Effiziente Laufzeit-Transformation für die aspektorientierte Programmiersprache ObjectTeams/Java

Diplomarbeit im Fachbereich Softwaretechnik

der Technischen Universität Berlin

Timo Sellin

Berlin, den 28. September 2006

Inhaltsverzeichnis

Kapitel 1

Einleitung

Das Hauptziel der Softwaretechnik ist es, die Qualität von Software zu verbessern, Entwicklungskosten zu reduzieren sowie Wartbar- und Erweiterbarkeit zu ermöglichen. Dazu sucht die Softwaretechnik nach Methoden, um die Komplexität von Software zu verringern und die Verständlichkeit zu erhöhen. Eine gute Modularisierung der Software ist ganz entscheidend, um diese Ziele zu erreichen.

Die meisten, der bis heute geschaffenen Programmiersprachen, unterstützen Modularisierung nur entlang einer Dimension. Für viele Probleme ist dies nicht ausreichend. Aspektorientierte Programmierung greift dieses Problem auf und unterstützt die Modularisierung von Software, entlang von zwei Dimensionen. Die eine Dimension ist die aus der objektorientierung bekannte hierarchische Dekomposition mit Klassen. Die andere Dimension, ist die Dekomposition mit Hilfe sogenannter Aspekte. Aspekte definieren Punkte – sogenannte *Joinpoints* – an denen ein objektorientiertes Programm erweitert werden soll und definieren, womit es erweitert werden soll – nennen wir es Aspektcode. Der Aspektcode wird dann automatisch, an den entsprechenden Joinpoints eingefügt. Diesen Vorgang nennt man *weben*.

Auch *ObjectTeams/Java* ist eine aspektorientierte Programmiersprache. Die Joinpoints, die *ObjectTeams/Java* unterstützt, sind die Punkte, vor und nach der Ausführung einer Methode. Obwohl dies für viele Anwendungszwecke ausreichend ist, wünscht man sich weitere Joinpoints für *ObjectTeams/Java*. Ziel dieser Diplomarbeit ist es, zu untersuchen, welche weiteren Joinpoints für *ObjectTeams/Java* in Frage kommen und wie ihre Benutzung für *ObjectTeams/Java* realisiert werden kann.

Die Arbeit ist dazu wie folgt aufgebaut: In Kapitel 2 wird zunächst Aspektorientierung

und *ObjectTeams/Java* genauer erklärt. Im Anschluß in Kapitel 3, folgt ein Einblick in die Infrastruktur der Programmiersprache *Java*, weil die weiteren Kapitel Verständnis darüber erfordern. Dann soll in Kapitel 4 geklärt werden, welche Joinpoints es in Java-Anwendungen gibt. Direkt im Anschluß in Kapitel 5 wird eine Querysprache vorgestellt, die es ermöglich, die in Kapitel 4 identifizierten Joinpoints, programmatisch zu beschreiben, um sie geeignet weiterverarbeiten zu können. In Kapitel 6 werden Muster zum Beschreiben von Joinpoints, vorgestellt, die vom Compiler erstellt werden sollen, um zur Laufzeit, die vom Compiler gefundenen Joinpoints, wiederfinden zu können. Diese Muster dienen also als Schnittstelle zwischen Compiler und Laufzeitumgebung. Desweiteren wird beschrieben, wie diese Muster ausgewertet werden, um den durch das Muster beschriebenen Joinpoint zu finden, ohne jedoch dabei auf die konkrete Implementierung einzugehen.

Die den Kapiteln 7-8 wird auf die Implementierung, der in Kapitel 6 vorgestellten Konzepte, eingegangen. Dazu wird zunächst in Kapitel 7 beschrieben, wie die jetztige *ObjectTeams*-Laufzeitumgebung arbeitet und dann in Kapitel 8, wie die bestehende Laufzeitumgebung erweitert werden kann, um die musterbasierte Suche zu implementieren.

Danach folgt mit den Kapiteln 9-11 ein weiterer Themenkomplex, der erarbeiten soll, wie an die gefundenen Joinpoints, Aspektcode gewoben werden kann. Dazu werden in Kapitel 9 zuerst die Grundlagen gelegt, indem beschrieben wird, wie es möglich ist, den Kontrollfluss in einen Aspekt umzuleiten. Kapitel 10 nennt mögliche Probleme, die dabei auftreten und gibt Lösungsvorschläge. In Kapitel 11 folgt dann die Darstellung, wie die in Kapitel 9-10 beschriebenen Verfahren in der *ObjectTeams*-Laufzeitumgebung implementiert werden können.

Kapitel 12 ist ein *Fazit*, das eine Zusammenfassung gibt, beschreibt, was noch nicht funktioniert und was noch zu tun ist. Anhang A listet alle Joinpoints die in Rahmen dieser Arbeit eine Rolle spielen auf und beschreibt wichtige Eigenschaften zu den Themen *Matching und Weaving*. Wie der im Rahmen dieser Diplomarbeit entstandene Prototyp erweitert werden kann, um ihn zu vervollständigen, wird in Anhang B beschrieben.

Kapitel 2

Aspektorientierte Programmierung mit OT/J

Das Hauptziel der Softwaretechnik ist es, die Qualität von Software zu verbessern, Entwicklungskosten zu reduzieren, sowie Wartbar- und Erweiterbarkeit zu ermöglichen. Dazu sucht die Softwaretechnik nach Methoden, um die Komplexität von Software zu verringern und die Verständlichkeit zu erhöhen. Eine gute Modularisierung der Software ist ganz entscheidend, um diese Ziele zu erreichen [7, 6, 17].

Separation of Concerns. Statt Modularisierung, spricht man heute auch allgemeiner von *Separation of Concerns*. Unter einem *Concern* versteht man ein „*Konzept, Ziel*“ [8], „*Ding von Interesse*“ [4], Feature, Anforderung, Anliegen oder eine Zuständigkeit. Im Folgenden werden nur die Begriffe *Anliegen* und *Concern* verwendet. *Separation of Concerns* bezeichnet also, die Trennung der Anliegen. Die *Concerns* sollen räumlich voneinander getrennt realisiert werden; in sogenannten Moduln. Modul meint in diesem Zusammenhang, kein Konstrukt einer bestimmten Programmiersprache, es meint das allgemeine Konzept, Software in Teile – eben Moduln – zu zerlegen. Ein Modul soll *genau ein Concern* realisieren. Wenn es eine eineindeutige Relation zwischen Concern und Modul gibt, dann wird dadurch Nachvollziehbarkeit (*Tracability*), logischer Zusammenhang (*Cohesion*), Verständlichkeit, Wiederverwendbarkeit und niedrige Kopplung (*low coupling*) ermöglicht.

Die meisten Programmiersprachen bieten deshalb Möglichkeiten zur Modularisierung von Software an. Beispielsweise kann man in *Java* die Software in Klassenhierarchien unterteilen und in *Pascal* in *Units*. Sie stellen dafür Mechanismen zur Dekomposition – um Software in Teile zu untergliedern – und Komposition – um diese Teile wieder zusammenzusetzten – zur Verfügung. Software zur modularisieren ist für einen Softwareentwickler nicht einfach, denn es gibt nicht nur eine Möglichkeit Software zu modularisieren. Der Softwareentwickler

trifft die Entscheidung, wie er dies tut. Die Effektivität der Modularisierung hängt dabei ganz entscheidend davon ab, wie gut er diese Entscheidung trifft bzw. welches Kriterium er verwendet, um die Entscheidung zu treffen. [7, 6]

Crosscutting Concerns. Doch die Qualität der Modularisierung hängt nicht nur vom Softwareentwickler ab. Es gibt *Concerns*, die in vielen heutigen Programmiersprachen nicht getrennt werden können, weil die Dekompositionsmechanismen dieser Programmiersprachen es nicht erlauben. Diese *Concerns* heißen *Crosscutting Concerns*. Wenn man in solchen Programmiersprachen ein *Crosscutting Concern* implementiert, verteilt sich der Code des *Concerns* auf unterschiedliche Moduln – scattering – oder steht in Moduln, die ein anderes *Concern* implementieren sollen – tangling. Das liegt daran, dass die Dekomposition in diesen Programmiersprachen nur eine Dimension der Zerlegung kennt. Auch objektorientierte Programmiersprachen haben mit ihrer hierarchischen Modularisierung nicht die Möglichkeit geschaffen, alle *Concerns* modularisieren zu können. In [17] wird von der „*Tyrannei der dominanten Dekomposition*“ gesprochen. Typische *Crosscutting Concerns*, sind Aufgaben wie Logging und Persistenz. Der Logging- und Persistenzcode verstreut sich im ganzen System und verursacht *scattering* und *tangling*. Außerdem erhöht er die Kopplung zwischen Moduln und verringert die *Cohesion* eines Moduls. Es gibt deshalb viele Ansätze, die versuchen, alle Anliegen voneinander trennen zu können. [17]

2.1 Aspektorientierte Programmierung

Aspekte. Aspektorientierte Programmierung (AOP) ist ein Konzept, um *Crosscutting Concerns* modular implementieren zu können. AOP bietet eine Dimension mehr, als dies bei herkömmlichen Programmiersprachen der Fall ist. Viele aspektorientierte Programmiersprachen bauen auf objektorientierten Programmiersprachen auf und unterstützen deshalb zum einen die bekannte hierarchische Dekomposition mit Klassen und zu anderen die Dekomposition mit Hilfe sogenannter Aspekte. Ein *Crosscutting Concern* wird in einem Aspekt implementiert, das heißt der Aspekt enthält den Code (Aspektcode), der sonst querschneiden (*to crosscut*) würde und modularisiert ihn damit.

Joinpoints. Ein so zerteiltes Programm muss auch wieder zusammengesetzt werden können. Um die Komposition zu ermöglichen, definiert ein Aspekt bestimmte Punkte im Kontrollfluss des Programms – sogenannte *Joinpoints* – an denen der Aspektcode ausgeführt werden soll. Der Aspektcode wird dann automatisch an den entsprechenden Joinpoints eingefügt und kann dann ausgeführt werden. Diesen Vorgang nennt man *weben*.

Weben. Weben kann in aspektorientierten Programmiersprachen auf verschiedene Weise realisiert werden. Es ist möglich, den Aspektcode in den Quelltext der Klasse zu weben, bevor diese übersetzt wird. Das hat jedoch den großen Nachteil, dass man den Quelltext der Klassen benötigt. Das ist beispielsweise bei gekauften Bibliotheken, nicht der Fall. Deshalb ist es wünschenswert, das Weben so zu realisieren, dass der Aspektcode auch in bereits compilierte Programme gewoben werden kann.

Begriffsklärungen. In der Literatur über AOP findet man unterschiedliche Begriffe, die teilweise unterschiedliche Konzepte bezeichnen. An dieser Stelle soll erklärt werden, wie diese Begriffe im Rahmen dieser Diplomarbeit verstanden werden und wie sie sich zu anderen Konzepten abgrenzen.

Joinpoint
: In *AspectJ* und zahlreichen anderen Publikationen werden Joinpoints als Punkte in der Ausführung eines Programms betrachtet. Gefunden werden sie durch einen Pointcut-Bezeichner, der zur Laufzeit ausgewertet wird [8]. Es entscheidet sich also erst zur Laufzeit, ob eine Stelle im Programm ein Joinpoint ist. In dieser Arbeit sollen Joinpoints anders verstanden werden, nämlich als Punkte in der statischen Struktur des Programms [19].

Joinpoint-Shadow
: Die Arbeiten, die Joinpoints als Punkte in der Ausführung eines Programms betrachten, verwenden zusätzlich häufig den Begriff Joinpoint-Shadow. Er bezeichnet das statische Vorkommen eines Joinpoints, also genau das Konzept, das in dieser Arbeit als *Joinpoint* bezeichnet wird.

Joinpoint-Queries
: Werden benutzt, um eine Menge von Joinpoints zu definieren. Es sind Queries – Abfrageausdrücke – über dem abstrakten Syntaxbaum eines Programms.

Basisprogramm
: Alle Bestandteile des Systems, die nicht zum Aspektteil gehören und durch Aspekte erweitert werden können.

Pointcut
: *AspectJ* versteht Pointcuts als „*[...] eine Menge von Joinpoints, plus, optional, einige der Werte im Ausführungskontext dieser Joinpoints.*“ [8][sic]. Im Rahmen dieser Arbeit wird der Begriff *Pointcut* als nicht notwendig erachtet. Die Notwendigkeit von Pointcuts als eigenständiges Konzepts, wird in [19] leicht nachvollziehbar widerlegt. Es ist nicht notwendig Pointcuts als Menge von Joinpoints zubetrachten, weil dies durch

das Konzept von Queries vollständig abgedeckt wird. Die dynamischen Eigenschaften von Pointcuts lassen sich, wie gezeigt wird, als normale Klassen realisieren.

Bindung
Bezeichnet das Konzept, um Aspektmethoden an das Basisprogramm zu binden. Man findet viele Bezeichungen dafür: Advice in AspectJ, Joinpoint- bzw. Method-Interception in [19] und *Method Binding* in *ObjectTeams/Java*. In dieser Arbeit wird von Joinpoint- bzw. Methoden-Bindungen die Rede sein.

2.2 ObjectTeams/Java

Das Konzepte von *ObjectTeams* sollen mit unterschiedlichen Programmiersprachen benutzbar sein. Die Anwendung mit *Java* ist jedoch der De-facto-Standard. Im Folgenden werden deshalb die Begriffe *ObjectTeams* und *ObjectTeams/Java* synonym verwendet.

In diesem Kapitel werden nur die *ObjectTeams/Java*-Konzepte erklärt, die für diese Arbeit relevant sind. Eine weitergehende Einführung bzw. eine genaue Sprachdefinition, kann man in [18, 20] finden.

Rolle. *ObjectTeams/Java* (OT/J) ist eine aspektorientierte Programmiersprache und kennt daher Aspekt – in OT/J heißen sie Rollen. Eine Rolle in OT/J wird immer an genau eine Basisklasse gebunden. Die Basisklasse wird durch das Schlüsselwort `playedBy`, im Kopf der Rolle, in der Form „`class` *Rollenklasse* `playedBy` *Basisklasse*“, definiert:

```
1 class MyRole playedBy MyBase {
2         ...
3 }
```

Es gibt zwei Möglichkeiten, wie Rollen eine Basisklasse adaptieren können – *Callout*- und *Callin*-Bindung.

Callout-Bindung. Bei der Callout-Bindung verwendet die Rolle Funktionen der Basisklasse wieder, indem sie Methoden definiert, die einen Aufruf an die Basisklasse delegieren. Definiert wird eine Callout-Bindung nach dem Schema „*Rollenmethode* `->` *Basismethode*`;`“:

```
1 class MyRole playedBy MyBase {
2         ...
3         roleMethod -> baseMethod;
4         ...
5 }
```

Callin-Bindung. Während Callout-Bindungen die Wiederverwendung von Code der Basisklasse ermöglichen, sind Callin-Bindungen der Mechanismus von OT/J, um Aspektorientierung zu ermöglichen. Die Aufrufrichtung verläuft bei einem Callin entgegengesetzt zum Callout. Die Basis ruft die Rollenmethode auf. Allerdings steht der Aufruf der Rollenmethode nicht im Sourcecode der Basisklasse. Der Aufruf wird nachträglich in die Basisklasse hineingewoben. Wo und wie der Aufruf hineingewoben wird, hängt auch von der Art der Callin-Bindung ab. Es gibt drei Arten: `before`, `after` und `replace`. Eine Callin-Bindung erfolgt nach dem Schema „*role_method_designator* `<-` *callin_modifier base_method_designator*`;`“ und sieht folgendermaßen aus:

```
1 class MyRole playedBy MyBase {
2         ...
3         roleMethod1 <- before baseMethod1;
4         roleMethod2 <- after baseMethod2;
5         roleMethod3 <- replace baseMethod3;
6         ...
7 }
```

role_method_designator
: Der Name einer Rollenmethode. Sollte der Name alleine nicht eindeutig sein, beispielsweise weil die Methode überladen ist, muss auch die Signatur angegeben werden.

callin_modifier
: Eins, der drei Schlüsselwörter `before`, `after` oder `replace`, definiert die Bindungsart. Das Schlüsselwort `before` bedeutet, dass die Rollenmethode *vor* der Basismethode aufgerufen wird, `after` heißt, dass sie *danach* aufgerufen wird und `replace` bedeutet, dass die Basismethode durch die Rollenmethode ersetzt wird, also *stattdessen* aufgerufen wird. Bei `replace` hat die Rollenmethode die Möglichkeit, die Basismethode aufzurufen. Dies nennt man *Basecall*.

base_method_designator
: Der Name einer Methode, der Basisklasse, falls er nicht eindeutig ist, muss die Signatur

angegeben werden. Die Basisklasse wird durch das Schlüsselwort `playedby` im Kopf der Rollenklasse definiert (siehe oben).

Basecall. Wird eine Callin-Bindung mit `replace` definiert, hat die Rollenmethode die Möglichkeit, die ersetzte Basismethode aufzurufen. Dafür gibt es den *Identifier* `base`. Er wird benutzt, wie `this` oder `super` in *Java*:

```
1 class MyRole playedBy MyBase {
2         ...
3         roleMethod3 <- replace baseMethod3;
4         callin void roleMethod3() {
5                 ...
6                 base.roleMethod3(); // Basecall
7                 ...
8         }
9 }
```

Parametermappings. Mit Parametermappings ist es möglich, Parameter der Basismethode auf Parameter der Rollenmethode abzubilden oder umgekehrt. Dann ist relevant, weil eine Rollenmethode und ihre Basismethode die unterschiedliche Signatur haben können. In der Bindung muss die vollständige Signatur angegeben werden. Die Mappings erfolgen mit Hilfe des Schlüsselwortes `with` und den Operatoren `->` und `<-`. Auf Rückgabewerte wie mit dem *Identifierer* `result` zugegriffen [4]:

```
1 int roleMethod(int i) <- replace Integer baseMethod(int i1, int i2) with {
2         i <- i2;
3         new Integer(result) -> result;
4 }
```

Die Rollenmethode erhält beim Aufruf den Parameter `i2` der Basismethode als Parameter `i` übergeben. Der Parameter `i1` ist nicht verfügbar. Das Ergebnis von `roleMethod`, wird als `Integer` geboxt und wird das Ergebnis eines Aufrufes von `baseMethod` sein.

Teams. Bis jetzt wurde unterschlagen, dass OT/J auch sogenannte Teams kennt. Obwohl sie ein wichtiges Konstrukt von OT/J sind, sind sie für das Verständnis dieser Arbeit nicht

so wichtig. Einige Eigenschaften sollen jedoch nicht unerwähnt bleiben: Rollen werden innerhalb eines Teams definiert. In [19] werden sie als „*compound module of aspect classes*“ bezeichnet, also als Zusammenfassungsmodul von Aspektklassen. Die Definition eines Teams geschieht wie folgt:

```
1 team class MyTeam {
2         class MyRole ...
3 }
```

Außerdem können Teams aktiviert und deaktivert werden. Die Bindungen der Rollen sind nur dann aktiv, wenn ein Team aktiv ist. Wenn ein Team deaktiviert ist, verhalten sich die Basisklassen, die durch Rollen des Teams adaptiert werden, als wenn die Rollen gar nicht da wären.

2.3 ObjectTeams erweitern

Der obere Abschnitt hat gezeigt, dass die Bindung der Aspekte – in OT/J Rollen – an die Basisklassen auf Methodenebene mit Callin-Bindungen passiert. In der Literatur wird dies auch als *Method Call Interception* (MCI) bezeichnet. MCI kommt der Unterstützung eines einzigen Joinpoints gleich – dem Execution-Joinpoint. Ein Execution-Joinpoint bezeichnet die tatsächliche Ausführung einer Methode, nicht etwa den Aufruf.

Die Beschränkung auf MCI ist zu groß. Man benötigt stattdessen *Joinpoint-Interception*, um alle Arten von Joinpoints abzudecken [19]. *Joinpoint-Interception* bezeichnet natürlich die Bindung auf Joinpointebene. Ein anderer Nachteil ist, dass jede MCI explizit formuliert werden muss. Es ist nicht möglich, eine Menge von Joinpoints an einen Aspekt zu binden, weil Joinpoint – bis jetzt – nicht *quantifiziert* werden können. Deshalb ist bereits eine Querysprache in Entwicklung, die die Quantifizierung von Joinpoints ermöglichen soll.

In [19] wird ein Modell für AOP vorgestellt, das nur aus den Konzepten *Joinpoints*, *Joinpoint-Queries* und *Joinpoint-Interception* besteht. Dieses Modell soll für OT/J realisiert werden. Ziel dieser Diplomarbeit ist es deshalb, zu untersuchen, welche weiteren Joinpoints für *ObjectTeams/Java* in Frage kommen und wie die Joinpoint-Interception für *ObjectTeams/Java* realisiert werden kann. Eine prototypische Implementierung für Joinpoint-Queries existiert bereits und kann mit Einschränkungen verwendet werden.

Kapitel 3

Java

ObjectTeams/Java basiert, wie der Name vermuten lässt, auf *Java*. Große Teile der Java-Infrastruktur werden von *ObjectTeams/Java* wiederverwendet. Dieses Kapitel soll deshalb erklären, wie die Infrastruktur von Java funktioniert. Damit sind vor allem die Vorgänge des Compilierens und der Ausführung von Java-Anwendungen gemeint. Vom Leser wird vorausgesetzt, dass er zumindest grundlegende Kenntnisse in der Programmierung mit Java besitzt.

In Abbildung 3.1 wird die Compilierung und Ausführung einer Java-Anwendung veranschaulicht. Der Java-Quelltext (eine `java`-Datei) wird von einem Compiler verarbeitet. Compiler herkömmlicher Programmiersprachen erzeugen Maschinencode. Maschinencode oder *nativer Code*, wie er auch genannt wird, ist direkt auf einem Prozessor ausführbar, für den er erzeugt wurde[1]. *Java*-Compiler verhalten sich anders. Sie erzeugen sogenannten *Bytecode*. Dieser Bytecode wird zusammen mit anderen Informationen in `class`-Dateien gespeichert. Die `class`-Dateien können dann von der Java-Laufzeitumgebung ausgeführt werden.

Ziel dieses Kapitels ist es, dem Leser ein grundlegendes Verständnis für diese Vorgänge zu geben. Dazu wird zuerst erklärt, worum es sich bei Bytecode handelt. Danach sollen Javas

[1] Natürlich muss das Betriebssystem noch das Format der ausführbaren Dateien (*Binaries*) kennen, um den Maschinencode vor der Ausführung in den Speicher zu laden.

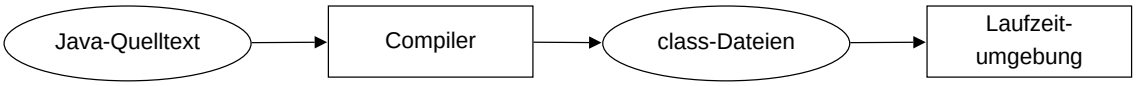

Abbildung 3.1: Compilierung und Ausführung einer Java-Anwendung.

`class`-Dateien erklärt werden und schließlich die *Java Virtual Machine*, die Hauptbestandteil der Java-Laufzeitumgebung ist und die Aufgabe hat, den Bytecode auszuführen.

3.1 Bytecode

Für jede Plattform (oder jeden Prozessortyp), die ein Compiler unterstützt, muss er anderen Maschinencode erzeugen. Das hat den Nachteil, dass die Komplexität von Compilern unnötig erhöht wird, denn für jede neue Plattform muss ein zusätzlicher Codegenerator programmiert werden. Deshalb kam man schon früh auf die Idee, den Compiler nur einen Code erzeugen zu lassen[2]. Um diese Art von Code auch namentlich von nativem Code unterscheiden zu können, wird sie im Folgenden *Bytecode* genannt. Bytecode ist Maschinencode sehr ähnlich. Er hat auch einen definierten Befehlssatz. Dieser Befehlssatz stellt, wie bei nativem Code, Befehle für arithmetische, logische und andere Operationen bereit.

Virtuelle Maschine. Der Bytecode kann von einem Prozessor in der Regel nicht ausgeführt werden, deshalb muss er von einer Software interpretiert werden. Diese Software nennt man auch *Virtuelle Maschine* (VM). Sie führt den Bytecode aus, als wäre sie ein Prozessor, dessen nativer Code dem Bytecode entspricht.

Vorteile. Die Verwendung von Bytecode hat viele Vorteile. Ein Vorteil wurde oben bereits angedeutet. Ein Compiler muss bei der Codegenerierung nur noch einen Code unterstützen. Außerdem ist der Bytecode plattformunabhängig[3]. Häufig wird als Vorteil angeführt, dass die Portierung der virtuellen Maschine auf andere Plattformen leicht ist. Das ist aber nur der Fall, wenn man auf Just-in-Time-Compilation – die Umwandlung in nativen Code zur Laufzeit, um die Performance zu verbessern – verzichtet. Dann reicht es die virtuelle Maschine, die in *Ansi C* geschrieben ist, auf der neuen Plattform zu compilieren. Natürlich muss ein *Ansi-C*-Compiler für diese Plattform existieren, aber das ist fast immer der Fall. Die resultierende VM wird den Code leider nur langsam ausführen, weil sie ihn interpretieren muss. Um die Vorteile von Just-in-Time-Compilation zu nutzen, ist eine aufwendigere Portierung notwendig. Ein weiterer Vorteil ist der geringere Unterschied zwischen Quelltext und Compilat, der Bytecodemanipulationen, wie in dieser Arbeit, erheblich vereinfacht. Fast möchte man sagen, überhaupt erst möglich macht.

[2] Ende der 60er gab es den sogenannten O-Code für die Programmiersprache BCPL. In den frühen 70ern gab es den von Niklaus Wirth (Erfinder der Programmiersprache Pascal) erdachten P-Code.

[3] Der gleiche Bytecode läuft auf unterschiedlichen Maschinen.

3.2 class-Dateien

Auch Java setzt auf die Vorteile von Bytecode, deshalb erzeugen die meisten Java-Compiler *Java-Bytecode*[4]. Der Bytecode wird in `class`-Dateien gespeichert. Für jede Klasse erzeugt der Compiler eine `class`-Datei. `class`-Dateien enthalten zahlreiche Informationen. An dieser Stelle sollen nur die für uns relevanten Informationen beschrieben werden. Eine detailierte Beschreibung findet man in [21]:

- *Meta-Informationen*: Informationen über die Schnittstelle der Klasse, implementierte Interfaces, die Superklasse etc.
- *Bytecode der Methoden*: Die `class`-Dateien enthalten natürlich den Bytecode. Programmcode steht in compilierten Java-Programmen nur innerhalb einer Methode, deshalb wird der Bytecode für jede Methode separat gespeichert. Im Sourcecode kann auch außerhalb von Methoden Code stehen. Es ist Klassen- und Objektinitialisierungs-Code. Objektinitialisierungs-Code wird vom Java-Compiler in die Konstruktoren, Klasseninitialisierungs-Code wird in eine spezielle Methode `<clinit>` verschoben.
- *Konstanten-Pool*: Der sogenannte *Constant Pool* enthält unter anderem Literale und Referenzen. Literale sind konstante numerische Werte und Zeichenketten, die in der Klasse benutzt werden. Die Referenzen verweisen auf Klassen, Interfaces, Felder und Methoden. Da solche Informationen häufig mehrfach vom Bytecode benutzt werden, ist es sinnvoll, sie nur einmal zu speichern und bei Bedarf auf sie zu verweisen. Dazu dient ein Index, der einen Eintrag im *Constant Pool* referenziert.
- *Bytecode-Attribute*: Bytecode-Attribute ermöglichen es, weitere Informationen in einer `class`-Datei abzulegen. Es gibt einige vordefinierte Attribute, die beispielsweise das Verhalten bei Exceptions definieren oder die Informationen über den Gültigkeitsbereich von lokalen Variablen enthalten. Das Verhalten bei Exceptions wird durch eine Tabelle definiert. Sie weist einem Codebereich und einem Exceptiontyp ein Sprungziel zu. Tritt in einem bestimmten Codebereich eine Exception auf und existiert in der Exceptiontabelle für den Codebereich und die Exception ein Sprungziel, dann veranlasst die VM einen Sprung an dieses Ziel. Dort steht der Exceptionhandler-Code. Uns interessieren vor allem die benutzerdefinierten Attribute, da sie im Folgenden benutzt werden, um Informationen vom Compiler in die Laufzeitumgebung zu transportieren. Gemein ist allen Attributen, dass sie einen Namen besitzen und die Daten als `byte`-Array speichern. Das soll uns an dieser Stelle reichen, gegebenenfalls werden konkrete Attribute vorgestellt werden, wenn dies zum Textverständnis nötig ist.

[4] Es gibt auch Java-Compiler, die nativen Code generieren können.

3.3 Java Virtual Machine

Die *Java Virtual Machine* (*JVM*) ist Teil der Java-Laufzeitumgebung, engl. *Java Runtime Environment* (*JRE*). Die Klassen der Java-Standard-Bibliothek gehören auch zum *JRE*. Die *JVM* interpretiert den Bytecode[5]. Sie ist eine sogenannte *Stackmaschine* (0-Adress-Architektur). Das heißt, dass die Operanden einer Bytecode-Instruktion nicht direkt als Parameter übergeben werden, sondern vorher auf den Operanden-Stack gelegt werden müssen.

Stack. Für jeden Thread der in der *JVM* läuft, existiert ein Stack. Möchte man beispielsweise eine Addition $a + b$ durchführen, legt man zuerst die beiden Operanden a und b auf den Stack und ruft dann `add` auf. Das Ergebnis der Addition wird ebenfalls auf dem Stack abgelegt. Ein ausführlicheres Beispiel mit Stack kommt im Abschnitt *Beispiel*.

Eine Instruktion erwartet Operanden auf dem Stack. `add` erwartet z. B. zwei Operanden. Wenn alle Operanden, wie erwartet, auf dem Stack liegen, ist der Stack in Balance. Ob ein Stack in Balance ist, lässt sich leicht berechnen. Jede Bytecodeinstruktion produziert und konsumiert eine bestimmte Anzahl Operanden. Wenn man vom Anfang bis zum Ende der Methode alle Instruktionen durchläuft und dabei die Differenz, der von einer Instruktion produzierten und konsumierten Elemente summiert, fällt folgendes auf: Ist die Zahl zwischenzeitlich negativ, liegt ein Fehler vor. Ist die Zahl nach Abschluss der Berechnung nicht 0, liegt ein Fehler vor. Sollte der Stack einmal nicht in Balance sein, stellt dies einen schwerwiegenden Fehler dar. Ein solcher Fehler kann nur durch eine fehlerhafte Implementierung des Compilers, der Laufzeitumgebung, des Betriebssystems oder durch Hardwarefehler verursacht werden. Der Programmierer einer Java-Anwendung kann einen solchen Fehler nicht verursachen. Da im Rahmen dieser Diplomarbeit der Bytecode manipuliert werden wird, wird uns das Thema Stackbalance noch beschäftigen.

Typen. Die *Java Virtual Machine* unterstützt zahlreiche Type, das sind in im Prinzip die gleichen Typen, die auch Java kennt. Obwohl Java Pointer kennt (Referenzen), gibt es in Java keine Pointerarithmetik. So werden viele, der damit verbundenen Probleme (Overflows) vermieden [21]:

- *Ganzzahlige Typen*: `byte, short, int, long, char`

[5] Genaugenommen wird der Bytecode schon lange nicht mehr interpretiert, sondern Just-In-Time compiliert (dazu später mehr).

- *Gleitkomma Typen*: `float, double`
- *Wahrheitswerte*: `boolean`
- *Referenztypen*: Referenzen zeigen auf Arrays oder Instanzen von Klassen, also Objekte.
- *Rücksprung-Adresse*: `returnAddress` wird von den Instruktionen `jsr*` und `ret` benutzt, um bei einem Aufruf von `ret` hinter das dazugehörige `jsr*` springen zu können. Sie besitzen keine Entsprechung in Java und können auch nicht manipuliert werden.

Bytecode Verifier. Die *JVM* kann, während sie Klassen lädt, eine Reihe von Überprüfungen durchführen. Dazu gehört auch eine Verifizierung des Bytecodes durch den *Bytecode Verifier*. Er überprüft beispielsweise, ob Sprungziele existieren oder bei Methodenaufrufen die Typen der übergebenen Parameter konform mit der Signatur der aufgerufenen Methode sind. Diese Überprüfungen sind standardmäßig größtenteils deaktiviert, können aber aus Sicherheitsgründen aktiviert werden. Wenn man – und das soll in dieser Arbeit geschehen – den Bytecode manipuliert, ist es wichtig sicherzustellen, dass der manipulierte Code alle diese Überprüfungen besteht.

Overflows. Während der Laufzeit werden durch die *JVM* Überprüfungen durchgeführt. Lästige Buffer- und Heap-Overflows gehören damit der Vergangenheit an, da sich z. B. Array-Grenzen zur Laufzeit überprüfen lassen. Der Verzicht auf Pointerarithmetik ist, neben der Ausführung in einer *VM*, eine grundlegende Voraussetzung dafür, dass diese Überprüfungen zur Laufzeit effizient möglich sind.

Just-In-Time Compilation. Wie oben bereits kurz angesprochen, wird Java-Bytecode heute meist nicht mehr ausschließlich interpretiert, sondern Just-In-Time compilert. Dabei wird der Bytecode, wenn er ausgeführt werden soll, zuerst von der *JVM* in nativen Code umgewandelt. Das passiert nur, wenn diese Umwandlung semantikerhaltend möglich ist. Erst dann wird der Code ausgeführt. Das ermöglicht erhebliche Performancegewinne. Die *JVM*s der früheren Tage haben den Bytecode noch komplett interpretiert, deshalb steht Java bis heute in dem Ruf langsam zu sein, obwohl dies nicht den Tatsachen entspricht.

Garbage Collection. Eine der wichtigsten Neuerungen von Java ist *Garbage Collection*. Sie entbindet den Programmierer von der Verantwortung, Speicherverwaltung zu betreiben, also reservierten Speicher wieder freizugeben. Der *Garbage Collector* sucht dazu mit Hintergrund nach Objekten, die nicht mehr referenziert werden und gibt sie frei.

3.4 Befehlssatz

Der Befehlssatz der *Java Virtual Machine* (*JVM*) ist auf die Bedürfnisse von Java zugeschnitten und bietet auch objektorientierte Features wie Objekterzeugung. Normale Assembler verfügen in ihrem Befehlssatz nicht über solche Instruktionen. Die *JVM* kennt für einige ansonsten gleiche Befehle unterschiedliche Ausprägungen. Je nach Typ der Operanden ist dem Befehl ein Buchstaben vorangestellt:

Kürzel	***Verwendbare Typen***
`i`	`int, byte, boolean, short, char`
`l`	`long`
`f`	`float`
`d`	`double`
`a`	*Referenztypen*

Im Folgenden kommt eine Auflistung des Befehlssatzes der JVM, gegliedert nach Bereichen. Die Aufgliederung und Benennung wurde *The Java Virtual Machine Specification* entnommen. Um nicht alle Befehle einzeln auflisten zu müssen, werden reguläre Ausdrücke verwendet. Die Notation `{a,b}{c,d}` bezeichnet dann `ac, ad, bc, bd`. Die Liste ist nicht ganz vollständig, denn sie soll nur einen Überblick über den Befehlssatz geben. Eine vollständige Liste findet sich in [21]:

Laden- und Speichern

Lokale Variablen auf den Stack laden: `{i,l,f,d,a}{load,load_<n>}`
Wert vom Stack in lokaler Variable speichern: `{i,l,f,d,a}{store,store_<n>}`
Konstante auf den Stack laden: `{i,l,f,d}{const_<c>}, ldc*, aconst_null`

Die Bezeichnung `<n>` ist eine Zahl zwischen 0 und 3. Befehle mit dem Postfix `_<n>` können mit nur einem Byte dargestellt werden. Die anderen load/store-Befehle benötigen zwei Bytes. Weil eine Methode selten mehr als vier lokale Variablen benutzt, spart diese Maßnahme Speicherplatz. Die Bezeichung `<c>` meint den Wert einer Konstanten.

Arithmetik

Addition: `{i,l,f,d}add`
Subtraktion: `{i,l,f,d}sub`
Multiplikation: `{i,l,f,d}mul`

Division: `{i,l,f,d}div`
Divisionsrest: `{i,l,f,d}rem`
Negation: `{i,l,f,d}neg`
Bitschieben: `{i,l}shl, {i,l}{shr,ushr}`
Bitweises OR: `ior, lor`
Bitweises AND: `iand, land`
Bitweises XOR: `ixor, lxor`
Inkrementierung: `iinc`
Vergleich: `{d,f}cmp{g,l}`

Typumwandlung

Befehle für die Umwandlung von einem numerischen Typ in einen anderen numerischen Typ. Beim *Widening* tritt kein Informationsverlust auf. Beim *Narrowing* kann ein Informationsverlust auftreten. Typcasts gehören nicht zu den Typumwandlungsbefehlen.

Widening: `i2{l,f,d}, l2f, l2d, f2d`
Narrowing: `i2{b,c,s}, l2i, f2{i,l}, d2{i,l,f}`

Objekterzeugung und -manipulation

Objekterzeugung: `new`
Arrayerzeugung: `newarray, anewarray, multinewarray`
Lesende und schreibende Feldzugriffe: `getfield, putfield, getstatic, putstatic`
Lesende und schreibende Arrayzugriffe: `{b,c,s,i,l,f,d,a}{aload,astore}`
Arraylänge: `arraylength`
Typ-Überprüfungen: `instanceof, checkcast`

Operanden-Stack-Management

Typische Stack-Operationen: `pop*, dup*, swap`

Kontroll-Transfer

Bedingte Sprünge: `if{eq,lt,le,ne,gt,ge,null,nonnull}`
`if_icmp{eq,ne,lt,gt,le,ge}, if_acmp{eq,ne}`

Switch: `tableswitch, lookupswitch`
Unbedingte Sprünge: `goto*, jsr*, ret`

Bedingte Sprünge mit Objekt- oder Integervergleich sind am häufigsten, deshalb gibt es für sie kombinierte Befehle. Die anderen bedingten Sprungbefehle erfordern einen vorhergehenden Aufruf einer Vergleichsoperation `cmp*`.

Methodenaufrufe und -rückkehr

Aufrufe: `invokevirtual, invokestatic, invokespecial, invokestatic`
Rückkehr: `{i,l,f,d,a}return`

Die `invoke`-Befehle werden später noch genau erklärt. Die Rückgabe von den Typen `boolean, byte, char, short` erfolgt mit `ireturn`.

Sonstige

Werfen von Exceptions: `athrow`
Synchronisierung: `monitorenter, monitorexit`
No Operation (Nichts machen): `nop`

Die Synchronisierung auf Methodenebene erfolgt nicht durch die obigen Befehle, sondern implizit durch die VM. Der Befehl `nop` macht nichts. Obwohl das auf den ersten Blick unsinnig erscheint, kann der Befehl doch nützlich sein. Beispielsweise um unerwünschte Instruktionen zu überschreiben.

3.5 Beispiel

Die bisherige Beschreibung des Java-Bytecode war relativ abstrakt, deshalb soll in diesem Kapitel einmal gezeigt werden, wie der Bytecode eigentlich aussieht und was während seiner Ausführung passiert.

In Listing 1 ist eine Java-Methode dargestellt und in Listing 2 der dazugehörige Bytecode. Der Bytecode ist wie eine Aufspaltung in die elementaren Bestandteile und liest sich relativ gut:

```
1 public static void main(String[] args) {
2     Beispiel bsp = new Beispiel();
3     bsp.hallo(args);
4 }
```

Listing 1: Eine Java-Methode.

```
1 new #2; //class Beispiel
2 dup
3 invokespecial #3; //Method "<init>":()V
4 astore_1
5 aload_1
6 aload_0
7 invokevirtual #4; //Method hallo:([Ljava/lang/String;)V
8 return
```

Listing 2: Java-Bytecode der Methode aus Listing 1.

- *Zeile 1:* Zu Beginn der Methode hat der Stack irgendeinen Zustand. Das ist in Abbildung 3.2-0 durch Punkte dargestellt. Ein neues Objekt der Klasse `Beispiel` wird erzeugt und eine Referenz auf dieses Objekt auf den Stack gelegt (Abbildung 3.2-1). Das neue Objekt ist noch uninitialisiert, da noch kein Konstruktor aufgerufen wurde. Der Parameter `#2` von `new` ist ein Index in den *Constant Pool*. An dieser Stelle liegt ein Verweis auf Klasse des zu erzeugenden Objekts.

- *Zeile 2:* Die Referenz auf das neue Objekt wird dupliziert, weil sie später noch benötigt wird. Nun liegt sie zweimal auf dem Stack (Abbildung 3.2-2).

- *Zeile 3:* Der Konstruktor wird jetzt aufgerufen. Er initialisiert das neue Objekt. Durch diesen Aufruf wird auch die oberste Referenz auf das Objekt konsumiert (Abbildung 3.2-3).

- *Zeile 4:* Da in *Zeile 2* die Referenz kopiert wurde, liegt noch eine Referenz oben auf dem Stack (Abbildung 3.2-3). Diese Referenz wird in der ersten lokalen Variable gespeichert; womit sie auch von Stack konsumiert wird (Abbildung 3.2-4). Die Bytecode-Befehle 1-4 sind die Übersetzung der ersten Zeile von Methode `main` in Listing 1.

- *Zeile 5:* Der Wert der lokalen Variable 1 wird auf den Stack gelegt. Dieser Wert ist

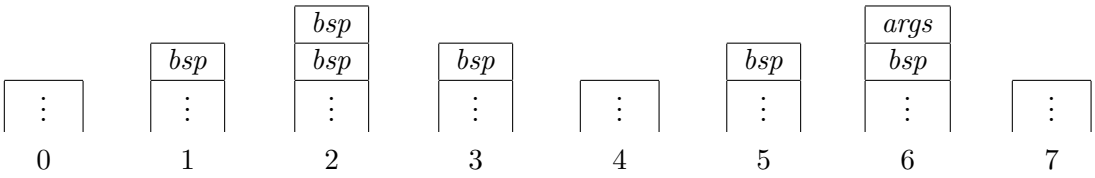

Abbildung 3.2: Zustände des Stacks nach Ausführung von Zeile x des Listings 2.

eine Referenz auf ein Objekt, an dem eine Funktion aufgerufen werden soll. In diesem Fall eine Referenz, auf das vorher erzeugte Objekt `bsp` (Abbildung 3.2-5).

- *Zeile 6:* Eine Referenz auf ein Objekt, das als Parameter der Funktion `hallo` dienen soll, wird auf den Stack gelegt. In diesem Fall ist es die lokale Variable 0, die nichts anderes ist als `args` (Abbildung 3.2-6). Bei statischen Methoden ist die lokale Variable 0 der erste Parameter (falls vorhanden). Bei dynamischen Methoden ist die lokale Variable 0 immer mit `this` belegt, das impliziter erster Parameter ist.
- *Zeile 7:* Die Methode `hallo` wird aufgerufen und konsumiert beide Elemente vom Stack (Abbildung 3.2-7).
- *Zeile 8:* Die Methode ist nun zu Ende und kehrt durch `return` zurück zum Aufrufer.

Natürlich ist der oben gezeigte Code nur ein Beispiel, aber es ist erkennbar, dass auf Grund der größeren Abstraktion und des geringeren Umfangs des Befehlssatzes der *JVM*, der Bytecode besser verständlich ist, als der Assemblercode gängiger Prozessoren. In diesem Beispiel ist der Bytecode relativ gut lesbar. Das ist nicht immer der Fall. Wenn eine Methode beispielsweise Schleifen enthält, sind diese, wie in echten Assemblern, durch Labels als Sprungziele realisiert und man kann schnell den Überblick verlieren.

Die gute Lesbarkeit ist jedoch nur ein Nebeneffekt. Das Hauptaugenmerk bei der Entwicklung des Java-Bytecode lag natürlich in der Plattformunabhängigkeit und der Möglichkeit, Laufzeitüberprüfungen durchzuführen. Auch die bereits angesprochene Ähnlichkeit zwischen Quelltext und Compilat dürfte eine Rolle gespielt haben. Ob die Java-Entwickler dabei auch die leichter durchzuführende Manipulation des Compilats im Kopf hatten, ist mir unbekannt.

Kapitel 4

Joinpoints in Java

Im Abschnitt über aspektorientierte Programmierung wurden Joinpoints als „Stellen, an den das Programm erweitert werden kann“ beschrieben. Diese Definition ist zwar richtig, dennoch soll der Begriff Joinpoint etwas genauer betrachtet werden. Es sei an dieser Stelle noch einmal darauf hingewiesen, dass *Joinpoint* in dieser Arbeit ein statisches Vorkommen im Programmcode bezeichnet. Manchmal wird auch der Begriff *Shadow* dafür verwendet.

Es gibt in der Literatur keine genaue Definition, was genau ein Joinpoint ist. In einem Paper ist zu lesen: „*[...] join points are certain well-defined points in the execution of the program.*“ [8]. Danach könnte ein Joinpoint beispielsweise als Folge von Knoten im Kontrollflussgraphen einer Methode betrachtet werden. In Abbildung 4.1 ist ein solcher Kontrollflussgraph dargestellt. Die Knoten 1 bis 3 stellen eine Schleife dar. Der Knoten 4 ist die erste Anweisung nach der Schleife. Wenn eine beliebige Folge von Knoten ein Joinpoint sein könnte, dann wären die Knoten 1 und 3 ein Joinpoint. Also die letzte und die erste Anweisung der Schleife. Ob diese Definition sinnvoll ist, sei dahin gestellt. Sie ist aber nicht besonders hilfreich, weil sie nicht sagt, welche Joinpoints sinnvoll sind.

In [9] werden Joinpoints als „*[...] those elements of the component language semantics that the aspect programs coordinate with.*“ beschrieben. Diese Aussage ist schon etwas aufschlussreicher. Das Paper [19] enthält eine noch konkretere Aussage, was Joinpoints sind: „*[...] those program elements correspond to nodes in the abstract syntax tree (AST) of a program.*“ Statt an dieser Stelle eine wasserdichte Definition des Begriffs Joinpoint zu liefern, soll im Geist der präsentierten Zitate eine Aussage gemacht werden, welche Joinpoints sinnvoll sind. Die Vorstellung von Joinpoints als Stellen, an denen das Programm erweitert werden kann, ist als generische Definition weiter geeignet.

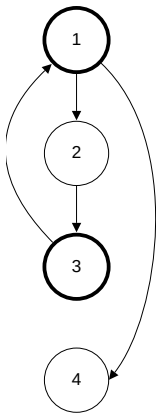

Abbildung 4.1: Die Knoten 1 und 3 könnten ein Joinpoint sein.

4.1 Sinnvolle Joinpoints

In diesem Abschnitt soll ein Kriterium vorgestellt werden, mit dem es möglich ist zu unterscheiden, was ein sinnvoller Joinpoint ist. Dieses Kriterum soll einerseits sinnlose Joinpoints aussortieren, andererseits sollen diejenigen Joinpoints dem Kriterium genügen, die ein Programmierer benötigt, um maximalen Nutzen aus der Aspektorientierung ziehen zu können. Aus dieser Anforderung ergab sich folgendes Kriterium:

Ein Joinpoint ist dann sinnvoll, wenn er genau ein Hochsprachkonstrukt der zugrunde liegenden Programmiersprache repräsentiert.

Die Joinpoints sollen auf dem gleichen Abstraktionsniveau wie die Programmiersprache liegen, schließlich soll die Aspektorientierung eine Bereicherung sein und nicht die wertvolle Abstraktion aufbrechen. Die Einschränkung auf *ein* Hochsprachkonstrukt ist keine wirkliche Einschränkung. Ein Joinpoint, der aus mehreren Konstrukten besteht, kann in seine Bestandteile zerlegt werden.

Da viele Hochsprachkonstrukte eine direkte Entsprechung im Bytecode-Befehlssatz haben und da das Weben und das Suchen nach Joinpoints im Bytecode stattfinden kann, soll das gerade genannte Kriterium verwendet werden, um im Befehlssatz der JVM nach möglichen Joinpoints zu suchen. Die Frage ist also, welche Instruktionen repräsentieren kein Hochsprachkonstrukt:

- *Laden- und Speichern*: Sowohl das Laden- und Speichern in lokale Variablen, als auch die Benutzung von Literalen sind ein Hochsprachkonstrukt.

- *Arithmetische Operationen*: Ganz offensichtlich gibt es für jede der Instruktionen auch einen Operator in Java.
- *Typumwandlung*: Auch wenn die Umwandlung in andere numerische Typen teilweise implizit erfolgt, ist es dennoch gerechtfertigt, hier von einem Java-Konstrukt zu sprechen.
- *Objekterzeugung und -manipulation*: Bis auf `new` gibt es für jede der Operationen eine entsprechende Java-Anweisung. Von `new` kann man dies nicht behaupten, weil es eigentlich nur ein uninitialisiertes Objekt erzeugt. Die wirkliche Erzeugung beginnt erst mit dem Konstruktoraufruf. Objekterzeugung ist aber definitiv ein Joinpoint.
- *Operanden-Stack-Management*: Die Stackoperationen sind definitiv keine Hochsprachkonstrukte.
- *Kontroll-Transfer*: Auch wenn diese Operationen benutzt werden, um Hochsprachkonstrukte zu implementieren, sind sie selbst keine.
- *Methodenaufrufe und -rückkehr*: Dies sind eindeutig Hochsprachkonstrukte.
- *Werfen von Exceptions*: Ein Hochsprachkonstrukt.
- *Synchronisierung*: Ein Hochsprachkonstrukt, benutzbar durch das Schlüsselwort `synchronized`.
- *No Operation (nop)*: Kein Hochsprachkonstrukt.

In dieser Liste werden eine Menge Java-Konstrukte genannt, aber nur die, die eine direkte Bytecode-Entsprechung besitzen. Nochmal zusammengefasst sind das: Methodenaufrufe (auch statische), Methodenrückkehr, Objekterzeugung, Feldzugriffe (auch statische), Arrayzugriffe, Casts, Werfen von Exceptions, Synchronisierung mit Monitoren, Variablenzugiffe, Typüberprüfung, arithmetische, boolesche und bitweise Operatoren.

Nicht dabei sind folgende Konstrukte, weil sie im Bytecode nicht oder nur implizit vorhanden sind: Dereferenzierung, Schleifen, Bedingte Ausdrücke: switch, if, ternärer Operator; Bedingungen, Methodenkörper, Fangen von Exceptions, statische Klassen- und Feldinitialisierer.

4.2 Atomare Joinpoints

Obwohl alle diese Joinpoints sinnvoll sind, sollen nicht alle in dieser Arbeit in gleichem Maße berücksichtigt werden. Außen vor bleiben sollen Schleifen und bedingte Ausdrücke,

weil ihre Realisierung sehr aufwendig ist.

In [3] wird beispielsweise die Forderung nach einem Joinpoint für Schleifen gestellt und beschrieben, wie er sich realisieren lässt. Ein Schleifenjoinpoint würde es erlauben, die Ausführung der Schleife durch einen Aspekt steuern zu lassen. Der Aspekt könnte bei Zählschleifen den Start- und Endwert manipulieren und die Schrittweite. Außerdem könnte er die Iterationen einzeln anstoßen. Das wäre sehr sinnvoll, um Parallelisierung als Aspekt zu realisieren. Die Umsetzung ist allerdings kompliziert. Man weiß beispielsweise vor der Ausführung einer Schleife nicht immer, welcher der erste Befehl nach der Schleife ist. Verschachtelte Schleifen und Fallunterscheidungen können in Java, durch die Verwendung von `break` im Zusammenspiel mit einem Label, mehrere Nachfolge-Anweisungen haben. Die Verwendung von replace ist in diesem Fall nicht möglich, weil der Kontrollfluss undefiniert ist, wenn kein base-Call erfolgt. Denn man weiß nicht, wenn man die Schleife nicht ausführt, welches der nachfolgende Befehl ist, an dem die Programmausführung fortgesetzt werden soll.

Diese Arbeit soll sich hauptsächlich auf Joinpoints konzentrieren, die im Bytecode durch eine verzweigungsfreie Instruktionssequenz realisiert werden. Diese Art von Joinpoints heißt *Atomare Joinpoints*. Trotzdem wird hier und da auch auf andere Joinpoints eingegangen, beispielsweise das Fangen von Exceptions, das nicht nur im Bytecode definiert wird.

Execution- vs. Call-Joinpoint. Diese Stelle scheint außerdem geeignet, um den Unterschied zwischen Execution- und Call-Joinpoints anzusprechen, weil Execution-Joinpoints schon ein paar Mal angesprochen wurden, es sich aber nicht um einen atomaren Joinpoint handelt. Einfach gesagt, meint ein Execution-Joinpoint die Ausführung einer Methode. Ein Call-Joinpoint meint den Aufruf einer Methode. Der Unterschied wird klar, wenn man an den Bytecode dieser Joinpoints denkt. Der Code des Call-Joinpoint ist einfach ein `invoke`-Befehl. Der Code des Execution-Joinpoints dagegen ist ein gesamter Methodenrumpf.

4.3 Besonders wertvolle Joinpoints

Da im Rahmen dieser Diplomarbeit nicht alle der obengenannten Joinpoints umgesetzt werden können, muss *ObjectTeams* Schritt für Schritt um Joinpoints erweitert werden. Deshalb soll dieser Abschnitt betrachten, welches die wichtigsten Joinpoints sind, die in einer aspektorientierten Programmiersprache auf gar keinen Fall fehlen dürfen. Dazu werden zwei Kriterien definiert und dann Joinpoints anhand dieser Kriterien ausgewählt und schließlich ein Vergleich mit der aspektorientierten Sprache AspectJ gezogen.

Blackbox-Verwendbarkeit. *Die Informationen die benötigt werden, um einen Joinpoint zu spezifizieren, sind allein aus dem Interface einer Klasse abzulesen.*

Elementares Sprachkonstrukt. *Bei dem Joinpoint handelt es sich um ein zentrales Konstrukt, des zugrunde liegenden Programmierparadigmas der Programmiersprache.*

Erklärung

Kapselung von Klassen. Eine zu genaue Kenntnis des Aspekts über die Implementierung einer Klasse, hebt in gewisser Weise die Kapselung auf und erhöht die Kopplung des Aspekts an die Klasse. Das zeigt, dass Joinpoints, die blackboxverwendbar sind, eine besondere Stellung unter den Joinpoints haben.

Kein Sourcecode. Wenn kein Quelltext für eine Klasse existiert, was bei *ObjectTeams* der Fall sein kann, sind nur die Interfaces der Klassen bekannt. Diese sind dann Grundlage für die Auswahl von Joinpoints. Weitere Informationen über die Basisklassen können nur über komplizierte Maßnahmen wie Dekompilierung, Deassemblierung oder andere *Reverse-Engineering*-Techniken gewonnen werden. Also Aufwand, den es in der Regel zu meiden gilt.

Objektorientierte Dekomposition. Das Geschehen in objektorientierten Anwendungen spielt sich zwischen Klassen bzw. Methoden ab. In gut entworfenen objektorientierten Anwendungen ist zu beobachten, dass viele kleine Objekte miteinander kommunizieren. Klein meint in diesem Zusammenhang sowohl die Größe der Klassen, also die Anzahl der Methoden, als auch die Länge der Methoden selbst. Martin Fowler sagt in *Refactoring* dazu [13]:

„*Das objektorientierte Programm, das am besten und längsten lebt, ist das mit den kürzesten Methoden. Programmierer, für die Objekte etwas Neues sind, haben oft den Eindruck, dass nie Berechnungen gemacht werden, dass objektorientierte Programme eine endlose Folge von Delegationen sind.*“

„*Wie eine Klasse mit zu vielen Instanzvariablen ist auch eine Klasse mit zu viel Code eine hervorragende Brutstätte für duplizierten Code, Chaos und Tod.*“

Diese Zitate unterstreichen den hohen Stellenwert den die Kommunikation zwischen Objekten hat. Da diese Kommunikationsmöglichkeiten durch die Interfaces von Klassen bestimmt

werden, ist es nur recht, der Blackboxverwendbarkeit von Joinpoint auch einen so großen Stellenwert einzuräumen.

Vorraussetzungen. Eine der Vorraussetzungen für sinnvolle Verwendung von blackboxverwendbaren Joinpoints ist sicherlich eine geeignete funktionale Zerlegung, wie sie von Martin Fowler (siehe oben) oder Parnas [6] gefordert wird. Ein schlechtes Interface kann die Verwendung von Aspektorientierung verhindern oder zumindest erschweren.

Elementares Sprachkonstrukt. Im Fall von *ObjectTeams/Java* ist die zugrunde liegende Programmiersprache Java, also eine objektorientierte Programmiersprache (OOP). Es gibt Konstrukte, die nicht elementarer Bestandteil einer OOP sind. Als Beispiel sei die *Synchronisierung mit Monitoren* genannt. Sie ist *kein* elementares Sprachkonstrukt einer OOP. Methodenaufrufe sind dagegen das wohl wichtigste Konstrukt.

Hinweis. An dieser Stelle soll noch einmal ausdrücklich darauf hingewiesen werden, dass auch andere Joinpoints, die die gerade genannten Kriterien nicht erfüllen, trotzdem sehr sinnvoll sind. Zum Beispiel die angesprochenen Schleifen [3], auch wenn sie in dieser Arbeit nicht weiter betrachtet werden. Es gibt weitere Joinpoints, die man sinnvoll verwenden kann, obwohl man aus dem Klasseninterface keine Informationen darüber erhält, wie z. B. Verzweigungen, um die Überdeckung von Testfällen zu messen [11]. Damit kann automatisch gemessen werden, wieviel Prozent der Ausführungspfade im Kontrollflussgraphen durch einen Testfall abgedeckt werden. Die hier genannten Kriterien sollen nur motivieren, in welcher Reihenfolge die Joinpoints realisiert werden sollten.

Joinpoints

Wendet man das Kriterium für besonders wertvolle Joinpoints auf die sinnvollen und atomaren Joinpoints an, erhält man folgende Joinpoints:

- *Methodenaufrufe*: Aufrufe von Klassen- und Instanzmethoden.
- *Objekterzeugung*: Die Erzeugung von Objekten.
- *Feldzugriffe*: Sowohl lesend, als auch schreibend, dynamisch und statisch.
- *Exceptions*: Werfen bzw. Auftreten von Ausnahmen. Exceptions werden zwar nicht so häufig verwendet wie Methodenaufrufe – schließlich sind es Ausnahmen –, dennoch sind sie hier zu nennen.

AspectJ. Die eben genannten Joinpoints entsprechen ziemlich genau den statischen Joinpoints, die AspectJ überstützt [12, 23]. Das unterstreicht noch einmal deren Stellenwert.

4.4 Joinpointkatalog

Da Joinpoints ein zentrales Thema dieser Arbeit sind, wurde ein Joinpointkatalog zusammengetragen. Man findet ihn in Anhang A. Er enthält alle identifizierten Joinpoints. Weitere Kapitel reichern die Joinpointeinträge um zusätzliche Informationen an, so wie es nötig ist. Matching führt Einträge über die Eigenschaften von Joinpoints ein, die Kapitel übers Weben, wie ein entsprechender Joinpoint gewoben werden kann. Die Joinpoints sind nach Kategorien sortiert: *Methodenaufrufe, Objekterzeugung und -initialisierung, Feldzugriffe, Arrayzugriffe, Typprüfungen, Exceptions und Synchronisation, Lokale Variablen und Arithmetik, Bedingungen und Vergleiche und Dereferenzierung.* Für dieses Kapitel interessant sind die Einträge *Name, Beschreibung, Java und Bytecode.* Der Name soll eine griffige Bezeichung sein um den Joinpoint zu benennen. In Beschreibung wird der Joinpoint beschrieben. Java und Bytecode präsentieren den Javacode eines Joinpoint und den daraus resultierenden Bytecode.

4.5 Multihierarchie

Um ähnliche Joinpoints zusammenzufassen oder von ihrem konkreten Typ abstrahieren zu können, bieten sich sogenannte Multihierarchien an. In Abbildung 4.2 ist eine kleine Multihierarchie dargestellt. getField und setField können beispielsweise als fieldAccess abstrahiert werden. Die abstrakten Bezeichner können dann in Joinpointqueries (siehe Kapitel 5) verwendet werden, um einen Joinpoint zu beschreiben.

Allerdings werden Multihierarchie schnell sehr groß und unübersichtlich. Abbildung 4.3 zeigt, dass wenn man zum Beispiel Instanz- und Klassenfeatures unterscheiden will die Größe der Klassenanzahl schnell anschwillt. Es wurde trotzdem der Versuch unternommen, eine Multihierarchie für Joinpoints zu entwickeln. Diese Multihierarchie kann allerdings grafisch nur schlecht dargestellt werden. Sie wird deshalb nach folgendem Prinzip textuell definiert:

Klasse ▷ {*Superklassen ...*}

value ▷ {joinpoint}
hasOperands ▷ {joinpoint}

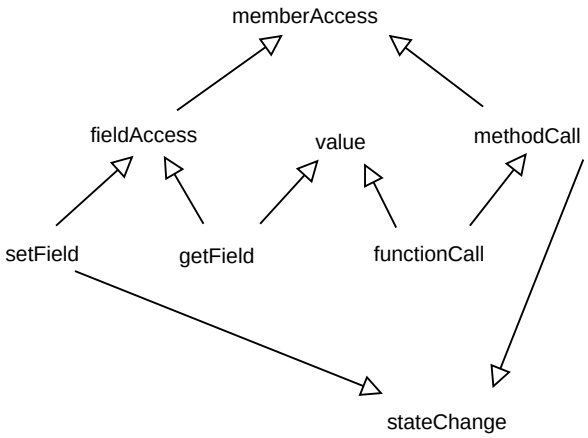

Abbildung 4.2: Eine Multihierarchie für Joinpointarten.

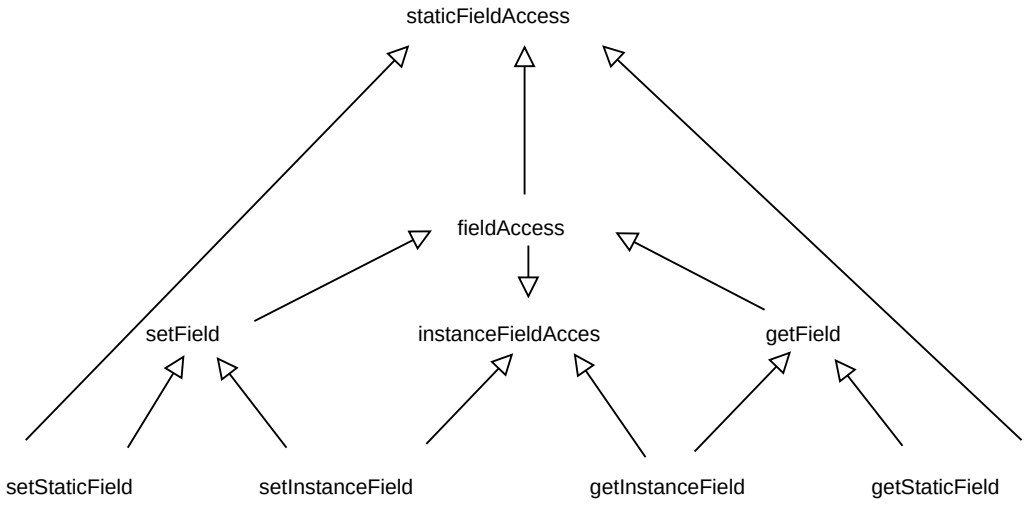

Abbildung 4.3: Teil einer Multihierarchie, hier Feldzugriffe.

access ▷ {joinpoint}
memberAccess ▷ {access}
staticMemberAccess ▷ {memberAccess}
instanceMemberAccess ▷ {memberAccess}
methodCall ▷ {memberAccess}
parameterMethodCall ▷ {hasOperands, methodCall}
noParameterMethodCall ▷ {methodCall}
functionCall ▷ {value, methodCall}
parameterFunctionCall ▷ {value, hasOperands, functionCall}
noParameterFunctionCall ▷ {value, functionCall}
fieldAccess ▷ {memberAccess}
staticFieldAccess ▷ {staticMemberAccess, fieldAccess}
instanceFieldAccess ▷ {instanceMemberAccess, fieldAccess}
getField ▷ {fieldAccess, value}
getStaticField ▷ {staticFieldAccess, getField}
getInstanceField ▷ {instanceFieldAccess, getField}
setField ▷ {fieldAccess, hasOperands, stateChange}
setStaticField ▷ {staticFieldAccess, setField, classStateChange}
setInstanceField ▷ {instanceFieldAccess, setField, objectStateChange}
stateChange ▷ {joinpoint}
objectStateChange ▷ {stateChange, instanceMemberAccess}
classStateChange ▷ {stateChange, staticMemberAccess}
localStateChange ▷ {stateChange}
arrayAccess ▷ {joinpoint}
arrayRead ▷ {value, arrayAccess, hasOperands}
arrayWrite ▷ {hasOperands, stateChange, arrayAccess}
variableAccess ▷ {noMemberAccess}
variableRead ▷ {value}
variableWrite ▷ {hasOperands, localStateChange}
primitiveType ▷ {value}
referenceType ▷ {value}
arrayType ▷ {referenceType}
objectType ▷ {referenceType}

Kapitel 5

Joinpoint-Querysprache

Um die zahlreichen von *ObjectTeams/Java* unterstützten Joinpoints zu verwenden, muss es möglich sein zu formulieren, welcher Aspekt bzw. welche Rollenmethode an welchen Joinpoint gebunden werden soll. Die Auswahl der Joinpoints erfolgt durch Queries, die in diesem Kapitel vorgestellt werden sollen. Welche Rollenmethode an einen Joinpoint gebunden werden soll, wird durch Callinbindungen definiert. Callinbindungen für Methoden wurden bereits beschrieben. Im Folgenden wird deshalb nur beschrieben, was sich ändert, wenn in einer Callinbindung eine Query statt eines Basismethodenbezeichners verwendet wird. Danach wird genauer darauf eingegangen, wie Joinpoint-Queries definiert werden und wie sie funktionieren.

5.1 Callinbindungen

query. Wie man Callinbindungen zur Zeit benutzt, wurde im Kaptiel über *ObjectTeams/Java* bereits vorgestellt (`roleMethod <- after baseMethod`). Wenn in einer Callinbindung statt eines Basismethodenbezeichners eine Query benutzt werden soll, wird dies durch Angabe des Schlüsselwörtes `query` kenntlich gemacht:

role_method_designator `<-` *callin_modifier*
`query` $join\_point\_signature_{opt}$ *query_method_designator*`;`

join_point_signature kann optional die Signatur der Joinpoints, die die Query zurückliefert, deklarieren. Notiert wird die Signatur wie in Java, beispielsweise: `void(int val)`. *query_method_designator* ist der Name der Query inclusive der Parameter der Query in spitzen

Kammern (z. B. `setterCalls<MyBase>`). Eine Callin-Bindung mit Query könnte dann wie folgt aussehen:

```
test <- before query setterCalls<MyBase>
```

In diesem Beispiel soll *vor* jedem Joinpoint, den die Query `setterCalls` zurückliefert, die Methode `test` ausgeführt werden.

playedBy. Da Joinpoint-Queries die Joinpoints aus unterscheidlichen Basisklassen zurückliefern sollen, muss ein neues Konstrukt eingeführt werden, das es ermöglicht mehrere Basisklassen auf einmal zuzulassen. Zur Zeit kann eine Rolle nur eine Basisklasse haben, die durch `playedBy` im Kopf der Klasse festgelegt wird. Um mehr als eine Basisklasse zuzulassen, wird ein Wildcard als Klassenname erlaubt. Das Wildcard `?` drückt aus, dass eine beliebige Basisklasse zulässig ist[1]:
`class` *role_class_name* `playedBy ? { ... }`

5.2 Allgemeine Eigenschaften von Queries

Queries werden ganz ähnlich wie Methoden innerhalb einer Rollenklasse definiert. Von Methoden unterscheiden sie sich vor allem durch das Schlüsselwort `query`. Eine ausführliche Beschreibung findet sich in der *ObjectTeams/Java Language Definition* [20].

Schlüsselwörter. `query` ist zur Zeit das einzige Schlüsselwort, das unterstützt wird. `abstract` soll aber auch realisiert werden, weil es nützlich sein wird, um in abstrakten Teams und Rollen die konkrete Implementierung der Query offen zu lassen. Dies wird der Wiederverwendbarkeit von abstrakten Teams zugute kommen und ihren Einsatz in Framework, die nach dem *Template-and-Hook*-Prinzip funktionieren, ermöglichen.

Queryname. Der Name einer Query muss innerhalb der beinhaltenden Klasse eindeutig sein. Es gibt kein Overloading. Eindeutig identifizierbar sind Queries durch die Angabe von Paketnamen, Klassennamen und Querynamen.

[1] Es ist zur Compilezeit nicht einmal bekannt, welche Klassen alles als Basisklasse in Frage kommen. Das klärt sich erst zur Laufzeit.

Signatur. Die Signatur einer Query unterscheidet sich syntaktisch nicht von einer Methodensignatur. Es können jedoch nur bestimmte Typen in der Signatur angegeben werden:

- Javatypen: `int`, `boolean` und `String`.
- Ein Typ aus dem OT/J-Metamodell[2]
- Eine Menge `Set<T>`, wobei `T` ein Typ aus dem Metamodell ist.

Verwendung. Es gibt zwei Möglichkeiten, Queries zu verwenden. Die Eine ist, in einer Callin-Methoden-Bindung und die Andere ist, in einer anderen Query. Bei der Verwendung in einer Query in einer Callin-Methoden-Bindung ist zu beachten, dass der Rückgabewert der Query von einem der Typen `Expression`, `Set<Expression>`, `Method` oder `Set<Method>` sein muss. Die Rückgabe einer `Expression` wird für die Definition einer Joinpointinterception benutzt, die Rückgabe einer Methode für Methodcallinterception. Bei einem `Set` werden mehrere Interceptions definiert.

Die Einschränkung auf `Expression` ist allerdings zur restiktiv. Es sollte auch `Statement` zugelassen werden, um zum Beispiel Schleifen als Joinpoints zu erlauben. Statements sind Schleifen, Fallunterscheidungen, Sprunganweisungen, Anweisungsblöcke, Verwendung von Marken und Expressions. Eine Expression ist also ein Statement. Im Bytecode gibt es diese Unterscheidung nicht mehr. Dort gibt es nur noch Instruktionen, weil Statement, und damit auch Expressions, aus einzelnen Instruktionen zusammengesetzt sind.

Auswertung. Da Queries auch zur Compilezeit ausgewertet werden sollen, haben Queries keine Möglichkeit auf Informationen zuzugreifen, die nur zur Laufzeit verfügbar sind, also z. B. Variablen der umgebenden Klasse. Genaueres zur Auswertung von Queries steht in einem gesonderten Kapitel. Queries werden im Kontext der Klasse ausgewerten, die die Query in einer Callinbindung benutzt.

Vererbung. Queries werden wie Methoden vererbt. Es findet jedoch kein dynamisches Binden statt. Die Queries werden im Kontext der sie benutzenden Klasse ausgewertet.

[2]Das *ObjectTeams/Java*-Metamodell wird später genauer beschrieben.

5.3 Queryausdrücke

Nachdem der vorherige Abschnitt die allgemeinen Eigenschaften von Queries beschrieben hat, wie man sie verwendet und deklariert, soll dieser Abschnitt sich mit dem Schreiben von Queries beschäftigen. Queries sind nicht wie Java-Methoden imperativ, sondern sie sind vielmehr mit Funktionen einer funktionalen Programmiersprache vergleichbar. Ihr Rumpf besteht nicht aus einer Vielzahl von Anweisungen, sondern aus einem einzigen seiteneffektfreien Ausdruck. Dennoch erinnert die Syntax der Queries an die einer imperativen Programmiersprache. Denn es gibt einen `for`-Ausdruck und lokale Variablen. Es handelt sich dennoch um funktionale Ausdrücke, wie die nächsten Abschnitte zeigen werden.

for-Ausdruck

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

Ein for-Ausdruck sieht aus, wie ein foreach-Konstrukt in Java. Es handelt sich aber trotzdem um einen funktionalen Ausdruck. Das Verhalten dieses Ausdrucks ist mit der Funktion `map` aus funktionalen Programmiersprachen vergleichbar. Ein Ausdruck wird auf alle Elemente einer Collection angewandt. Auch die Auswertung des for-Ausdrucks liefert eine Collection zurück. Die Elemente der Collection sind die Ergebnisse der Auswertungen von `expression`.

if-Ausdruck

```
if (condition) thenExpression else elseExpression
```

Die Syntax eines if-Ausdrucks ist identisch zu der in Java. Wenn `condition` wahr ist, wird das Ergebnis der Auswertung von `thenExpression` zurückgeliefert, sonst das von `elseExpression`. Innerhalb von for-Ausdrücken ist die Angabe eines Elsezweiges optional. Ist `condition` unwahr, wird kein Element von if zurückgeliefert. Damit läßt sich die Higher-Order-Funktion `filter` realisieren.

Lokale Variablen

```
Type v = expression1; expression2
```

Die Definition von lokalen Variablen erinnert wieder an die Definition in Java. Der Unterschied ist jedoch, dass die Zuweisung nur einmal erfolgen kann. Damit gleicht der Ausdruck eher dem `let`-Konstrukt funktionaler Sprachen.

Spezielle Identifier

```
this
allpackages
packageroot
alltypes
allmethods
```

Von den hiergenannten Identifiern gibt es bislang nur `this`. Er verweist auf die Klasse in deren Kontext die Query ausgewertet wird. Die anderen Identifier sind bislang nur angedacht. `allpackages`, `alltypes` und `allmethods` sollen Collections werden, die alle Packages, Typen und Methoden des Systems als Metamodell-Objekte enthalten. `packageroot` soll die Wurzel der Pakethierarchie sein.

Java-Ausdrücke

```
m.toString.startsWith("get")
```

Innerhalb von Queries können vereinzelt Java-Ausdrücke stehen. So ist es innerhalb einer Query möglich, an einem String die Methode `startsWith` aufzurufen.

Pfad-Ausdrücke

```
for (MyType t: elements) if(constraint) t.feature
```
oder kürzer
```
t.feature[constraint]
```

Um eine kürzere Notation für Ausdrücke zu haben, gibt es sogenannte Pfad-Ausdrücke. Die zweite Notation ist ein Pfadaufdruck. Genauer sollen Pfadausdrücke hier nicht erklärt werden. Eine genaue Beschreibung findet man in [20].

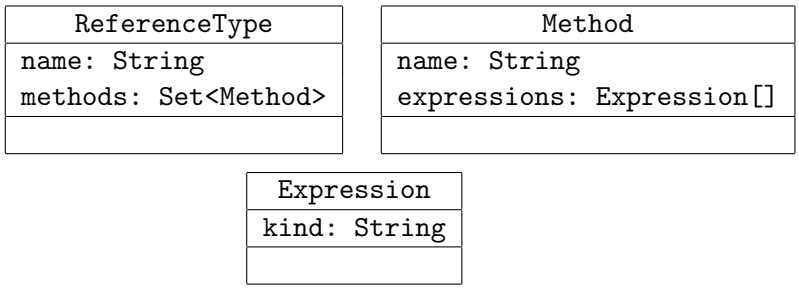

Abbildung 5.1: Klassen des *ObjectTeams/Java*-Meta-Modells.

Queryeinführung

```
query boolean Expression.isExpression() { true }
```

Es ist möglich, Typen des Metamodells um Queries zu erweitern. Diese Definition erweitert den Metamodelltyp `Expression` um die Query `isExpression`. Dieser Mechanismus wird als Queryeinführung (engl. *query introduction*) bezeichnet.

5.4 ObjectTeams/Java-Meta-Modell

Das *ObjectTeams/Java*-Meta-Modell bietet Queries die Möglichkeit, Metainformationen, also Informationen über die Beschaffenheit des Programms, zu erlangen. Das Prinzip ist vergleichbar mit Introspektion, wie es Java-Reflections bieten, nur dass die Metainformationen statisch sind und damit keine Informationen bereitstellen, die erst zur Laufzeit verfügbar sind.

Für die Definition von Queries sind besonders die Typen `ReferenceType`, `Method` und `Expression` von Bedeutung (siehe Abbildung 5.1). `ReferenceType` repäsentiert Klassen und Interfaces und besitzt damit unter anderem die Eigenschaften `name`, den eigenen Name und `methods`, die enthaltenen Methoden. Eine `Method` hat einen Namen `name` und einen Methodenrumpf, der aus `Expressions` besteht. `Expressions` wiederum sind von einer bestimmten Art (*engl.* `kind`). Ein Methodenaufruf ist beispielsweise vom der Art `"MessageSend"`, da Methodenaufrufe mit dem Senden von Nachrichten zwischen Objekten vergleichbar sind. Nach dieser etwas trockenen Einführung in Joinpoint-Queries, soll ein vollständiges Beispiel präsentiert werden:

```
1 query boolean Expression.isGetterCall() {
2         if (kind.equals("MessageSend"))
```

```
3                 toString.startsWith("get")
4 }
5
6 query Set<Expression> getterCalls(ReferenceType t) {
7         for (Method m : t.methods)
8                 m.expressions[isGetterCall]
9 }
```

In Zeile 1 findet eine *Queryeinführung* statt. Der Metamodelltyp `Expression` wird um die Methode `isGetterCall` erweitert. `isGetterCall` liefert `true`, wenn die `kind` der `Expression` gleich `MessageSend` ist und die Stringrepräsentation mit `get` beginnt.

In Zeile 5 wird eine weitere Query definiert. Sie wird diesmal nicht *eingeführt*, sondern in der umgebenen Klasse, die hier nicht angegeben ist, erzeugt. `getterCalls` erhält als Parameter einen Referenztyp, traversiert über alle Methoden von `t`, darin über alle Expressions und liefert alle Aufrufe eines Getters zurück. Um den Code kompakter zu gestalten, wurde ein Pfadausdruck verwendet. Um zu prüfen, ob eine Expression ein Getteraufruf ist, wird die in Zeile 1 in `Expression` *eingeführte* Query `isGetterCall` verwendet.

Die Art einer Expression mit einem Stringvergleich festzustellen, ist aber lediglich eine provisorische Lösung. Es muss möglich sein, die Art einer Expression explizit zu erfassen. Prinzipiell ist das möglich, denn `MessageSend` ist eine der über 120 Subklassen von `Expression` und gehört damit auch zum OT-Metamodell. Allerdings muss die Querysprache erweitert werden. Da die Querysprache in javaähnlicher Syntax entworfen wurde, bieten sich dafür zwei Javakonstrukte an – `instanceof` und Casts:

```
1 query boolean Expression.isMessageSend() {
2         if (e instanceof MessageSend)
3                 true
4         else
5                 false
6 }
7
8 query Set<MessageSend> getMessageSends(ReferenceTyp t) {
9         for (Method m: t.methods)
10                (MessageSend) m.expressions[isMessageSend]
11 }
```

5.5 Queries auswerten

Die Queries müssen irgendwann ausgewertet werden. Es gibt mehrere Varianten, wie dies geschehen kann. Queries können sowohl zur Compile-, als auch zur Ladezeit ausgewertet werden. Die Vor- und Nachteile und die Konsequenzen beider Varianten sollen hier besprochen werden.

Compilezeitauswertung von Queries

Vorteile:

- *Geschwindigkeit*: Da ein Programm öfter gestartet wird, als es compiliert wird, liegt es auf der Hand, dass es performancetechnisch von Vorteil ist, die Queries zur Compilezeit auszuwerten und nicht bei jedem Programmstart.

Nachteile:

- *Unbekannte Klassen*: Es gibt Klassen, die dem Compiler zur Compilezeit nicht bekannt sind. Deshalb kann für diese Klassen nicht festgestellt werden, ob sie Joinpoints enthalten.
- *Overhead*: Es ist mehr Aufwand zu treiben, um der Laufzeitumgebung mitzuteilen, wo sich der gefundene Joinpoint befindet.

Auf die beiden Nachteile wird weiter unten noch genauer eingegangen. Zunächst soll gezeigt werden, welche Möglichkeiten der Compiler hat die Queries auszuwerten. Damit ist vor allem gemeint, welche Informationen der Compiler heranzieht, um einen konkreten Joinpoint zu finden und nicht etwa das Parsen der Queries etc.

Auf den Sourcecode zurückgreifen. Wenn die Queries zur Compilezeit ausgewertet werden, hat der Compiler die Möglichkeit, auf den Quelltext zurückzugreifen. Das hat zwar einen Vorteil, bedeutet jedoch auch eine große Einschränkung, wenn man ausschließlich auf diese Variante setzt.

Vorteil:

- *Leichtere Umsetztbarkeit*: Da die Implementierung der Methoden als abstrakter Syntaxbaum verfügbar ist, können Joinpoints auf einem abstrakteren Niveau gesucht werden.

Nachteile:

- *Nur zur Compilezeit möglich*: Der Quelltext steht nur zur Compilezeit zur Verfügung, daher muss die Auswertung zur Compilezeit erfolgen. Es ist nicht möglich den Code, der die Auswertung durchführt, zur Laufzeit wieder zu verwenden.
- *Basisklassen ohne Quelltext*: Es ist nicht möglich, in Basisklassen nach Joinpoints zu suchen, für die kein Quelltext existiert, obwohl gerade das Adaptieren solcher Klassen einer der Vorteile von *ObjectTeams/Java* ist und bleiben soll.

Auf den Bytecode zurückgreifen. Wenn der Compiler die Queries mit Hilfe des Bytecodes aufwertet, sind die Queries natürlich die gleichen. Nur die Art und Weise, wie der Compiler an die notwendigen Informationen kommt, um die Metainformationen bereitzustellen, ist unterschiedlich. Er muss die Informationen aus dem compilierten Bytecode gewinnen. Ein Unterschied ist z. B. die Bereitstellung von Informationen über die Implementierung einer Methode. Die Implementierung einer Methode ist, wie oben bereits dargestellt, im Metamodell über die Klasse `Method` und deren Feld `expressions` verfügbar. Die `Expressions` aus diesem Feld müssen immer die gleichen sein, egal ob es aus dem Bytecode oder dem Quelltext erstellt wurde. Der Aufwand für den Compiler ist entsprechend größer, weil er nicht auf den AST zurückgeifen kann. Es ist aber notwendig, um die Nachteile zu umgehen.

Klassen die zur Compilezeit nicht existieren. Es gibt Klassen von denen der Compiler zur Compilezeit nicht weiß, dass sie existieren, beispielsweise Laufzeitbibliotheken. Wenn man in diese Klassen Aspektcode weben möchte, muss für sämtliche Queries überprüft werden, ob sie Joinpoints in diesen Klassen haben. Dazu muss die Laufzeitumgebung alle benutzten Queries kennen und die Queries müssen in geeigneter Form an die Laufzeitumgebung übergeben werden. Wie dies geschehen kann, soll im nächsten Kapitel beleuchtet werden. Um Rechenzeit zu sparen, sollten die Queries nicht für Klassen ausgewertet werden, für die sie bereits zur Compilezeit auswertet wurden. Dazu bräuchte man eine Liste aller bekannter Klassen, die zwar lang, aber endlich wäre.

Ladezeitauswertung

Das Problem mit den unbekannten Klassen zeigt, dass sich zur Compilezeit trotz allem Aufwands nicht alle Anwendungsfälle abdecken lassen. Deshalb muss eine umfassende Lösung in der Lage sein, Queries zur Laufzeit anhand des Bytecodes auszuwerten. Dabei gilt das bereits oben gesagte, über die Verwendung von Bytecode zur Auswertung von Queries. Es bietet sich an, Code aus dem Compiler, der für die Auswertung von Queries zuständig ist, an dieser Stelle wiederzuverwenden.

Gegenwärtige Realisierung

Die gegenwärtige Situation in *ObjectTeams/Java* sieht so aus, dass der Compiler die Queries unter Zuhilfenahme des Quelltextes der Basisklassen auswertet, Muster erstellt, die die Joinpoints beschreiben und das ein Matcher in der Laufzeitumgebung diese Muster auswertet, um die Treffer dann an den Weber zu übergeben. Die Muster zur Beschreibung von Joinpoints werden im nächsten Kapitel beschrieben.

Kapitel 6

Muster zur Beschreibung von Joinpoints

Egal wie der Compiler die Queries auswertet – mit Hilfe des Quelltexts oder anhand des Bytecodes – die Ergebnisse der Auswertung müssen irgendwie an die Laufzeitumgebung übergeben werden, denn erst hier soll der Aspektcode in die Basisklasse gewoben werden.

Es wäre möglich, den Joinpoint durch Methodenbezeichner und einen Index zu referenzieren. Der Methodenbezeichner würde eindeutig die Methode bezeichnen, die den Joinpoint enthält. Dazu muss er nur den Paketnamen, den Klassen-Namen, den Methoden-Name und die Methoden-Signatur enthalten. Der Index würde auf einen Bytecode-Befehl in der Methode verweisen. Wenn der Compiler den Bytecode benutzt hat, um den Joinpoint zu finden, stünde der Index sofort zur Verfügung. Würde der Quelletext benützt, wäre es für den Compiler ein geringes Problem, den Index zu berechnen, da der Compiler in der Lage ist, aus dem abstrakten Syntaxbaum der Methode den Bytecode dieser zu generieren. Allerdings lässt sich schwer sicherstellen, dass sich der Code der Basisklasse nicht geändert hat, seit dem Kompilieren der Rollenklasse und der damit verbundenen Queryauswertung. Der Compiler kann nicht immer mitbekommen, dass sich eine Basisklasse geändert hat. Wenn die Basisklasse z. B. Fremdsoftware ist und es ein Update gegeben hat. Deshalb ist es sinnvoll, einen solchen Index als nicht robust genug einzustufen oder ein Verfahren vorzusehen, das erkennt, dass sich der Code geändert hat und nicht mehr der gleiche, wie zur der Zeit ist, als die Rollenklasse kompiliert wurde.

Auf der anderen Seite ist es möglich, ein abstraktes Muster anzugeben, das den Joinpoint beschreibt und robuster ist als ein Index. In diesem Kapitel sollen Muster (engl. *patterns*) zur Beschreibung von Joinpoints vorgestellt werden. Diese Muster sollen vom Compiler eingesetzt werden, um der Laufzeitumgebung mitzuteilen, wo Joinpoints existieren, die durch die Auswertung von Queries gefunden wurden. Zuerst wird erklärt, warum Muster so vorteilhaft

sind. Danach sollen einige Muster vorgestellt werden und dann soll über die Eindeutigkeit und Robustheit von Mustern gesprochen werden. Abschließend wird präsentiert, wie Muster gespeichert werden und auf welchem Weg sie zur Laufzeitumgebung gelangen. Nun zu den Vorteilen:

Performance. Die Queries zur Compilezeit auszuwerten, erspart die Arbeit sie jedes Mal zu Laufzeit auszuwerten zu müssen. Die Patterns dienen in diesem Fall dazu, die Ergebnisse der Auswertung in die Laufzeitumgebung zu transportieren.

Robustheit. Das Ergebnis der Auswertung kann in Form eines einfachen Index gespeichert werden. Sobald sich der Bytecode der Basisklasse in geringster Weise ändert, verweist er nicht mehr auf den Joinpoint. Das Ergebnis kann auch in robusterer Form gespeichert werden, die geringe Änderungen des Bytecodes toleriert. Dies geschieht, indem ein Muster einen Joinpoint in einer abstrakteren Weise beschreibt, als das durch einen Index möglich ist.

Abstraktion. Wenn der Compiler die Queries anhand des Quelltextes auswertet, ist die genaue Bytecodeposition eines Joinpoints nicht bekannt. Der Compiler wird von dem Auswand befreit die genaue Position herauszufinden, indem er den Joinpoint in einer abstrakteren Form beschreibt. Die Muster dienen als abstrahierte Schnittstelle zwischen Compiler und Laufzeitumgebung.

6.1 Muster

Der Compiler erzeugt Muster, die einen Joinpoints beschreiben. Zur Laufzeit werden die Muster ausgewertet, um wieder genau den Joinpoint zu erhalten, den der Compiler ursprünglich gefunden hat. Welche beschreibaren Eigenschaften ein Joinpoint besitzt, wurde in Abschnitt 8.1 gesagt. Diese Eigenschaften können benutzt werden, um einen Joinpoint zu beschreiben. Wie dies in einem Muster geschehen kann und was es sonst noch für Muster geben kann, folgt jetzt:

- *Kind*: Die Angabe des Typs kann dadurch erfolgen, dass man für jede Art von Joinpoint eine Typ-ID definiert und diese dann im Muster benutzt.

- *Signatur*: Da sich die Signatur eines Joinpoints nicht von der Signatur einer Methode unterscheidet, kann sie auch wie diese angegeben werden – durch Aufzählen der Parameter und des Rückgabewertes. Die Typen können dabei durch Strings angegeben werden. Typ-IDs kommen natürlich nicht in Frage.

- *Ziel*: Aus dem Joinpoin-Katalog wird ersichtlich, dass ein Ziel bzw. Target immer ein Feld, eine Methode oder eine Klasse ist. Ein String ist ausreichend, um dieses Ziel eindeutig zu referenzieren.

- *Kontext*: Wie man den Kontext eines Joinpoints beschreibt, ist ein umfangreiches Thema. Denkbar sind Lösungen wie komplexe reguläre Ausdrücke oder die Angabe des Vorgängers oder des Nachfolgers des Joinpoints. Der Compiler könnte ein solches Feature benutzten, um statt einer Positionsangabe den Joinpoint allein durch seinen Kontext zu beschreiben. Für die Queries würde dies nichts ändern. Es wäre nur ein Feature, das in Mustern Verwendung findet. Das BCEL-Bytecodetransformations-Toolkit, das später vorgestellt wird, kann Bytecodebefehle mit Hilfe von regulären Ausdrücken suchen. Wie man diese Fähigkeit nutzt, wird später gezeigt.

- *Position*: Eine Position anzugeben, ist verhältnismäßig einfach. Hierzu reicht eine einfache Zahl. Wie diese Zahl zu interpretieren, ist hängt davon ab, ob diese Zahl – wie oben schon angesprochen – eine Position im Bytecode-Array angibt oder das x-te Auftreten einer bestimmten Joinpointart beschreibt.

- *Unausgewertete Query*: Es wurde bereits gesagt, dass nicht alle Queries zur Compilezeit ausgewertet werden können. Deshalb muss es Muster geben, die diesem Umstand Rechnung tragen. Ob man in diesem Fall noch von Muster sprechen kann, ist eine reine Formulierungssache. Fakt ist, dass man Queries zur Laufzeit ausgewerten können muss. Mit diesem Thema befasst sich der Abschnitt *Queries zur Laufzeit auswerten.*

6.2 Eindeutigkeit

In diesem Abschnitt wird gezeigt, dass Muster, die nur bestimmte Eigenschaften von Joinpoints beschreiben, nicht eindeutig sind. Dann soll erklärt werden, warum eindeutige Muster notwendig sind und wie man sie eindeutig macht.

Selbst wenn alle Eigenschaften eines Joinpoints – außer der Position – durch ein Muster festgelegt werden, ist dies keineswegs eindeutig. Es kann zwei oder mehr Joinpoints in einer Methode geben, die sich in allen Eigenschaften – außer der Position – gleichen. Interessant ist vorallem die Frage, ob es Queries gibt, die von zwei unterschiedlichen – aber sonst

identischen Joinpoints – den einen auswählt, den anderen aber nicht. Man betrachte dazu folgende Query:

```
1 query Set<Expression> getJoinpoints(Method m) {
2         if (m.expressions.length > 0)
3                 m.expressions[0]
4 }
```

Nehmen wir an, eine Methode `foo` enthalte zwei, bis auf die Position, identische Joinpoints. Der erste Joinpoint stehe in der ersten Expression von `foo`. Die Query `getJoinpoints` würde den ersten der beiden Joinpoints zurückliefern. Allerdings wurde bei dieser Query etwas gemogelt. Sie ist in zweierlei Hinsicht nicht richtig. Es fehlt erstens der Elsezweig und zweitens kann das `expressions`-Array nicht mit eckigen Klammern indiziert werden, weil die Klammern für Pfadausdrücke reserviert sind. Aber auch wenn hier gemogelt wurde, muss man beachten, dass die Entwicklung der Querysprache noch nicht abgeschlossen ist. Es ist wahrscheinlich, dass es Queries geben wird, die Joinpoints unterscheiden können, die sich nur in ihrer Position voneinander unterscheiden. Es muss daher möglich sein, einen Joinpoint eindeutig zu beschreiben.

Die erste und einfachste Möglichkeit ist, die Position des Joinpoints durch einen Index, der auf die Position im Bytecodearray zeigt, zu benennen. Dies wird im Folgenden als *absolute Position* bezeichnet. Eine andere Möglichkeit wäre durch eine Zahl, die beschreibt, der wievielte von ansonsten gleichen Joinpoints gemeint ist (z. B. den dritten Methodenaufrufjoinpoint). Dies Art der Positionsangabe soll *relative Position* heißen, weil sie relativ zu gleichen Joinpoints erfolgt.

6.3 Robustheit

In diesem Abschnitt soll die Robustheit von Mustern untersucht werden. Dazu wird zuerst der Begriff Robustheit definiert, dann erklärt, warum Robustheit wichtig ist – nämlich weil sich der Bytecode ändert –, warum sich Bytecode ändert und abschließend wie man Änderungen des Bytecodes erkennen und mit ihnen umgehen kann.

Definition. *Robustheit bezeichnet – im Kontext von Joinpointmustern – wie sicher ein Joinpoint, der durch ein Muster beschrieben wird, wiedergefunden werden kann.*

Wenn sich der Bytecode nicht ändert, kann es nicht passieren, dass ein Joinpoint nicht wiedergefunden wird. Wenn sich der Bytecode jedoch ändert, der Joinpoint jedoch immer noch existiert – nur vielleicht an anderer Stelle –, dann ist es bei robusten Mustern wahrscheinlicher, diesen Joinpoint wiederzufinden.

Wieso ändert sich Bytecode?

Aber warum kann sich der Bytecode eigentlich ändern? Die Antwort ist: Durch Evolution. Wenn eine Basisklasse geändert wird, der Aspekt aber nicht neu kompiliert wird, dann weiß er nichts von der Änderung der Basisklasse. Es kann auch sein, dass die Basisklasse mit einem anderen Compiler oder mit anderen Optionen, zum Beispiel zur Optimierung, neukompiliert wurde. Kann man nicht erkennen, dass die Basisklasse geändert wurde und den Aspekt neukompilieren? Das geht nicht immer oder ist nicht immer erwünscht:

- *Inkrementelle Compilierung*: Quelltext wird inkrementell kompiliert, d. h. wenn sich eine Klasse ändert, soll auch nur diese neu kompiliert werden. Wenn es nötig wird, einen Aspekt neu zu kompilieren, weil sich die Basisklasse – vielleicht unwesentlich – geändert hat, kostet es viel Rechenzeit festzustellen, welche Aspekte überhaupt betroffen sind. Auch der Kompilierungsvorgang wird verkompliziert.
- *Updates*: Bei Updates von Basisklassen oder wenn unterschiedliche Bibliotheken zum Einsatz kommen, muss nur eine Version der kompilierten Software zur Verfügung stehen, weil sie die Joinpoints in unterschiedlichen Versionen wieder findet.

Robustheit der Muster

Die Frage in diesem Abschnitt ist, wie robust sind die bis jetzt vorgestellten Muster. Am wenigsten robust ist offenbar ein Index in das Bytecode-Array. Wenn der Joinpoint nur um einen Befehl verrutscht, kann der Joinpoint nicht wiedergefunden werden. Die anderen Muster sind schon robuster. Hier ist die exakte Stelle egal, nur an der Reihenfolge der Joinpoints darf sich nichts ändern.

Trotz robuster Muster kann nicht garantiert werden, dass der Joinpoints bei verändertem Code wiedergefunden wird. Deshalb ist es notwendig, diesen Umstand zu erkennen. Dazu muss zum einen erkannt werden, dass der Bytecode geändert wurde und zum anderen, dass der Joinpoint nicht auffindbar ist. Die einzige sichere Lösung, um einen Joinpoint wiederzufinden ist, die Query erneut auszuwerten, wenn sich der Bytecode geändert hat.

Veränderungen erkennen

Auch sehr robuste Muster können einen Joinpoint nicht immer wiederfinden. Die Frage ist also, wie können Veränderungen des Bytecodes erkannt werden.

Trefferanzahl. Wenn in einem Muster vermerkt ist, wieviele Joinpoints eines Typs in der Methode enthalten sind und der wievielte davon der Gesuchte ist, dann kann an gewissen Unstimmigkeiten erkannt werden, dass sich der Bytecode geändert hat.

- *Der Joinpoint ist nicht auffindbar*: Der Compiler hat den Joinpoint gefunden – sonst hätte er das Muster nicht erzeugt –, also muss sich der Bytecode geändert haben. Denn die Muster sind so konstruiert, dass sie bei der Auswertung exakt den gleichen Joinpoint finden, den der Compiler gefunden hat.

- *Anzahl der gleichen Joinpoints stimmt nicht*: Auch in diesem Fall ist klar, dass sich der Bytecode geändert haben muss. Der Compiler hat mehrere Joinpoints mit den gesuchten Eigenschaften gefunden und diese Zahl im Muster vermerkt. Diese Zahl stimmt aber nicht mit der aktuellen Zahl überein. Also muss eine Veränderung stattgefunden haben.

Selbstverständlich kann sich der Bytecode so verändern, dass die Anzahl der Joinpoints gleich bleibt, der gesuchte Joinpoint aber seine relative Position geändert hat. Dann ist der gefundene Joinpoint nicht der ist, der zur Compilezeit gefunden wurde. In diesem Fall ist die Veränderung des Bytecodes nach dem obigen Verfahren nicht feststellbar.

CRC. Das eben vorgestellte Verfahren ist also zu unsicher. Deshalb soll ein Verfahren vorgestellt werden, das zuverlässig Veränderungen des Bytecodes erkennt. Dieses Verfahren arbeitet mit Prüfsummen, kurz CRC (*Cyclic Redundancy Check*).

Der Compiler erstellt eine CRC des Bytecodearray einer Methode und speichert sie zusammen mit dem Muster ab. Die Laufzeitumgebung überprüft diese Prüfsumme noch bevor sie das Muster auswertet. Veränderungen werden so mit sehr hoher Wahrscheinlichkeit erkannt[1].

[1] Die Wahrscheinlichkeit, dass eine Änderung nicht erkennt wird, ist viel niedriger als die Wahrscheinlichkeit, dass die Software durch einen zufälligen Hardwarefehler abstürzt.

Behandlung

In diesem Abschnitt soll uns beschäftigen, welche Möglichkeiten es gibt, wenn eine Veränderung des Bytecode festgestellt wurde. Es kann gundsätzlich nicht sichergestellt werden, dass das Programm – selbst wenn die Query neu ausgewertet wird – noch wie erwartet funktioniert. Die Änderungen in der Basisklasse können einfach zu groß sein und eine Änderung der Query oder gar des ganzen Aspektes fordern. In diesem Fall würde natürlich selbst ein Neukompilieren des System das Problem nicht beheben. Eine Warnung von der Laufzeitumgebung ist auf jeden Fall angebracht. Gehen wir einen Moment davon aus, dass ein Neukompilieren das Problem beheben würde. Dann ist es, wenn die Bytecodeänderung erkannt wurde, möglich, das Problem auch ohne Neukompilierung zu lösen. Alles was zu tun ist, ist die Query neu auszuwerten.

6.4 Queries zur Laufzeit auswerten.

Um Queries zur Laufzeit auszuwerten, müssen die Queries entweder an die Laufzeitumbegung übergeben werden oder der Compiler erzeugt Methoden mit denen man die Queries auswerten kann.

Wenn die Queries übergeben werden sollen, dann müssen sie in irgendeiner Form abgespeichert und per Bytecodeattribut in den Klassendateien gespeichert werden. Als Format bietet sich entweder Klartext oder eine bereits geparste Form, also der abstrakte Syntaxbaum (AST), an. Die Frage ist allerdings, in welcher Klasse die Queries abgespeichert werden. Wenn man sie in der gleichen Klasse K_1 speichert, wo sie auch definiert werden, was am sinnvollsten wäre, dann kann der Fall auftreten, dass eine Query eine andere Query, in einer anderen Klasse K_2, aufrufen möchte. Dann müsste zuvor die Klasse K_2 geladen werden. Was aber macht man, wenn K_2 eine Query aus K_1 aufrufen möchte? Außerdem ist das Speicherformat der AST-Lösung sehr komplex und beide Varianten erfordern, dass große Teile des Compilers, in der Laufzeitumgebung verfügbar sind.

Deshalb soll die andere Lösung propagiert werden, also vom Compiler erzeugte Methoden mit denen man eine Query auswerten kann. Alles was man wissen muss, um die Query auszuwerten, ist ihre Klasse und ihr Name. Die Klasse, die die Query enthält, definiert eine Methode mit dem gleichen Namen wie die Query. Diese Methode nimmt als Parameter den Bytecode der Klasse entgegen, in der nach Joinpoints gesucht werden soll. Als Ergebnis liefert sie beispielsweise die Methoden, die die Joinpoints enthalten und einen Index in das Bytecodearray dieser Methoden. Die Funktionalität, Queries anhand des Bytecodes auszuwerten muss sowieso in Compiler und Laufzeitumgebung verfügbar sein.

Wenn ein Verfahren, das mit Prüfsummen und Neuauswertung von Queries zur Laufzeit arbeitet, eingesetzt wird, könnte die Verwendung einiger Muster überflüssig werden. Der Compiler speichert einfach einen Index auf den Bytecodebefehl, der den Joinpoint repräsentiert und eine CRC. Stimmt die CRC nicht – was eher selten passieren dürfte – wird die Query neu ausgewertet. Robustheit ist kein Thema mehr. Mit dem Verfahren ist es außerdem leicht möglich, Queries für Klassen auszuwerten, die bis dato unbekannt waren. Wenn sich die Laufzeitumgebung die Queries merkt, die für alle noch zuladenden Klassen ausgewertet werden müssen, dann kann dies beim Laden einer Klasse nach dem obigen Verfahren geschehen. Das ist allerdings noch Zukunftsmusik.

6.5 Muster speichern

Dieser Abschnitt soll erklären, wie Muster an die Laufzeitumgebung übergeben werden können. Dazu werden zuerst Java-Bytecode-Attribute erklärt und dann eigene Bytecode-Attribute definiert, die es ermöglichen, Muster zu speichern.

Bytecode-Attribute

Bytecode-Attribute wurden bereits im Kapitel *Java* angesprochen. Es ist möglich, neben vordefinierten Attributen auch eigene Attribute zu definieren. Alle Attribute haben das gleiche Format. Es ist in Abbildung 3 dargestellt. Die Notation stammt aus der *JVM Spezifikation* und ist an *Structs* der Programmiersprache *C* angelehnt[2] [2, 21]:

```
1 attribute_info {
2     u2 attribute_name_index;
3     u4 attribute_length;
4     u1 info[attribute_length];
5 }
```

Listing 3: Allgemeiner Aufbau von Bytecode-Attributen, als Struktur dargestellt.

- *Zeile 1*: `attribute_info` ist der Name dieser Struktur. Die Namen von Strukturen werden im Folgenden benutzt, um in der Definition einer Struktur auf andere Strukturen verweisen zu können. Es sei nochmal darauf hingewiesen, dass die angegebene

[2]Structs werden in *C* benutzt, um eigene Datentypen zu definieren. Man kann sich darunter eine Klasse vorstellen, die nur `public`-Felder enthält.

Offset	*Inhalt*
0x00	*Attribut-Name (Index)*
0x02	*Daten-Länge*
0x06	*Daten*
...	...

Abbildung 6.1: Allgemeines Attribut in tabellarischer Darstellung.

Syntax keine praktische Bedeutung hat. Sie wird nur in der *JVM Spezifikation* verwendet [21].

- *Zeile 2 (Name)*: Der erste Eintrag ist der Attribut-Name, wobei `attribute_name_index` in den Konstantenpool verweist, in dem der Name abgelegt ist. Der Name kann eine beliebige Zeichenkette sein. `u2` ist eine 2-Byte-Ganzzahl.
- *Zeile 3 (Größe)*: Der Eintrag `attribute_length` gibt die Länge der Daten in Bytes an. Die Länge wird als 4-Byte-Ganzzahl `u4` angegeben.
- *Zeile 4 (Daten)*: Zuletzt kommen die Daten, die einfach in einem Byte-Array der Länge `attribute_length` abgelegt werden.

Abbildung 6.1 zeigt zum besseren Verständnis den Aufbau von Attributen noch einmal in tabellarischer Form.

Muster in Bytecode-Attributen speichern

Der Compiler funktioniert zur Zeit so, dass er für jede Methode die gebundene Joinpoints enthält, ein Attribut erzeugt. Dieses Attribut trägt den Namen `OTJoinPoints` (Listing 4). In den Zeilen 4-6 wird eindeutig – durch Klassen-, Methodennamen und -signatur – die Methode definiert, die den gesuchten Joinpoint enthält. In Zeile 7 steht die Anzahl der Muster, die danach folgen.

Das einzige Muster, das zur Zeit vom Compiler erzeugt wird, ist ein Muster das Art, Ziel und Signatur eines Joinpoints beschreibt (Listing 5). Es enthält als erstes den Querynamen der Query, die den Joinpoint gefunden hat (Zeile 2). Dann kommt eine ID, um die Art des Musters unterscheiden zu können (Zeile 3). Dadurch können auch anders aufgebaute Muster definiert werden. Der nächste Eintrag ist wieder eine ID (Zeile 4). Diesmal bezeichnet die Art des Joinpoint, nach dem gesucht werden soll. Jede Joinpointart besitzt eine eigene

ID. Bisher kennt der Compiler nur drei Arten – Methodenaufrufe, lesen und schreiben von Feldern. In den Zeilen 5-6 werden Ziel und Signatur des Joinpoints definiert. Es sind Indizes in den Konstantenpool, wo die entsprechenden Strings liegen.

```
1 OTJoinPoints {
2     u2 attribute_name_index;     // index to "OTJoinPoints"
3     u4 attribute_length;
4     u2 location_class_index;     // fully qualified class name
5     u2 location_selector_index;  // method selector (name)
6     u2 location_signature_index; // method signature
7     u2 joinpoint_patterns_count;
8     OTJoinPointPattern joinpoint_patterns[joinpoint_patterns_count];
9 }
```

Listing 4: Das benutzerdefinierte Attribut `OTJoinPoints`.

```
1 OTSignatureJoinPointPattern {
2     u2_query_name_index; // index to the query name
3     u1 pattern_kind;     // == 1
4     u1 joinpoint_kind;   // MethodCall=0, FieldGet=1, FieldSet=2
5     u2 pattern_name_index;
6     u2 pattern_signature_index;
7 }
```

Listing 5: Muster, das Ziel und Signatur eines Joinpoints beschreibt.

Das Muster in Listing 5 berücksichtigt noch nicht die in der obigen Diskussion gewonnenen Erkenntnisse. Deshalb soll an dieser Stelle ein weiteres Muster vorgeschlagen werden (Listing 6). Das Muster `OTAbsPositionCRCJoinPointPattern` beschreibt die Position durch einen Index ins Bytecodearray (Zeile 5). Um Robustheit zu garantieren, wird ebenso eine CRC definiert (Zeile 6). Nur wenn die CRC des Bytecodes mit der CRC im Muster übereinstimmt, darf der Index verwendet werden. In allen anderen Fällen muss die Query neu ausgewertet werden. Wie das geht, wurde bereits beschrieben.

Eine nicht ausgewertete Query könnte, wie in Listing 7 definiert, aussehen. Wenn ein solches Muster geladen wird, merkt sich die Laufzeitumgebung, dass es eine nicht ausgewertete Query gibt und wertet sie beim Laden neuer Klassen aus. Wie das geht, wurde bereits in Abschnitt 6.4 beschrieben.

```
OTAbsPositionCRCJoinPointPattern {
    u2_query_name_index;
    u1 pattern_kind;
    u1 joinpoint_kind;
    u2 byte_code_index;
    u4 crc;
}
```

Listing 6: Muster, das einen Joinpoint durch die absolute Position beschreibt.

```
OTUnevaluatedQueryJoinPointPattern {
    u2_query_name_index;
    u1 pattern_kind;
}
```

Listing 7: Muster für eine nicht ausgewertete Query.

Kapitel 7

Die ObjectTeams-Laufzeitumgebung

In diesem Kapitel soll beschrieben werden, wie die *ObjectTeams*-Laufzeitumgebung (OTRE) funktioniert. Denn diese muss geändert werden, um Joinpointinterception für OT/J zu implementieren. Dazu folgt zuerst ein Überblick über die Architektur von OT/J, mit anschließender genauerer Betrachtung der einzelnen Bestandteile der Laufzeitumgebung.

7.1 Architektur

Abbildung 7.1 zeigt die Architektur von *ObjectTeams/Java*, wie sie in [4] beschrieben wird. *ObjectTeams/Java* besteht nicht nur aus der Laufzeitumgebung, sondern auch aus einem eigenen Compiler. Dieser besteht aus einem Standard-Java-Compiler und einer *ObjectTeams* spezifischen Erweiterung. Damit können normale Java-Quelltexte und *ObjectTeams*-Quelltext compiliert werden. Der Compiler erzeugt normale class-Dateien mit Java-Bytecode. *ObjectTeams* relevante Informationen sind in Form von Bytecode-Attributen darin abgelegt.

Obwohl der Compiler standardkonforme Klassendateien erzeugt, können diese nicht ohne weiteres in einer normalen Java-Laufzeitumgebung (JRE) ausgeführt werden. Es bedarf einer speziellen Laufzeitumgebung – des OTRE. Das OTRE besteht, wie das JRE, aus der JVM und der Java-Standardbibliothek. Allerdings sind zusätzlich Mechanismen vorhanden, um den Bytecode der Klassen transformieren zu können. Der Bytecode wird vom *Classloader*, der die Klassen bei Bedarf lädt, zur Ausführung an den *OTRE-Transformationstreiber* `ObjectTeamsTransformer` übergeben. Erst nachdem der Treiber den Bytecode transformiert hat (wenn das nötig war), wird er an die *Execution Engine* übergeben (Abbildung 7.1).

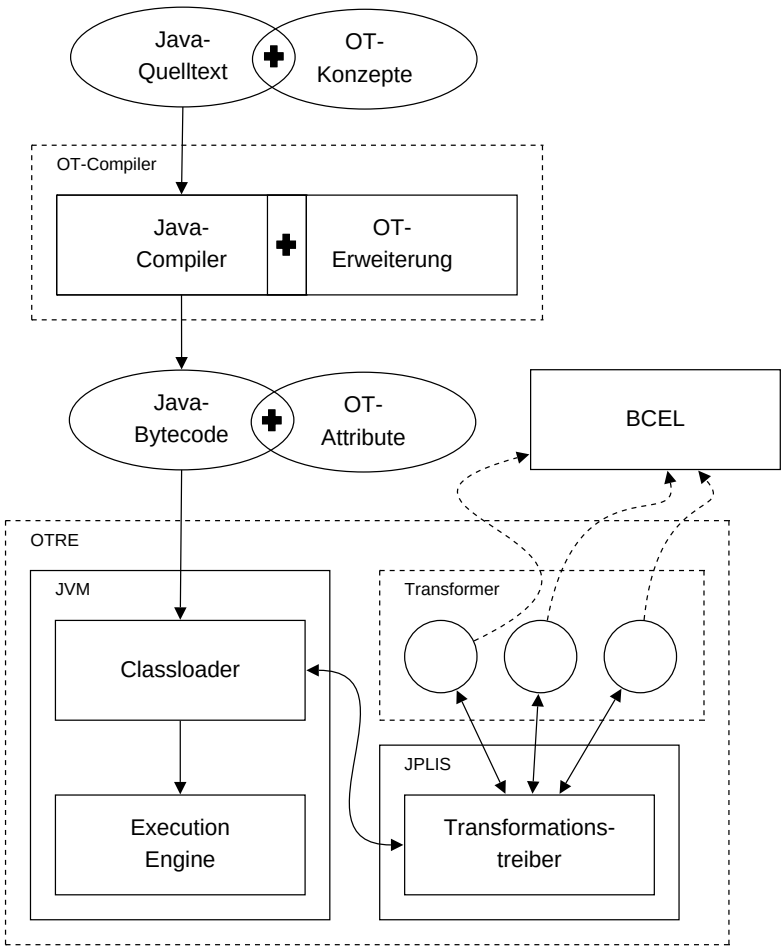

Abbildung 7.1: Architektur von *ObjectTeams/Java*.

Der Transformationstreiber ändert, wie die Bezeichnung Treiber vermuten lässt, den Bytecode nicht alleine, sondern er führt die Änderungen mit Hilfe von zahlreichen Transformern durch. Die Transformer verwenden ihrerseits eine Bibliothek für Bytecodemanipulationen (BCEL). Diese einzelnen Schritte sollen im Folgenden etwas näher beschrieben werden.

7.2 Zugriff auf den Bytecode

Seit *Java J2SE 5.0* besteht standardmäßig die Möglichkeit, den Bytecode von Klassen zur Lade- und Laufzeit zu ändern. Dazu definiert man einen sogenannten *Agent*, der einen *JPLIS*-Transformer bei der JVM registriert. Der Transformationstreiber des OTRE ist ein solcher *JPLIS*-Transformer. Damit er vom *Classloader* aufgerufen wird, muss der Treiber also erst bei der JVM registriert werden. Dazu existiert ein sogenannter Agent, der die Methode `premain` besitzt:

```
public static void premain(String agentArgs, Instrumentation inst);
```

Damit `premain` noch vor dem Aufruf der `main`-Methode aufgerufen wird, wird der Agent in die jar-Datei `otre_agent.jar` gepackt. Das *Manifest* der jar-Datei definiert eine Premainmethode. Dann wird die jar-Datei beim Start der Laufzeitumgebung an diese übergeben:

```
java -javaagent:otre_agent.jar ...
```

Der *Agent* macht eigentlich nichts weiter, als den Transformationstreiber `ObjectTeamsTransformer` bei der JVM zu registieren:

```
inst.addTransformer(new ObjectTeamsTransformer());
```

Damit ist der Transformationstreiber registiert. Der Treiber wird nun vom *Classloader* aufgerufen, deshalb implementiert er das Interface `java.lang.instrument.ClassFileTransformer`. Das Interface enthält nur eine Methode:

```
byte[] transform(
        ClassLoader loader,
        String className,
        Class<?> classBeingRedefined,
        ProtectionDomain protectionDomain,
        byte[] classfileBuffer) throws IllegalClassFormatException
```

Besonders interessant ist der Parameter `classfileBuffer`. Über diesen kann die Klasse eingelesen werden. Die neue Klasse wird von der Methode als `byte[]` zurückgeliefert.

Der geneigte Leser findet weitere Informationen über *Instrumentation* in der Java-API-Dokumentation [22] für das Paket `java.lang.instrument`.

Weil eine class-Datei einen relativ komplexen Aufbau hat und Bytecode komplexe interne Zusammenhänge, ist es aufwendig und fehleranfällig, Code zu schreiben, der die Änderungen auf sehr niedrigem Abstraktionsniveau durchzuführt – also direkt auf dem Bytecode. Glücklicherweise gibt es eine Bibliothek, die Funktionen zur komfortablen Analyse, Manipulation und Erzeugung von class-Dateien bzw. Bytecode bereit stellt – die *Bytecode Engineering Library* (BCEL) [1].

7.3 Transformation

Der Transformationstreiber erzeugt aus dem an ihn übergebenen Bytecode eine BCEL-Repräsentation. Diese wird nacheinander an verschiedene Transformer übergeben, die alle notwendigen Transformationen durchführen. Danach wird aus der BCEL-Repräsentation wieder Bytecode generiert und an den *Classloader* zurückgegeben. An dieser Stelle sollen die wichtigsten Transformationsschritte erklärt werden, weil sie im Rahmen dieser Diplomarbeit erweitert werden müssen.

Transformer. Die Transformation ist nach der *Pipes-and-Filters-Architektur* realisiert. Der Datenstrom ist ein `ClassGen`-Objekt, das ist die BCEL-Repäsentation einer Klasse. Alle Transformationsschritte werden nacheinander durchgeführt. `ObjectTeamsTransformer` ruft dazu nacheinander, alle ihm bekannten Transformer auf. Hier ist auch die Stelle, um *ObjectTeams* um neue Transformationsschritte zu erweitern.

Alle Transformer sind Ableitungen der abstrakten Superklasse `ObjectTeams-Transformation` – nicht zu verwechseln mit `ObjectTeamsTransformer`. Bevor sie mit der eigentlichen Transformation beginnen, stellen sie sicher, dass alle Attribute der Klasse eingelesen werden.

Bytecode-Attribute. Bytecode-Attribute dienen in *ObjectTeams/Java* als die Schnittstelle zwischen Compiler und Laufzeitumgebung. Alle für die Transformation relevanten Daten sind als Bytecode-Attribute abgelegt. Deshalb werden, bevor die eigentliche Transformation durchgeführt wird, alle Bytecode-Attribute der zu transformierenden Klasse eingelesen und teilweise ausgewertet. Dies geschieht, indem ein Transformer die von seiner Superklasse geerbte Methode `checkReadClassAttributes` aufruft.

`checkReadClassAttributes` ruft die Methode `scanClassOTAttributes` auf, die das eigentliche Einlesen durchführt, alle *ObjectTeams*-Attribute werden von dieser Methode eingelesen. Diese detaillierte Angabe erfolgt deshalb, weil eine der Methoden geändert werden muss, wenn man neue Attribute einführen möchte und die andere von einem neuen Transformer aufgerufen werden muss.

Ladereihenfolge. Ein andere Aufgabe der Methode `scanClassOTAttributes` ist es, festzustellen, ob es nötig ist, vor der aktuellen Klasse, andere Klassen zu laden. `scanClassOTAttributes` liefert eine Liste aller Klassen zurück, die *vor* der aktuellen Klasse geladen werden müssen.

Transformation. Nachdem alle diese Arbeiten erledigt sind, kann die eigentliche Transformation beginnen. Welche Transformationen im Einzelnen durchgeführt werden, soll uns an dieser Stelle nicht interessieren, bei Bedarf wird darauf eingegangen.

Kapitel 8

Finden von Joinpoints

Dieses Kapitel soll beschreiben, wie anhand eines gegebenen Musters, der darin beschriebene Joinpoint gefunden wird. Der Vorgang des Findens wird im Folgenden auch als *Matching* bezeichnet. Danach soll beschrieben werden, wie die *ObjectTeams*-Laufzeitumgebung prototypisch um die vorher beschriebenen Konzepte des Matchings erweitert wurde. Die Laufzeitumgebung wurde dazu um einen *Transformer* erweitert und in die Lage versetzt, die neu eingeführten Bytecode-Attribute zu lesen.

8.1 Joinpoint-Eigenschaften

Bevor man Muster zur Suche nach Joinpoints definieren kann, muss man überlegen, welche Eigenschaften ein Joinpoint eigentlich hat, die man beschreiben kann. Es wurden die Eigenschaften *Art, Ziel, Signatur, Kontext und Position* identifiziert:

- *Art*: Jeder Joinpoint hat eine Art. Ein Methodenaufruf-Joinpoint hat die Art Methodenaufruf. Die Art resultiert aus dem Hochsprachkonstrukt des Joinpoints und geht aus dem Joinpointkatalog, aus dem Namen des Joinpoints hervor (Anhang A).
- *Signatur*: Man kann jedem Joinpoint eine Signatur zuordnen, genau wie dies bei Methoden passiert. Ein Methodenaufruf-Joinpoint hat dann die gleiche Signatur, wie die Methode selbst. Feldzugriff-Joinpoints haben die gleiche Signatur, wie eine entsprechende getter/setter-Methode. Ein lesender Arrayzugriff (`a[23]`) auf ein Array des Typs `T` hat zum Beispiel die Signatur `T (T[], int)`. Parameter sind das Array

und ein Index. Der Rückgabewert hat den Typ `T`. Der Joinpointkatalog in Anhang A enthält für jeden Joinpoint einen Signatureintrag, in dem seine Signatur angegeben wird. Zu beachten ist, dass bei Methodenaufruf-Joinpoints der implizite Parameter this explizit aufgelistet ist.

- *Target*: Viele Joinpoints haben ein Ziel, auf das sie zugreifen. Bei Methodenaufrufen wäre das die aufgerufenene Methode. Bei Feldzugriffen das Feld. Bei Casts könnte man gewissermaßen den Zieltyp des Casts als Ziel sehen, also die Klasse. Die Ziele aller Joinpoints sind im Joinpointkatalog beschrieben.
- *Kontext*: Ein Joinpoint kann in einem bestimmten Kontext stehen, z. B. hinter einem Inkrement einer Schleife, in der Bedingung einer Fallunterscheidung.
- *Position*: Die Position muss keine Position im Bytecode-Array der Methode sein. Wenn einige der obigen Eigenschaften angegeben sind, kann sie auch durch abzählen angegeben werden (beispielsweise der fünfte Methodenaufruf).

Die Eigenschaften *Art, Ziel und Signatur* eignen sich, um sie im Joinpointkatalog in Anhang A zu beschreiben, deshalb hat jeder der Joinpoints im Katalog einen entsprechenden Eintrag. Im Folgenden soll beschrieben werden, wie ein Joinpoint mit der angegbenen Eigenschaft gefunden werden kann.

Joinpoints einer Art finden

Nach einer bestimmten Art von Joinpoints zu suchen ist relativ einfach. Viele Joinpoints sind eindeutig durch einen Bytecode-Befehl identifizierbar. Doch auch wenn sie das nicht sind, so deutet zumindest ein Bytecode-Befehl auf ihre Existenz hin. Durch weitere Überprüfungen kann ein Joinpoint dann als solcher indentifiziert werden. Bytecode-Befehle, die auf die Existenz eines bestimmten Joinpoints hinweisen, sollen im Folgenden dessen *Spur* genannt werden.

Die *Spur* für jeden Joinpoint ist in der Joinpoint-Liste angegeben, zusammen mit der Information, ob die Spur eindeutig ist. Wenn dies nicht der Fall ist, erfolgt die Angabe, welche weiteren Analysen nötig sind, um festzustellen, ob es sich um einen Joinpoint der gesuchten Art handelt.

Es sei noch angemerkt, dass unterschiedliche Joinpoints die gleiche *Spur* haben können. Das passiert, weil unterschiedliche Joinpoints zusammen auftreten. Beispielsweise treten Dereferenzierung und Feldzugriff immer zusammen auf. Außerdem kann der gleiche Joinpoint unterschiedliche Spuren haben. Der Aufruf einer Instanzmethode kann beispielsweise durch

`invokevirtual` oder `invokespecial` realisiert werden, je nachdem, ob es sich um den Aufruf einer öffentlichen oder privaten Methode handelt. Es muss dann nach dem Auftreten einer der Spuren gesucht werden.

Zugriffsziel eines Joinpoints

Ein Muster beschreibt nie nur das Zugriffsziel eines Joinpoints, sondern gibt immer auch dessen Art an, deshalb wird zuerst ein Auftreten eines Joinpoints der gegebenen Art gesucht – nach obiger Methode – und dann das Zugriffsziel des Joinpoints mit dem mit Muster angegebenen verglichen. Zugriffsziel kann eine Methode, eine Klasse oder ein Feld sein. Ein Muster verweist auf sie mit einer Zeichenkette. Was das Ziel eines Joinoints ist, steht im Eintrag *Ziel* des Joinpoint-Katalog (Anhang A). Der Rest ist einfacher Stringvergleich.

Signatur eines Joinpoints

Wie man die Signatur eines Joinpoints bestimmt, wird für jeden Joinpoint im Joinpointkatalog beschrieben – im Eintrag Signatur. Die Signatur ist im Muster als einfacher String angegeben. Die Verwendung von Signaturen in Mustern ist, auch ohne Angabe eines Ziels, sinnvoll.

Kontext eines Joinpoints

Dieses Thema ist, wie schon angesprochen, sehr umfangreich und soll in dieser Arbeit nicht in aller Tiefe analysiert werden. Dennoch soll an dieser Stelle motiviert werden, wie ein Joinpoint gesucht werden kann, dessen Kontext man kennt.

Der Kontext kann angegeben sein durch einen regulären Ausdruck, dessen Terminalsymbole einfache Bytecode-Befehle sind. Auf Grund der linearen Struktur von Bytecode können solche regulären Ausdrücke gut ausgewertet werden. Die Muster, die Bytecode-Befehle in einem Ausdruck benutzen, verlieren jedoch an Abstraktion. Ermöglicht man höhere Abstraktion, wie beispielsweise: „Ein Joinpoint innerhalb eines `if`“, dann muss, um den Joinpoints zu finden, eine nicht unerhebliche Analyse des Bytecodes erfolgen, die einer Dekompilierung nahe kommt.

Im Kapitel über die Implementierung des Matchings soll beschrieben werden, wie ein weniger abstraktes Verfahren realisiert werden kann. Also ein Verfahren, das den Kontext eines Joinpoints mit regulären Ausdrücken beschreibt, dessen Terminale Bytecodebefehle sind.

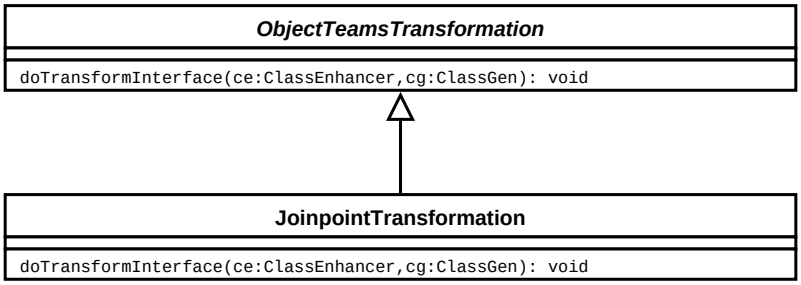

Abbildung 8.1: Alle Transformer sind Unterklassen von ObjectTeamsTransformation.

Position eines Joinpoints

Einen Joinpoint anhand eines Index ins Bytecode-Array zu finden, ist trivial und bedarf keiner weiteren Erläuterung. Wenn die Position relativ zu anderen Treffern angegeben wird (der x-te Treffer), dann ist die Suche auch nicht schwer. Es werden alle Joinpoints gesucht und dann der x-te ausgewählt. Dieses Verfahren funktioniert für alle Joinpoints gleich.

Queries zur Lade- oder Laufzeit auswerten

Es ist vorstellbar, dass es Patterns gibt, die Queries enthalten, die noch nicht ausgewertet sind. Die Auswertung muss dann zur Laufzeit passieren. Das Auswerten erfolgt nur für die aktuelle Klasse. Wie Laufzeitauswertung funktioniert, wurde in Abschnitt 6.4 besprochen.

8.2 Neuer Transformer

Zuerst wird ein neuer Transformer erstellt. Er erhält den Namen `JoinpointTransformation`. Dies passiert, wie in Abschnitt 7.3 beschrieben, durch Erben der abstrakten Klasse `ObjectTeamsTransformation` und Redefinition der Methode `doTransformInterface` (siehe Abbildung 8.1).

Damit der neue Transformer auch aufgerufen wird, muss der Transformationstreiber `ObjectTeamsTransformer`, um einen Aufruf des neuen Transformers ergänzt werden. Der Aufruf der Joinpoint-Transformation muss relativ früh erfolgen, weil die vom Joinpoint-Transformer erstellten Wrapper-Methoden (später mehr) durch bereits existierende Transformer weiter transformiert werden sollen:

```
1 private JoinPointTransformation joinPointTransformation =
2         new JoinPointTransformation();
3 // ...
4 joinPointTransformation.doTransformInterface(jpe, cg);
5 // alle weiteren Transformeraufrufe
6 // ...
```

Im Transformationstreiber wird eine Instanz des neuen Transformers erzeugt (Zeile 1 und 2) und als erster aller Transformer aufgerufen (Zeile 4).

8.3 Neue Bytecode-Attribute

Die *ObjectTeams*-Laufzeitumgebung liest die Bytecode-Attribute, wie wir bereits gesehen haben, alle an einer zentralen Stelle ein: In der Methode `scanClassOTAttributes` der Transformer-Superklasse `ObjectTeamsTransformation`. Hier ist der richtige Ort, um neue Attribute hinzuzufügen und bestehende Attribute zu ändern.

OTJoinPoints-Attribut. Um das neue Attribut `OTJoinPoints` – zur Speicherung der Patterns – einzulesen, wird diese Methode um einen Fall erweitert:

```
} else if (attrName.equals("OTJoinPoints")) {
        OTJoinPoints otJPs =
                new OTJoinPoints(indizes, cg.getConstantPool());
        JoinPointRepository.getInstance().insert(otJPs);
}
```

Aus der Byte-Array-Repräsentation der Patterns (`indizes`) wird ein `OTJoinPoints`-Objekt erzeugt. Dieses Objekt enthält mehrere Patterns, die einer Methode zugeordnet sind. Die Muster werden in abstrakter Form, als `JoinpointPattern` – einer Superklasse für alle Muster – referenziert. Außerdem enthält `OTJoinPoints` einen eindeutigen Verweis auf die Methode, in der nach den Joinpoints gesucht werden soll, also Klassenname, Methodenname und Signatur.

Weil die eingelesenen Informationen erst später benötigt werden, werden sie in dem `JoinPointRepository` gespeichert. Es enthält zu einer Methode alle Joinpoint-Muster. Intern werden vollständig qualifizierte Methodennamen auf

mehrere `OTJoinPoints`-Objekte abgebildet. Das passiert mit einer Hashmap: `HashMap<String, Collection<OTJoinPoints>>()`.

CallinMethodMappings-Attribut. Das Attribute `CallinMethodMappings`, das speichert, welche Rollenmethode an welchen Joinpoint gebunden werden soll, wurde um ein Flag `isQuery` erweitert, um zu unterscheiden, ob es sich bei der rechten Seite der Callinmethodenbindung um eine Query oder einen Basismethodenbezeichner handelt. Der Fall Basismethodenbezeichner ist für uns nicht interessant, er wird bereits jetzt vom OTRE behandelt. Wenn es sich jedoch um eine Query handelt, wird ein Aufruf von `JoinPointRepository.addQueryBinding` ergänzt:

```
1 if (!isQuery)
2         ...
3 else
4         JoinPointRepository.addQueryBinding(className, ...);
```

`addQueryBinding` sorgt dafür, dass die Querybindung gespeichert wird, denn auch sie soll erst später verwendet werden, nämlich beim Weben. `addQueryBinding` erzeugt zu diesem Zweck ein `CallinMethodMappings`-Objekt, das alle wichtigen Informationen aus dem CallinMethodMappings-Attribut enthält, und legt es – referenziert durch den Querynamen – in einer Hashmap ab. Diesmal hat die Hashmap folgende Signatur: `HashMap<String, Collection<CallinMethodMapping>>`.

Damit sind alle für uns wichtigen Attribute eingelesen und für die spätere Verwendung aufbereitet.

Ladereihenfolge. Das Matching und Weben einer Klasse soll zu deren Ladezeit durchgeführt werden. Deshalb müssen alle OTJoinPoints- und CallinMethodMappings-Attribute, die die Klasse betreffen, eingelesen worden sein. Wenn nicht, dann müsste nach dem Finden weiterer Queries bzw. Querybindungen, weiterer Aspektcode in die entsprechenden Klassen gewoben werden. Obwohl Mehrfachweben möglich ist, möchte man doch, wenn es nicht unbedingt nötig ist, darauf verzichten. Mehr zum Thema Mehrfachweben steht in Abschnitt 10.5.

Für den Prototypen ist die derzeitige Ladereihenfolge ausreichend, weil Joinpointinterception (JPI) die gleiche Ladereihenfolge erfordert, wie Methodcallinterception (MCI). Da MCI vom OTRE bereits unterstützt wird, wird die Ladereihenfolge als ausreihend betrachtet. Es sei allerdings darauf hingewiesen, dass die derzeitige Releaseversion von *ObjectTeams/Java* einige pathologische Fälle besitzt, in denen die notwendige Ladereihenfolge nicht sichergestellt werden kann, sowohl für MCI, als auch für JPI.

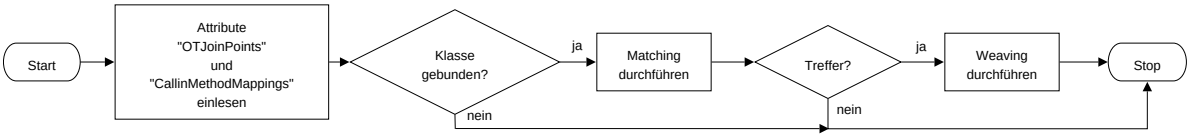

Abbildung 8.2: Ablauf der Transformation als Flowchart.

8.4 Ablauf der Transformation

In diesem Abschnitt soll auf den Ablauf der Transformation und danach auf die das Matching betreffenden Teile eingegangen werden. Abbildung 8.2 zeigt den allgemeinen Ablauf als Flowchart. Abbildung 8.3 wird etwas detaillierter und zeigt den Ablauf der Transformation als Collaboration-Diagramm. Im Flowchart sieht man, dass zuerst die Attribute eingelesen werden. Dieser Teil wurde bereits beschrieben. Da OT-Attribute nur in Team- und Rollenklassen stehen und nie in gewöhnlichen Basisklassen, meistens aber an eben diesen Basisklassen gewoben wird, endet die Transformation in Rollenklassen häufig nach dem Einlesen der Attribute. Wenn eine Basisklasse geladen wird, stehen die Attribute schon zur Verfügung und es kann überprüft werden, ob eine Klasse gebunden ist. Nach dem Matching kann dann das Weben erfolgen.

Nun zu Abbildung 8.3. Dass der Transformer über die geerbte Methode `doTransformInterface` aufgerufen wird, wurde bereits gezeigt. Jetzt soll betrachtet werden, was der Transformer macht. Die Aktionen beziehen sich auf die Nummerierung im Collaboration-Diagramm:

- *Aktion 1*: Durch den Aufruf der geerbten Methode `checkReadClassAttributes` stellt der Transformer sicher, dass alle Attribute eingelesen und alle notwendigen Klassen geladen worden sind. Dafür ist, wie wir gesehen haben, die Methode `scanClassOTAttribute` zuständig. Sie wird von `checkReadClassAttributes` aufgerufen.
- *Aktion 2*: Der Transformer lässt sich von der BCEL-Repräsentation der Klasse alle Methoden geben.
- *Aktion 3*: Der Transformer erzeugt einen Matcher.
- *Aktion 5*: Alle Methoden werden einzeln nacheinander transformiert. Dazu wird zuerst eine BCEL-Repräsentation `MethodGen` der Methode erzeugt.
- *Aktion 6*: Die BCEL-Repräsentation von Methode und Klasse werden an den Matcher übergeben, der das eigentlich Matching durchführt und alle Treffer zur weiteren Verarbeitung zurückliefert.

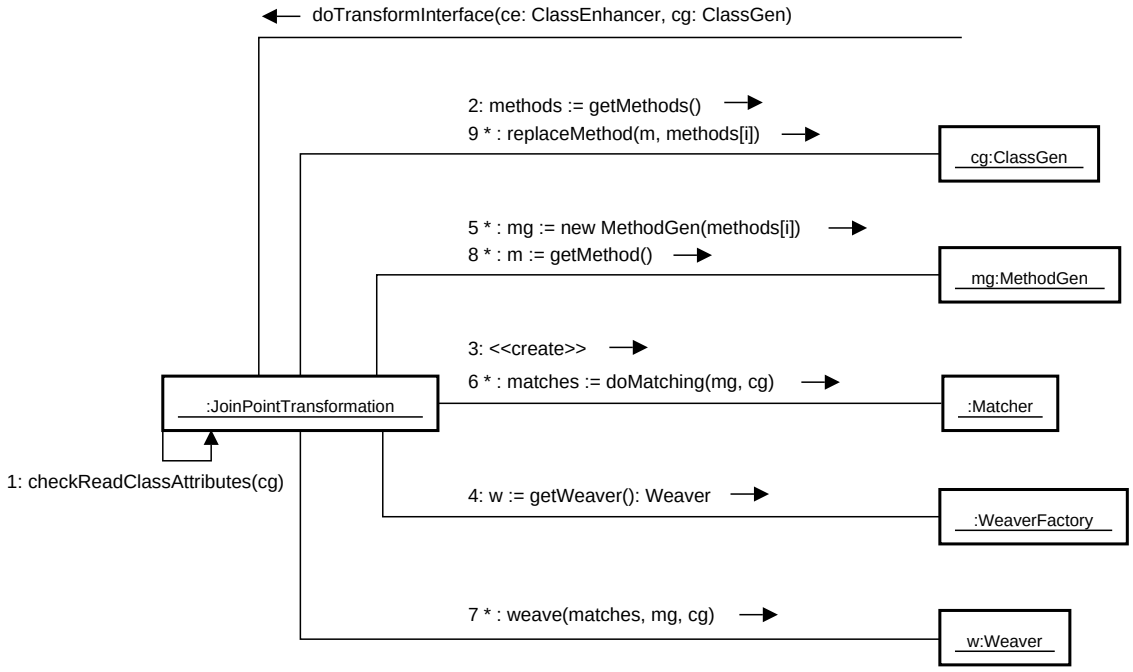

Abbildung 8.3: Ablauf der Transformation als Collaboration-Diagram.

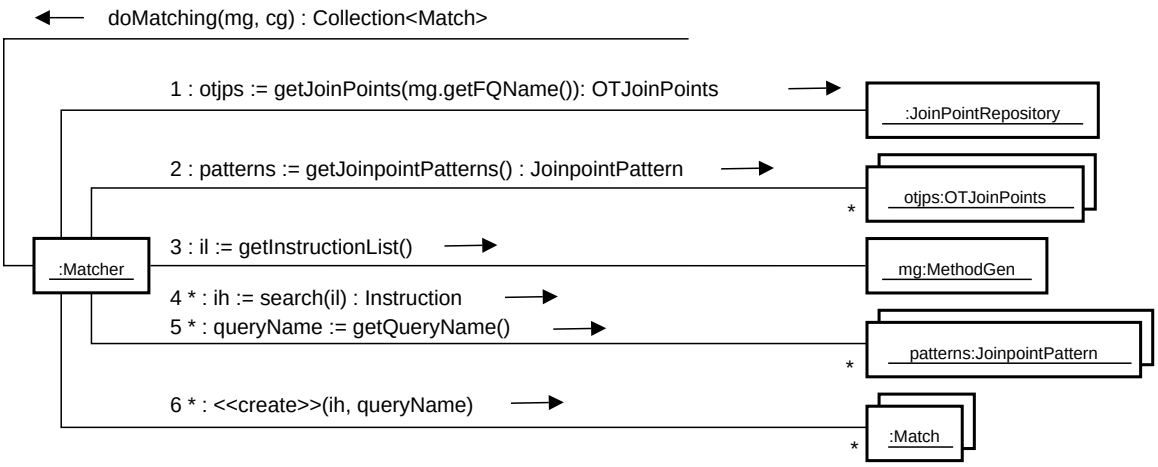

Abbildung 8.4: Ablauf des Matchings.

Die anderen Punkte betreffen das Weben und werden deshalb an dieser Stelle nicht erklärt. Selbstverständlich folgt eine genauere Erklärung im Kapitel über die Implementierung des Webens (Abschnitt 11.2).

Der grundsätzliche Ablauf der Transformation ist klar. Deshalb soll jetzt die Arbeit des Matchers, der durch die Aktion 6 aufgerufen wurde, dargestellt werden. Man betrachte dazu Abbildung 8.4.

- *Aktion 1*: Zuerst werden alle zu einer Methode gehörenden `OTJoinPoints`, die vorher beim Einlesen der Attribute ins `JoinpointRepository` gelegt wurden, abgefragt. Eindeutig gekennzeichnet wird die Methode durch Klassenname, Methodenname und Signatur.
- *Aktion 2*: Der Matcher holt sich die JoinPointPatterns.
- *Aktion 3*: Der Matcher lässt sich von der BCEL-Repräsentation die `InstructionList` der Methode geben. Das ist die BCEL-Repräsentation des Bytecodes.
- *Aktion 4*: Der Matcher gibt dem JoinpointPattern die Anweisung, in der Instruktions-Liste nach einem Auftreten von sich selbst zu suchen. Das Muster liefert diese Information als `InstructionHandle` – eine Referenz auf ein Element der Instruktions-Liste.
- *Aktion 5*: Die Funktion `getQueryName` liefert den Namen der Query, die für die Erzeugung des Musters verantwortlich ist.

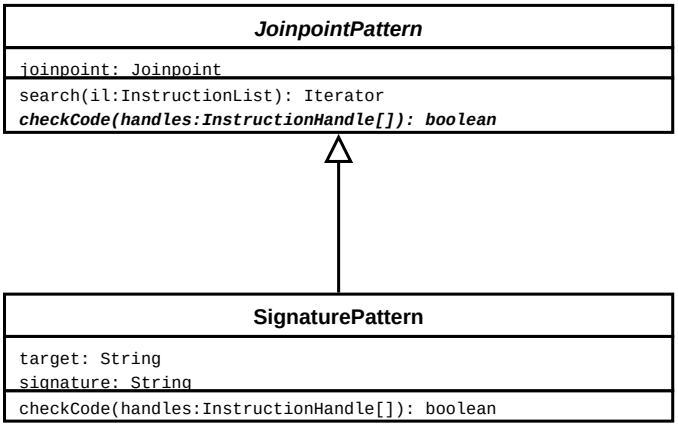

Abbildung 8.5: Die Pattern-Hierarchie.

- *Aktion 6*: Diese beiden Informationen – Queryname und InstructionHandle – stellen einen Treffer dar und werden als solcher gespeichert.

search. Die `search`-Methode der Klasse `JoinpointPatterns` soll näher betrachtet werden. Denn hier findet das eigentliche Matching statt. `search` benutzt zur Suche die `InstructionFinder`- bzw. `FindPattern`-Klasse von BCEL – je nach benutzter BCEL-Version. Sie bietet die Möglichkeit, mit regulären Ausdrücken nach Instruktionen zu suchen. Die hier vorgestellten Beispiele verwenden die Syntax der aktuellen BCEL-Version, weil sie erheblich besser lesbar ist. Im Prototypen wird jedoch aus Kompatibilitätsgründen eine ältere BCEL-Version verwendet. Ein regulärer Ausdruck könnte z. B. wie folgt aussehen:

```
search("BranchInstruction NOP ((IfInstruction|GOTO)+ ISTORE Instruction)*");
```

Unsere Ausdrücke sehen *noch* nicht so komplex aus, da die Muster nur verwendet werden, um nach der Spur eines Joinpoints zu suchen. Zusätzlich zu einem Ausdruck kann man `search` ein Constraint übergeben. Dieses Constraint überprüft, ob der durch den regulären Ausdruck gefundene Codeabschnitt tatsächlich ein Treffer ist:

```
public final Iterator search(String pattern, CodeConstraint constraint);
```

Constraints müssen die Methode `checkCode` implementieren. Dies erledigen die einzelnen Patterns (siehe Abbildung 8.5). Die Implementierung von `search` sieht daher wie folgt aus:

```
1 public Iterator search(InstructionList il) {
2         InstructionFinder finder = new InstructionFinder(il);
3         return finder.search(joinpoint.getPattern(), this);
4 }
```

Jedes JoinpointPattern hat einen Joinpoint. Dieser beschreibt die Art des zu suchenden Joinpoints. Jeder Joinpoint kennt einen regulären Ausdruck, um nach seiner Spur zu suchen. Der Ausdruck ist über `getPattern` verfügbar (Abbildung 8.6). `getPattern` wird in Zeile 3 benutzt. Als Constraint dient `this`, also das `JoinpointPattern` selbst. Für `Signature-JoinpointPattern`, das Ziel und Signatur eines Joinpoints prüft, sieht der Fall für einen Methodenaufruf wie folgt aus:

```
1 public boolean checkCode(InstructionHandle[] ihs) {
2         Instruction i = ihs[0].getInstruction();
3         if (i instanceof InvokeInstruction) {
4                 InvokeInstruction ii = (InvokeInstruction) i;
5                 if (ii.getName(cpg).equals(name) &&
6                         ii.getSignature(cpg).equals(signature))
7                         return true;
8         ...
```

`checkCode` wird aufgerufen und bekommt den Treffer des InstructionFinders übergeben (Zeile 1). Wenn es sich bei dem Treffer um eine InvokeInstruction, also einen Methodenaufruf handelt, wird ein Cast durchgeführt (Zeile 4). Dann werden in den Zeilen 5-6 die Eigenschaften Name und Signatur durch einen einfachen Stringvergleich verglichen.

8.5 Sonstiges

Matching ohne Bytecode-Toolkit. Das Matching lässt sich auch ohne Einsatz eines Bytecode-Toolkits – in diesem Fall BCEL – durchführen. Der Hauptgrund, wieso man dies tun möchte, sind Performanceüberlegungen. Es ist allerdings fragwürdig, ob die Performanceverbesserung tatsächlich so ausgeprägt ist. Dennoch ist das Matching ohne Bytecode-Toolkit einfacher durchzuführen, als das Weben, wie wir noch sehen werden. Matching ohne Bytecode-Toolkit ist in diesem Prototypen nicht realisiert und es kann daher keine Performancemessung durchgeführt werden. Dennoch soll kurz skizziert werden, wie der Prototyp um einen anderen Matcher erweitert werden könnte.

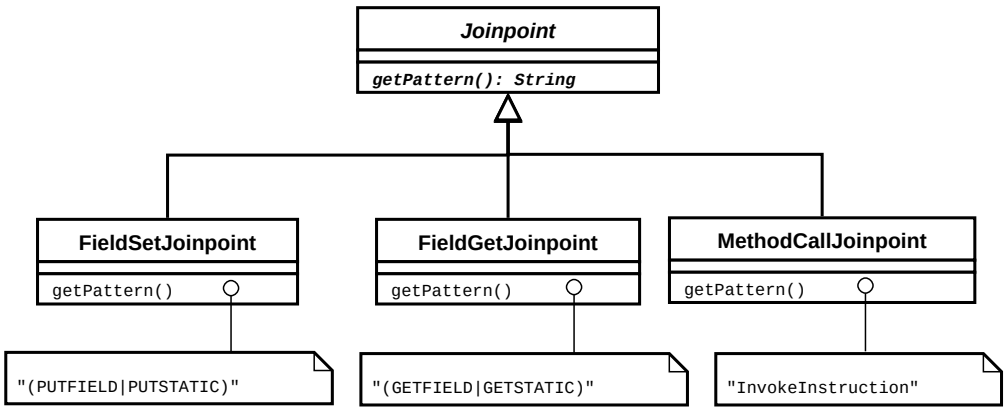

Abbildung 8.6: Ein Teil der Joinpoint-Hierarchie.

Einsatz unterschiedlicher Matcher. Die Möglichkeit den Matcher auszutauschen, ist nicht so interessant, wie die Möglichkeit den Weber auszutauschen. Darum ist in diesem Prototypen nur letzteres realisiert. Dennoch ist es sehr einfach möglich, den Prototypen entsprechend zu erweitern.

Dazu wird eine Schnittstelle definiert, die z. B. die Methode `doMatching` enthält, so wie das jetzt schon beim Matcher der Fall ist. Eine Fabrik abstrahiert davon, welcher Matcher erzeugt wird. Die JoinpointPattern- und Joinpoint-Hierarchien werden um eventuell benötigte Methoden erweitert. Diese Realisierung wird für den Weber benutzt und später vorgestellt.

Matching im Kontext. Im Kapitel 8 wurde der Kontext eines Joinpoints als Eigenschaft von Joinpoints benannt und erklärt, dass die Kontextangabe mit Hilfe regulärer Ausdrücke erfolgen könne. BCEL bietet mit dem InstructionFinder eine komfortable Möglichkeit, solche regulären Ausdrücke auszuwerten. Der Compiler könnte Muster erzeugen, die einen für InstructionFinder verwertbaren regulären Ausdruck enthalten. Dieser könnte dann einfach an die search-Methode (siehe oben) übergeben werde. Mit dieser kurzen Ausführung soll dieses Thema aber beendet sein.

Performance. Das hier besprochene Matching wirkt sich ausschließlich auf die Ladezeit aus. Es sei denn, es soll Laufzeitweben gemacht werden. Das ist jedoch Zukunftsmusik. Das Matching, das der Compiler macht, wirkt sich auf die Compilezeit aus. Die größten Verbesserungen sind dadurch zu erreichen, indem viel zur Compilezeit ausgewertet wird.

Auf die Benutzung eines Bytecode-Toolkit zu verzichten, könnte vielleicht auch die Performance beim Matchen verbessern. Dennoch sei angemerkt, dass von der Laufzeitumgebung zur Zeit immer ein ClassGen-Objekt erzeugt wird. Das Matching ohne Toolkit würde also lediglich die Erzeugung der MethodGen-Objekte ersparen. Beim Weben werden diese Objekt aber ohnehin benötig, wie wir noch sehen werden. Also spart man sich die Erzeugung nicht wirklich. Es ist allerdings die Änderung geplant, dass die ClassGen-Objekte nur bei Bedarf erzeugt werden. Dann wäre es von Vorteil, möglichst viele Transformationsschritte direkt auf dem Bytecode durchzuführen. Natürlich ist das auch umständlich und fehleranfällig zu implementieren.

Kapitel 9

Weben von Aspektcode an Joinpoints

Wie man Joinpoints findet, ist jetzt bekannt. Was fehlt ist der zweite große Teil, das Weben. Die Stellen, an denen gewoben werden soll – die Joinpoints –, sind also bekannt. Doch wie kann an diesen Stellen der Aspektcode ausgeführt werden? Davon soll dieses Kapitel berichten.

9.1 Wrapper-Methode

Es reicht nicht, den Aspektcode in die Methode, die den Joinpoint enthält – davor und dahinter – einzufügen, denn bei `replace`-Bindung kann der Aspekt entscheiden, ob der Joinpoint-Code ausgeführt werden soll oder nicht. Die Ausführung kann sogar mehrfach erfolgen.

Aus diesem Grund soll der Joinpoint aus der Methode herausgetrennt und in einer eigenen Methode ausgeführt werden. Vereinfacht gesagt, wird eine neue Methode erstellt. Im Folgenden wird sie Wrappermethode oder kurz *Wrapper* genannt werden. Der Joinpointcode wird in die neue Methode verschoben und anstelle des Joinpointcodes wird ein Aufruf der Wrappermethode eingefügt. Es ist wichtig, dass das Programm durch diese Änderung semantisch nicht verändert wird.

Das eigentliche Weben wird dadurch vereinheitlicht. Denn dadurch, dass der Joinpoint in einer eigenen Methode steht, kann man ihn wie einen *Execution-Joinpoint* behandeln. Das Weben funktioniert für alle Joinpoints gleich. Nur das Erzeugen und Aufrufen der

Wrappermethode hängt von der Art des Joinpoints ab. Beide Probleme können getrennt voneinander betrachtet werden.

Dieser Abschnitt soll beschreiben, wie eine solche Wrappermethode aussehen muss, wie sie aufgerufen wird und wie der Bytecode der Methode, die den Joinpoint enthält, geändert werden muss. Wie Aspektcode an Execution-Joinpoints gewoben wird, wird in Abschnitt 11.3 erklärt.

Parameter und Rückgabewert. Das Laden der Parameter eines Joinpoint gehört nicht zum Joinpoint und soll daher nicht in die Wrappermethode verschoben werden – was sich auch schwierig darstellt, weil dabei unter anderem auf lokale Variablen zugegriffen werden kann. Deshalb müssen die Parameter des Joinpoints stattdessen an die Wrappermethode übergeben werden. Diese kann sie dann für den Joinpointcode auf den Stack legen. Der vom Joinpoint produzierte Wert – es ist höhstens einer – kann als Rückgabewert der Wrappermethode geliefert werden.

Signatur. Um die Wrappermethode so zu realisieren, muss nur die Signatur geeignet definiert werden. Die Signatur der Wrappermethode muss mit der Signatur des Joinpoints übereinstimmen (Joinpointkatalog A). Dabei ist zu beachten, dass Instanzmethoden einen impliziten ersten Parameter haben (`this`). Im Joinpointkatalog wird dieser Parameter expizit aufgeführt.

Stackbalance. Wenn die Wrappermethode das Vorhandensein von impliziten Parametern berücksichtigt, kann der Aufruf der Wrappermethode, die zuvor für den Joinpointcode auf den Operanden-Stack gelegten Werte konsumieren und produziert den gleichen Wert, wie der zuvor entfernte Joinpointcode. Der Stack bleibt damit in Balance. Man muss dabei unterscheiden, ob der Wrapper als Instanz- oder Klassenmethode realisiert ist.

Instanz- oder Klassenmethode. Die Realisierung als Instanzmethode erfordert, dass noch bevor die Parameter des Joinpoints auf den Stack gelegt werden, eine Referenz von dem Objekt auf den Stack gelegt wird, an dem die Wrappermethode aufgerufen werden soll. Das bedeutet auch, dass man den Anfang des Codes finden muss, der für das Laden der Parameter verantwortlich ist und davor Code einfügt, der die benötigte Instanz lädt. Den Anfang zu finden, ist nicht ganz trivial. Deshalb soll eine Lösung gefunden werden, die weniger Analyse des Bytecodes erfordert. Ziehen wir also in Betracht, die Wrappermethode als Klassenmethode zu implementieren.

Wenn die Wrappermethode als Klassenmethode realisiert ist, hat das zur Konsequenz, dass die Wrappermethode nicht weiß, wer sie aufgerufen hat. Diese Information ist aber, wenn der Aufrufer ein Objekt ist – der Aufruf also aus einer Instanzmethode erfolgt –, wichtig; denn in *ObjectTeams* benötigen die Rollenobjekte eine Referenz auf ein Basisobjekt.

Der Aufrufer der Wrappermethode muss also, falls er ein Objekt ist, eine Referenz auf sich selbst an die Wrappermethode übergeben. In diesem Fall ist es möglich den Anfang des Parameterladecodes nicht suchen zu müssen. Dazu kann die Referenz als der letzte Parameter der Wrappermethode übergeben werden. Als letzter Parameter der Wrappermethode, kann der Code zum Laden der Referenz direkt vor dem Aufruf der Wrappermethode eingefügt werden. Beide Varianten – der Wrapper als Instanz- oder Klassenmethode – sind also denkbar und werden uns im Folgenden noch beschäftigen.

Wrappercode. Der Wrapper macht nichts weiter, als die ihm übergebenen Joinpointparameter auf den Stack zu legen, den Joinpointcode auszuführen und den vom Joinpoint produzierten Wert zurückzugeben. Je nach Joinpointart unterscheidet sich die Implementierung des Wrappers. Die genaue Realisierung wird im Joinpointkatalog im Anhang A angegeben. Dazu dienen die Einträge *Geänderter Code* und *Wrapper*.

Wrappername. Weil für jeden gebundenen Joinpoint eine Wrappermethode erstellt werden muss, kann es viele Wrapper geben und die müssen – sofern sie alle in der gleichen Klasse definiert werden – einen eindeutigen Namen haben. Der entwickelte Prototyp benutzt folgende Benennung:

Name`$`*Signatur*`$wrapper$`*Nummer*

- *Name*: Name der Methode M, in den Joinpoint enthält.
- *Signatur*: Index des Konstantenpool-Eintrags, der die Signature von M enthält.
- *Nummer*: Eine fortlaufende Nummer.

Bevor an dieser Stelle ein Beispiel folgt, um das bereits Gesagte weiter zu beleuchten, sollen noch einmal die wichtigsten Punkte zur Wrappermethode zusammengefasst werden:

- Die Wrappermethode kann als Klassenmethode – also statisch – realisiert werden oder als Instanzmethode.

- Die Signatur der Wrappermethode stimmt größtenteils mit der Signatur des Joinpoints überein (implizite Parameter).
- Wenn der Wrapper als eine Klassenmethode realisiert ist und der Joinpoint sich in einer Instanzmethode befindet, erhält die Parameterliste der Wrappermethode einen zusätzlichen Parameter am Ende. Dieser dient dazu, eine Referenz auf die Instanz zu übergeben. Dazu wird direkt vor dem Aufruf des Wrappers Code zum Laden dieser Referenz eingefügt.
- Wenn der Wrapper eine Instanzmethode ist und auch der Joinpoint sich in einer Instanzmethode befindet, muss vor den Parametern eine Referenz auf das Objekt mit dem Wrapper geladen werden.
- Befindet sich der Joinpoint in einer Klassenmethode kann der Wrapper bedenkenlos als Klassenmethode realisiert werden und seine Signatur entspricht der des Joinpoints.
- Der Wrapper lädt die Parameter, führt den Joinpoint aus und gibt das Ergebnis zurück.
- Das Programm wird semantisch nicht verändert.

9.2 Beispiele

Um die Benutzung von Wrappermethoden besser verstehen zu können, sollen an dieser Stelle zwei Beispiele präsentiert werden. Das eine Beispiel demonstriert, wie ein Methodenaufruf-Joinpoint in einer Instanzmethode durch einen statischen Wrapper ersetzt wird. Das andere Beispiel zeigt, wie ein Konstruktor-Joinpoint in einer Instanzmethode durch einen dynamischen Wrapper ersetzt wird.

Methodenaufruf-Joinpoint

Listing 9 ist der Bytecode der Methode `bar` aus Listing 8. In Zeile 1-5 wird `i`, in 6-10 `j` erzeugt und in den lokalen Variablen 1 und 2 gespeichert. In Zeile 11 wird zuerst `i`, in Zeile 12 wird dann `j` auf den Stack geladen. In Zeile 13 wird die Methode `compareTo` aufgerufen.

Der Joinpoint sei der Aufruf der Methode `compareTo`. Er soll in einen als Klassenmethode realisierten Wrapper verschoben werden. Bei dem Joinpoint handelt sich um den Aufruf einer Instanzmethode der Klasse `Integer` mit der Signatur `int compareTo(Integer)`. Der Joinpoint hat daher die Signatur `int (Integer, Integer)`. Die Wrappermethode muss die

```
1  class Foo {
2      public void bar() {
3          Integer i = new Integer(1);
4          Integer j = new Integer(2);
5          int result = i.compareTo(j);
6      }
7  }
```

Listing 8: Instanzmethode `bar` enthält einen Methodenaufruf-Joinpoint.

```
1  new Integer;
2  dup
3  iconst_1
4  invokespecial Integer.<init>(I)V;
5  astore_1
6  new Integer;
7  dup
8  iconst_2
9  invokespecial Integer.<init>(I)V;
10 astore_2
11 aload_1
12 aload_2
13 invokevirtual Integer.compareTo(Ljava/lang/Integer;)I;
14 istore_3
15 return
```

Listing 9: Bytecode der Methode `bar` aus Listing 8.

gleiche Signatur haben und einen zusätzlichen Parameter, um eine Referenz auf das Basisobjekt empfangen zu können, denn der Aufruf von `compareTo` erfolgt in einer Instanzmethode von `Foo`.

Der Wrapper soll hier der Einfachheit halber `wrapper` heißen. Tatsächlich würde er gemäß dem oben genannten Schema einen Namen wie `bar$43$wrapper$1` erhalten. Als Signatur ergibt sich: `int wrapper(Integer, Integer, Foo)`. Der Aufruf von `compareTo` muss durch einen Aufruf von `wrapper` ersetzt werden. Der geänderte Bytecode sieht dann wie in Listing 10 aus. Vor dem Aufruf des Wrappers wird noch eine Referenz auf die Instanz von `Foo` auf den Stack gelegt (Zeile 4). Das ist nichts weiter als `this`. Dann wird `wrapper` aufgerufen (Zeile 5).

```
1 ...
2 aload_1
3 aload_2
4 aload_0
5 invokestatic wrapper(Ljava/lang/Integer;Ljava/lang/Integer;LFoo;)I;
6 istore_3
7 return
```

Listing 10: Methode `bar` benutzt jetzt `wrapper`.

Wenn man dem transformierten Bytecode zurück in Java übersetzten würde, sähe er aus wie in Listing 11.

```
1  class Foo {
2      public void bar() {
3          Integer i = new Integer(1);
4          Integer j = new Integer(2);
5          int result = wrapper(i, j, this);
6      }
7      static int wrapper(Integer i, Integer j, Foo f) {
8          return i.compareTo(j);
9      }
10 }
```

Listing 11: Methode `bar` benutzt jetzt `wrapper`.

Die hier vorgestellte Variante funktioniert nicht, wenn man das OTRE den Wrapper als

Execution-Joinpoint weben lässt, weil das Basisobjekt dann nicht bekannt ist. Es wird zwar als letzter Parameter übergeben, aber der Basismethodentransformer, der das Weben in diesem Fall durchführt, weiß dies nicht. Wenn man das Weben selbst implementiert, würde die Variante aber funktionieren und vorteilhaft sein, weil man den Anfang des Parameterladecodes nicht suchen muss. Allerdings erfordert es auch erheblichen Aufwand, das Weben komplett neu zu implementieren. Der Prototyp verwendet daher den Basismethodentransformer und erstellt Wrapper, wenn nötig, als Instanzmethode. Wie das Weben am besten im OTRE zu implementieren ist, wird in aller Breite in Abschnitt 11.1 beschrieben.

Konstruktoraufruf-Joinpoint

```
1  class Foo {
2          int i;
3          public Foo(int i) {
4                  this.i = i;
5          }
6  }
7
8  class Bar {
9          public Foo bar() {
10                 return new Foo(0);
11         }
12 }
```

In der Methode `Bar.bar` befindet sich ein Konstruktoraufruf- bzw. Objekterzeugungs-Joinpoint. Der Bytecode von `bar` sieht folgendermaßen aus:

```
1  new Foo
2  dup
3  iconst_0
4  invokespecial Foo.<init>
5  areturn
```

Im Kapitel *Java* wurde bereits gesagt, dass `new` ein uninitialisiertes Objekt auf den Stack legt, das vom Konstruktoraufruf konsumiert wird. Es ist allerdings nicht möglich, ein uninitialisiertes Objekt als Parameter an eine normale Methode wie den Wrapper zu übergeben. Der *Bytecode-Verifier* erlaubt es nicht. Deshalb müssen `new` und `invokespecial` in der

gleichen Methode stehen. Sie müssen also beide in den Wrapper verschoben werden. Der Parameterladecode kann nicht einfach verschoben werden, weil er auf lokale Variablen zugreifen kann, die im Wrapper nicht existieren. Er soll deshalb nicht verschoben werden. Dieses Prinzip angewendet auf die Methode `bar`, führt zu folgendem Code:

```
1 aload_0
2 iconst_0
3 invokespecial wrapper
4 areturn
```

Die Anweisungen `new` und `dup` wurden in den Wrapper verschoben und tauchen nicht mehr auf. Stattdessen wird eine Referenz auf `this` auf den Stack gelegt (Zeile 1). An diesem Objekt soll der Wrapper aufgerufen werden. Dann folgt in Zeile 2 der ursprüngliche Parameterladecode (o. B. d. A. nur ein Parameter). In diesem Beispiel gibt es nur einen Parameter, aber die Transformation funktioniert bei mehreren Parametern analog. In Zeile 3 erfolgt statt des Konstruktoraufrufs der Wrapperaufruf. Der Parameter mit dem Wert 0 der vorher an den Konstruktor übergeben wurde, wird jetzt an den Wrapper übergeben. Der Rest der Methode bleibt unverändert. Der Wrapper wird, wie bereits beschrieben, als Instanzmethode in der gleichen Klasse wie `bar` erzeugt:

```
1 private Foo wrapper(int arg1) {
2         new Foo
3         dup
4         aload_1
5         invokespecial Foo.<init>
6         areturn
7 }
```

Die Zeilen 2 und 3 sind in den Wrapper verschoben worden. Der Parameterladecode wurde nicht verschoben. Der Parameter für den Konstruktoraufruf wurde stattdessen an den Wrapper übergeben. In Zeile 4 wird er auf den Stack gelegt. Dann erfolgt der Konstruktoraufruf, auch er ist verschoben worden (Zeile 5). Zum Schluss wird das neue Objekt zurückgegeben. Der Joinpoint wurde in eine eigene Methode ausgelagert. Das Programm wurde trotzdem semantisch nicht verändert.

9.3 Sonstiges

Joinpoint Variablenzugriff. In den beiden gerade präsentierten Beispielen war es so, dass der Joinpoint in eine andere Methode verschoben wurde. Beim Joinpoint Variablenzugriff ist das nicht möglich, weil die Variable nur im Gültigkeitsbereich der umgebenden Methode existiert. Der Kontrollfluss wird in den Wrapper umgeleitet und der Wert, der geladen oder gespeichert werden soll, kann manipuliert werden. Um den Variablenwert aber an den Wrapper zu übergeben bzw. den Rückgabewert vom Wrapper in der Variablen zu speichern, wird trotzdem ein Lade- bzw. Speicherbefehl benutzt. Der Joinpoint ist also eigentlich noch vorhanden. Wie man sich aber leicht überlegen kann, ist das nicht ganz so. Der Lade- bzw. Speicherbefehl ist eigentlich ein anderer Joinpoint. Der ursprüngliche Joinpoint existiert nicht mehr als Bytecode. Er wurde komplett durch den Wrapperaufruf ersetzt. Für das Lesen von Variablen verhält sich das, wie der Refactoringschritt *Temporäre Variable durch Abfrage ersetzen*. Die Semantik bleibt erhalten. Auch die unterschiedlichen Bindungsarten sind sinnvoll. Auch wenn der Wrapper nichts weiter macht, als den übergebenen Wert zurück zu geben, also eine Identitätsfunktion ist.

Verwandte Arbeiten. Das in diesem Kapitel vorgestellte Verfahren, den Joinpoint in einer eigenen Methode zu kapseln, ähnelt einem Ansatz der sich *Envelope Based Weaving* nennt [5]. Beim *Envelope Based Weaving* werden für Methodenaufrufe und Feldzugriffe *Envelope* genannte Wrappermethoden erstellt. Die *Envelopes* befinden sich in der gleichen Klasse wie das Feld oder die Methode, die sie kapseln und können so eine identische Signatur haben. *Envelopes* für Feldzugriffe sind also nichts weiter als *getter*- bzw. *setter*-Methoden. Dadurch wird das Weben vereinfacht, denn der Joinpointcode muss nur durch einen Methodenaufruf ersetzt werden. Ein Einfügen von `this`, wie es für OT benötigt wird, entfällt.

Allerdings ist dieses Verfahren für OT nicht anwendbar. Die Klassen der Objekte, auf die zugegriffen wird, könnten bereits geladen sein. Das lässt sich auch durch eine geeignete Ladereihenfolge nicht verhindern. Eine vorher unbekannte Klasse könnte plötzlich geladen werden und eine bereits geladene Klasse benutzen. Java erlaubt aber keine schemaändernde[1] Transformation von Klassen, die für die Erstellung von Evelopes benötigt würde.

Kollisionen. Es kann passieren, dass mehrere Queries den gleichen Joinpoint beschreiben oder dass eine Query mehrmals benutzt wird. Dieser Fall muss beim Weben berücksichtigt werden. Der Joinpoint kann und wird nur einmal aus der Methode entfernt und in einen Wrapper verschoben. Wie Kollisionen behandelt werden steht in 10.1.

[1] Das Interface der Klasse ändernt.

Weben an einen Executionjoinpoint. Durch die Erstellung eines Wrappers ist es möglich, den Joinpoint wie einen *Execution-Joinpoint* zu behandeln. Wie ein Execution-Joinpoint gewoben wird, steht in 11.3.

Wrapper-Klasse. Während diese Diplomarbeit entsteht, entstehen auch einige andere Diplomarbeiten, die sich mit Bytecodetransformation im OTRE beschäftigen. Eine davon ist [14]. Sie soll ergründen, wie es möglich ist, die Transformationen des OTRE so durchzuführen, dass zur Laufzeit schemaerhaltend weiterhin alle notwendigen Transformationen durchgeführt werden können. Die Forderung nach Schemaerhaltung ist notwendig, weil Java zur Zeit nur solche Transformationen zur Laufzeit erlaubt.

Die ursprünglich Idee in der Planung von [14] war es, für jede Basisklasse eine weitere Schattenklasse zu erstellen, in der sämtlicher generierter Code steht. Auch Wrapper hätten dort landen sollen, was Konsequenzen für die Wrapper gehabt hätte; vor allem, was die Sichtbarkeit von Methoden und Feldern betrifft. Allerdings wurde die Lösung als Suboptimal verworfen und eine andere, sehr viel elegantere Lösung, gefunden. Die neue Lösung hat noch größere Konsequenzen für die Generierung der Wrappermethoden. Im nächsten Kapitel, wenn über Vererbung gesprochen wird, soll darauf eingegangen werden (Abschnitt 10.6).

Kapitel 10

Weitere Aspekte des Webens

In diesem Kapitel sollen einige Themen betrachtet werden, die bis jetzt keine Erwähnung gefunden haben. Zuerst wird besprochen, wie Kollisionen beim Matching behandelt werden. Kollisionenen treten auf, wenn unterschiedliche Queries die gleichen Joinpoints beschreiben. Alsdann soll beschrieben werden, wie es Aspektcode ermöglicht werden kann, auf die Parameter eines Joinpoints zuzugreifen, wie gewoben wird, wenn Vererbung oder Exceptions ins Spiel kommen und wie weitere Aspekte in eine Methode gewoben werden können, in die bereits gewoben wurde – Mehrfachweben – und ob Nebenläufigkeit ein Problem darstellt.

10.1 Kollisionen beim Matching

Es kann passieren, dass beim Matching Kollisionen auftreten und zwar immer dann, wenn unterschiedliche Queries den gleichen Joinpoint beschreiben. Dies ist beim Matching völlig unproblematisch, muss aber beim Weben berücksichtigt werden. Schließlich gibt es nur einen Wrapper pro Joinpoint, der zum Weben benutzt werden kann. Deshalb muss vor dem Weben festgestellt werden, welche Treffer sich überschneiden. Dann können sie geeignet behandelt werden. Dieser Abschnitt soll erklären, welche Überschneidungen es gibt und wie man sie erkennen und behandeln kann.

Theoretisch betrachtet kann es drei mögliche Überschneidungsarten geben – *Teilweise Überlappend*, *Umschießend* und *Identisch*. Eine teilweise Überlappung kann bei Joinpoints, die ein Hochsprachkonstrukt repräsentieren, nicht vorkommen, deshalb werden im Folgenden nur die anderen beiden Arten betrachtet.

Erkennung. Die Erkennung ist sehr einfach. Jeder Joinpoint besitzt zugehörigen Bytecode. Wenn unterschiedliche Joinpoints teilweise den selben Bytecode besitzen, liegt eine Überschneidung vor. Wie in diesem Prototypen eine Erkennung implementiert ist, wird in Abschnitt 11.4 beschrieben.

Umschließend. Umschließend ist eine Überschneidung, wenn ein Joinpoint innerhalb eines anderen vorkommt. Im Joinpointkatalog (Anhang A) stehen keine solchen Joinpoints, weil sie alle atomar sind. Auch bei einen Konstruktoraufruf-Joinpoint, der zwar zwischen `new` und dem Aufruf der `<init>`-Funktion andere Joinpoints enthalten kann, kann man in diesem Fall nicht wirklich von einer Überschneidung sprechen, weil die Joinpoints zwischen `new` und dem Aufruf der `<init>`-Funktion nicht in den Wrapper des Konstruktor-Joinpoints verschoben werden.

Dennoch könnte der Fall bei einem Schleifen-Joinpoint auftreten. Die Behandlung ist aber denkbar einfach. Die inneren Joinpoint müssen einfach zuerst ersetzt werden. Wenn alle inneren Joinpoints ersetzt wurden, wird der äußere Joinpoint ersetzt. Das Weben findet wie gewohnt durch Ersetzten der Wrapper statt.

Identisch. Eine Überschneidung kann identisch sein, d. h. dass unterschiedliche Queries den gleichen Joinpoint beschrieben haben. Behandeln lässt sich dieser Fall wie eine mehrfache Verwendung der gleichen Query. Man erkennt an der Bytecodeposition beider Joinpoints, dass sie identisch sind. Der Joinpoint wird natürlich nur einmal durch den Wrapper ersetzt und es werden beide Queries an den gleichen Wrapper gewoben. Eine Implementierung wird in Abschnitt 11.4 vorgestellt. Allerdings ist für den Fall zweier kollidierender Queries keine Präzedenz definiert. Im Fall der Verwendung der gleichen Query erkennt der Compiler zuverlässig, dass eine Präzedenz definiert werden muss und fordert dies ein. In diesem Fall kann der Compiler nicht immer erkennen, dass bei unterschiedlichen Queries eine solche Kollision auftritt, weil Klassen erst zur Ladezeit bekannt werden können. Man muss in diesem Fall davon ausgehen, dass die Ausführungsreihenfolge der Aspektmethoden undefiniert ist.

Präzedenz. Es gibt einige impizite Präzedenzregeln, die definieren in welcher Reihenfolge Rollenmethoden, die an den gleichen Joinpoint oder die gleiche Basismethode gebunden sind, ausgeführt werden. Zuerst werden alle mit dem *Bindungsmodifikator* `before` gebundenen Rollenmethoden ausgeführt, dann folgen die mit `replace`, dann die mit `after` gebundenen Rollenmethoden. Bei der mehrfachen Verwendung eines Basismethoden- oder Querybezeichners in unterschiedlichen Callinbindungen mit dem gleichen *Bindungsmodi-*

fikator liegt eine Kollision vor, die durch die implizit verwendeten Präzedenzregeln nicht aufgelöst werden kann. Deshalb ist es möglich eigene Präzedenzregeln zu deklarieren:

```
1 foo: test <- before query getterCalls<MyBaseSameQueryTwoTimes>;
2 bar: test2 <- before query getterCalls<MyBaseSameQueryTwoTimes>;
3 precedence bar, foo;
```

Diese Präzedenzdeklaration definiert, dass zuerst `test2` und danach `test` ausgeführt werden soll. Dazu werden die miteinander in Konflikt stehenden Callinbindungen mit einem *Label* markiert (Zeile 1-2). Dann findet eine Präzedenzdeklaration mit dem Schlüsselwort `precedence` statt. Dabei werden die vorher definierten Label verwendet.

10.2 Parameter

Jeder Joinpoint hat Parameter und einen Rückgabewert. Dies sind die Werte, die vor der Ausführung des Joinpoint-Codes, auf den Stack gelegt werden und der Wert, der nach der Ausführung auf dem Stack liegt. Welche Parameter und Rückgabewerte ein Joinpoint genau hat, ist im Joinpointkatalog im Eintrag Signatur vermerkt. *ObjectTeams/Java* bietet für Execution-Joinpoints bereits jetzt die Möglichkeit, Parametermapping durchzuführen. Das wurde bereits in Kapitel 2 beschrieben. Diese Variante setzt jedoch voraus, dass alle von der Query gelieferten Joinpoints die gleiche Signatur haben. Wenn diese Voraussetzung gegeben ist, ist Parametermapping dieser Art möglich. In Callinbindungen kann die Signatur der zurückgelieferten Joinpoints angegeben werden. Was im Abschnitt 5.1 beschrieben wurde. Die Zuordnung der Parameter erfolgt mit dem `with`-Konstrukt. Es sieht etwa so aus:

```
1 void roleMethod(int i) <- replace
2         query void(int value, int v2) setterCalls<MyBase> with {
3                 i <- value
4         }
```

Eine andere Möglichkeit, wie man Aspektmethoden Zugriff auf die Parameter eines Joinpoints geben könnte, wäre introspektiver Art. Hierbei könnte man in jeder Aspektmethode einen Identifier `thisJoinPoint` zur Verfügung stellen. Über dieses Objekt könnten dann alle notwendigen Abfragen gemacht werden. Einen ähnlichen Ansatz geht AspectJ. Jedoch bietet `thisJoinPoint` dort noch zahlreiche andere Informationen. Da die Joinpointinterception mit Wrappermethoden realisiert ist und OTRE bereits Parametermapping für Methodcallinterception unterstützt, funktioniert das Prinzip auch im Prototypen.

10.3 Nebenläufigkeit

Die Methodcallinterception in *ObjectTeams* ist so realisiert, dass sie mit Nebenläufigkeit zurecht kommt. Durch die Erzeugung eines Wrappers entstehen keine neuen Probleme, die in einem nebenläufigen Programm Probleme verursachen könnten, deshalb muss das Thema Nebenläufigkeit nicht weiter betrachtet werden.

10.4 Exceptions

Exceptions wurden bis jetzt nicht behandelt. Dies soll jetzt nachgeholt werden. *Was ändert sich für den Wrapper, wenn ein Joinpoint eine Exception wirft?* Um das Programm semantisch nicht zu verändern, deklariert der Wrapper die einsprechenden Exceptions als `throws`. Im Joinpointkatalog in Anhang A wird im Eintrag *Exceptions* beschrieben, welche Exceptions ein Joinpoint werfen kann. Ein Dereferenzierung-Joinpoint kann beispielsweise eine `NullPointerException` werfen.

Was ist, wenn eine Rollenmethode eine Exception werfen kann und dies definiert? Das ist keine gute Idee! Der Wrapper und die Basismethode könnten zwar so geändert werden, dass sie die Exception weiter reichen, aber was macht der Aufrufer der Basismethode, wenn eine unbekannte Exception fliegt?

Ist Exception fangen ein Joinpoint? Ja! Allerdings ist er nicht atomar und deshalb schwer umzusetzten. Seine Spur ist ein Eintrag in der Exceptiontabelle:

```
u2 exception_table_length;
{          u2 start_pc;
           u2 end_pc;
           u2  handler_pc;
           u2  catch_type;
}        exception_table[exception_table_length];
```

Der Anfang ist leicht zu finden. Es ist `handler_pc`. Das Ende und die `finally`-Klausel sind nur durch aufwendige Analyse zu finden. Das Verschieben in einen Wrapper stellt sich deshalb als schwierig heraus. Auch lokale Variablen stellen ein Problem dar, weil sie im Wrapper nicht ad-hoc verfügbar sind. Allerdings lässt sich das Problem mit den Variablen noch am leichtesten lösen, indem die Variablen alle eingepackt und an den Wrapper übergeben werden. Es sei noch angemerkt, dass Basismethoden relativ leicht um einen neuen Exceptionhandler erweitert werden können, der *nicht* folgende Form besitzt:

```
1 callin roleMethod() {
2         try {
3                 base.roleMethod();
4         } catch (NewException e) {
5                 ...
6         }
7 }
```

Die Exceptiontabelle einer Basismethode kann auch um eine neue Exception erweitert werden. Denn das gerade gezeigte Beispiel, funktioniert nicht immer, nämlich dann, wenn die Basismethode einen generischen catch-Block hat, der alle Exceptions fängt. Um in diesem Fall eine neue Exception einzuführen, könnte die Exceptiontabelle um eine zusätzlich Exception erweitert werden.

10.5 Mehrfachweben

Mehrfachweben bedeutet, dass nachdem bereits Aspektcode in eine Methode gewoben wurde, noch einmal Aspektcode in eine Methode gewoben werden soll. Dieser Fall tritt vor allem dann auf, wenn zur Laufzeit einer Klasse – also nachdem die Klasse bereits geladen wurde – weitere Aspekte in die Klasse gewoben werden sollen. Es ist in Java grundsätzlich möglich, eine Klasse zur Laufzeit zu ändern. Allerdings gibt es die Einschränkung, dass nur Methodenimplementierungen geändert werden können. Die Transformation muss also schemaerhaltend sein.

Um zusätzliche Rollenmethoden in die Basisklasse zu weben, muss erstmal das Matching für die neuen Patterns durchgeführt werden. Dazu wird die Originalmethode benötigt, das heißt es muss vor der ersten Transformation ein Backup erstellt werden. Das ist kein Problem. Dann kann mit dem Backup das Matching neu durchgeführt werden. Wenn die Auswertung Joinpoints findet, die bisher nicht gebunden waren, müsste ein neuer Wrapper erstellt werden. Wenn die Auswertung einen Joinpoint findet, der bereits gebunden ist, müsste die Rollenmethoden zu den anderen Rollenmethoden gewoben werden. In beiden Fällen müsste zuerst festgestellt werden, wo der gefundene Joinpoint in der transformierten Methode liegt, denn er kann sich verschoben haben. Für diesen Zweck könnte ein Array verwaltet werden, dass diese Zuordnung möglich macht. Das Array würde mit der Position des Joinpoint in der Original-Methode indiziert werden und als Eintrag die Position des Joinpoints in der transformierten Methode enthalten, sowie ein Flag das ausdrückt, ob an dieser Stelle bereits ein Wrapper eingefügt wurde. Wurde kein Wrapper eingefügt, wird wie gehabt gewoben. Wurde ein Wrapper eingefügt, wird der entsprechende Aspektcode in dem Wrapper hinzugefügt.

Allerdings gibt es ein Problem, das bis jetzt unterschlagen wurde, aber dem aufmerksamen Leser sicher ausgefallen ist: Die Transformation zur Laufzeit muss schemaerhaltend sein! Die Einführung von Wrappermethoden ist nicht möglich. In einer parallel laufenden Diplomarbeit wurde aber eine Möglichkeit gefunden, die Transformationen des OTRE so durchzuführen, dass zur Laufzeit schemaerhaltend weiterhin alle notwendigen Transformationen durchgeführt werden können [14]. Das Prizip soll hier kurz dargestellt werden.

Schemaerhaltende Transformation zur Laufzeit

Die hier vorgestellte Lösung arbeitet nicht nur mit geschickter Transformation zur Laufzeit, sondern führt auch noch Transformationen zur Ladezeit durch. Diese sollen die Laufzeit-Transformationen vereinfachen bzw. erst ermöglichen. Der Vorteil der Ladezeit-Transformation ist, dass beliebige Transformationen durchgeführt werden können.

Es werden zur Ladezeit zwei Methoden – `callOrig` und `callAllBindings` – erzeugt. Zur Laufzeit wird `callAllBindings` von der neuen Basismethode benutzt, um den Aspektcode aufzurufen. Der Aspektcode benutzt `callOrig`, um den Originalcode der Basismethode aufzurufen. Jede Basismethode erhält eine eindeutige ID, um von `callOrig` und `callAllBindings` unterschieden werden zu können.

Ladezeit-Transformation. Jede Basis-Klasse, die geladen wird, erhält zwei zusätzliche Methoden – `callOrig` und `callAllBindings` – mit der folgenden Default-Implementierung:

```
Object callOrig(int methodID, Object[] args) {
        switch (methodID) {
        default:
                return null;
        }
}

Object callAllBindings(int methodID, Object[] args) {
        // sehen wird später
}
```

Außerdem erhalten die Basis-Klassen ein neues Interface, das sie implementieren. Der Basecall kann so generisch erfolgen:

```
Interface IBase {
        Object callOrig(int methodID, Object[] args);
}
```

Die neu eingeführten Methoden werden vorerst nicht ausgerufen und tun nicht viel. Das Verhalten der Klasse wird also nicht geändert. Diese Vorbereitung reicht, um eine schemaerhaltende Transformation zur Laufzeit durchführen zu können.

Laufzeit-Transformation. Nehmen wir an, zur Laufzeit soll eine neue Callin-Bindung für die Methode `m` realisiert werden:

```
class MyBase {
    int m(int i) {
        return i;
    }
}
team MyTeam {
    role MyRole playedBy MyBase {
        void rm() {}
        rm <- before m;
    }
}
```

Die Basismethode erhält dann zunächst eine eindeutige ID – zum Beispiel 1. Der Originalcode von `m` wird in die Methode `callOrig` verschoben – in das `switch`. Als `case`-Label wird ihre eindeutig ID benutzt:

```
Object callOrig(int methodID, Object[] args) {
    switch (methodID) {
    case 1:
        return ((Integer) args[0]).intVal();
    default:
        return null;
    }
}
```

Auch in `callAllBindings` wird Code ergänzt, er sorgt dafür, dass der Aspektcode ausgeführt wird:

```
Object callAllBindings(int methodID, Object[] args) {
        TeamNode first;
        // Liste der aktiven Teams erstellen,
        // sortiert nach Ausführungsreihenfolge.
        // first ist erstes Team und kennt den Nachfolger
        switch(methodID) {
        default:
                return first.callAllBindings(
                        this, first.next(), methodID, args);
        }
}
```

Man sieht, dass ein Team eine andere `callAllBindings`-Methode hat. Sie kann Aspektcode ausführen, Basecalls machen und das Nachfolge-Team aufrufen. Die Art und Reihenfolge der Ausführung hängt von der Art der Bindung und der `methodID` ab:

```
Object callAllBindings(
        IBase base, TeamNode successor,
        int methodID, Object[] args) {
        // Das Team weiß, was zu tun ist.
}
```

Der Code von `m` wird durch einen Aufruf von `callAllBindings` ersetzt:

```
int m(int i) {
        callAllBindings(1,
                new Object[] { /* Parameter einpacken */ }
        );
}
```

Konsequenzen

Diese Art der Methodcallinterception hat erhebliche Konsequenzen für diese Arbeit und konnte, weil dieses Konzept erst spät entwickelt wurde, in dieser Arbeit nicht von Anfang an berücksichtigt werden. An dieser Stelle soll trotzdem eine kleine Folgenabschätzung erfolgen und die Konsequenzen für diese Arbeit besprochen werden, also, wie die beiden Konzepte zusammen arbeiten können.

Keine Wrapper mehr. Um schemaerhaltend zu transformieren, können keine Wrappermethoden mehr benutzt werden. Der Joinpointcode muss stattdessen nach `callOrig` verschoben werden und statt des Wrappers muss `callAllBindings` aufgerufen werden. Da die Signatur von `callAllBindings` generisch ist, stimmt sie nicht mit der Joinpointsignatur überein. Deshalb muss zusätzlich zu dem Aufruf von `callAllBindings` nach dem Parameterladecode, Code eingefügt werden, der die auf dem Stack liegenden Parameter des Joinpoints, in ein `Object[]` einpackt. Alle Informationen, die man benötigt, um die Parameter einzupacken, sind die Parameteranzahl und die Typen der Parameter. Da die Signatur des Joinpoints bekannt ist, sind diese Informationen vorhanden. Die Parameter können eingepackt werden. Erst wird ein neues Array erstellt, dann wird Parameter für Parameter in das Array eingefügt. Referenztypen können direkt eingefügt werden. Primitive Datentypen müssen zuvor in entsprechende Wrapperobjekte eingepackt werden. So verpackt können die Parameter bis in `callOrig` transportiert werden. Sie dort wieder auszupacken und auf den Stack zu legen, ist trivial. Jetzt kann der Joinpointcode ausgeführt werden. Vor dem Parameterladecode müssen selbstverständlich ein `IBase`, das erste Team und die Methoden-ID geladen werden, damit `callAllBindings` aufgerufen werden kann.

Nochmal Mehrfachweben. Das Mehrfachweben kann mit schemaerhaltender Transformation durchgeführt werden. Es braucht, wie bereits gesagt, den Bytecode der Originalmethode und eine Tabelle, um die Joinpointpositionen in der alten Methode der neuen Methode zuzuordnen. Das Weben erfolgt schemaerhaltend mit `callAllBindings` und `callOrig`.

10.6 Vererbung

Der Grund, weshalb Vererbung beim Weben ein Problem machen kann, ist der, dass die Implementierung einer geerbten Methode in einer der Superklassen steht. Wenn Aspektcode in eine geerbt Methode gewoben werden soll, muss er in die Methode der Superklasse gewoben werden, was aber auch zu Folge hat, dass die Methode für alle Klassen, die sie erben geändert wird. Ein Umstand, den man vielleicht nicht wünscht. Wenn in Abbildung 10.1 die Implementierung der Methode `m` in Klasse `B1` geändert wird, dann ändert sich auch das Verhalten aller Subklassen (`B2`, `B3`).

Es soll möglich sein, die Implementierung für nur eine der Subklassen zu ändern. Die derzeitige Lösung in *ObjectTeams/Java* sieht so aus, dass die betreffende Methode redefiniert wird mit einer Implementierung, die einfach den Aufruf an die Superklasse delegiert. Der Aspektcode kann dann in die redefinierte Methode gewoben werden. Dieser Ansatz ist hier so nicht möglich denn die Joinpoints existieren nach so einer Redefinition trotzdem nur in der entsprechenden Methode der Superklasse.

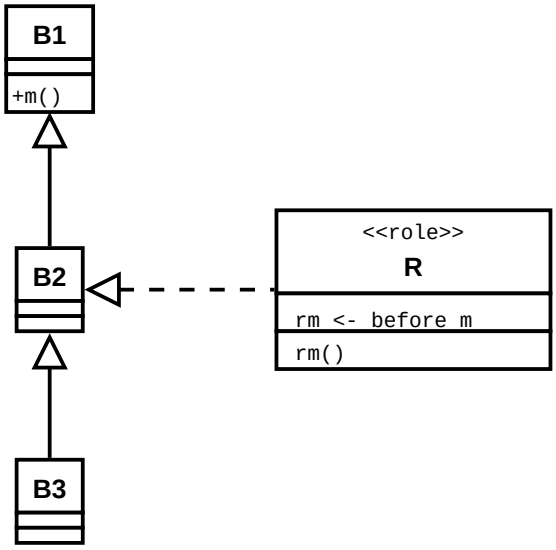

Abbildung 10.1: Auf einen Teil der Hierarchie beschränkter Aspekt.

Um das Weben hier dennoch selektiv zu erlauben, könnte man auf die Idee kommen, die Methode zu redefinieren und die Implementierung aus der Superklasse in die der Subklasse kopieren. Dies erfordert aber einiges an zusätzlichem Aufwand, weil die Implementierung der Superklasse auf Features – Felder und Methoden – mit der Sichbarkeit `private` zugreifen kann. Diese Features müssten entkapselt werden. In diesem Fall heißt das, die Sichbarkeit von `private` auf `protected` ändern und der Bytecode, der vormals private Methoden aufruft, muss von `invokespecial` in `invokevirtual` geändert werden.

Wenn die Subklasse `B2` weitere Subklassen hat, für die diese Änderung nicht erfolgen soll, dann muss in allen direkten Subklassen von `B2` – in Abbildung 10.1 wäre das `B3` – die Methode ebenfalls redefiniert werden und die Implementierung die Methoden der Superklasse von `B2` aufrufen.

Allerdings funktioniert dieses Verfahren nicht. Es ist nicht möglich, Felder in einer Klasse von private auf protected zu ändern. Weil protected-Felder vererbt werden und private-Felder nicht, gibt es das Problem, dass, wenn man eine solche Änderung durchführt, eine der Subklassen bereits ein private-Feld mit dem gleichen Namen, wie das neue protected-Feld besitzen kann. Das ergäbe eine Situation, die in Java nicht auftreten darf. Allerdings gibt es eine elegante Lösung für dieses Problem und sie basiert auf der im vorherigen Abschnitt vorgestellten schemaerhaltenden Transformation.

Weben in Klassenhierarchien mit schemaerhaltender Transformation

Wenn in einer Vererbungshierarchie (z. B. `MyBase2 extends MyBase`) eine geerbte Methode (in diesem Beispiel `m`) gebunden werden soll, besteht das Problem, dass die Implementierung nur in der Superklasse steht. Dort darf sie nicht geändert werden, wenn sie nur für eine bestimmte Unterklasse geändert werden soll. Das hier vorgestellte Verfahren bietet eine elegante Lösung für das Problem.

Die Methode `m` bekommt wieder eine ID, die Implementierung wird nach `callOrig` verschoben, `m` ruft callAllBindings auf – alles wie gehabt. Nur der Code von `callAllBindings` sieht anders aus, der ruft einfach nur den Originalcode auf:

```
Object callAllBindings(int methodID, Object[] args) {
        // ...
        switch(methodID) {
        case 1:
                return callOrig(methodID, args);
        default:
                return first.callAllBindings(
                        this, first.next(), methodID, args);
        }
}
```

Das Verhalten von `MyBase` wurde nicht geändert, aber das Verhalten von `MyBase2` kann jetzt geändert werden, indem `callAllBindings` redefiniert wird:

```
Object callAllBindings(int methodID, Object[] args) {
        // ...
        switch(methodID) {
        case 1:
                return first.callAllBindings(
                        this, first.next(), methodID, args);
        default:
                return super.callBindings();
        }
}
```

Wird `callAllBindings` der Superklasse aufgerufen, wird der Originalcode ausgeführt. Wenn man die Subklasse aufruft, wird die gebundene Version aufgeführt.

Kapitel 11

Implementierung des Webens

In diesem Kapitel wird die Implementierung des Webens erklärt. Dazu werden zuerst die verschiedenen Implementierungsmöglichkeiten besprochen und dann, wie das Weben mit Hilfe der bereits in der *ObjectTeams*-Laufzeitumgebung vorhandenen Funktionalität implementiert wurde.

11.1 Implementierungsmöglichkeiten

Es gibt verschiedene Implementierungsmöglichkeiten. Sie sollen – zusammen mit ihren Konsequenzen – in diesem Abschnitt besprochen werden.

Neuimplementierung. Die erste Möglichkeit wäre, das benötigte Verhalten komplett neu zu implementieren. Wenn man dabei nicht auf Reflections zurückgreift, was wegen der Performance nicht erwünscht ist, muss man sehr viel Entwicklungsarbeit leisten, nur um einen Ansatz zu erhalten, der arbeitet, wie das OTRE jetzt. Ein solcher Ansatz dürfte kaum mit dem OTRE mithalten, da in die Entwicklung des OTRE über zwei Jahre investiert wurden.

Gezielte Änderungen im OTRE. Es liegt also nahe, das gewünschte Verhalten nicht von Grund auf neu zu programmieren, sondern Verhalten des OTRE wieder zu verwenden. Dazu könnten zusätzlich zum Hinzufügen eines neuen Transformers bestehende Transformer geändert werden. Dies stellt sich jedoch aus zwei Gründen als problematisch heraus.

Der Code der *ObjectTeams*-Laufzeitumgebung ist in zwei Jahren Entwicklungszeit deutlich gewachsen, was Spuren hinterlassen hat. Es treten verschiedene Probleme auf: *Lange Methode, Große Klasse, Lange Parameterliste, Duplizierter Code* und *Divergierende Änderungen* [13]. Selbst kleinere Änderungen sind, deshalb nur mühsam umzusetzten. Eigentlich müssten zahlreiche Refactorings durchgeführt werden. Ein Problem dabei ist die Integration. Es existieren verschiedene Entwicklungszweige des OTRE. In allen Entwicklungszweigen wird an unterschiedlichen Stellen des OTRE gearbeitet. Damit die Entwicklungszweige dem Entwicklungsstand des Hauptzweigs entsprechen, müssen sie regelmäßig zum Hauptzweig synchronisiert werden. Damit sind größere Änderungen – zum Beispiel *Große Refaktorisierungen* – im Entwicklungszweig unmöglich, da sie das Design des Hauptzweigs erheblich ändern können. Außerdem erschweren große Änderungen eine erfolgreiche Integration der einzelnen Entwicklungszweige mit dem Hauptzweig erheblich. Wenn irgendwo große Änderungen möglich sind, dann nur im Hauptzweig direkt, darauf hat der Autor jedoch keinen Zugriff. Aus diesen Gründen sollen am bestehenden OTRE zu wenig Änderungen wie möglich gemacht werden.

Copy'n'Paste-Wiederverwendung. Um von dem Know-how des OTRE zu profitieren und trotzdem größere Änderungen durchführen zu können, wäre es möglich, benötigten Code einfach aus einem anderen Transformer zu kopieren. Allerdings würde dies zu einer erheblichen Menge dupliziertem Code führen, der vermutlich unwartbar würde. Copy'n'Paste ist daher nie eine wirkliche Option und sollte hier nur der Vollständigkeit halber angesprochen werden.

Derzeitiges Weben wiederverwenden. Das OTRE unterstützt bereits jetzt Weben an Execution-Joinpoints (Methodcallinterception). Durch den Einsatz von Wrappern kann jeder Joinpoint wie ein Execution-Joinpoint behandelt werden. Dieser Ansatz soll daher weiterverfolgt werden, denn er bietet einerseits die Möglichkeit vom OTRE-Know-how zu profitieren und andererseits den Vorteil, sehr wenig Code des OTRE modifizieren zu müssen.

11.2 Ablauf der Transformation

Eine Teil der Transformation wurde bereits vorgestellt, als über das Weben gesprochen wurde (Abbildung 11.1). An dieser Stelle soll der Rest besprochen werden, der fürs Weben zuständig ist:

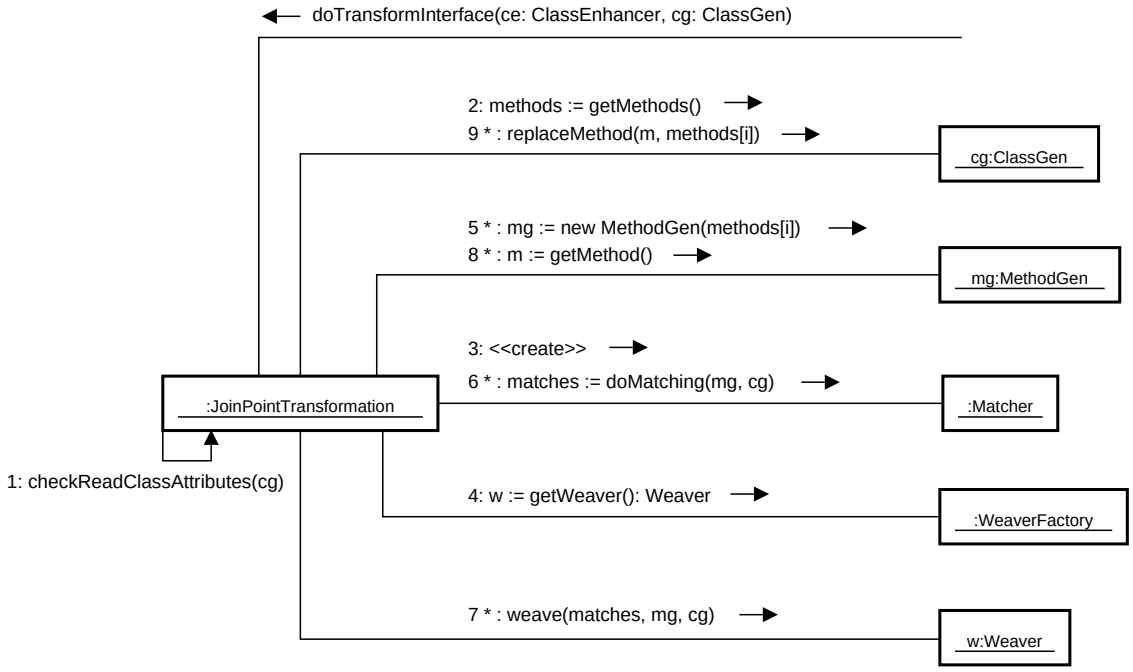

Abbildung 11.1: Ablauf der Transformation.

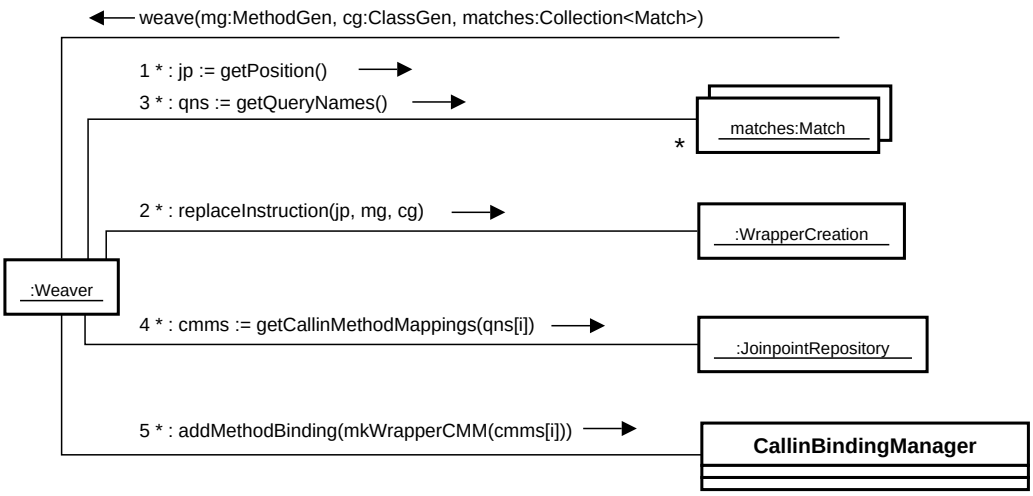

Abbildung 11.2: Ablauf des Webens.

- *Aktion 4*: Der Transformer lässt sich von der `WeaverFactory` einen Weber geben. Welcher das ist, weiß er nicht, es kann anhand von Java-Properties gesteuert werden.
- *Aktion 7*: Die Treffer werden – zusammen mit der zu transformierenden Klasse und Methode – an den Weber übergeben, damit dieser das Weben durchführt.
- *Aktion 8*: Der Bytecode, der transformierten Methode, wird neu generiert.
- *Aktion 9*: Dann wird die alte, durch die neue Methode ersetzt.

In diesem Kapitel interessiert natürlich besonders, der Teil des Weben. Er ist in Abbildung 11.2 genauer dargestellt:

- *Aktion 1*: Der Weber arbeitet alle `matches` nacheinander ab. Zuerst lässt er sich von einem `Match` die Position geben.
- *Aktion 2*: Die Position übergibt er samt `MethodGen` und `ClassGen` an `WrapperCreation`. `MethodGen` und `ClassGen` sind die BCEL-Repäsentationen von Klasse und Methode. `WrapperCreation` erzeugt für den Joinpoint einen Wrapper, verschiebt den Joinpointcode in den Wrapper und sorgt dafür, dass stattdessen der Wrapper aufgerufen wird. Die genaue Implementierung von `WrappperCreation` ist nicht weiter interessant. Sie verwendet BCEL, um die im Joinpointkatalog beschriebenen Transformationen durchzuführen. Einzig, wie der Anfang des Parameterladecode gefunden wird ist interessant und wird, im Anschluss auf diese Aufzählung erklärt.

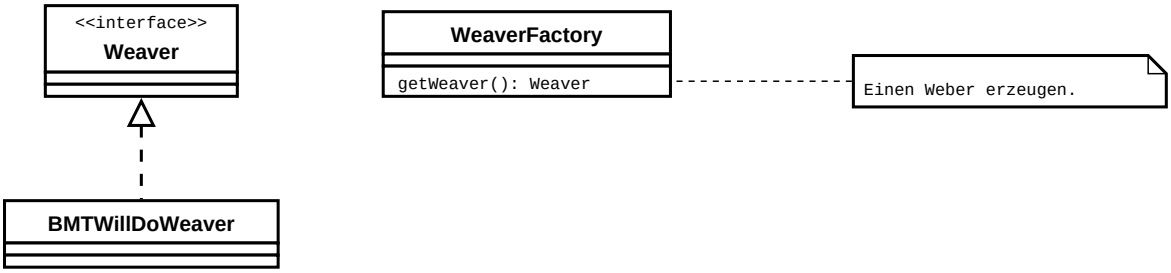

Abbildung 11.3: Die Weber-Hierarchie mit Factory.

- *Aktion 3*: Als nächstes interessieren die Queries, die an den Joinpoint gebunden werden sollen. In dieser Aktion werden ihre Namen aus dem `Match` geholt.

- *Aktion 4*: `JoinpointRepository` verwaltet die `CallinMethodMappings` und kann zu einem Querynamen eben diese liefern.

- *Aktion 5*: Dem `CallinBindingManager` – über ihn wird gesteuert, wie die anderen Transformer Methoden ersetzten – wird eine neue Callin-Methoden-Bindung übergeben. Diese Bindung teilt den anderen Transformern mit, dass an die neuen Wrappermethode eine Rollenmethode gewoben werden soll. Dazu werden dem `CallinBindingManager` Informationen wie Basismethodenname (In diesem Fall der Wrapper), Rollenmethodenname, Bindungsmodifikator, Rollenmethodenwrapper etc. übergeben. Den Rest erledigen die anderen *ObjectTeams*-Transformer. Im Abschnitt 11.3 wird erklärt, was sie machen.

WrapperCreation. Um den Anfang des Parameterladecode zu finden, geht `WrapperCreation`, beginnend bei der Spur des Joinpoints, rückwärts durch die Instruktionsliste und berechnet die Stackdifferenz (Anzahl produzierte Befehle - Anzahl konsumierte Befehle). Angenommen, der Joinpointcode konsumiere 4 Operanden vom Stack, dann ist -4 der Startwert. Auf den Startwert wird jetzt solange die Stackdifferenz der Vorgängerbefehle summiert, bis der Wert 0 erreicht wurde. Jetzt befindet man sich am Anfang des Parameterladecodes.

11.3 Execution-Joinpoint in ObjectTeams

In diesem Abschnitt soll beschrieben werden, wie Methodcallinterception bzw. das Weben eines Execution-Joinpoint zur Zeit realisiert ist. Die Basismethode heiße `T m(a1, ...)`. Zuerst wird die Basismethode umbenannt. Sei heißt dann `T _OT$m$orig(a1, ...)`. An die Stelle von `m` tritt ein sogenannter *initialer Wrapper*. Wenn die Basismethode aufgerufen wird, wird also der initiale Wrapper ausgeführt, denn er hat die Basismethode ersetzt. Jetzt sollen die gebundenen Rollenmethoden aller aktiver Teams aufgerufen werden. Der Wrapper erstellt deshalb zuerst ein Array aller aktiver Teams. Durch das Kopieren wird sichergestellt, dass sich die Menge der Teams, an denen Rollenmethoden aufgerufen werden sollen, nicht ändert. OT/J unterstützt nämlich Threads, d. h. ein Team könnte während der Ausführung einer Rollenmethode aktiv oder passiv werden. Dann ruft der initiale Wrapper einen sogenannten *Chainingwrapper* auf. Er sieht folgendermaßen aus[1]: `T _OT$m$chain(Team[], int index, a1, ...)`. Der Chainingwrapper traversiert rekursiv über die Teams und sorgt dafür, dass die Rollenmethoden aufgerufen werden. Die Rollenmethoden werden allerdings nicht direkt aufgerufen, sondern es wird ein Rollenmethodenwrapper, der sich im Team befindet, aufgerufen. Die Rollenmethodenwrapper werden vom Compiler erzeugt.

Bei `replace`-Bindung kann die Rollenmethode einen Basecall machen, also die Basismethode aufrufen. Dies wird durch einen geeigneten Aufruf des Chainingwrappers realisiert. Das genau Vorgehen kann in [4] nachgelesen werden (*3.4.2 Realisierung des base-calls*).

11.4 Sonstiges

Kollisionen. Es wurde bereits in Abschnitt 10.1 angesprochen, dass beim Matching Kollisionen auftreten können und wie man sie behandelt. Allerdings würde noch nicht gesagt, wie der Prototyp mit solchen Kollisionen umgeht.

Die Kollisionen werden im Matcher behandelt. Es existiert ein Array `Match[] matches`, mit so vielen Einträgen, wie die entsprechende Methode Instruktionen hat. Ein Match kann genau eine Position und mehrere Queries haben. Wenn eine Query auf einen Joinpoint matcht, wird die Bytecodeposition der Spur des Joinpoints als Index benutzt. An der entsprechenden Position im `matches`-Array wird ein neuer `Match` erzeugt oder wenn schon einer existiert, zu diesem eine neue Query hinzugefügt. Queries können nicht doppelt hinzugefügt werden. Dadurch wird sichergestellt, dass für jeden gefundenen Joinpoint alle Queries bekannt sind, die gewoben werden sollen. Sie werden alle zusammen behandelt.

[1] Die Signatur ist etwas vereinfacht.

Andere Webestrategien. Der Prototyp ist so konzipiert, dass auch andere Webstrategien leicht realisiert werden können. Soll eine andere Webestrategie eingeführt werden, muss einfach ein neuer Weber geschrieben werden. Dieser Weber muss lediglich das `Weaver`-Interface implementieren. Danach muss nur noch der `WeaverFactory`, der neue Weber, bekannt gemacht werden. Dann kann über *Java-Properties* ausgewählt werden, welcher Weber verwendet werden soll (Abbildung 11.3). Man siehe dazu in den Code. Er ist selbst erklärend.

Toolkit-Benutzung. Einen Joinpoint in einen Wrapper zu kapseln, wird durch die Verwendung eines Toolkits erheblich vereinfacht. Der Aufruf eines Wrappers kann nämlich, vorhandenen Code verschieben oder die maximale Stackgröße verändern. Solche Fälle müssen beim Weben ohne Toolkit besonders behandelt werden, z. B. um die korrekten Sprungziele wiederherzustellen. Mit BCEL entfällt eine besondere Behandlung.

Performance. Das Weben kann sich in zweierlei Hinsicht auf die Performance einer Software auswirken.

Beim Weben kommt, wie auch beim Matching, Overhead zur Ladezeit hinzu. Dieser wird durch den Vorgang des Webens verursacht. Dieser Vorgang lässt sich vielleicht durch Verzicht auf ein Bytecode-Toolkit beschleunigen. Aber ob diese Beschleunigung so groß ausfällt, ist auch in diesem Fall nicht sicher.

Nach dem Weben wird nicht mehr der Originalcode ausgeführt, sondern der modifizierte. Er enthält Aufrufe des Chaining-Wrappers, Joinpoint-Wrappers, des Aspektwrappers etc. Auf die von der herkömmlichen Laufzeitumgebung erzeugten Konstrukte hat der Prototyp keinen Einfluss. Nur der Wrapperaufruf wird vom Prototypen erzeugt. Methodenaufrufe können in Java unter bestimmten Voraussetzungen sehr schnell durchgeführt werden. Im *Handbuch der Javaprogrammierung* [10] werden für unterschiedliche Methodenaufrufe folgende Wert angegeben:

Signatur	***Zeit (ms)***
public	280
public, mit 4 Parametern	390
public static	110
protected	280
private	50
public synchronized	4660
public final	50

Ein Instanzmethoden-Wrapper sollte also als `final` deklariert werden, dann kann der Vorgang unter bestimmten Umständen bis zu fünf Mal schneller durchgeführt werden, als wenn man die Signatur falsch wählt. Fakt ist, dass durch den Wrapper kein nennenswerter Overhead erzeugt wird. Nicht mehr als durch den Refactoringschritt *Methode extrahieren*.

Kapitel 12

Fazit

Keine Diplomarbeit löst das an sie gestellte Problem vollständig, deshalb soll dieses Kapitel zusammenfassen, was im Rahmen dieser Diplomarbeit erreicht wurde und was in Zukunft noch getan werden muss, um die gestellten Fragen vollständig zu lösen. Außerdem sollen einige Themen, die bisher keine Erwähnung gefunden haben, kurz angesprochen werden.

Was war die Aufgabe?

Kurz gesagt, bestand die Aufgabe darin, Konzepte zu erarbeiten, um Joinpointinterception für *ObjectTeams/Java* (OT/J) zu realisieren und dies prototypisch zu implementieren. Der Wunsch nach Joinpointinterception war aufgekommen, weil OT/J bis heute nur Methodcallinterception unterstützt.

12.1 Was wurde erreicht?

Es wurden zahlreiche Konzepte entwickelt und beschrieben, wie die facettenreichen Probleme, die bei der Realisierung von Joinpointinterception auftreten, gelöst werden können. Dazu gehört zum Beispiel ein Joinpointkatalog, der die identifizierten Joinpoints enthält und ihre Eigenschaften beschreibt. Zum Beispiel ihre Signatur und wie man an ihren Aspektcode weben kann. Dazu wurde unter anderem das Konzept der Wrappermethoden entwickelt, das es ermöglicht, Joinpoints in eine eigene Methode auszulagern, um das Problem der Joinpointinterception, auf das der Methodinterception herunter zu brechen. Die Identifizierung von

Joinpointeigenschaften ermöglichte, es das Konzept der Muster zu entwickeln, mit denen Joinpoints abstrakt bzw. robust beschrieben werden können. Weiterhin wurde erklärt, wie dieses Konzept mit anderen Konzepten von Programmiersprachen, z. B. Vererbung, Exceptions und Nebenläufigkeit, zusammenspielen. Einige, wenn auch nicht alle dieser Konzepte, wurden prototypisch implementiert, um ihr prinzipielles Funktionieren zu bestätigen.

Welche Grundlagen gab es bereits?

Diese Diplomarbeit baut auf einer Vielzahl an bereits vorhandenen Techniken auf, von denen einige selbst erst prototypisch realisiert sind. Damit ist vor allem die Querysprache und ihre Realisierung im *ObjectTeams/Java*-Compiler gemeint. Auch die bereits vorhandene Unterstützung für *Muster* im Compiler, ist zur Zeit nur rudimentär. Der Compiler unterstützt nur ein Signatur-Ziel-Muster. Schemaerhaltende Transformation ist eine weitere Grundlage auf der, wenn auch erst zu Ende dieser Arbeit, aufgebaut werden konnte[1]. Die funktionsfähige Laufzeitumgebung von *ObjectTeams* und das Bytecode-Toolkit BCEL waren weitere wichtige Grundlagen.

12.2 Was ist noch zu tun?

Um die Aufgabe Joinpointinterception zu realisieren und vollständig abzuschließen, sind noch einige Dinge zu tun, die in dem Prototypen noch nicht funktionieren, deshalb sollen sie hier beschrieben werden.

Laufzeitauswertung von Queries

Zu den wichtigsten Dingen, die noch vervollständigt werden müssen, gehört wohl die Laufzeitauswertung von Queries. Sie sollte, wie in Abschnitt 6.4 beschrieben, durch vom Compiler generierte Methoden realisiert werden.

Neue Muster

Das in Abschnitt 6.5 vorgestellte Pattern, das mit Indizes ins Bytecode-Array einer Methode und CRCs arbeitet, sollte vom Compiler benutzt werden, weil es schnell auszuwerten ist und

[1] Das Konzept wurde erst kürzlich in einer anderen Diplomarbeit entwickelt.

trotzdem die beste Robustheit liefert. Dazu wird allerdings auch die Laufzeitauswertung von Queries benötigt.

Itegration mit anderen Prototypen

Der hier entwickelte Prototyp muss zusammen mit anderen Prototypen in den Entwicklungshauptzweig des OTRE integriert werden. Andere Prototypen meint vor allem den in [14] entwickelten Prototypen, weil dieser eine erhebliche Relevanz für diese Arbeit besitzt (siehe auch 10.5).

Mehr Joinpoints

Zur Zeit werden nur die Joinpoints Feldzugriff (auch statische), Methodenaufrufe (auch statische) und Objekterzeugung unterstützt. Im Joinpointkatalog ist beschrieben, wie die anderen Joinpoint realisiert werden können, dies ist noch offen. Ein Joinpoint für Schleifen ist auch eine sinnvolle Erweiterung (siehe Abschnitt 4.2).

Afterbindung

Im Prototypen funktioniert derzeit keine Afterbindung, weil die für Queries von Compiler erzeugten Wrappermethoden inkompatibel sind, wenn die Joinpointbindung folgende Struktur hat:

```
test <- after query getterCalls<MyBase>;
```

Dann wird vom Compiler folgende Wrappermethode generiert:

```
public void _OT$R1$test$getterCalls(MyBase, java.util.Set);
```

Die OTRE Basismethodentransformation, die mit dem Wrapper Methodecallinterception durchführen soll, erzeugt im Chaining-Wrapper folgenden Aufruf:

```
1 ...
2 54: lookupswitch{ //1      1: 72;       default: 92 }
3 72: aload 9
4 74: checkcast MyTeam
5 77: aload_0
```

```
6  78: aload 8
7  80: checkcast Integer
8  83: invokevirtual Integer.intValue:()I
9  86: invokevirtual _OT$R1$test$getterCalls:
10         (LMyBase;Ljava/util/Set;)V
11 ...
```

Das passt natürlich nicht zusammen (Zeilen 8-9) und die Bytecode-Verifikation scheitert, weil statt einer Referenz ein int auf den Stack liegt:

```
Exception in thread "main" java.lang.VerifyError:
(class: MyBase, method: _OT$_OT$foo$8$proxy$1$chain
signature: ([Lorg/objectteams/Team;
        [IIII[Ljava/lang/Object;LMyBase;)Ljava/lang/Object;)
Expecting to find object/array on stack
```

Replacebindung

Im Prototypen funktioniert derzeit keine Replacebindung, weil kein Basecall möglich ist. Das liegt daran, dass Basecalls über ein *Basecall-Surrogate* realisiert sind, das nur zur Ladezeit eines Teams implementiert werden kann. Zu diesem Zeitpunkt ist aber noch nicht bekannt, welche gebundenen Joinpoints bzw. Wrapper es in der Basisklasse gibt – schließlich wurde noch kein Matching durchgeführt. Durch eine generische Defaultimplementierung ließe sich das Problem wahrscheinlich lösen.

12.3 Sonstiges

Systemtests mit jacks

Im Rahmen dieser Diplomarbeit wurde die umfangreiche Testsuite von *ObjectTeams/Java* um einige Testfälle erweitert. Die Testsuite basiert auf dem *Jacks* Java-Compiler-Testing-Framework. Um den Rahmen dieser Diplomarbeit nicht zu sprengen, wurde auf eine Beschreibung, wie dies passiert ist, verzichtet.

Performanceoptimierungen

Mit dem entwickelten Prototypen wurden keine Performanceoptimierungen durchgeführt. Dazu sollten bei Gelegenheit Performancetests nachgeholt werden, um Schwachstellen in der Implementierung des Prototypen zu finden. Auch ein Vergleich, wie lange das Weben mit Methodcallinterception dauert und wie lange mit Joinpointinterception, ist wünschenswert. Es wird zeitgleich an einer anderen Diplomarbeit gearbeitet, die Performanceuntersuchungen am OTRE durchführt.

Inspiration

Bevor die Implementierung des Prototypen erfolgte, wurde der AspectJ-Compiler *Aspect Bench Compiler* (ABC) [15, 16] genauer betrachtet. Auch wenn keine Design-Entscheidungen direkt übernommen wurden, ist es doch möglich, dass die Kenntnis des *abc* einen Einfluss darauf hatte.

Joinpoints für OT-Konstrukte

Es wurden ausschließlich Joinpoint betrachtet, die in Java vorkommen, aber auch *Object-Teams* könnte Joinpoints enthalten, die in Java gar nicht enthalten sind. Hier sollte eine entsprechende Analyse durchgeführt werden.

Anhang A

Joinpointkatalog

Der Joinpointkatalog enthält alle im Rahmen dieser Arbeit wichtigen Joinpoints. Jeder Joinpointeintrag fasst die wichtigsten Eigenschaften eines Joinpoints zusammen. Die einzelnen Felder sind größenteils selbsterklärend. Bei Joinpoints, wo einer der Einträge überflüssig ist, fehlt dieser. Die einzelnen Einträge haben folgende Bedeutung:

Beschreibung
: Beschreibung des Joinpoints.

Java
: Java-Syntax des Joinpoints.

Bytecode
: Bytecode in den das Javakonstrukt übersetzt wird.

Ziel
: Zugriffsziel des Joinpoints.

Signatur
: Signatur des Joinpoints. Alle Parameter werden aufgelistet, auch implizite.

Spur
: Spur, an der man den Joinpoint erkennt.

Exception
: Exceptions, die der Joinpoint werfen kann.

Neuer Bytecode
Bytecode nach der Transformation.

Wrapper
Code des Wrappers.

Pattern
Regulärer Ausdruck für den BCEL-`InstructionFinder`, mit dem nach dem Joinpoint gesucht werden kann.

Notation. Im Joinpointkatalog werden einige Abkürzungen benutzt. Ihre Bedeutung wird in der folgenden Tabelle erklärt:

Abkürzung	***Beschreibung***
`o`	*Objekt*
`m`	*Methode*
`I`	*Interface*
`C`	*Klasse*
`f`	*Feld*
`T`	*Referenz- oder Elementartyp*
`i`	*Index*
`e`	*Exception*
`T`x	*Typ von* x
`C`x	*Klasse von* x
`v`	*Wert*
`x`	*Lokale Variable*
args	*Parameterliste*
$x \sqsubseteq y$	x *ist konform zu* y
$x \triangleright y$	x *ist Subtyp von* y
x `(` y `)`	*Signatur mit Rückgabewert* x *und Parameterliste* y
`{a,b,c}`	`a`, `b` oder `c`
Joinpointcode	*Der gleiche Code, wie in Bytecode*
Parameterladecode	*Code, der in Basismethode die Parameter lädt*
Parameter laden	*Code, der in Wrapper die Parameter lädt*
`*return`	*Geeigneter* `return`*-Befehl*

A.1 Methodenaufrufe

Instanzmethode aufrufen

Beschreibung
Aufruf einer Instanzmethode.

Java
`o.m(`*args*`)`

Bytecode
`invokevirtual`
`invokespecial`
`invokeinterface`

Ziel
Methode `m` (*Außer* `<init>`)

Signatur
`T (Co,` *args*`)`

Spur
Wie Bytecode

Exception
Wie Methode

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`*return`

Pattern
`"(invokevirtual|invokeinterface|invokespecial)"`

Klassenmethode aufrufen

Beschreibung
Aufruf einer Klassenmethode.

Java
`C.m(`*args*`)`

Bytecode
`invokestatic`

Ziel
Methode `m`

Signatur
`T (`*args*`)`

Spur
Wie Bytecode

Exception
Wie Methode

Neuer Bytecode
Parameterladecode
`invokestatic wrapper`

Wrapper
Parameter laden
Joinpointcode
`*return`

Pattern
`"invokestatic"`

A.2 Objekterzeugung und Initialisierung

Objektinstanzierung

Beschreibung
Instanzierung eines Objekts.

Java
`new C(`*args*`)`

Bytecode
`new C`
args laden
`invokespecial C.<init>`

Ziel
Klasse `C`

Signatur
`C (`*args*`)`

Spur
`invokespecial <init>`
Das zugehörige `new` *muss gefunden werden. Rückwärts gehen und andere* `invokespecial <init>` *zählen. Dann findet man das zugehörige* `new`.

Exception
Wie Konstruktur

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
`new`
`dup`
Parameter laden
Joinpointcode
`*return`

Pattern
`"invokespecial"`

Statische Klasseninitialisierung

Beschreibung
Initialisierung der Felder einer Klasse `C`, passiert in der Methode `<clinit>`, die implizit von der *JVM* aufgerufen wird.

Java
implizit

Bytecode
implizit

Ziel
Klasse `C`

Signatur
keine

Spur
Existenz der Methode `<clinit>`

Feld-Initialisierung

Beschreibung
Initialisierung eines Feldes `f` einer Klasse `C` oder eine Objekts `o`. Die Initialisierung passiert innerhalb der Methoden `<init>` (für Instanzfelder) und `<clinit>` (für Klassenfelder). Die Initialisierungen sind – teilweise – ohne Hinweise durch den Compiler nicht von Initialisierungen, die im Konstruktor definiert wurden, unterscheidbar.

Java
`T f = v` *oder*
`static T f = v`

Bytecode
`putfield` (*innerhalb von Methode* `<init>`)
`putstatic` (*innerhalb von Methode* `<clinit>`)

Ziel
Feld `f`

Signatur
`void (Tf)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
Wie Feld lesen (siehe unten)

Wrapper
Wie Feld lesen (siehe unten)

Pattern
Wie Feld lesen (siehe unten)

A.3 Feldzugriffe

Feld schreiben

Beschreibung
Einen Wert in das Feld eines Objekts schreiben.

Java
`o.f = v`

Bytecode
`putfield`

Ziel
Feld `f`

Signatur
`void (To, Tf)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
`putfield`
`return`

Pattern
`"putfield"`

Feld lesen

Beschreibung
Einen Wert aus einem Feld eines Objekts lesen.

Java
`o.f`

Bytecode
`getfield`

Ziel
Feld `f`

Signatur
`Tf ()`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
`getfield`
`*return`

Pattern
`"getfield"`

Statisches Feld schreiben

Beschreibung
Einen Wert in das Feld einer Klasse schreiben.

Java
`C.f = v`

Bytecode
`putstatic`

Ziel
Feld `f`

Signatur
`void (Tf)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
Parameterladecode
`invokestatic wrapper`

Wrapper
Parameter laden
`putstatic`
`return`

Pattern
`"putstatic"`

Statisches Feld lesen

Beschreibung
Einen Wert aus einem Feld einer Klasse lesen.

Java
`C.f`

Bytecode
`getstatic`

Ziel
Feld `f`

Signatur
`Tf ()`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
Parameterladecode
`invokestatic wrapper`

Wrapper
Parameter laden
`getstatic`
`*return`

Pattern
`"getstatic"`

A.4 Arrayzugriffe

Wert aus Array lesen

Beschreibung
Einen Wert aus einem Array lesen.

Java
`a[i]`

Bytecode
`{b,s,i,l,f,d,c,a}aload`

Ziel
keins

Signatur
`T (Ta, int)`

Spur
Wie Bytecode

Exception
`ArrayIndexOutOfBoundsException`

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`*return`

Pattern
`"(baload|saload|caload|iaload|aaload|faload|laload|daload)"`

Wert in Array schreiben

Beschreibung
Einen Wert in ein Array schreiben.

Java
`a[i] = v`

Bytecode
`{b,s,i,l,f,d,c,a}astore`

Ziel
keins

Signatur
`void (Ta, int, T)`

Spur
Wie Bytecode

Exception
`ArrayIndexOutOfBoundsException`

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`return`

Pattern
`"(aastore|fastore|lastore|castore|iastore|bastore|sastore|dastore)"`

Arraylänge abfragen

Beschreibung
Die Länge eines Arrays abfragen.

Java
`a.length`

Bytecode
`arraylength`

Ziel
keins

Signatur
`int (Ta)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`ireturn`

Pattern
`"arraylength"`

A.5 Typprüfungen

Typcast

Beschreibung
Einen Typ in einen anderen casten.

Java
`(C) o`

Bytecode
`checkcast`

Ziel
Klasse `C`

Signatur
`C (Co)`

Spur
Wie Bytecode

Exception
`ClassCastException`

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`areturn`

Pattern
`"checkcast"`

Dynamische Typüberprüfung

Beschreibung
Ein Objekt zur Laufzeit auf seinen Typ prüfen – mit `instanceof`.

Java
`o instanceof C`

Bytecode
`instanceof`

Ziel
Klasse `C`

Signatur
`boolean (Co)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`ireturn`

Pattern
`"instanceof"`

A.6 Exeptions und Synchronization

Ausnahme werfen

Beschreibung
Eine Ausnahme werfen.

Java
`throw e`

Bytecode
`athrow`

Ziel
keins

Signatur
`void (Ce) throws Ce`

Spur
Wie Bytecode

Exception
`Ce`

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode

Pattern
`"athrow"`

Ausnahme fangen

Beschreibung
Eine Ausnahme fangen.

Java
`catch (Exception e) { ... }`

Bytecode
Taucht nur in der Exceptionhandler-Tabelle auf.

Ziel
keins

Signatur
`void (Ce) throws Ce`

Spur
Eintrag in der Exceptionhandler-Tabelle.

Exception
`Ce`

Monitor betreten

Beschreibung
In einen Monitor eintreten. Synchronisation auf Methodenebene wird implizit durch die VM realisiert.

Java
```
synchronized (o) {
    Monitor wird implizit betreten
    ...
    Monitor wird implizit verlassen
}
```

Bytecode
`monitorenter`

Ziel
Klasse `Co`

Signatur
`void (Object)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`return`

Pattern
`"monitorenter"`

Monitor verlassen

Beschreibung
In einen Monitor verlassen.

Java
```
synchronized (o) {
    Monitor wird implizit betreten
    ...
    Monitor wird implizit verlassen
}
```

Bytecode
`monitorexit`

Ziel
Klasse `Co`

Signatur
`void (Object)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`return`

Pattern
`"monitorenter"`

A.7 Dereferenzierung

Dereferenzierung

Beschreibung
Ein Objekt wird dereferenziert, d. h. es wird auf ein Feature des Objekts zugegriffen – das kann ein Feld oder eine Methode sein. *feature* sei also ein Feld oder eine Methode. Die Signature von *feature* sei `X (Y)`.

Java
`o.`*feature*

Bytecode
Objekt `o` *laden, dann einer der folgenden Befehle ...*
`invokevirtual`
`invokeinterface`
`invokespecial` (*kein* `<init>`)
`putfield`
`getfield`

Ziel
Feature, als Methode `m` oder Feld `f`

Signatur
`X (Co, Y)`

Spur
Wie Bytecode

Exception
`NullPointerException`

Neuer Bytecode
Siehe betreffende Befehle

Wrapper
Siehe betreffende Befehle

Pattern
`"(invokevirtual|invokeinterface|invokespecial|putfield|getfield)"`

A.8 Lokale Variablen und Arithmetik

Lokale Variable lesen

Beschreibung
Eine lokale Variable lesen.

Java
`x`

Bytecode
`{i,l,f,d,a}load`

Ziel
keins

Signatur
`Tx ()`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
```
aload_0
{i,l,f,d,a}load
invokespecial wrapper
```

Wrapper
Parameter laden
`*return`

Pattern
`"loadinstruction"`

Lokale Variable schreiben

Beschreibung
Einen Wert in einer lokalen Variablen speichern.

Java
`x = v`

Bytecode
`{i,l,f,d,a}store`

Ziel
keins

Signatur
`Tv (Tv)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper {i,l,f,d,a}store`

Wrapper
Parameter laden
`*return`

Pattern
`"storeinstruction"`

Variable in- oder dekrementieren

Beschreibung
Eine Variable um einen Wert in- oder dekrementieren.

Java
`i++` *oder*
`i--` *oder*
`i += 2`

Bytecode
`iinc`

Ziel
keins

Signatur
`int (int, int)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Variable laden
Inkrementwert laden
`invokespecial wrapper`
Wert speichern

Wrapper
`aload_1`
`aload_2`
`iadd`
`ireturn`

Pattern
`"iinc"`

Binäre Operation

Beschreibung

Eine binäre Operation. Das Ergebnis der Operation habe Typ `T`. Vergleichsoperatoren sowie `&&` und `||` mit Short-Circuit-Evaluation sind hier nicht enthalten, denn sie werden mit Hilfe von Sprungbefehlen realisiert.

Java

`a + b` *oder*
`a - b` *oder*
`a * b` *oder*
`a / b` *oder*
`a % b` *oder*
`a << b` *oder*
`a >> b` *oder*
`a >>> b` *oder*
`a & b` *oder*
`a | b` *oder*
`a ^ b`

Bytecode

`{i,l,f,d}add` *oder*
`{i,l,f,d}sub` *oder*
`{i,l,f,d}mul` *oder*
`{i,l,f,d}div` *oder*
`{i,l,f,d}rem` *oder*
`{i,l}shl` *oder*
`{i,l}shr` *oder*
`{i,l}ushr` *oder*
`{i,l}and` *oder*
`{i,l}or` *oder*
`{i,l}xor`

Ziel

keins

Signatur

```
T (Ta, Tb)
```

Spur

Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode `*return`

Pattern
`"arithmeticinstruction"`

Unäre Operation

Beschreibung
Anwendung eines unären Operators. Der Operator ~ ist nicht enthalten. Er wird mit Hilfe von XOR realisiert.

Java
`-a`

Bytecode
`{i,l,f,d}neg`

Ziel
keins

Signatur
`T (Ta)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`*return`

Pattern
`"arithmeticinstruction"`

Konstante benutzen

Beschreibung
Benutzung einer Konstante `c`. Es werden hier nur einige Möglichkeiten gezeigt.

Java
`null` *oder*
`0` *oder*
`"ein string"`

Bytecode
`aconst_null` *oder*
`iconst_0` *oder*
`ldc`

Ziel
keins

Signatur
`Tc ()`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Joinpointcode
`invokespecial wrapper`

Wrapper
Parameter laden
`*return`

Pattern
`"(*const|ldc*)"`

Numerische Typumwandlung

Beschreibung
Eine Konversion zwischen zwei numerischen Typen (`x2y`).

Java
`i = (int) d` *oder implizit*

Bytecode
`i2{b,s,l,f,d}` *oder*
`l2{i,f,d}` *oder*
`f2{i,l,d}` *oder*
`d2{i,l,f}`

Ziel
keins

Signatur
`Ty (Tx)`

Spur
Wie Bytecode

Exception
keine

Neuer Bytecode
`aload_0`
Parameterladecode `invokespecial wrapper`

Wrapper
Parameter laden
Joinpointcode
`*return`

Pattern
`"conversioninstruction"`

A.9 Bedingungen und Vergleich

Bedingung, Vergleichs- oder ternärer Operator

Beschreibung

Es gibt zwar separate Vergleichsbefehle im *JVM*-Befehlssatz. Das Ergebnis wird jedoch nicht auf den Stack gelegt, sondern durch den Zustand der internen Register repäsentiert. Die Auswertung eines Vergleichs erfolgt durch einen bedingten Sprungbefehl. Für `int` und Referenztypen kennt die *JVM* kombinierte Vergleichs- und Sprungbefehle (`if_{i,a}cmp`*OP*). Vergleich und Sprung erfolgt in einem Befehl.

Java

`c == d` *oder*
`(c == d) ? s : t`

Bytecode

`if`*

Der resultierende Bytecode unterscheidet sich teilweise erheblich. Allen gemein ist nur ein Aufruf; einer der vielen if-Befehle.

Ziel

keins

Signatur

`void (Tc, Td)` (*Bei if*)
`boolean (Tc, Td)` (*Bei Zuweisung*)
`Tc (Tc, Td)` (*Bei ternärem Operator*)

Spur

Wie Bytecode

Exception

keine

Neuer Bytecode

`aload_0`
Parameterladecode
`invokespecial wrapper`

Wrapper

Parameter laden
Joinpointcode
`*return`

Pattern

`ifinstruction`

Anhang B

Erweiterung des Prototypen

Dieses Kapitel soll kurz beschreiben, wie der mit Rahmen dieser Diplomarbeit entstandene Prototyp erweitert werden kann, um ihn zu vervollständigen. Dabei interessiert vor allem die Erweiterung um neue Joinpoints, neue Muster und andere Webestrategien.

B.1 Neue Joinpoints hinzufügen

Um den Prototypen um neue Joinpoints zu erweitern, muss die in Abbildung 8.6 dargestellte Joinpointhierarchie um die neuen Joinpoints erweitert werden. Damit das Matching funktioniert, muss der neue Joinpoint ein `InstructionFinder`-Pattern definieren. Man kann die Patterns aus dem Joinpointkatalog dafür verwenden (siehe Anhang A). In der Klasse `OTJoinPoints` muss für den neuen Joinpoint eine eindeutige ID vergeben werden, damit beim Einlesen der `OTJoinPoints`-Attribute eine Zuordnung der ID im Muster zu einer Joinpointklasse möglich ist. Außerdem sind die Methoden `checkConstraint` der `JoinpointPattern`-Hierarchie um einen Fall für die neuen Joinpoints zu erweitern, damit der Joinpoint beim Matching gefunden werden kann. Um den neuen Joinpoint auch beim Weben durch einen Wrapper ersetzen zu können, muss außerdem die Klasse `WrapperCreation` erweitert werden.

B.2 Neue Muster hinzufügen

Eingelesen werden die Muster gänzlich in der Klasse `OTJoinPoints`. Sie repäsentiert die gleichnamigen Bytecode-Attribute. Wenn der Compiler also neue Ausprägungen des `OTJoinpoints`-Attributs erzeugt, dann ist diese Klasse zu ändern. Muster werden außerdem durch die Hierarchie `JoinpointPattern` repräsentiert (8.5) und durch die `OTJoinPoints`-Klasse erzeugt. Ein neues Muster macht es nötig, dass die Patternhierarchie um eine neue Klasse erweitert wird. Alle geerbten abstrakten Methoden müssen implementiert werden. Dazu zählt `checkConstraint`, die Funktion, die das eigentliche Matching durchführt. Hier werden alle notwendigen Überprüfungen durchgeführt.

B.3 Neue Webestrategien realisieren

Es ist außerdem möglich, den Prototypen um neue Webestrategien zu erweitern. Wenn eine neue Webestrategie eingeführt werden soll, muss einfach ein neuer Weber geschrieben werden. Dieser Weber muss lediglich das `Weaver`-Interface implementieren (Abbildung 11.3). Danach muss nur noch die `WeaverFactory` erweitert werden. Sie erzeugt nach der Auswertung des *Java-Properties* `ot.weaver` den definierten Weber oder einen Defaultweber. Man siehe dazu in den Code der Klasse `WeaverFactory`. Er ist selbst erklärend.

B.4 Anderes Matching

Es ist derzeit nicht vorgesehen, den Prototypen um andere Matchingstrategien zu erweitern. Allerdings ist dies nach dem gleichen Prinzip wie beim Weben möglich. Der Weber müsste ein Interface bekommen und es müsste eine `MatcherFactory` erstellt werden. Außerdem sind die `Joinpoint-` und die `JoinpoinPattern`-Hierarchie zu erweitern, denn hier wird zur Zeit das Matching realisiert.

Literaturverzeichnis

[1] Byte Code Engineering Library (BCEL) Homepage. `http://jakarta.apache.org/bcel/`.

[2] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C.* Hanser, 2nd edition, 1990.

[3] Bruno Harbulot and John R. Gurd. *A join point for loops in AspectJ.* In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 63–74, New York, NY, USA, 2006. ACM Press.

[4] Christine Hundt. *Bytecode-Transformation zur Laufzeitunterstützung von aspektorienterter Modularisierung mit Object-Teams/Java*, 2003. Diplomarbeit, Technische Universität Berlin.

[5] Christoph Bockisch and Michael Haupt and Mira Mezini and Ralf Mitschke. *Envelope-Based Weaving for Faster Aspect Compilers.* In *NODe/GSEM*, pages 3–18, 2005.

[6] D. L. Parnas. *On the Criteria To Be Used in Decomposing Systems into Modules. Comm. ACM*, 15(12):1053–1058, December 1972.

[7] Frederick P. Brooks, Jr. *No silver bullet: essence and accidents of software engineering. Computer*, 20(4):10–19, 1987.

[8] Gregor Kiczales and Erik Hilsdale and Jim Hugunin and Mik Kersten and Jeffrey Palm and William G. Griswold. *An Overview of AspectJ.* In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[9] Gregor Kiczales and John Lamping and Anurag Menhdhekar and Chris Maeda and Cristina Lopes and Jean-Marc Loingtier and John Irwin.

Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[10] Guido Krüger. *Handbuch der Java-Programmierung*. Addision-Wesley, 4th edition, 2006.

[11] Hridesh Rajan and Kevin Sullivan. *Aspect language features for concern coverage profiling*. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 181–191, New York, NY, USA, 2005. ACM Press.

[12] Laurie Hendren, Oege De Moor, Aske Simon Christensen et al. *The abc scanner and parser, including an LALR(1) grammar for AspectJ*, 2004. `http://abc.comlab.ox.ac.uk/documents/scanparse/index.html`.

[13] Martin Fowler. *Refactoring - Wie Sie das Design vorhandener Software verbessern*. Addision-Wesley, 1st edition, 2000.

[14] Michael Flueh. *Schemaerhaltende Bytecodetransformation zum Aspektweben zur Programmlaufzeit*, 2006. Diplomarbeit, Technische Universität Berlin.

[15] Pavel Avgustinov and Aske Simon Christensen and Laurie Hendren and Sascha Kuzins and Jennifer Lhoták and Ondřej Lhoták and Oege de Moor and Damien Sereni and Ganesh Sittampalam and Julian Tibble. *abc: an extensible AspectJ compiler*. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.

[16] Pavel Avgustinov and Aske Simon Christensen and Laurie Hendren and Sascha Kuzins and Jennifer Lhoták and Ondřej Lhoták and Oege de Moor and Damien Sereni and Ganesh Sittampalam and Julian Tibble. *Optimising aspectJ*. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 117–128, New York, NY, USA, 2005. ACM Press.

[17] Peri Tarr and Harold Ossher and William Harrison and Stanley M. Sutton, Jr. *N degrees of separation: multi-dimensional separation of concerns*. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 107–119, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.

[18] Stephan Herrmann. *Object Teams: Improving Modularity for Crosscutting Collaborations*. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264, London, UK, 2003. Springer-Verlag.

[19] Stephan Herrmann. *Are Pointcuts a First-Class Language Feature?*, 2005. FOAL'06 Workshop (Foundation of Aspect-Oriented Languages), at AOSD'06, Bonn, Germany.

[20] Stephan Herrmann, Christine Hundt. *ObjectTeams/Java Language Definition*, 2006.
`http://objectteams.org/def/0.9/index.html`.

[21] Sun Microsystems, Inc. *The Java(TM) Virtual Machine Specification 2nd Ed.*, 1999.
`http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`.

[22] Sun Microsystems, Inc. *JavaTM 2 Platform Standard Edition 5.0 API Specification*, 2004.
`http://java.sun.com/j2se/1.5.0/docs/api/index.html`.

[23] Xerox Corporation. *The AspectJ Programming Guide*, 2003.
`http://www.eclipse.org/aspectj/doc/released/progguide/index.html`.