mit der wir ausgehend von einer Klasse den Generalisierungspfad verfolgen und prüfen, ob wir irgendwann wieder zum Ausgangspunkt zurückkehren. Um zu verhindern, dass wir dabei von außen in einen Zyklus stoßen und in einer Endlosschleife landen müssen wir uns zudem alle besuchten Klassen merken und bei jedem Schritt abprüfen, ob wir die aktuelle Klasse bereits zuvor angelaufen haben.

```
1  private static Boolean cyclicGeneralizationCheck(MyClass self) {
2          MyClass parent;
3          MyGeneralization gen = self.getMyGeneralization();
4          List<MyClass> list = new LinkedList<MyClass>();
5          while (gen != null) {
6                  parent = gen.getTarget();
7                  if (parent == self) return false;
8                  if (list.contains(parent)) {
9                          break;
10                 } else {
11                         list.add(parent);
12                 }
13                 gen = parent.getMyGeneralization();
14         }
15         return true;
16 }
```

Listing 7.8: Constraint zum Zyklencheck, formuliert in Java

7.2.7 Erweiterbarkeit des Editors

Eine positive Eigenschaft der modellgetriebenen Entwicklung ist die Flexibilität gegenüber Änderungen. Sollte die Notwendigkeit entstehen, die Anwendung zu verändern bzw. zu erweitern, muss im idealen Fall nur eine entsprechende Anpassung am Metamodell vorgenommen werden, und mittels erneuter Codegenerierung wird diese ohne weiteren Aufwand in der Implementierung umgesetzt. Im Normalfall, wie etwa bei diesem Editor, geht es aber meist nicht ganz so einfach, da die Implementierung auch handgeschriebenen Code enthält. Die Schnittstellen der generierten und handgeschriebenen Teile des Codes passen nicht unbedingt perfekt zusammen, und je größer der Anteil der letzteren ist, um so größer wird auch der manuelle Aufwand zur Abstimmung dieser Teile mit den Rest der Anwendung sein. Ein Problem, was bei jeder Neugenerierung nach einer Metamodell-Änderung aufs neue auftreten kann. Bei dem vorliegenden Query-Editor sind die Anteile handgeschriebener Codemanipulation aber gering im Verhältnis zum Umfang der Anwendung, so dass auch bei Anpassungen des Metamodells der Aufwand zur Umsetzung verhältnismäßig gering ausfällt. Die Änderungen betreffen primär die Editparts (insbesondere den Notification-Mechanismus) sowie die verschiedenen Parser zur Anpassung der angezeigten Texte. Ein anderes Problem könnte von der Abwärtskompatibilität neuer GMF-Versionen ausgehen. GMF befindet sich derzeit (aktuell Version 2.0) in kontinuierlicher Entwicklung und erfährt gelegentlich größere Änderungen, die mitunter auch Kompatibilitätsprobleme der Modelle und Codeerweiterungen mit sich bringen. Ein weiteres Problem stellt die bislang unzureichende Dokumentation dar. Die Möglichkeiten zur manuellen Anpassung des GMF-Quellcodes sind sehr vielfältig, die Dokumentation dagegen ist für viele dieser Bereiche noch unvollständig. Besonders im Bereich der MDSD, wo große Teile des Frameworks unter der Oberfläche arbeiten, ist eine gute Dokumentation eine wichtige Vorraussetzung für den Einstieg in Technik ist. Vor allem fehlen noch ≪Best Practices≫ für die individuelle Konfiguration der Software, welche den Entwicklern als Leitfaden bei der Anpassung dient. Auch die in dieser Arbeit vorgestellten Modifikationen zeigen selbstverständlich nur jeweils eine von mehreren verschiedenen Möglichkeiten.

8 Transformation und Codegenerierung

Mit dem Editor existiert ein Werkzeug, das es uns ermöglicht, eine Query grafisch zu modellieren. Wirklich gewinnbringend einsetzen lässt sich der Editor aber erst dann, wenn aus dem Modell auch eine Implementierung folgt. Es ist nahe liegend, für das vorhandene Modell den entsprechenden Quellcode automatisiert generieren zu lassen. Im Sinne der modellgetriebene Softwareentwicklung existieren hier mehrere Vorteile: zum einen muss sich der Anwender nicht mit konkreten Implementierungsdetails (wie etwa spezieller Syntax) befassen, zum anderen kann er die vom Modell gebotene Abstraktion nutzen, ohne diese selbst auf Code-Ebene auflösen zu müssen.

8.1 OpenArchitectureWare

Zur Implementierung des Code-Generators wird die Plattform openArchitectureWare (oAW) verwendet. oAW vereint eine Vielzahl von Werkzeugen zur Modell-Transformation. Die Funktionalität der Plattform ist zu umfangreich, als dass sie hier vollständig wiedergegeben werden könnte, daher sollen nur die Kernkomponenten kurz erläutert werden, um einen Überblick zur Funktionsweise zu vermitteln.

oAW bietet mehrere Komponenten, die flexibel miteinander kombiniert werden können, zur Erstellung eines Codegenerators für verschiedenen Modelle. Dabei wird insbesondere das Ecore-Format unterstützt, darüber hinaus aber auch weitere, wie z.B. UML2 oder XML. Die Komponenten ermöglichen es u.a. ein bestehendes Modell zu parsen, zu validieren und zu transformieren.

Die konkrete Einbindung und Konfiguration der Komponenten, die letztlich den Generator bilden, findet in der Definition eines Workflows statt. Der Workflow steuert den gesamten Ablauf und veranlasst die einzelnen Arbeitschritte zur Bearbeitung eines Modells. Die Komponenten erhalten an dieser Stelle die Programmmodule zugewiesen (Modell und Transformationsbeschreibung), welche in dem Prozess verarbeitet werden sollen.

8.1.1 Sprachen

Zur Programmierung in den verschiedenen Aufgabenbereichen kommen in oAW mehrere Sprachen zum Einsatz.

Expression Allen Sprachen zugrunde liegend ist eine funktionale Expression-Sprache, die fundamentale Ausdrücke, Steuerkonstrukte und das Typsystem definiert. Die Sprache wird auf allen Modellstufen (vom Modell bis zum Meta-Metamodell) angewandt. Das Typ-System integriert vordefinierte Basistypen (String, Boolean, Collection etc.) sowie die Typen eines Metamodells. Die Basistypen sind grundsätzlich vorhanden, das jeweilige Metamodell (und seine Implementierung) muss im Workflow registriert werden, um seine Typen ebenfalls verfügbar zu machen.

xPand xPand ist eine Transformationssprache, die für die Umwandlung des Modells in ein textuelles Format, z.B. Implementierungscode, sorgt. Mit xPand können Templates für die diversen Modellelemente definiert werden, sie bilden den inhaltlichen Kern eines Generators. Die generierte Ausgabe kann in neu anzulegende Dateien geschrieben werden. Die Templates können in direkter Weise oder aber auch mithilfe aspektorientierter Techniken an den Generator gebunden werden.

xTend Die Sprache xTend dient als Erweiterungsmechanismus und Schnittstelle für andere Sprachen, sie ermöglicht die Intertype Declaration. Mit xTend können Funktionsbibliotheken angelegt oder zugänglich gemacht werden, mit denen die Funktionalität des Metamodells erweitert werden kann, ohne das Metamodell direkt zu verändern.

Check Zur Validierung wird in oAW die Sprache Check verwendet. Sie ist vergleichbar mit der OCL, allerdings mit dem Unterschied, dass das Typ- und Sprachsystem von oAW integriert wird.

8.1.2 Codegenerierung

Zum besseren Verständnis des Ablaufs der Codegenerierung in oAW soll die grundlegende Funktionsweise eines Generators hier beispielhaft erläutert werden.

Der Generator wird über einen *Workflow* konfiguriert. Im Workflow werden verschiedene Komponenten zusammengestellt, welche die Ein- und Ausgabe übernehmen. Variable Angaben, wie etwa ein Dateipfad, können dabei in eine *properties*-Datei ausgelagert werden, um die variablen und fixen Anteile eines Workflows sauber voneinander getrennt zu halten.

Von einer Query, die wir z.B. im GMF-Editor modelliert haben, existiert eine XML-Datei mit allen relevanten Informationen. Ein xmiParser übernimmt die Aufgabe, diese Datei einzulesen und das Modell zu instanziieren, wobei er auf die zugrunde liegende Implementierung des Metamodells zurückgreifen muss. Für die Query starten wir nun einen Generator, der mit entsprechenden Templates die Modell-Query in eine Query-Implementierung transformieren soll.

```
1 <component id="generator"
2   class="org.openarchitectureware.xpand2.Generator">
3   <metaModel id="mm"
4     class="org.openarchitectureware.type.emf.EmfMetaModel">
5     <metaModelPackage value="de.tuberlin.cs.amertgen.query.QueryPackage" />
6   </metaModel>
7   <expand value="templates::Root::Root FOR model" />
8 </component>
```

Listing 8.1: Die Generator-Komponente im oAW-Workflow. Der Generator erhält das QueryPackage als Metamodell. Das Template Root wird für den Slot «model» aufgerufen, der mit der vom xmiParser erstellten Query-Instanz belegt ist.

Als Einstiegspunkt dient uns hier das Template `Root`. Zur besseren Strukturierung und Kontrolle des Generierungsprozesses sind auch die Templates modularisiert. Vom Template `Root` aus werden weitere Templates aufgerufen, zunächst ein Template zur Generierung einer Query, darin wiederum Templates zur Generierung einzelner Komponenten, wie etwa Klassen, Methoden usw.

```
1 «DEFINE Root FOR query::MyQuery»
2   «FILE getName(this)+".otquery"»
3     // generated at «timestamp()»
4   «EXPAND exQuery(this)»
5   «ENDFILE»
6 «ENDDEFINE»
```

Listing 8.2: Das Root-Template. Eine neue Datei wird angelegt (2), deren erste Zeile mit einem Kommentar zur Generierungszeit beginnt (3). Danach wird das Template zur Generierung des Quellcodes für eine Query aufgerufen (4).

Innerhalb der Templates können wir nun vom xTend-Mechanismus Gebrauch machen. Innerhalb einer ext-Datei kann z.B. auf eine Java-Methode verwiesen werden, die somit auch in einem Template, das diese Extension verwendet, verfügbar wird. Die Erweiterungen müssen dabei grundsätzlich statisch getypt sein. Eine Erweiterung um einfache Getter-Methoden ist übrigens nicht unbedingt notwendig, da der Zugriff auf die Attribute eines Objekts bereits vom oAW-Typsystem ermöglicht wird.

```
1 String getName(MyQuery q) :
2     JAVA de.tuberlin.cs.amertgen.query.generator.util.
3             QueryHelper.getName(de.tuberlin.cs.amertgen.query.MyQuery);
```

Listing 8.3: Eine oAW Extension. Die Methode *getName(MyQuery q)* der Klasse QueryHelper wird im Template verfügbar gemacht.

Zur Validierung wird eine Check-Komponente in den Workflow eingebunden. In einem zugehörigen Programmmodul können dann Constraints für die Validierung des Modells formuliert werden. Sollte die Validierung fehlschlagen, wird der Workflow-Ablauf mit einer entsprechenden Fehlermeldung unterbrochen.

```
1 context MyQueryData ERROR
2 "Query must be named" :
3 name.length > 0;
```

Listing 8.4: Ein oAW-Check-Constraint, äquivalent zur OCL

Ein vollständiger Durchlauf des Workflows generiert schließlich eine Datei, die in diesem Fall den Quellcode für die eingangs geparste Query enthält. Da in den Templates die Formatierung des generierten Outputs nur bedingt beeinflusst werden kann ist ein nachgeschalteter Code-Beautifier unerlässlich. Auch hierzu existiert eine Komponente, die entsprechende Hooks bereitstellt, mit denen selbstdefinierte Routinen zur Code-Formatierung aufgerufen werden können.

8.2 Modell-Transformationen

Für die Umsetzung der Query-Transformationen stellt sich die Frage, wie aus dem Struktur-Diagramm einer Query eine funktional formulierte Implementierung generiert werden kann. Es wird eine allgemein gültige Abbildungsvorschrift benötigt.

8.2.1 Grundkonzept

In einem Query-Diagramm sollen letztlich Elemente anhand ihrer Eigenschaften und Beziehungen selektiert werden. Das korrespondierende Konzept einer funktionalen Sprache ist die Filter-Anweisung.

Betrachten wir die einfache Query `classesEndingWithItem`, in der eine Menge von Klassen anhand eines Namens-Musters selektiert werden soll.

```
<query:MyQuery xmi:version="2.0">
  <classes name="*Item" identifier="C"/>
  <data name="classesEndingWithItem" result="<C?>"/>
</query:MyQuery>
```

Listing 8.5: XML-Modell einer einfachen Query

```
query Set<ClassType> classesEndingWithItem() {
  for(ClassType c_id_C: allClasses[name.endsWith("Item")])
    c_id_C
}
```

Listing 8.6: Implementierung für das Modell

Es ist leicht ersichtlich, wie mithilfe des `for`-Ausdrucks und Angabe von Constraints ein entsprechender Filter konstruiert werden muss. Für eine automatisierte Abbildungsvorschrift gilt es jedoch mehrere wichtige Dinge zu berücksichtigen.

Scope Zunächst einmal ist der Scope eines Elements zu beachten. Auf oberster Ebene starten wir bei einer der vordefinierten Mengen wie `allClasses` oder `allPackages`. Ggf. wird dieser Scope aber auch eingeschränkt, z.B. wenn eine Klasse innerhalb eines Pakets definiert wurde.

Wildcards Selektion, die auf variablen String-Mustern (*) oder Parameter-Anordnungen (..) basiert, muss gesondert behandelt werden. Es müssen entsprechende Funktionen zur Auflösung dieser Muster vorhanden sein.

Eindeutigkeit Bei Queries mit mehreren Elementen, in denen es zu geschachtelten Filter-Anweisungen kommt, müssen wir uns um die eindeutige Zuordnung von Elementen kümmern. Um eine korrekte Referenzierung aller Elemente innerhalb einer Query sicherzustellen muss jedes Element einen eindeutigen Bezeichner erhalten. Dazu verwenden wir eine Methode `getUniqueId(Object o)`, die für jedes Element einen eindeutigen String liefern soll. Im Falle eines vorgegebenen Identifiers können wir diesen bei der Erstellung eines Bezeichners benutzen, andernfalls machen wir Gebrauch von der Objektreferenz selbst, von der wir wissen, dass sie grundsätzlich eindeutig ist.

Die Definition sieht weiterhin vor, dass zwei verschiedene Elemente einer Query niemals zur gleichen Zeit ein und dasselbe Objekt repräsentieren können (Mehrfachbelegung). Aus diesem Grund müssen einmal zugewiesene Variablen von nachfolgend deklarierten Mengen des selben Scopes abgezogen werden (siehe Zeile 7 in Beispiel-Listing 8.8).

8.2.2 Filter-Konstruktion

Wie bereits in Kapitel 6 erläutert gilt für die Elemente einer Query eine implizite AND-Verknüpfung, d.h. alle Elemente müssen zwingend einen positiven Match finden; können die Elemente nur teilweise gematcht werden, so bleibt die Ergebnismenge leer. Die funktionalen Sprachkonzepte eignen sich gut zur Abbildung dieser Eigenschaft. Die Kombination von Filtern findet dabei in geschachtelten for-Ausdrücken statt. Findet sich für ein gefordertes Element keine Repräsentation, so bleibt die Menge dieses Scopes leer und der gesamte Ausdruck liefert damit eine leere Ergebnismenge. Nur dann, wenn für alle for-Ausdrücke entsprechende nicht-leere Mengen selektiert werden, kann überhaupt am Ende ein Element in die Ergebnismenge aufgenommen werden.

Ein Filter wird aus zwei Bestandteilen konstruiert: einer Menge von Elementen (Scope) und einer Menge von Selektionskriterien (Constraints). Bei der Generierung muss allerdings zwischen zwei Arten von Selektionskriterien unterschieden werden, den selbstbezogenen und den auf andere Elemente bezogenen. Selbstbezogene Constraints, wie beispielsweise ein Namens-Muster, beziehen sich allein auf das eigene Element und haben keinerlei Abhängigkeiten zu anderen Elementen. Sie können daher der Einfachheit und Lesbarkeit halber in Form der Pfadsyntax (in eckigen Klammern [...]) an Ort und Stelle der Definition dieses Elements angegeben werden. Bezieht sich ein Constraint allerdings auf andere Elemente, so müssen wir sicherstellen, dass an dieser Stelle der für das fremde Element stehende Bezeichner bereits deklariert wurde, also in einem gültigen Scope steht. Da die Elemente sich dabei auch gegenseitig referenzieren können existiert aber keine eindeutige Reihenfolge, mit der ein Konflikt grundsätzlich vermieden werden kann. Aus diesem Grund werden die Filter-Anweisungen in zwei Teile gespalten. Zuerst werden alle Elemente deklariert und rein strukturell vorgefiltert (for-Anweisungen), danach werden alle auf fremde Referenzen bezogenen Constraints gesondert aufgeführt (if-Anweisungen). So ist sicher gestellt, dass alle Referenzen in einem gültigen Scope angegeben werden.

Template-Darstellung

Die genannten Transformationen werden in Templates implementiert, welche den Generator bilden. Ein solches Template hat (vereinfacht dargestellt) die folgende Struktur.

```
1  «FOREACH classes AS c»
2  «LET getUniqueID(c) AS cvar»
3     ...
4     for(ClassType «cvar»: «scope»«getConstraint(c)»«cremove»)
5     ...
6     «FOREACH c.operations AS m»
7     «LET getUniqueID(m) AS mvar»
8           ...
9           for(Method «mvar»: «cvar».methods«getConstraint(m)»«mremove»)
10          ...
```

```
11      «ENDFOREACH»
12      «ENDLET»
13  «ENDLET»
14  «ENDFOREACH»
```

Listing 8.7: Ausschnitt aus dem Template zur Generierung des strukturellen Filters

Erläuterung:
Zeile 1 - Für jedes Element aus der Menge der Klassen soll der folgende Template-Abschnitt ausgeführt werden.
Zeile 2 - Zum Zwischenspeichern benötigter Werte werden lokale Variablen wie `cvar`, welche die ID der Klasse als String enthält, definiert.
Zeile 4 - Außerhalb der Guillemots stehende Zeichen werden in selbiger Form in die Ausgabe geschrieben, innerhalb der Guillemots wird eine Auswertung vorgenommen, etwa bei den lokalen Variablen `cvar`, `scope` und `cremove`. Gleichermaßen wird die Funktion `getConstraint` aufgerufen, die den als String formulierten Constraint für dieses Element liefert.
Zeilen 6-11 - Analog werden die Kind-Elemente behandelt, in diesem Fall die Methoden.

Generierter Code

Der Generator wendet die Templates nun entsprechend seiner Konfiguration auf Query-Modelle an, um deren Implementierung zu erzeugen. Am Beispiel der Query `intSetterCaller` sehen wir das Ergebnis.

Abbildung 8.1: Eine Query zur Selektion von Aufrufern von int-Setter-Methoden

```
1  // generated at 12.09.2007 13:01:07
2  query Set<Method> intSetterCaller(ClassType c_id_T) {
3    // Structural Filter
4    for(Method m_d6a05e: c_id_T.methods[isPublic && isSetter])
5    for(Parameter par_ba8602: m_d6a05e.arguments[
6          type.class.getName().equals("int")])
7    for(ClassType c_1b3f8f: allClasses - {c_id_T})
8    for(Method m_id_m: c_1b3f8f.methods)
9    // Reference Filter
10   if ((m_d6a05e.arguments.size() == 1) &&
```

```
11          (m_d6a05e.arguments.indexOf(par_ba8602) == 0))
12      if (m_id_m.call(m_d6a05e))
13      // Result
14      m_id_m
15  }
```

Listing 8.8: Generierter Code für die Query intSetterCaller

Erläuterung:
Die Zeilen 4 bis 8 beschreiben den strukturellen Part der Filter, danach, in den Zeilen 10 bis 12, werden die Referenz-abhängigen Teile des Filters angefügt.
Zeile 4 - das Stereotyp ≪setter≫ wird in der Generierung zu einem `isSetter`-Constraint. Die Generierung erfolgt allein per String-Umwandlung, so bleibt das Modell flexibel und es können beliebige Stereotypen verwendet werden. Der Anwender muss allerdings sicherstellen, dass eine entsprechende Query Introduction dieser Signatur existiert oder diese ggf. implementieren.
Zeile 7 - der Abzug der Klasse `c_id_T` aus der Menge allClasses (generiert aus der ≪cremove≫-Anweisung des Templates) verhindert eine Mehrfachbelegung dieses Elements. Es gilt also immer, dass `c_1b3f8f` ungleich `c_id_T` ist.
Zeilen 10 u. 11 - die Parameter der Methode `m_d6a05e` werden geprüft. Da die Parameter u.U. auch von Typ-Referenzen abhängig sein können, wird diese Bedingung erst nach dem strukturellen Teil des Filters aufgeführt.
Zeile 12 - die Abhängigkeitsbeziehung zwischen `m_id_m` und `m_d6a05e` wird ebenfalls erst im Referenz-Teil aufgeführt, damit sicher gestellt ist, dass beide Variablen in einem gültigen Scope stehen.
Zeile 14 - am Ende des Filters steht die Rückgabe. Nur wenn alle Elemente des Durchlaufs einen positiven Match gefunden haben wird das entsprechende Element in die Ergebnis-Menge aufgenommen.

Vollständigkeit und Korrektheit

Ziel der vorgestellten Transformationen ist eine vollständige und korrekte Abbildung eines OTQD in Form einer textuellen Query. Das Ergebnis ist eine Selektion über eine Menge von Elementen, für die Umsetzung wurde daher naheliegenderweise das Konzept des Filters herangezogen. Der Filter wird dabei elementweise Stück für Stück zusammengesetzt. Die modulare Bauweise soll sicher stellen, dass alle Informationen aus dem dem OTQD-Modell in der Query-Implementierung umgesetzt werden. Alle Elemente des OTQD-Metamodells finden eine direkte Abbildung ohne interpretatorische Spielräume in der Querysprache, was eine straight-forward Implementierung der Einzelteile ermöglicht. Die vom Metamodell vorgegebene Baumstruktur stellt sicher, dass alle Elemente eindeutig einem Container zugeordnet sind. Sämtliche Verbindungen sind eindeutig gerichtet, es können keine zyklischen Containment-Beziehungen existieren. Jedes Element wird bei der Traversierung genau einmal berücksichtigt. Da die Querysprache funktional arbeitet, spielt die Reihenfolge der Selektion inhaltlich keine Rolle.

8.2.3 OTQD-Constraints

Eine besondere Herausforderung stellen die OTQD-Constraints NOT, OR und XOR dar. Für sie stehen in der Joinpoint-Querysprache keine direkt korrespondierenden Abbildungen zur Verfügung, die OTQD-Constraints müssen daher gesondert realisiert werden.

NOT

Im Kern lässt sich die Problematik auf den Constraint NOT reduzieren, denn wie wir nachfolgend noch sehen, lassen sich die Query-Constraints OR und XOR durch einfache logische Umformung auf (das implizit vorhandene) AND und NOT zurückführen. Die Aussage eines NOT-Constraint zu erfassen ist allerdings kompliziert. Die bisherigen Transformationen beschreiben einen reinen Positiv-Filter zur Selektion von Mengen diverser Elemente. Die Verknüpfung eines NOT-Constraints mit einer solchen Menge ist dabei keineswegs trivial. Natürlich lässt sich eine Menge nicht einfach negieren, ebenso wenig handelt es sich um eine inverse Menge. Die Umformung des Kriteriums $\neg \exists e \in A \bullet constraint(e)$ führt zu $\forall e \in A \bullet \neg constraint(e)$. Statt des bisherigen Positiv-Filters müssen wir also an dieser Stelle durch Verwendung von Constraints sicher stellen, dass Elemente mit diesen Bedingungen nicht existieren. Die Menge dieser Elemente selbst ist dabei nicht relevant, es ist allein entscheidend, ob die Menge leer ist oder nicht. Zur Umsetzung können wir den in 3.4 vorgestellten ≪?≫-Quantor verwenden.

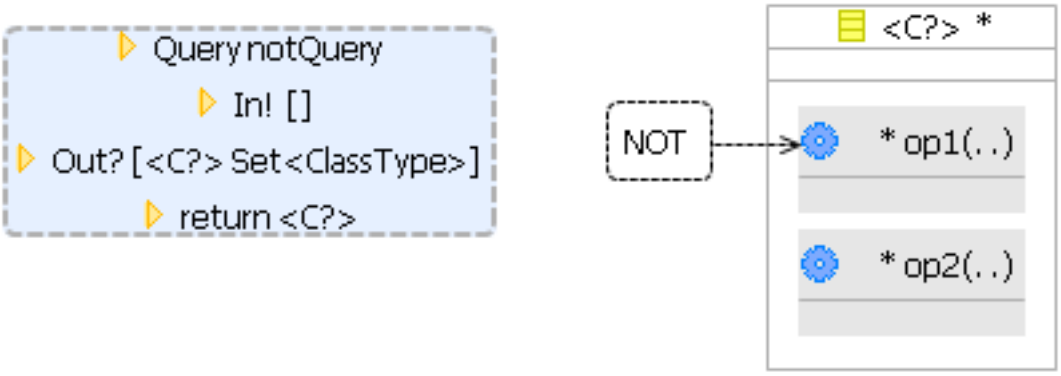

Abbildung 8.2: Query mit NOT-Constraint

```
1 query Set<ClassType> notQuery() {
2   for(ClassType c_id_C: allClasses)
3   if (! c_id_C.methods?[name.equals("op1")])
4   for(Method m_ba8602: c_id_C.methods[name.equals("op2")])
5   c_id_C
6 }
```

Listing 8.9: Implementierung der NOT-Query

Im Falle einzelner Elemente wie etwa einer Methode oder einer Dependency, die als Selektionskriterium für übergeordnete Elemente verstanden werden können, ist die Auswirkung des NOT-Constraints leicht überschaubar. Im Falle von übergeordneten Elementen, die ggf. als Container für weitere Elemente dienen, wie Klassen oder Pakete, kann die Auswirkung des Constraints allerdings schon sehr umfangreiche Dimensionen annehmen. In diesen Fällen ist mitunter bereits die Interpretation des Constraints schwierig und nicht in allen Fällen entsteht eine sinnvolle Aussage.

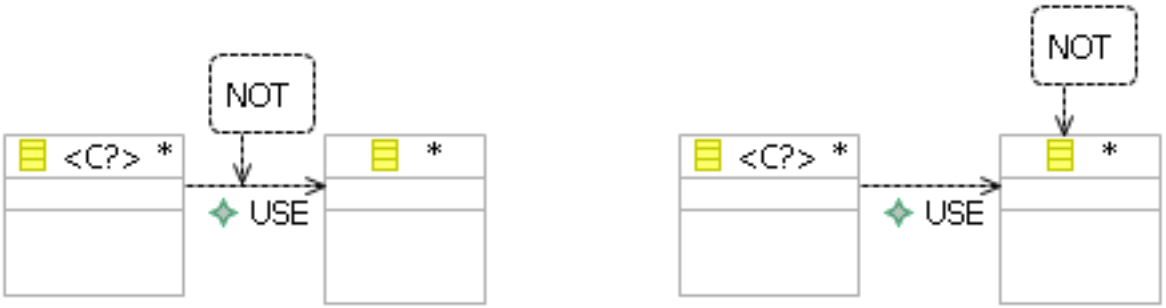

Abbildung 8.3: Ein sinnvoller (links) und ein sinnloser (rechts) NOT-Constraint

Im linken Fall von Abbildung 8.3 drückt das NOT aus, dass eine Klasse keinen Gebrauch von anderen Klassen machen darf. Rechts dagegen ist das NOT widersprüchlich, denn es wird eine Dependency auf eine Klasse verlangt, die nicht existieren darf, was unweigerlich zu einer leeren Ergebnismenge führen muss.

Daraus eine generelle Einschränkung wie etwa «kein NOT auf Klassen» abzuleiten ist aber nicht praktikabel, denn es lassen sich auch sinnvolle Beispiele für solche Constraints finden. Beispielsweise wenn wir alle Pakete selektieren wollen, die keine Klassen mit einem bestimmten Merkmal enthalten (siehe Abbildung 8.4).

Abbildung 8.4: Ein sinnvolles NOT auf eine Klasse

Ein NOT muss also immer im Gesamtkontext beurteilt werden, die Zusammenhänge sind dabei aber zu komplex, um pauschal sinnvolle von nicht sinnvollen Fällen unterscheiden zu können. Der sinnvolle Einsatz des Constraints liegt in der Verantwortung des Anwenders.

OR

Ein OR-Constraint stellt uns vor ein anders geartetes Problem. Letztlich verlangt der Constraint eine Vereinigung von mehreren alternativen Mengen.

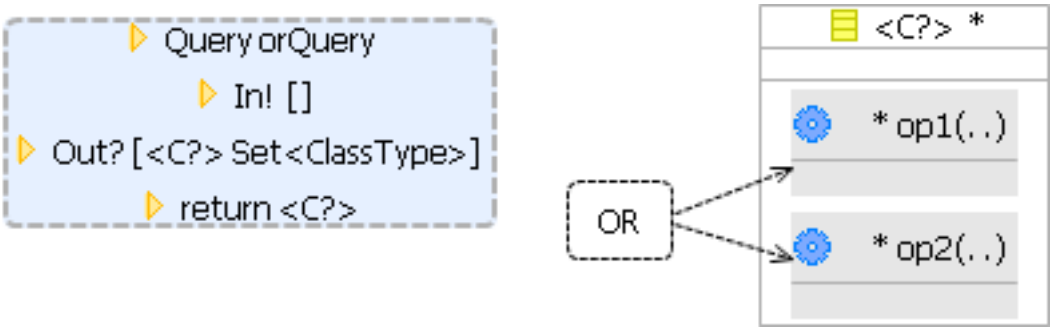

Abbildung 8.5: Query mit OR-Constraint

Um eine korrekte und typsichere Selektionsanweisung zu generieren, sollten die Mengen in der Query selbst strikt getrennt bleiben. Eine Auflösung erzielen wir stattdessen dadurch, dass wir die Query in Subqueries aufspalten, die jeweils eine Komponente der OR-Verknüpfung repräsentieren. Der Hauptquery kommt dann nur noch die Aufgabe zu, die Subqueries aufzurufen und ihre Ergebnismengen zu vereinen. Selbstverständlich könnte das alles auch in einer einzelnen Query zusammengefasst werden (in diesem einfachen Beispiel ließen sich auch direkt die Prädikate verknüpfen). Aus Gründen der Übersichtlichkeit und um dem Anspruch der Allgemeingültigkeit gerecht zu werden wird aber die modularisierte Variante vorgezogen. Alternative Varianten können bei Bedarf im Zuge einer Optimierung eingeführt werden (siehe Abschnitt 8.2.4).

```
1  query Set<ClassType> orQuery() {
2    (orQuery_sub0() | orQuery_sub1())
3  }
4
5  query Set<ClassType> orQuery_sub0() {
6    for(ClassType c_id_C: allClasses)
7    for(Method m_16dadf: c_id_C.methods[name.equals("op1")])
8    c_id_C
9  }
10
11 query Set<ClassType> orQuery_sub1() {
12   for(ClassType c_id_C: allClasses)
13   for(Method m_1e4f7c: c_id_C.methods[name.equals("op2")])
14   c_id_C
15 }
```

Listing 8.10: Zur Realisierung eines OR-Constraint werden die Mengen der Subqueries mittels Mengenoperation vereint

XOR

Mit dem Prinzip der Subqueries und der Realisierung des NOT-Constraints haben wir nun auch die notwendigem Mittel, um einen XOR-Constraint darzustellen. Die Aufspaltung in Subqueries erfolgt dabei in gleicher Weise wie bei einem OR-Constraint, allerdings handelt es sich hier um ein exklusives «entweder oder», weshalb die jeweils übrigen Komponenten in diesem Fall nicht weggelassen, sondern durch Negation explizit ausgeschlossen werden müssen.

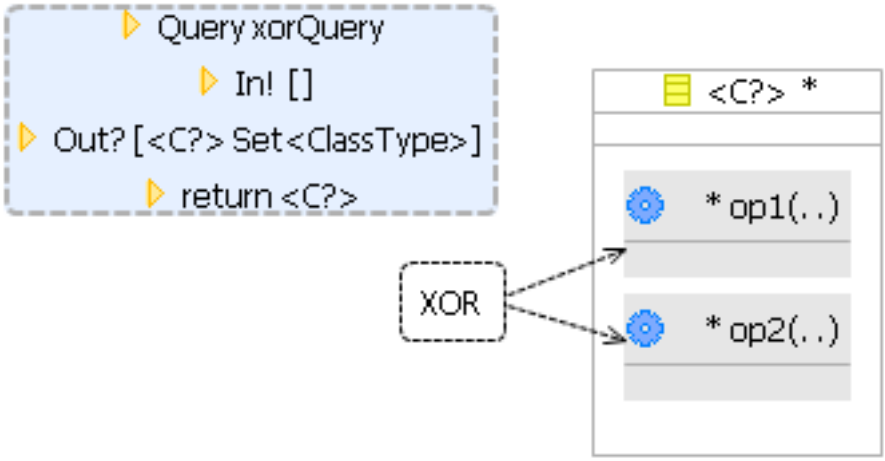

Abbildung 8.6: Ein XOR-Constraint

```
1  query Set<ClassType> xorQuery() {
2    (xorQuery_sub0() & xorQuery_sub1())
3  }
4
5  query Set<ClassType> xorQuery_sub0() {
6    for(ClassType c_id_C: allClasses)
7    for(Method m_5e1077: c_id_C.methods[name.equals("op1")])
8    if (! c_id_C.methods?[name.equals("op2")])
9    c_id_C
10 }
11
12 query Set<ClassType> xorQuery_sub1() {
13   for(ClassType c_id_C: allClasses)
14   if (! c_id_C.methods?[name.equals("op1")])
15   for(Method m_11671b: c_id_C.methods[name.equals("op2")])
16   c_id_C
17 }
```

Listing 8.11: Generierte Query mit XOR-Realisierung durch Aufspaltung in Subqueries

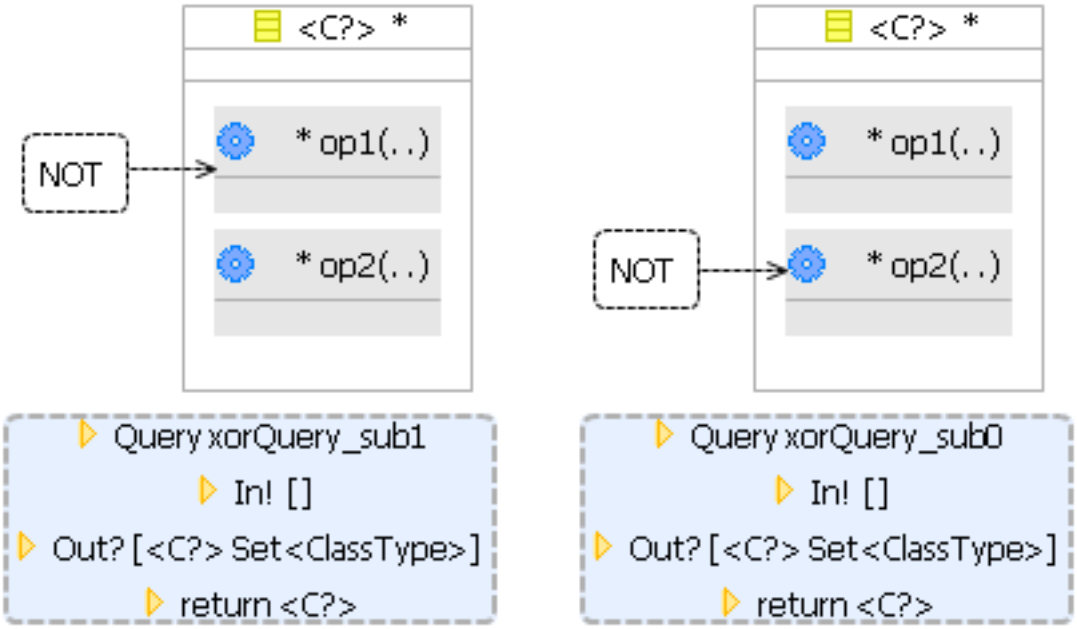

Abbildung 8.7: Die Subqueries des aufgespaltenen XOR-Constraints (die grafische Darstellung dient nur der Veranschaulichung und ist kein Bestandteil des Generierungsprozesses)

8.2.4 Ergebnisqualität

Mit den aufgeführten Transformationen lassen sich prinzipiell korrekte Implementierung für die Queries generieren. Bei automatisch generiertem Code sollte darüber hinaus aber auch Übersichtlichkeit und Performanz berücksichtigt werden.

Zu diesem Zweck stellen wir eine handgeschriebene Implementierung einer generierten gegenüber. Als Beispiel ziehen wir die Query `alienReferencer` heran, deren Auftrag lautet: ≪Finde alle Methoden aus anderen Paketen, die eine bestimmte Klasse benutzen≫ [Her06a].

```
1 query Set<Method> alienReferencer(ClassType c) {
2   for (ClassType c0: allClasses[package != c.package])
3   for (Method m: c0.methods)
4   if (m.use(c))
5   m
6 }
```

Listing 8.12: Handgeschriebene Implementierung von alienReferencer

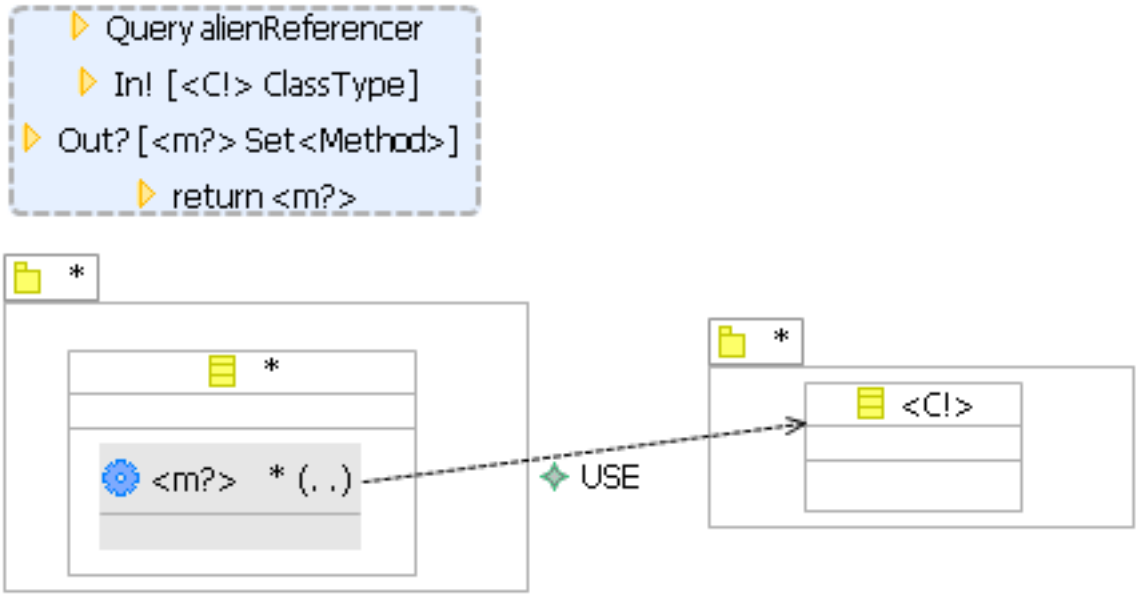

Abbildung 8.8: Query alienReferencer: Finde alle Methoden aus anderen Paketen, die eine bestimmte Klasse benutzen.

```
1 query Set<Method> alienReferencer(ClassType c_id_C) {
2   for(Package p_139b78: allPackages)
3   for(ClassType c_41d05d: p_139b78.classes - {c_id_C})
4   for(Method m_id_m: c_41d05d.methods)
5   for(Package p_1b3f8f: allPackages - {p_139b78})
6   if (m_id_m.use(c_id_C))
7   m_id_m
8 }
```

Listing 8.13: Generierte Implementierung von alienReferencer

Im Vergleich fällt zunächst auf, dass die generierten Bezeichner in Hinsicht auf die Lesbarkeit nur bedingt aussagekräftig und nachvollziehbar sind. Da der generierte Code im besten Falle vom Entwickler aber gar nicht weiter bearbeitet werden muss, sollte dieser Schönheitsfehler vorerst vernachlässigbar sein. Ggf. kann dem durch Einbindung einer Namenstabelle, welche die Bezeichner nachträglich durch verständlichere Namen ersetzt, begegnet werden.

Von grundsätzlicher Relevanz ist dagegen die Optimierung der Filter-Anweisung. Die beiden for-Ausdrücke zur Selektion der Pakete werden in der handgeschriebenen Variante nicht benötigt, die Selektion der Paket-Information findet innerhalb eines einzigen Constraints statt. Es ist daher zu vermuten, dass die Ausführung der handgeschriebenen Implementierung weniger Aufwand verursacht (was letztlich durch die interne Umsetzung der Ausdrücke im Compiler entschieden wird). In Hinblick auf die Performanz können also ggf. günstigere Implementierungen, als die durch den Generator erzeugte, gefunden werden. Sofern sich dieser Nachteil in der Praxis als tatsächlich relevant herausstellt, so könnte dem mit einer Optimierung der Transformationen entgegen gewirkt werden.

Eine Optimierung ist prinzipiell möglich, übersteigt aber in ihrer Komplexität die bisherige Abbildung bei weitem. Die Filter-basierte Abbildung aller Elemente kann vergleichsweise modular angewendet werden. Für jedes Diagramm-Element erhält die Query einen Baustein und wird so Stück für Stück zusammengesetzt, was allerdings in einer hohen Anzahl von ineinander verschachtelten Schleifen resultiert. Bereits bei dieser Filter-Abbildung findet sich Potential zur Optimierung, da ggf. schon die Reihenfolge der geschachtelten Anweisungen den Aufwand beeinflusst. Bei einer Optimierung über die modularen Filter hinaus lautet das Ziel, die einzelnen Bausteine des Filters zu reduzieren. Dazu müssen die Diagramm-Elemente allerdings auch in Kombination betrachtet werden. In Abbildung 8.8 zu `alienReferencer` erkennen wir beispielsweise erst aus dem Zusammenspiel der beiden Pakete, dass sie selbst für unser Ergebnis gar nicht von Bedeutung sind, sondern allein zur Unterscheidung eines beliebigen Pakets vom Paket der Klasse `<C!>` verwendet werden, wie auch in der Handimplementierung ersichtlich ist. Ein Umstand, der allerdings von vielen Faktoren abhängig ist; sobald etwa eines der Pakete ein zusätzliches Selektionskriterium besäße, wäre die Optimierung in dieser Form nicht mehr möglich. Die Fülle der kombinatorischen Möglichkeiten beim Zusammenspiel der Elemente dürfte eine vollständig optimierte Transformationsvorschrift auf unüberschaubaren Umfang anwachsen lassen. In der Praxis sollte daher vorerst geprüft werden, in welchen Bereichen eine Optimierung tatsächlich notwendig erscheint, um diese dann selektiv zu ergänzen.

8.2.5 Erweiterbarkeit des Generators

Wie für den GMF-Editor muss auch für den Generator die Frage nach der Anpassungsfähigkeit und Erweiterbarkeit der Transformationen gestellt werden. Teils sind diese bereits durch die flexible Grundstruktur des Generators gewährleistet, da, wie z.B. bei der Angabe von Stereotypen, einfache und zugleich allgemein gültige Abbildungsvorschriften existieren, für die zunächst keine Restriktionen vorgegeben sind, und es daher allein dem Anwender überlassen ist, diese Elemente sinnvoll einzusetzen. Änderungen am Metamodell sollten sich ohne besonderen Aufwand in die bestehende Generator-Struktur integrieren lassen. Das Metamodell wird selbst in den Generator eingebunden und ist dort direkt verfügbar. Bei Erweiterungen des Metamodells müssen entsprechend neue Template-Module erstellt und in die bestehende Struktur eingebunden werden, was durch den modularen Aufbau vergleichsweise einfach zu bewerkstelligen ist. Die Konfiguration der Generatorkomponenten kann dabei ohne Änderung übernommen werden bleiben.

9 Fazit und Ausblick

9.1 Fazit

In dieser Arbeit wurde das Konzept und die Modellierung der Querysprache für Object Teams untersucht. Konzeptionell hat sich gezeigt, dass die Integration der Querysprache in Object Teams noch nicht vollständig ausgereift ist. Anhand eines Beispiels im Vergleich zu AspectJ wurde erläutert, an welchen Stellen Probleme zu Tage treten. Speziell im Bereich der Bindung besteht noch Bedarf für Erweiterungen, damit die Queries ihrem Aufgabengebiet gerecht werden können. Daneben wurde eine Klassifizierung für Methoden und deren Integration vorgestellt, welche hilfreich sein kann für die Nutzung grundlegender semantischer Informationen, die sich aus der Programmstruktur ergeben.

Für die Modellierung von Queries wurde auf Basis der Joinpoint Designation Diagrams eine grafische Notation entwickelt. Die sogenannten Object Teams Query Diagrams sollen Object Teams-Entwicklern ein leicht verständliches und hilfreiches Mittel bei der Formulierung von quantifizierenden Aussagen an die Hand geben. Das Ziel, Programmeinheiten anhand verschiedener Kriterien selektieren zu können, wurde auf Ebene der üblichen Module wie Pakete, Klassen und Methoden erreicht. Die Abstraktion mithilfe grafischer Mittel bietet ein intuitives Bild der wesentlichen Kriterien. Der Abstraktionsgrad ist dabei aber differenziert genug, dass eine eindeutige Abbildung der Definition in die Implementierung möglich ist.

Durch die Implementierung eines entsprechenden Editor-Werkzeuges, samt Generator zur Modelltransformation, ist auch der Grundstein für eine Verwendung der Object Teams Query Diagrams in der Praxis gelegt. Die Werkzeuge sollen einen modellgetriebenen Entwicklungsprozess von Queries ermöglich. Der Editor zur Query-Modellierung wurde mithilfe des Eclipse Graphical Modelling Frameworks gebaut, der Generator für die automatisierte Codegenerierung aus diesen Modellen wurde mithilfe von openArchitectureWare erstellt. Die Nutzung dieser Werkzeuge des Model Driven Software Development soll dabei die Vorteile eines modellgetriebenen Entwicklungsprozesses aufzeigen. Das Prinzip, dass sich Programmänderungen direkt am Modell realisieren lassen und dank automatisierter Transformation in das bestehende System eingebettet werden, kann zu einer deutlichen Reduzierung des Entwicklungsaufwandes führen. Im Falle des Editors konnten etwa viele der vorgefertigten Komponenten des Frameworks (GUI-Anwendung, Serialisierung, Persistenz etc.) übernommen werden, womit der Großteil der Zeit blieb, die Arbeit auf die wesentlichen Bereiche der Kernanwendung zu konzentrieren. Allerdings sind dieser Vorgehensweise auch Grenzen gesetzt, was z.B. in der Notwendigkeit manueller Anpassungen zu Tage tritt. Dies führt auch dazu, dass der Entwickler sich letztlich doch mit den komplexen Details der Implementierung auseinandersetzen muss. Die Vision ≪einfach in das

MDSD-Auto einsteigen und losfahren» funktioniert in diesem Fall leider noch nicht - noch muss der Entwickler unter die Motorhaube schauen und dort herumschrauben.

9.2 Ausblick

Die OT/J-Querysprache existiert bisher nur prototypisch. Im Konzept bestehen noch einige offene Fragen. Das Rollen/Teams-Konzept in Object Teams sieht vor, dass jede Rolle fest an eine bestimmte Basisklasse gebunden ist. Wie gezeigt soll die Quantifizierung es aber auch ermöglichen, Joinpoints klassenübergreifend zusammenzufassen, womit diese Bindung nicht mehr eindeutig ist. In diesem Zusammenhang lässt sich fragen, ob das Rollen/Teams-Konzept im Fall einer Aufweichung der Bindung überhaupt sinnvoll ist, oder ob es hier um ein einfacheres Konzept ergänzt werden sollte. Nicht bei allen Aspekten erscheint die Modularisierung in Rollen und Teams notwendig.

Im Sinne praxisorientierter Anwendung der OTQDs muss geprüft werden, ob sie im Entwicklunsbetrieb wirklich Bestand haben können. Insbesondere kann die Frage gestellt werden, inwieweit aspektorientierte Programme tatsächlich sinnvollen Gebrauch von komplexen Queries machen, welche den Einsatz spezieller Werkzeuge rechtfertigen, oder ob die angeführten Anwendungsszenarien primär akademischer Natur sind. Die prinzipielle Relevanz eines solchen Modells und seiner Anwendung muss sich letztlich an praxisnahen Anforderungen messen lassen.

Literaturverzeichnis

[CB05] Clarke, Siobhan und Elisa Baniassad: *Aspect-Oriented Analysis and Design. The Theme Approach.* Object Technology Series. Addison-Wesley, Boston, USA, 2005.

[CRS+05] Chitchyan, Ruzanna, Awais Rashid, Pete Sawyer, Alessandro Garcia, Mónica Pinto Alarcon, Jethro Bakker, Bedir Tekinerdogan, Siobhán Clarke und Andrew Jackson: *Report synthesizing state-of-the-art in aspect-oriented requirements engineering, architectures and design.* Technischer Bericht AOSD-Europe Deliverable D11, AOSD-Europe-ULANC-9, Lancaster University, Mai 2005.

[DCM06] Dragan, Natalia, Michael L. Collard und Jonathan I. Maletic: *Reverse Engineering Method Stereotypes.* In: *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM'06)*, 2006. http://www.cs.kent.edu/ jmaletic/papers/ICSM06.pdf.

[DG07] Dvorak, Radomil und Richard Gronback: *GMF Constraints.* `http://wiki.eclipse.org/index.php/GMF_Constraints`, 2007.

[EF07a] Eclipse-Foundation: *Webseite von EMF.* URL: `http://www.eclipse.org/modeling/emf/`, 2007.

[EF07b] Eclipse-Foundation: *Webseite von GEF.* URL: `http://www.eclipse.org/gef/`, 2007.

[EF07c] Eclipse-Foundation: *Webseite von GMF.* URL: `http://www.eclipse.org/gmf/`, 2007.

[FF00] Filman, R. und D. Friedman: *Aspect-Oriented Programming is Quantification and Obliviousness*, 2000.

[FS07a] Fuentes, Lidia und Pablo Sánchez: *Designing and Weaving Aspect-Oriented Executable UML models.* Journal of Object Technology, August 2007.

[FS07b] Fuentes, Lidia und Pablo Sánchez: *Towards Executable Aspect-Oriented UML Models.* In: *Proceedings of AOM @ AOSD07 10th Int'l Workshop on Aspect-*

Oriented Modeling, März 2007.
http://www.aspect-modeling.org/aosd07/accepted_papers/a5-fuentes.pdf.

[Her06a] Hermann, Stephan: *Innovation by Not-Inventing.* Vortrag, Technische Universität Berlin, 2006.

[Her06b] Herrmann, Stephan: *Are pointcuts a first-class language feature?* In: *FOAL'06 Workshop (Foundation of Aspect-Oriented Languages), AOSD 2006*, 2006.
http://objectteams.org/publications/FOAL06.pdf.

[Her07] Herrmann, Stephan: *Webseite von Object Teams.* URL: `http://www.objectteams.org/`, 2007.

[HH07] Herrmann, Stephan und Christine Hundt: *Object Teams/Java - Language Definition v1.0*, 2007.

[HHM04] Herrmann, Stephan, Christine Hundt und Katharina Mehner: *Translation Polymorphism in Object Teams.* Technischer Bericht, Technische Universität Berlin, 2004.
http://www.objectteams.org/publications/TR2004-05.ps.

[HHMW05] Herrmann, Stephan, Christine Hundt, Katharina Mehner und Jan Wloka: *Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation.* In: *Dynamic Aspects Workshop'05, AOSD 2005*, 2005.
http://aosd.net/2005/workshops/daw/HerrmannHundtMehnerWloka.pdf.

[JC06] Jackson, Andrew und Siobhán Clarke: *Towards the Integration of Theme/UML and JPDDs.* In: *Aspect Oriented Modelling workshop, at AOSD 06*, Bonn, Germany, 2006.

[KGBM06] Kellens, Andy, Kris Gybels, Johan Brichau und Kim Mens: *A Model-driven Pointcut Language for More Robust Pointcuts.* In: *Proceedings of SPLAT'06 Workshop*, 2006.
http://www.info.ucl.ac.be/~km/MyResearchPages/publications/workshop_paper/WP_2006_AOSD_SPLAT.pdf.

[KM05] Kiczales, Gregor und Mira Mezini: *Separation of Concerns with Procedures, Annotations, Advice and Pointcuts.* In: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'05).* Springer LNCS, Juli 2005.
http://www.infosun.fmi.uni-passau.de/st/papers/EIWAS04/stoerzer04eiwas.pdf.

[KS04] Koppen, Christian und Maximilian Stoerzer: *Pcdiff: Attacking the fragile pointcut problem.* In: *First European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.
http://www.infosun.fmi.uni-passau.de/st/papers/EIWAS04/stoerzer04eiwas.pdf.

[LRC+06] Loughran, Neil, Awais Rashid, Ruzanna Chitchyan, Nicholas Leidenfrost, Johan Fabry, Nelio Cacho, Alessandro Garcia, Frans Sanen, Eddy Truyen, Bart De Win, Wouter Joosen, Nelis Boucké, Tom Holvoet, Andrew Jackson, Andronikos Nedos, Neil Hatton, Jenny Munnelly, Serena Fritsch, Siobhán Clarke, Mercedes Amor, Lidia Fuentes, Monica Pinto und Carlos Canal: *A domain analysis of key concerns - known and new candidates.* Technischer Bericht AOSD-Europe Deliverable D43, AOSD-Europe-KUL-6, Katholieke Universiteit Leuven, Februar 2006.

[Mos03] Mosconi, Marco: *Modularisierung und Adaptierung von Komponenteninteraktionen mit Object Teams und dem CORBA Komponentenmodell.* Diplomarbeit, Technische Universität Berlin, 2003.
http://www.objectteams.org/publications/Diplom_Marco_Mosconi.pdf.

[Sel06] Sellin, Timo: *Effiziente Laufzeit-Transformation für die aspektorientierte Programmiersprache ObjectTeams/Java.* Diplomarbeit, Technische Universität Berlin, 2006.
http://www.objectteams.org/publications/Diplom_Timo_Sellin.pdf.

[SHU04a] Stein, Dominik, Stefan Hanenberg und Rainer Unland: *A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models.* In: *Proc. of MDA-FA '04*, Seiten 77–92. Springer LNCS 3599, Oktober 2004.
http://www.springerlink.com/index/39D62RQ0MBAR2X9W.pdf.

[SHU04b] Stein, Dominik, Stefan Hanenberg und Rainer Unland: *Query Models.* In: *Proc. of UML '04*, Seiten 98–112. Springer LNCS 3273, Oktober 2004.
http://dawis.icb.uni-due.de/fileadmin/dawis_template/main/resources/publications/aosd/2004_SHU_QueryModels_UML.pdf.

[SHU07] Stein, Dominik, Stefan Hanenberg und Rainer Unland: *From Aspect-Oriented Design To Aspect-Oriented Programs: Tool-Supported Translation of JPDDs into Code.* In: *Proc. of AOSD '07*, März 2007.
http://dawis.icb.uni-due.de/resources/publications/aosd/2007_SHU_ToolSupportedTranslationOfJPDDs_AOSD.pdf.

[SS05] Schmauder, Ralf und Philipp Schill: *Codegenerierung mit dem Eclipse Modeling Framework und Jet.* ObjektSpektrum, Januar 2005.

[vdBCC05] Berg, Klaas van den, Jose Maria Conejero und Ruzanna Chitchyan: *AOSD Ontology 1.0 - Public Ontology of Aspect-Orientation.* AOSD-Europe-UT-01 D9, AOSD-Europe, Mai 2005.

[W3C06] W3C: *Object Constraint Language Specification, Version 2.0.* `http://www.omg.org/cgi-bin/doc?formal/06-05-01`, Mai 2006.

[W3C07] W3C: *XQuery 1.0: An XML Query Language*. `http://www.w3.org/TR/2007/REC-xquery-20070123/`, Januar 2007.

[Win05] Winkelmann, Jessica: *Spezifikation von Visual OCL: Eine Visualisierung der Object Constraint Language*. Diplomarbeit, Technische Universität Berlin, 2005. http://iv.tu-berlin.de/TechnBerichte/2005/2005-4.pdf.

Abkürzungsverzeichnis

AOEM	Aspect-Oriented Executable Modeling
AOP	Aspektorientierte Programmierung
AOSD	Aspect-Oriented Software Development
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
JPDD	Join Point Designation Diagram
MDSD	Model Driven Software Development
oAW	openArchitectureWare
OCL	Object Constraint Language
OOP	Objektorientierte Programmierung
OT/J	Object Teams/Java
OTDT	Object Teams Development Tooling
OTQD	Object Teams Query Diagram
UML	Unified Modeling Language
XML	Extensible Markup Language