BYTECODE-TRANSFORMATION ZUR LAUFZEITUNTERSTÜTZUNG VON ASPEKT-ORIENTIERTER MODULARISIERUNG MIT OBJECT-TEAMS/JAVA

Diplomarbeit
bei Prof. S. Jähnichen

vorgelegt von
Christine Hundt
Matr. Nr. 172349
am Fachbereich Informatik der
Technischen Universität Berlin

Berlin, 2. Februar 2003

Inhaltsverzeichnis

Einleitung

Ein Ziel der Softwaretechnik ist es, eine hohe Programmqualität bei möglichst geringem Aufwand an Entwickungs-Ressourcen zu erreichen. Verständlichkeit, Wartbarkeit, Wiederverwendbarkeit und Zuverlässigkeit sind Software-Eigenschaften, die das Erreichen dieses Ziels in entscheidendem Maße beeinflussen. Eine intelligente Modularisierung des Programmcodes ist Vorraussetzung dafür. Code für eine bestimmte Aufgabe soll nach Möglichkeit nur einmal geschrieben werden und überall dort, wo er benötigt wird, benutzbar sein.

Während der Evolution der Softwareentwicklung wurden immer bessere Modularisierungs-Konzepte entwickelt. Das objektorientierte Programmiermodell bietet bereits gute Möglichkeiten zur Strukturierung und Modularisierung von Software.

Aspektorientierte Softwareentwicklung ist eine Weiterentwicklung der objektorientierten Softwareentwicklung, mit Hilfe derer eine weitere Dimension der Modularisierbarkeit eingeführt wird. *Object Teams* ist ein Programmiermodell, das Modularisierung entlang dieser Dimension ermöglicht und darüber hinaus ein kollaborationsbasiertes Rollenmodell bietet.

ObjectTeams/Java (kurz: ot-java) ist eine auf Java basierende Programmiersprache, die es ermöglicht Implementierungen nach den Konzepten von Object Teams zu erstellen. Die Umsetzung der Sprache wurde in eine Erweiterung des Standard Java Compilers und die Entwicklung eines Laufzeit-Backends unterteilt. Die vorliegende Arbeit beschäftigt sich mit der Entwicklung der Laufzeitumgebung für ot-java.

Eine Einführung in die Konzepte von Object Teams wird in Kapitel 1 gegeben.

Kapitel 2 beschäftigt sich mit den verwendeten Techniken und gibt einen Überblick über Loadtime Adaption und Bytecode Transformation.

Kapitel 3 beschreibt die Entwicklung der Laufzeitumgebung für ot-java, welche für die Realisierung einiger der in Kapitel 1 dargestellten Konzepte verantwortlich ist.

In Kapitel 4 wird auf die Rahmenbedingungen der Softwareentwicklung mit ot-java eingegangen. Hier wird außerdem der aktuelle Entwicklungsstand von ot-java dokumentiert.

Kapitel 1

Aspektorientierte Programmierung mit *Object Teams*

1.1 Aspekt-Orientierung

Die meisten modernen Softwaresysteme sind groß und komplex. Um diese Komplexität beherrschbar zu machen, muss das Gesamtsystem in kleinere Teilstücke (*Module*) unterteilt werden. Auf diese Weise ist es möglich, die anfallenden Arbeiten auf mehrere Personen zu verteilen. Eine effektive *Teamarbeit* ist jedoch nur möglich, wenn es klare und sinnvolle Abgrenzungen und wohldefinierte Schnittstellen zwischen den einzelnen Teilen gibt. Es muss ausserdem möglich sein, die einzelnen Module möglichst unabhängig voneinander zu entwickeln und zu testen, damit die Qualität des Gesamtsystems gewährleistet werden kann. Tritt ein Fehler auf, so soll es möglichst nicht nötig sein in unübersehbar vielen weiteren Modulen nach den Ursachen zu forschen. Wenn Dinge, die zusammengehören auch an einem Ort versammelt sind, also eine Form der *Lokalität* gegeben ist, herrschen beste Voraussetzungen hierfür. Ein hoher Modularisierungsgrad erhöht neben der Wartbarkeit auch die Wiederverwendbarkeit einmal entwickelter Teilstücke.

Concerns in der Softwareentwicklung sind ”Dinge von Interesse”, die in einem Software-System modelliert werden sollen. Auch wenn diese Definition einiges an Interpretationsspielraum zuläßt, scheint es grundsätzlich wünschenswert, einzelne Concerns zu identifizieren und zu kapseln (*separation of concerns*).

Objektorientierte Programmierung ermöglicht die Kapselung und Vererbung bestimmter Concerns (Objekteigenschaften) in Klassen. Man kann dabei von einer *Modularisierung nach der Struktur* sprechen. Viele auftretende Concerns können so erfasst und gekapselt werden.

Es gibt jedoch System-Eigenschaften oder Funktionalitäten, die über sehr viele Klassen verteilt realisiert werden müssen (Beispiel: Logging Mechanismen oder Persistenz). Man spricht bei solchen, *quer* zur (Klassen-) Struktur des Systems liegenden Einheiten von *crosscutting Concerns* oder *Aspekten*. Sie stellen verschiedene Sichten auf das System dar, die die vorhandenen Modul-Grenzen durchschneiden und somit die Kapselung zerstören.

Gibt es bei beliebiger Klassenstruktur immer crosscutting Concerns, so reichen die vorhandenen objektorientierten Techniken scheinbar nicht aus, um eine optimale Modularisierung zu erreichen.

Die *Aspektorientierte Programmierung* kapselt einzelne Codefragmente, die crosscutting Concerns oder Aspekte darstellen, ermöglicht also eine *Modularisierung nach der Funktion*, oder allgemeiner gesagt nach verschiedenen Gesichtspunkten des Systems. Aspektorientierte Sprachen stellen explizite Konstrukte zur Modularisierung solcher Aspekte, sowie zur Integration zu einem Gesamtsystem, zur Verfügung.

1.1.1 scattering und tangling

Scattering (zerstreuen) meint die Tatsache, dass mit klassischem objektorientiertem Design oftmals Dinge, die dazu dienen dem Gesamtsystem eine bestimmte Eigenschaft oder Funktionalität hinzuzufügen, auf verschiedene Module (Klassen) ”verstreut” werden müssen. Der Code für die Funktionalität ”logge den Aufruf aller Methoden” beispielsweise, müsste in allen Methoden auftauchen. Betrachtet man den Sourcecode, so ist nicht ohne weiteres erkennbar, dass es sich bei dem Logging-Mechanismus um eine Funktions-Einheit (einen Concern) handelt.

Tangling (verflechten) beschreibt den Zustand, dass an einem Ort (in einer Klasse, einer Methode) Code für die Realisierung unterschiedlichster Funktionalitäten ”vermischt” wird.

Eine Vermeidung von scattering und tangling sorgt dafür, dass eine bestimmte Funktionalität an einem Ort definiert werden kann. Aspektorientierung ermöglicht eine Trennung der Implementierung verschiedener crosscutting concerns in verschiedene Module und vermeidet so scattering und tangling.

1.1.2 Statisches und dynamisches Weben

Modular entwickelte Aspekte müssen zur Programmausführung mit der bestehenden Klassenstruktur zu einem lauffähigen Programm zusammengeführt werden. Falls wie oben erwähnt, mehrere Methoden mit einem Logging-Mechanismus ausgestattet werden sollen, so müssen die dafür nötigen Instruktionen[1] letztendlich

[1] Dabei kann es sich natürlich auch um den Aufruf einer Methode, die für das Logging zuständig ist, handeln.

innerhalb der betreffenden Methoden erscheinen. Da dies natürlich vom Anwender der aspektorientierten Programmiersprache fernzuhalten ist, muss es dafür eine Automatisierung geben. Den Vorgang des (automatischen) Einfügens von Aspekten in den Applikations-Code bezeichnet man als *weben* (weaving).

Werden Aspekte zur Compilezeit in die Applikationsklassen hinein gewoben, so spricht man von *statischem* Weben. Eine andere Möglichkeit besteht darin, das Weben erst zur Laufzeit *dynamisch* durchzuführen. Mit dieser Technik würden also auch Klassen, die schon geladen wurden, nachträglich verändert werden. Ein solch spätes Weben hat den Vorteil, dass Informationen, die erst zur Laufzeit bekannt sind, mit einbezogen werden können. Davon unabhängig zu betrachten ist die Frage der dynamischen Aktivierbarkeit von Aspekten. Eine Realisierungs-Möglichkeit dafür ist das Einbauen von dynamischem Dispatcher-Code, der zur Laufzeit entscheidet welcher Aspekt wo wirksam wird.

1.1.3 Join Points

Join Points sind Stellen im Kontrollfluss eines Programms, an denen mit Hilfe einer Aspektsprache Adaptionen vorgenommen werden. Aspektorientierte Programmiersprachen stellen Mittel zur Verfügung, um diese Punkte festzulegen. Beim Zusammen*weben* des resultierenden Programms sind es genau diese Stellen, an denen der Aspektcode mit dem Code der Basis-Applikation *verbunden* wird.

1.2 Das Programmiermodell *Object Teams*

Das in [8] vorgestellte Programmiermodell *Object Teams* führt Sprachkonzepte ein, die eine Modularisierung von crosscutting Concerns (*separation of concerns*) in Form von Objekt-Kollaborationen, sowie deren unvorhergesehene Anbindung an existierende Anwendungen ermöglichen. Die wichtigesten Konzepte dieses Modells werden im folgenden vorgestellt. Dabei wird hauptsächlich auf diejenigen näher eingegangen, die für die vorliegende Arbeit von Bedeutung sind.

1.2.1 Team

Das Kernkonzept von Object Teams ist das *Team*, welches eine *Kollaboration* mehrerer Klassen modularisiert. Eine Kollaboration ist ein Zusammenschluss von Klassen, die miteinander agieren und einem gemeinsamen Zweck dienen. Die Klassen innerhalb eines Teams bezeichnet man als *Rollen*. Sie werden in Form von Inner-Classes notiert.

Teams können an Packages einer (bestehenden) Anwendung ”gebunden” werden, wodurch die im Team definierten Erweiterungen und Änderungen das (*Basis*-) Programm adaptieren. Ein Team wird mit dem Modifier `team` deklariert.

```
team class ATeam {
    class Role {...}
}
```

Team-Level Features

Ein Team hat die Aufgabe seine Rollenklassen zu verwalten und zu koordinieren. Es gewährleistet ausserdem eine Art Kapselung der Rollen von der ”Außenwelt”. *Team-Level Features* sind Felder und Methoden eines Teams, die der Interaktion mit den Rollen dienen. Zum einen fungieren sie dabei als Schnittstelle zwischen Aufrufern von aussen und den Rollenklassen (vergleiche Design-Pattern *Facade*). Andererseits spielen Team-Level Features auch die Rolle einer Vermittlungsinstanz zwischen den einzelnen Rollen, die sich unter Umständen nicht gegenseitig ”kennen”. Sie übernehmen dadurch die Funktion eines *Mediator*s, der eine weitgehende Unabhängigkeit der Rollenklassen untereinander sicherstellt (*loose coupling*).

implizite Rollen-Vererbung

Erbt ein Team von einem anderen, so werden dessen Rollen ebenfalls (implizit) mit vererbt. Im abgeleiteten Team können auch die Rollenklassen verfeinert werden. Dazu ist kein expliziter Modifier nötig, sondern die Rollen können einfach überschrieben werden. Ein Zugriff auf die Rollen-*Oberklasse* ist mit dem `tsuper()`-Konstrukt (analog zum Java *super*-Konstrukt) möglich.

Rollen sind in Form von Java InnerClasses realisiert. Während für Teams, die in einer Vererbungshierarchie stehen, Polymorphie gilt, sind die dazugehörigen Rollen deshalb nicht polymorph substituierbar. Aus Gründen der Typ-Sicherheit wird deshalb ein Austauschen von Rollen zwischen verschiedenen Teams verboten.

externalized roles

Rollen sind normalerweise an das sie enthaltende Team gebunden und dürfen nur in ihm verwendet werden. Unter bestimmten Voraussetzungen können Rollen jedoch auch ausserhalb des Teams verwendet werden. Dann muss jedoch sichergestellt werden, dass Rollen nicht in Teams gelangen, zu denen sie nicht gehören.

1.2.2 Rolle und Basis

Um ein Team mit einer Anwendung zu integrieren und ihr so zusätzliche Funktionalität hinzuzufügen, werden seine *Rollen* an Basisklassen *gebunden*. *Basis*klassen sind gewöhnliche Anwendungsklassen, deren Verhalten von Rollenklassen verändert, oder erweitert wird. Zum Binden einer Rollenklasse an eine Basisklasse wird das Schlüsselwort `playedBy` verwendet.

class Role **playedBy** Base {...}

Basisklassen wissen im Normalfall nichts von der Existenz der von ihnen gespielten Rollen. Wird Rollenfunktionalität nachträglich zu einer bereits fertig implementierten Basis-Anwendung hinzugefügt, so ist dies der einzig gangbare Weg. Aber auch wenn ein System von vornherein mit den ot-Konzepten entworfen und umgesetzt wird, sind Basisklassen relativ unabhängig von den Rollen implementiert. So haben Basisklassen explizit keine Referenzen auf Rollenklassen, d.h. sie kennen diese nicht. Basisklassen können immer auch unabhängig von Rollen benutzt, insbesondere instanziiert werden. Rollenklassen hingegen sind immer abhänging von Basisklassen, denn sie weisen im Normalfall unvollständig implementierte Methoden auf, die erst durch Bindung an Methoden der Basisklassen ausführbar werden.

Rollenklassen haben zwei unterschiedliche Möglichkeiten das Verhalten einer Basis-Anwendung zu adaptieren. In jedem Fall müssen dazu Rollenmethoden an bestimmte Basismethoden *gebunden* werden. Die Basismethoden müssen dabei zu der an die Rolle gebundenen Basis gehören. Je nach der beabsichtigten Art der Adaption werden Rollenmethoden dabei entweder per *callout* oder per *callin* an Basismethoden gebunden.

callout-Bindung - Delegation an die Basis

Durch callout-Bindung kann das Verhalten einer (Basis-) Klasse bzw. ihrer Methoden erweitert werden. Rollenmethoden, die selbst keine Implementierung besitzen (praktisch nur leere Hülsen sind), können per callout an eine Basismethode gebunden werden. Ein Aufruf der Rollenmethode bewirkt dann einen Aufruf der Basismethode. Es handelt sich dabei also um eine Form von *Delegation* des Rollenmethoden-Aufrufs an eine Methode der Basis-Applikation.

Rollenmethoden können eine bestimmte Funktionalität ausführen, dabei jedoch eine (noch) nicht implementierte Methode aufrufen. Diese kann nun entweder in einer Subklasse implementiert werden, oder aber durch callout-Bindung mit Inhalt gefüllt werden. Im Prinzip entspricht dieses Szenario dem *Template&Hook*-Designpattern: Die abstrakte Rollenmethode entspricht dem *Hook* und die, mit callout daran gebundene, Basismethode fungiert als *Template*. Eine callout-Bindung

einer Rollenmethode `roleMethod1` an eine Basismethode `baseMethod1` würde in der Rollenklasse folgendermaßen deklariert werden:

```
class Role playedBy Base {
    ...
    roleMethod1 -> baseMethod1
    ...
}
```

callin-Bindung - Veränderung von Basismethoden

Bei einem callin ist die Aufrufrichtung sozusagen umgekehrt zu der beim callout. Die Basis ruft die Rollenmethode auf. Dies widerspricht nur scheinbar dem, was über die Unabhängigkeit der Basisklassen gesagt wurde. Im Quellcode der Basis wird dies nämlich mit keiner Zeile Code erwähnt. Alles hierfür notwendige wird im nachhinein (bei ot-Java zur Loadtime; siehe Kapitel 3) hinzugeneriert.

Ein callin hat das Hineinweben von Rollenmethoden-Code in eine Basismethode zur Folge und bewirkt damit eine Veränderung im Verhalten der Basisklasse. Wird eine so gebundene Basismethode von einem beliebigen Client aufgerufen, so wird automatisch die dazugehörige Rollenmethode aufgerufen. Eine Rollenmethode, die per callin-Binding[2] an eine Basismethode gebunden wird, wird vor (*before*), nach (*after*), oder anstatt (*replace*) dieser ausgeführt.

```
class Role playedBy Base {
    ...
    roleMethod2 <- replace baseMethod2
    // das Keyword replace kann weggelassen werden
    roleMethod3 <- after baseMethod3
    roleMethod4 <- before baseMethod4
    ...
}
```

Eine Rollenmethode, die per *replace* eine Basismethode praktisch *überschreibt*, wird mit dem Modifier `callin` gekennzeichnet. Innerhalb solcher replacement-*callin*-Methoden besteht die Möglichkeit, die ursprüngliche (Basis-) Methode aufzurufen. Diese Funktionalität wird mit dem sogenannten *base-call* realisiert. Der base-call entspricht in Namen und Signatur der Rollenmethode unabhängig davon, ob die Basismethoden-Signatur eventuell davon abweicht (siehe Abschnitt 1.2.3 Signatur Differenzen) und wird an der Basisreferenz `base` getätigt. Die Technische Realisierung des tatsächlichen Aufrufs der Basismethode wird in Kapitel 3 (S.49) erläutert.

[2] Im folgenden wird der Ausdruck ”Binding” verwendet, um die Deklaration einer Bindung zu bezeichnen.

```
class Role playedBy Base {
    ...
    roleMethod2 <− replace baseMethod2;
    callin void roleMethod2() {
        ...
        base.roleMethod2();      // base−call
        ...
    }
}
```

In Abbildung 1.1 sind die oben aufgeführten Syntaxbeispiele für die einzelnen Object Teams Konstrukte zusammengefasst in Form eines UFA-Diagramms dargestellt. *UFA* (UML for Aspects) ist eine Erweiterung der UML, zur Modellierung von Object Teams Anwendungen, die in [7] vorgestellt wird.

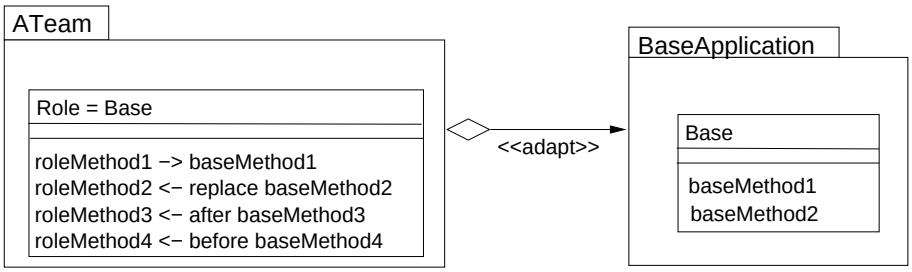

Abbildung 1.1: UFA - Diagramm für die Syntax-Beispiele

Konnektoren

In den Vorläufermodellen von Object Teams wurde zwischen *Kollaborationen* (jetzt Teams) und *Konnektoren* unterschieden. Konnektoren wurden dafür benutzt Bindings zu deklarieren. Konzeptionell ist es in der Tat sinnvoll die Implementierung eines Teams unabhängig von der konkreten Bindung an bestimmte Basisklassen zu halten. Um jedoch vorab eine Teilintegration zu ermöglichen, wurde diese strikte syntaktische Trennung aufgegeben.

Konzeptionell kann man jedoch sagen, dass ein Team als Konnektor fungiert, wenn es mindestens ein Binding enthält. Ein ”idealer” Konnektor enthält nur Bindings, und zwar alle.

Implizite Umwandlung zwischen Rolle und Basis

Betrachtet man eine (Basis-)Klasse aus der Sicht einer von ihr gespielten[3] Rolle, so wechselt man zwar den Kontext, hat aber eigentlich immernoch dasselbe Objekt

[3] also an sie gebundenen

vor sich. Trotzdem haben Rolle und Basis unterschiedliche Eigenschaften[4]. Um zu gewährleisten, dass jeweils die Felder und Methoden eines Objektes verfügbar sind, die der jeweilige Kontext erfordert, ermöglicht Object Teams eine *implizite Umwandlung* zwischen Rollen- und dazugehörigen Basisobjekten.

- Die Umwandlung eines Basisobjekts zu einem Rollenobjekt nennt man *lifting*.
- Wird zu einem Rollenobjekt das dazugehörende Basisobjekt gesucht, so findet es sich durch *lowering* der Rolle an.

Lifting und lowering sind Vorgänge, die nicht explizit vom Programmierer durchgeführt werden. Diese Umwandlungen werden zu gegebener Zeit vom Typsystem automatisch durchgeführt.

1.2.3 Signatur Differenzen

Stimmen die Signaturen aneinander gebundener Rollen- und Basismethoden überein, so können Argumente und Rückgabewerte bei callout und callin einfach an alle beteiligten Methoden durchgereicht werden. Gerade bei einer a-posteriori Integration kann jedoch nicht grundsätzlich von diesem Idealfall ausgegangen werden. Es muss möglich sein, auch Methoden mit unterschiedlichen Signaturen aneinander zu binden. Dafür müssen Signatur-Anpassungen durchgeführt werden.

Parameter-Mappings

Durch explizite *Parameter-Mappings* kann genau angegeben werden, welcher Parameter der Basismethode auf welchen Parameter der Rollenmethode abgebildet werden soll. Um bei der Deklaration des Parameter-Mappings auf die Signaturelemente zugreifen zu können, müssen beim Binden der Methoden die vollständigen Signaturen von Rollen- und Basismethode angegeben werden. Ohne Parameter-Mappings reicht dazu die Angabe der Methodennamen.

Beim callout sollen die Argumente für den Aufruf der Basismethode spezifiziert werden. Beim callin dagegen soll den Parametern der Rollenmethode etwas zugeordnet werden. Bei diesen Mappings stehen rollenspezifische Elemente jeweils auf der linken und basisspezifische Elemente auf der rechten Seite[5]. Verbunden werden diese Ausdrücke durch einen Pfeil, der in die Richtung des zu spezifizierenden Parameters zeigt. Im Fall eines callout Parameter-Mappings also nach rechts ($->$) und bei einem callin Parameter-Mapping nach links ($<-$).

[4] Durch callout-Bindings können sich diese teilweise überschneiden.

[5] Das Binden von Klassen und Methoden folgt übrigens dem gleichen Schema.

Desweiteren ist es möglich das Ergebnis der aufgerufenen Methode nochmals zu adaptieren. Durch Verwendung des Identifiers `result` kann man auf den Rückgabewert der aufgerufenen Methode zugreifen, sowie dem Rückgabewert der aufrufenden Methode etwas zuweisen. Ein `result` links vom Pfeil meint also das Resultat der Rollenmethode, während eines rechts davon den Rückgabewert der Basismethode referenziert. Da der Datenfluss hierbei andersherum, nämlich von der aufgerufenen zur aufrufenden Methode verläuft, ist der Pfeil umgedreht.

Zugewiesen werden können den Parametern und dem Resultat Ausdrücke, die die Parameter der jeweils anderen Methode, sowie sichtbare Felder und Methodenaufrufe enthalten können. Hier können also nicht bloß Parameter aufeinander abgebildet werden, sondern auch weitere explizite Konvertierungen vorgenommen werden.

Folgendes Beispiel für ein callout-Binding mit Parameter-Mapping verdeutlicht die Möglichkeiten:

```
int roleMethod(int i) -> Integer baseMethod(int i1, int i2) with {
    i -> i1;
    i * 2 -> i2;
    result <- result.intValue()
}
```

Hier wird die Basismethode mit `i` und `i * 2` aufgerufen. An dem von der Basismethode zurückgegebenen `Integer`-Objekt wird die `intValue()`-Methode aufgerufen und das Ergebnis dieses Ausdrucks wird dem Rückgabewert der Rollenmethode zugeordnet.

Implizite Signatur-Anpassungen

Für callin-Bindings werden bestimmte Signatur-Anpassungen automatisch durchgeführt. Für die Signaturen von mit callin aneinander gebundenen Rollen- und Basismethoden (ohne explizite Parameter-Mappings) gilt, dass alle Typen übereinstimmen müssen.
Ausnahmen hiervon sind:

- Die Rollenmethode darf weniger Argumente haben als die Basismethode.
- Es ist erlaubt, dass die Rollenmethode keinen Rückgabewert (`void`) hat, obwohl die Basismethode einen besitzt.
- Basistypen sind kompatibel zu ihren Rollentypen und werden automatisch *geliftet*.
- Typen der Rollensignatur dürfen allgemeiner sein als die der Basissignatur, falls sie in der Rollenmethode *final* deklariert sind; sie werden beim Aufruf der Basismethode (base-call) bei Bedarf gecastet.

- Ein von *void* verschiedener Rückgabewert der Rollenmethode wird (auch im replace-Fall!) von der Basismethode ignoriert, falls die Basismethode void zurückgibt.

Solange durch Parameter-Mappings nichts anderes festgelegt wurde, werden überschüssige Teile beim Rollenmethoden-Aufruf einfach weggeschnitten. Hat eine Rollenmethode weniger Argumente (*n*), als ihr Basis-Pendant, so werden ihr nur soviele übergeben, wie sie erwartet (nämlich die ersten n).

Einem Programmierer, der Object Teams benutzen will, reicht es die oben genannten Regeln zu kennen. Für die tatsächliche Realisierung dieser Anpassungen ist darüber hinaus einiges an technischem Aufwand zu betreiben. Schließlich muss gewährleistet sein, dass im Programmcode ausgeblendete, intern jedoch benötigte Daten an der richtigen Stelle zur Verfügung stehen. So muss u.a. dafür gesorgt werden, dass bei der Umsetzung des base-calls, der ja aus Programmierer-Sicht die Signatur der Rollenmethode hat, die Argumente zugänglich sind, welche die Basismethode erwartet.

Die technische Realisierung der Signaturanpassungen wird in 3.4.3 (S. 50) dargestellt.

1.2.4 Team Aktivierung

Callin-Bindings einer Rolle werden erst wirksam, wenn das umschließende Team *aktiviert* wird. Diese Aktivierung kann *explizit* durchgeführt werden, indem die dafür zuständige Methode `activate()` auf der Teaminstanz ausgeführt wird. Der Sinn dieser Aktivierung ist es, zu ermöglichen, dass die Team-Funktionalität gezielt an bestimmten Stellen (z.B. für bestimmte Objekte) ”eingeschaltet” werden kann. Entsprechend werden callins eines Teams mit `deactivate()` wieder ”ausgeschaltet”. Soll ein Team `t` nur für eine kurze Zeit aktiviert werden, so gibt es dafür die abkürzende Schreibweise:

```
in(t) do {
    // hier ist Team t aktiv
}
```

Teams werden *implizit* (automatisch) aktiviert, falls nötig. Werden Team-level-Methoden aufgerufen, so muss das Team aktiv sein, da diese in der Regel mit den ansonsten teilweise nicht funktionstüchtigen Rollenmethoden interagieren.

Statische Team Aktivierung

Soll ein Team im System generell wirksam sein, so können die dazugehörigen callin-Bindings statisch in die Basisklassen hineingewebt werden. Eine Aktivierung/Deaktivierung ist dann nicht mehr nötig. Statische Teams werden mit dem Modifier `static` deklariert.

static team STeam {...}

1.2.5 Grad der Realisierung

Methoden und Klassen können verschiedene Realisierungslevel aufweisen. *Abstrakte* Rollenmethoden zeigen fehlende Funktionalität (Implementation) an. Diese kann in Subteams, entweder durch ”normale” Implementierung, oder aber durch Delegation an eine vorhandene (Basis-) Methode (callout-Binding) erbracht werden.

Rollenmethoden mit dem Modifier *callin* benötigen Funktionalität einer weiteren Klasse (genauer einer speziellen Methode) um base-calls zu realisieren. Solche gebundenen Einheiten sind also nur funktionstüchtig, falls eine externe Einheit identifiziert wurde, die das fehlende Verhalten einbringt. Da callin-Methoden dafür konzipiert sind aus Basismethoden heraus aufgerufen zu werden, können sie nicht explizit aufgerufen werden[6].

Nur vollständig realisierte Teams sind zur Benutzung bereit und können instanziiert und aktiviert werden.

Die ausführliche Object Teams Sprachdefinition ist auf der Object Teams Homepage [3] zu finden.

[6] In 3.4.2 wird deutlich, dass dies wegen der unvollständigen Realisierung des base-calls in der ursprünglichen Rollenmethode technisch auch gar nicht funktionieren würde.

Kapitel 2

Bytecode Transformation zur Loadtime

Der Code für die Realisierung einiger der in Kapitel 1 vorgestellten ot-java-Konzepte wird erst zur Laufzeit, beim Laden der beteiligten Klassen in die Anwendung gewoben. Das folgende Kapitel beschäftigt sich mit den, für die Laufzeitumgebung von ot-java verwendeten, Techniken und Werkzeugen zur Durchführung von Code-Transformationen zur Laufzeit.

Java-Programme werden vom Java-Compiler in plattformunabhängigen Bytecode übersetzt und in *class files* gespeichert. Es handelt sich dabei nicht um Maschinencode, sondern um eine Art ”Zwischencode”, der erst bei Programmausführung in Maschinenbefehle für die spezielle Zielrechner-Architektur umgewandelt wird. Dies übernimmt die *Java Virtual Machine* (JVM), die sozusagen als virtuelle CPU fungiert und den Bytecode interpretiert.

Sollen Modifikationen an bereits kompilierten Java Klassen vorgenommen werden, zum Beispiel weil deren Sourcecode nicht vorliegt, so besteht die Möglichkeit den Bytecode nachträglich zu verändern. Zur *Bytecode Transformation* existieren diverse Ansätze und Klassenbibliotheken. Für diese Arbeit wurde die *BCEL* (Byte Code Engineering Library) API [5] zur Erzeugung und Manipulation von Java Bytecode verwendet.

Wird der Bytecode beim Laden einer Klasse transformiert, so spricht man von *Loadtime Adaption*. Während der Ausführung eines Java-Programms werden die kompilierten Java-Klassen in die JVM geladen. Vor der eigentlichen Ausführung können diese ”abgefangen”, transformiert und in einer modifizierten Form an die *Execution Engine* übergeben werden. Das *JMangler* Framework greift auf diese Weise in den Ladeprozess der JVM ein und ermöglicht so das Transformieren der geladenen Klassen vor ihrer eigentlichen Ausführung.

Abbildung 2.1[1] zeigt das Zusammenspiel der genannten Techniken zur Realisierung der Programmiersprache Object-Teams/Java. Genauere Erklärungen folgen im Laufe diese Kapitels.

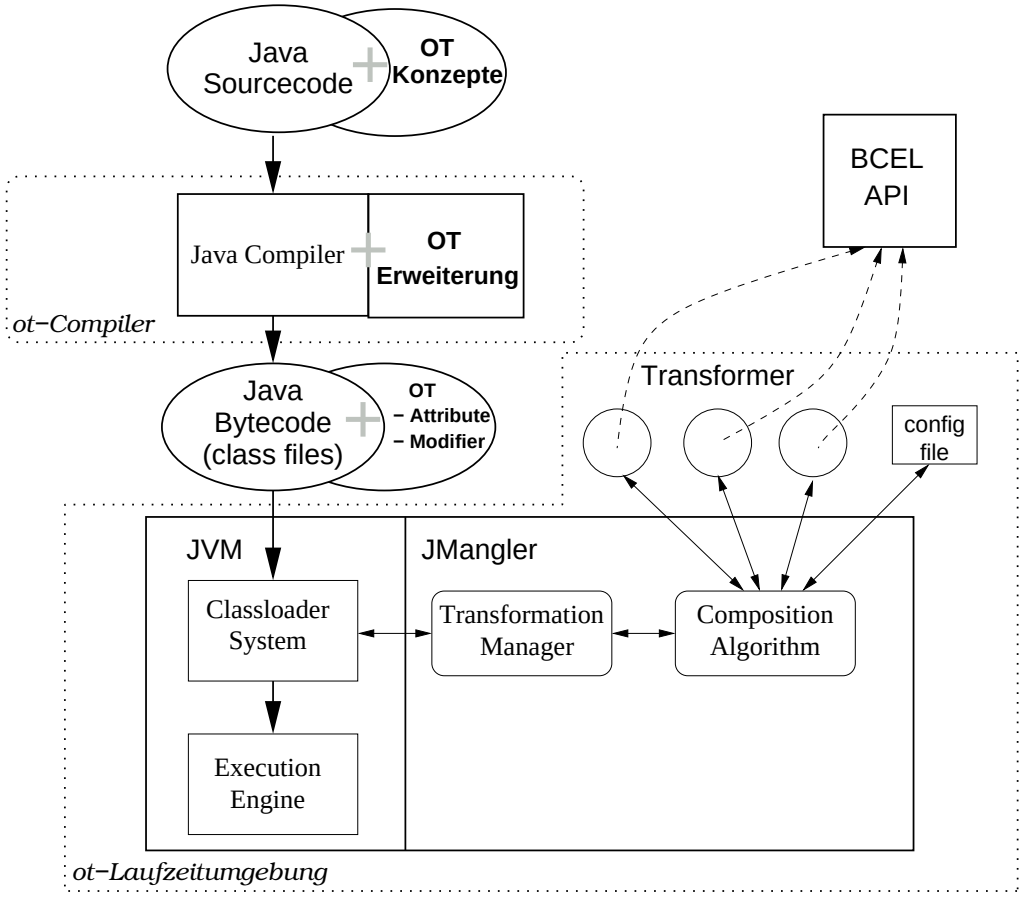

Abbildung 2.1: Realisierung von ot-java

2.1 Bytecode Transformation

Bytecode Transformation umfasst das Hinzufügen von Feldern und Methoden genauso wie das Verändern von Sichtbarkeits-Deklarationen oder der Implementierung von Methoden.

Das Java *class file format* gibt dem Bytecode eine Struktur, in welcher alle relevanten Informationen aus dem ursprünglichen Code erhalten bleiben (*information rich binaries*). Dies ermöglicht nicht nur das Dekompilieren übersetzter

[1]Die Abbildung wurde in Anlehnung an die Standard-Abbildung zur Darstellung der JMangler-Architektur erstellt.

Java-Klassen zurück in Sourcecode-Form, sondern auch nachträgliches (sinnvolles) Verändern des Bytecodes. Die *BCEL* API ermöglicht eine abstraktere Sicht auf den Bytecode und erleichtert so Erzeugung und Manipulation von class file format konformem Bytecode.

2.1.1 Das Java class file format

Ein class file ist das Resultat der Übersetzung einer Java Klasse und hat ein genau festgelegtes Format, über welches Abbildung 2.2 einen Überblick gibt. Nachfolgend werden die Einträge eines class files näher erläutert.

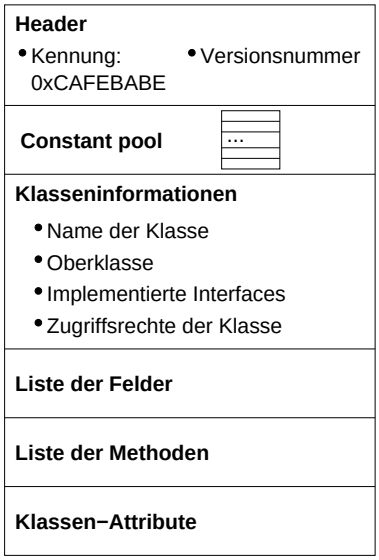

Abbildung 2.2: Das Java class file Format

Der **Header** enthält die *Kennung* 0xCAFEBABE, die dazu dient die Datei als ein Java class file zu identifizieren. Die *Versionsnummer* des Java Compilers mit dem das class file erzeugt wurde, wird benötigt um eventuelle Inkompatibilitäten von Klassen zu erkennen.

Der **Constant pool** einer Klasse kann als eine Art *Repository* für die konstanten Informationen einer Klasse angesehen werden. Es handelt sich um eine indizierte Tabelle, in welcher identische Einträge nur einmal gespeichert werden.
Je nach Art der Information unterscheidet man zwischen verschiedenen Typen von Constant pool Einträgen. Einige dienen zur Speicherung von String-Konstanten und Konstanten der numerischen Grundtypen. Andere stellen symbolische Referenzen auf Klassen, Felder oder Methoden dar. Alle Informationen, die benötigt werden um diese Referenzen zur Laufzeit dynamisch aufzulösen, sind als Strings kodiert.

Ein Eintrag vom Typ *CONSTANT_Methodref* enthält beispielsweise Verweise (Indizes) auf weitere Constant pool Einträge, welche die Klasse des methodenempfangenden Objekts, sowie den Methodennamen und die Signatur der Methode enthalten. Auch die Typen von Feldern sind im Constant pool abgelegt. Diese Typinformationen werden für Feldzugriffe und Methodenaufrufe benötigt und sind in Form von *Typdeskriptoren* abgelegt.
Der Constant pool macht durchschnittlich 60% der Größe einer Klassendatei aus.

Anschließend folgen Klasseninformationen, wie der **Name der Klasse**, die **Oberklasse** und die **implementierten Interfaces**. Die **Zugriffsrechte** einer Klasse sind in Form einer Bitmaske abgelegt. Um zum Beispiel zu prüfen, ob eine Klasse *public* deklariert ist, wird der für diesen Modifier stehende Wert mit dieser Bitmaske ver*und*et.

In der **Liste der Felder** befinden sich für jedes Feld der Klasse die *Zugriffsrechte* sowie *Name* und *Typ* des Feldes in Form von Indizes, die Verweise auf Einträge des Constant pools darstellen. Außerdem können hier, in den sogenannten *Attributen*, weitere Informationen abgelegt sein.

Die **Liste der Methoden** enthält für jede Methode der Klasse die *Zugriffsrechte*, die Constant pool Indizes der Einträge für *Methodenname* und *Signatur*, sowie zur Methode gehörende *Attribute*. Im *Code*-Attribut einer nicht-abstrakten Methode befindet sich die, zur Methode gehörige Sequenz von Bytecode-Instruktionen, welche im folgenden Abschnitt näher erläutert werden.

Zuletzt folgen die **Klassen-Attribute**. Das optionale *SourceFile*-Attribut beispielsweise zeigt an, in welcher Datei sich der Java Sourcecode der Klasse befindet. Auf die Bedeutung von Attributen wird im Anschluss näher eingegangen.

Attribute

Attribute stellen eine Möglichkeit dar, beliebige zusätzliche Informationen in class files zu speichern. Klassen, Felder und Methoden können auf diese Weise instrumentiert werden. Manche dieser Attribute sind obligatorisch, wie etwa das *Code*-Attribut bei nicht-*abstrakten* Methoden, welches die zur Methode gehörende Implementierung enthält.

Laut [10, §4] sollen JVM-Implementationen ihnen unbekannte Attribute in class files einfach ignorieren. Das kann dann sinnvoll sein, wenn die in den Attributen gespeicherten Informationen noch vor der Programmausführung von einer anderen Instanz als der JVM benutzt werden. Im Fall von ot-java geschieht dies, um den Bytecode nachträglich zu modifizieren (siehe 3.1.2). Auch die, im Rahmen des *Darwin Project*s entwickelte Programmiersprache Lava ([2]) benutzt diese Technik.

Typdeskriptoren

Bei den Typdeskriptoren handelt es sich um Strings, die in einem kompakten Format die Java Typdeklarationen widerspiegeln.

- *Grundtypen* werden dabei mit Großbuchstaben abgekürzt. So steht zum Beispiel ein `I` für den Typ `int`.
- *Referenztypen* werden durch vollständige Klassennamen dargestellt. Sie werden durch `L` und `;` eingeschlossen und enthalten ein `/` als Trennzeichen. Der Typ `org.objectteams.Team` würde also als `Lorg/objectteams/Team;` kodiert werden.
- Für *Arraytypen* wird die Anzahl der Dimensionen und der Elementtyp angegeben. Eine Dimension ist dabei durch ein `[` repräsentiert. Der Typ `int[][]` wäre in der Klassendatei als `[[I` gespeichert.

Methoden-Signaturen werden als kompakte Textdarstellung der Typen der Parameter und des Ergebnisses festgehalten. Sie haben die Form
(Parametertyp)Ergebnistyp*.
Für die Signatur der Methode `Team getTeam(int[], String)` würde also `([ILjava/lang/String;)Lorg/objectteams/Team;` stehen.

Die vollständige Aufstellung der Typdeskriptoren ist in [10, §4.3] nachzulesen.

JVM Instruktionen

Die Bytecode-Sequenz einer Methode besteht aus Instruktionen, die von der JVM ausgeführt werden können. Ein Bytecode-Befehl besteht aus einer eindeutigen Befehlsnummer, dem *Opcode*, sowie optionalen *Operanden*.

Die meisten JVM Instruktionen sind typspezifisch, wobei die Typinformation im Opcode kodiert ist und im Namen des Befehls als Präfix erscheint. Hierbei steht *i* für `int`, *l* für `long`, *s* für `short`, *b* für `byte`, *c* für `char`, *f* für `float`, *d* für `double` und *a* für *Referenztypen*. Der Befehlssatz der JVM lässt sich in folgende Gruppen von Instruktionen gliedern:

- **Lade- und Speicher-Instruktionen:**
  - Laden einer lokalen Variable vom Typ T$\in \{i, l, f, d, a\}$ auf den Stack:
  T*load*, T*load*<*n*>
  - Speichern eines Wertes des Typs T$\in \{i, l, f, d, a\}$ vom Stack in lokale Variablen:
  T*store*, T*store*<*n*>
  - Laden von Konstanten auf den Stack:

z.B. *bipush , sipush, ldc, iconst_*$< i >$
Lade- und Speicher-Instruktionen haben explizite Operanden: Constant pool Indizes.

- **Arithmetische Instruktionen:**
  - Addition T*add*, Subtraktion T*sub*, Multiplikation T*mul*, Division T*div*, Rest-Operationen T*rem*, Negation T*neg*; für T$\in \{i, l, f, d\}$
  - Arithmetischer Shift T*shl*, T*shr*, Logischer Shift T*ushr*, Bitweises Oder T*or*, Bitweises Und T*and*, Bitweises exklusives Oder T*xor*; für T$\in \{i, l\}$
  - Inkrementierung lokaler Variablen *iinc*
  - Vergleichsoperatoren

  Arithmetische Instruktionen greifen auf ihre Operanden über den Stack zu.

- **Typ-Konvertierungs Befehle:**
  - Erweiternde numerische Konvertierungen
  - Einschränkende numerische Konvertierungen

- **Objekt Erzeugung und Manipulation:**
  - Erzeugung von Objekten: *new*
  - Erzeugung von Arrays: *newarray*, *anewarray*, *multianewarray*
  - Zugriff auf statische Felder (Klassenvariablen): *getstatic*, *putstatic*
  - Zugriff auf Instanz-Felder (Instanzvariablen): *getfield*, *putfield*
  - Zugriff auf Array-Komponenten: T*aload*, T*astore*
  - Array-Länge: *arraylength*
  - Überprüfen von Typ-Kompatibilität: *instanceof*, *checkcast*

- **Operanden-Stack Management Befehle:**
  - *pop*, *dup*, *swap*, etc.

- **Kontrollfluss Befehle:**
  - Bedingter Sprung: z.B. *if_icmple*, *ifnonull*, *ifacmpeq*
  - Zusammengesetzte Sprung Befehle: *tableswitch*, *lookupswitch*
  - Unbedingter Sprung: z.B. *goto*, *jsr*, *ret*

- **Methoden Aufruf und Return Befehle:**
  - Aufruf einer Instanzmethode: *invokevirtual*
  - Aufruf einer Interfacemethode: *invokeinterface*
  - Konstruktor Aufruf, private- oder super-Methodenaufruf: *invokespecial*
  - Aufruf einer Klassenmethode (static): *invokestatic*
  - Rückgabe eines Wertes vom Typ T$\in \{i, l, f, d, a\}$: T*return*
  - Rückkehr aus einer `void`-Methode: *return*

Bytecode Offsets:
Ziele von Sprungbefehlen werden als relative Offsets von der aktuellen Bytecode-Posistion kodiert. In den folgenden Beispielen, wie auch in der Ausgabe des

`javap`-Kommandos zur Textdarstellung des Bytecodes, sind zur besseren Lesbarkeit jedoch absolute Zieladressen angegeben.

Werden bei der Veränderung des Bytecodes Instruktionen gelöscht oder neue hinzugefügt, so müssen diese Offsets aktualisiert werden. Wie man in 2.1.2 sehen wird, ist dies eine der Schwierigkeiten, die bei Benutzung von BCEL vom Anwender ferngehalten werden.

Bytecode Ausführung

Die *Execution Engine* der JVM ist eine abstrakte Stackmaschine. Bei vielen Operationen sind deshalb die benötigten Operanden nicht direkt der Instruktion mitgegeben, sondern müssen auf dem Stack übergeben werden. Bei Ausführung einer solchen Instruktion wird erwartet, dass sich die entsprechenden Operanden auf dem Stack befinden. Andere Befehle besitzen jedoch direkte Operanden, zum Beispiel Verweise in den Constant pool oder Bytecode-Offsets für Sprungbefehle.

Auch bei Methodenaufrufen wird der Stack zur Parameterübergabe verwendet. Der Aufrufer legt dabei die Operanden auf den Stack und die aufgerufene Methode empfängt diese in lokalen Variablen[2]. Ebenso befindet sich das Ergebnis einer Operation anschließend auf dem Stack und kann so weiter verwendet werden. Nichtstatische Methoden erhalten implizit als erstes Argument das empfangende Objekt mitübergeben. Diese Referenz (z.B. this) bekommt die Empfänger-Methode in ihrer lokalen Variable Nr.0 übergeben. Die eigentlichen Methodenargumente werden in aufsteigender Reihenfolge in den darauffolgenden Variablen empfangen. Lokale Variablen mit höherer Nummer können zum Zwischenspeichern von Werten in der Methode benutzt werden. Hat eine Methode also *n* Argumente, so sind die Variablen mit Indizes *0* bis *n* für die Methodenargumente reserviert.

Bei statischen Methoden entfällt der implizite Objekt-Parameter und die Methodenargumente werden schon ab Variablen-Index 0 übergeben.

Bytecode Beispiel

Sei in einer Klasse `Test` folgende Methode definiert:

```
public int differenz (int i1, int i2) {
    int result = 0;
    if (i1 < i2)
        result = i2 − i1;
    else if (i1 > i2)
        result = i1 − i2;
    return result;
}
```

[2] Tatsächlich sind lokale Variablen auch als ein (geschützter) Teil des Stacks realisiert. Von dieser Tatsache merkt ein BCEL-Anwender jedoch nichts.

Daraus würde die folgende Bytecode-Sequenz erzeugt werden:

```
 0:  iconst_0          // Laden einer (konstanten) 0 auf den Stack
 1:  istore_3          // Speichern in Variable 3 (result)
 2:  iload_1           // Laden des ersten Methodenarguments (i1)
 3:  iload_2           // Laden des zweiten Methodenarguments (i2)
 4:  if_icmpge  #14    // Größer-Gleich-Vergleich zweier int's; Sprung nach 14, falls wahr
 7:  iload_2           // Laden von i2
 8:  iload_1           // Laden von i1
 9:  isub              // Subtraktion zweier int's auf dem Stack
10:  istore_3          // Speichern des Subtraktionsergebnisses in result
11:  goto      #23     // Sprung nach 23 (else-Zweig wird übersprungen)
14:  iload_1
15:  iload_2
16:  if_icmple #23
19:  iload_1
20:  iload_2
21:  isub
22:  istore_3
23:  iload_3           // Laden von result
24:  ireturn           // Rückgabe eines int's (result) vom Stack
```

Ein Aufruf dieser Methode

```
int i = differenz (1, −7);
```

aus der selben Klasse heraus, würde folgendermaßen übersetzt werden:

```
0:  aload_0                                  // Laden von this
1:  iconst_1                                 // Laden der int Konstante 1
2:  bipush      −7                           // Laden eines größeren int Wertes
                                             // mit Operand -7
4:  invokevirtual  Test.differenz (II)I (2)  // (*)
7:  istore_1                                 // Speichern des Ergebnisses in Variable 1
```

(*) Aufruf der (virtuellen) Methode `differenz` aus der Klasse `Test` mit der Signatur `(II)I` (zwei `int`-Argumente; Rückgabewert `int`).
Die `(2)` ist der eigentliche Operand der *invokevirtual*-Instruktion und stellt den Constant pool Index des entsprechenden *CONSTANT_Methodref* Eintrags dar.

Zusätzlich zu der Sequenz von JVM-Instruktionen enthält das Code-Attribut einer Methode (wiederum in Attributen) noch Informationen über die maximale Anzahl von Elementen auf dem Stack (*max_stack*), die maximale Anzahl verwendeter lokaler Variablen (*max_locals*) und die Länge der Code-Sequenz (*code_length*). Die Werte dieser Variablen müssen, bei Transformation des Bytecodes, mit den tatsächlichen Gegebenheiten synchron gehalten werden.

2.1.2 BCEL

BCEL ist eine Bibliothek zur statischen Analyse sowie zur dynamischen Erzeugung und Manipulation von Java Bytecode. Das class file Format wird dabei auf eine Klassenstruktur abgebildet, die eine abstraktere Sicht auf die Einträge eines class files erlaubt.

Statischer Teil

Der statische Teil der BCEL API ist zum Analysieren von Java Klassen geeignet, wenn zum Beispiel deren Sourcecode nicht zur Verfügung steht. Ein Java class file wird dabei auf die Hauptstruktur, die Klasse `JavaClass` abgebildet. Die Abbildung 2.3 zeigt die Repräsentationen der Hauptelemente des class file formats: JavaClass, ConstantPool, AccessFlags, Field, Method und Attribute.

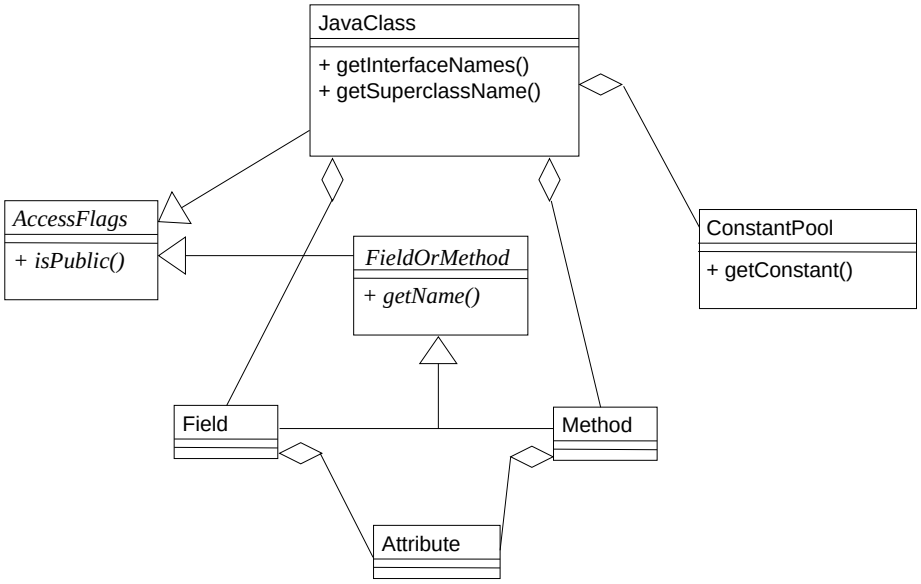

Abbildung 2.3: BCEL Klassendiagramm-Ausschnitt

Mit der Methode
`JavaClass Repository.lookupClass(String class_name)`
können Java Klassen parsiert, und die entsprechende `JavaClass`-Repräsentation erzeugt werden. Die *statischen* Repräsentationen eignen sich vor allem zum Lesen/Schreiben[3] von class files aus/in Dateien und nicht zur Transformation von Bytecode.

Dynamischer Teil

Zum Generieren oder Modifizieren von JavaClass Objekten wird der dynamische Teil der BCEL API verwendet. Hier finden sich Methoden zum Erzeugen von Klassen, Feldern und Methoden, sowie zum Hinzufügen und Entfernen von allen möglichen Teilen einer Klasse und ihrer Unterstrukturen. Auch zum Erstellen und Modi-

[3] JavaClass-Objekte können entweder durch einen Konstruktor, welcher alle Elemete eines class files spezifiziert, oder aus einem (mit dem dyamischen Teil der API erzeugten) ClassGen-Objekt erzeugt werden.

fizieren von Methoden-Code sind die nötigen Strukturen und Methoden gegeben. Für genauere Informationen sei auf die BCEL API Dokumentation verwiesen.

Im folgenden soll nun auf einige ausgewählte Besonderheiten bei der Implementierung von Bytecode Transformationen mit BCEL eingegangen werden.

Vorteile von BCEL's "Higherlevel-View" auf dem Bytecode

BCEL abstrahiert von bestimmten Details der class files und vereinfacht so die Implementierung von Bytecode-Transformationen.

- Abstraktion von konkreten Bytecode-Positionen:
  Man hat eine abstrakte Sicht auf den Kontrollfluss, ohne sich um Details wie konkrete Bytecode-Offsets kümmern zu müssen. Dies wird durch die weiter unten beschriebenen Klassen `InstructionList` und `InstructionHandle` realisiert.

- Abstraktion über spezielle Typen von Instruktionen:
  Dies ist durch die Klasse `InstructionFactory` realisiert. Beispielsweise können Lade-Befehle *iload, lload, fload, dload, aload* mit
  `InstructionFactory.createLoad(Type t, int CPIndex)`
  erzeugt werden. Man muss den speziellen Typ nicht kennen, sondern nur über *abstrakte Kopplung* auf ihn zugreifen können. So lassen sich zum Beispiel die Argumente einer Methode bequem in einer Schleife laden. Ansonsten müsste man erst per switch-Anweisung die einzelnen Typen überprüfen und jeweils den richtigen Ladebefehl auswählen.

Der Code einer Methode wird in BCEL in Form einer `InstructionList` verwaltet. Es existieren Methoden zum Hinzufügen und Löschen von Instruktionen aus der Liste. Einzelne Bytecode-Instruktionen werden durch `InstructionHandle`s repräsentiert. Dabei handelt es sich um symbolische Referenzen auf Instruktionen, die es ermöglichen während der Transformations-Phase von konkreten Bytecode Offsets zu abstrahieren. Dies vereinfacht das Einfügen und Löschen von Bytecode-Instruktionen, da der Programmierer nicht darauf achten muss, Offsets dadurch "verschobener" Instruktionen anzupassen. Erst wenn aus einer InstructionList der eigentliche Bytecode erzeugt wird, werden die konkreten Offsets automatisch berechnet.

Signaturveränderungen

Während das Verändern der Implementierung einer Methode also relativ gut von BCEL unterstützt wird, ist es recht umständlich, Signaturveränderungen zu realisieren. Zwar sollten Methoden-Signaturen genaugenommen gar nicht verändert

werden, weil dies die Binärkompatibilität (siehe S.28) einer Klasse zunichte machen würde, aber es besteht natürlich die Möglichkeit eine Methode zu kopieren und mit veränderter Signatur einer Klasse zusätzlich hinzuzufügen. Wird dabei die Signatur um zusätzliche Parameter erweitert, so werden die Indizes der danach folgenden alten Parameter sowie aller sonstigen lokalen Variablen der Methode ungültig, da die zusätzlichen Parameter dazwischen geschoben werden. Die Indizes von Lade- und Speicher-Instruktionen, die diese ”verschobenen” Variablen betreffen, müssen deshalb angepasst, d.h. um die Anzahl der hinzugefügten Parameter erhöht, werden. Dies wird dem Programmierer leider nicht durch BCEL erleichtert, sondern muss von Hand gemacht werden. Dazu muss jede Instruktion der `InstructionList` der entsprechenden Methode untersucht und gegebenenfalls angepasst werden. Gäbe es eine Abstraktion von konkreten lokalen Variablen-Indizes, ließe sich dieses Problem sicherlich eleganter lösen.

Beispiel: Erzeugung von Bytecode mit BCEL

Die folgenden Code-Auszüge stammen aus einem Beispiel, das im Anhang (4.4) als Ganzes zu finden ist. Es handelt sich dabei um eine Methode, die ein übergebenes `Method`-Objekt transformiert. Es werden Instruktionen erzeugt und in den vorhandenen Methoden-Code eingebaut. Sie sorgen dafür, dass am Anfang eines jeden Aufrufs der Methode alle übergebenen Argumente ausgegeben werden.

Der folgende Code erzeugt eine neue `InstructionList` `patch` und fügt ihr die Instruktionen hinzu, die nötig sind, um nacheinander alle Argumente der Methode auf den Stack zu laden und auszugeben. Dabei muss, wie in 2.1.1 (S. 21) beschrieben, für den Beginn der Parameter-Indizes zwischen statischen und nicht-statischen Methoden (`m`) unterschieden werden (Zeile 5-8).

`argTypes` ist ein Array, welches die Typen der Methoden-Argumente enthält. Die for-Schleife wird also für jedes Argument ausgeführt. Das eigentliche Erzeugen und Einfügen der Instruktionen in den `patch` geschieht in den Zeilen 10-11, 12 und 16-20.

In Zeile 10 u. 11 wird das statische Feld `java.io.PrintStream.out` der Klasse `java.lang.System` geladen. Zeile 12 lädt das aktuelle Methoden-Argument und in den Zeilen 16-20 wird ein Aufruf der Methode `print` mit den geladenen Parametern erzeugt.

Hier tritt der Vorteil der Abstraktion über konkrete Typen zutage: Obwohl, abhängig vom Typ des aktuell geladenen Parameters, u.U. unterschiedliche `print`-Methoden aufgerufen werden müssen, lassen sich alle mit ein und der selben Anweisung erzeugen. Der vierte Parameter (Zeile 19) der `createInvoke`-Methode bestimmt die Argument-Typen der aufzurufenden Methode. Er wird vom aktuellen Methoden-Argument determiniert. `factory` ist eine Instanz der `InstructionFactory`, die für alle möglichen getypten Instruktionen allgemeine (die Typen als Parameter erwartende) Funktionen bereitstellt.

In den Zeilen 14 und 15 wird der Argumenttyp der print-Methode auf `Object` gesetzt, falls es sich bei dem aktuellen Methoden-Argument um einen Referenztyp handelt. Dies ist nötig, weil es zwar print-Methoden für alle Grundtypen gibt, alle Referenztypen jedoch mit der Methode `print(Object o)` ausgegeben werden. Der Parameter darf natürlich von jedem beliebigen Referenztyp sein, bei der Erzeugung eines Methodenaufrufs müssen jedoch die statisch deklarierten Signatur-Typen der Methode angegeben werden.

: Erzeugung von Bytecode-Instruktionen

```
1  ObjectType p_stream = new ObjectType("java.io.PrintStream");
2   InstructionList patch = new InstructionList();
3
4  int index;
5  if (m.isStatic())
6      index = 0;
7  else
8      index = 1;
9  for (int i=0; i<argTypes.length; i++) {
10     patch.append(factory.createFieldAccess("java.lang.System", "out", p_stream,
11                                            Constants.GETSTATIC));
12     patch.append(factory.createLoad(argTypes[i], index));
13     Type printArgType = argTypes[i];
14     if (!(printArgType instanceof BasicType))
15         printArgType = new ObjectType("java.lang.Object");
16     patch.append(factory.createInvoke("java.io.PrintStream",
17                                       "print",
18                                       Type.VOID,
19                                       new Type[] { printArgType },
20                                       Constants.INVOKEVIRTUAL));
21     index += argTypes[i].getSize();
22 }
```

Das nächste Listing demonstriert das Einfügen des erzeugten Codes an den Anfang der Methode. `mg` ist eine `MethodGen`-Repräsentation der zu transformierenden Methode. Zunächst wird auf die `InstructionList` der Methode und deren Liste von `InstructionHandle`s zugegriffen (Zeilen 1 und 2).

Bei Konstruktoren (Methodenname `<init>`) muss darauf geachtet werden, dass `super()`-Aufrufe oder Aufrufe anderer Konstruktoren am Anfang des Codes bleiben müssen, das heißt diese müssen übersprungen werden. Dies wird durch die Zeilen 4-11 realisiert. Hier wird auf die einzelnen InstructionHandles zugegriffen und der im vorherigen Listing erzeugte `patch` wird in die Liste eingefügt, wenn alle initialen Konstruktor-Aufrufe getätigt wurden. Dass es sich bei einer Instruktion um einen solchen handelt, wird mit der Zugehörigkeit zur Instruktions-Klasse `INVOKESPECIAL` überprüft. Genaugenommen müsste hier noch überpüft werden, ob der Name der aufgerufenen Methode tatsächlich `<init>` ist, da nicht nur Konstruktor-Aufrufe, sondern auch Aufrufe von privaten Methoden sowie `super`-Aufrufe mit `invokespecial` durchgeführt werden.

Handelt es sich bei der zu transformierenden Methode nicht um einen Konstruktor, so wird der `patch` einfach an den Anfang, also vor (`insert`) die erste Instruktion in der InstructionList eingefügt (Zeile 13).

In den Zeilen 15 und 16 wird für ein Updaten der Code-Attribute (siehe 2.1.1, S.22) *max_stack* und *max_locals* gesorgt. Die hier verwendeten Methoden `setMaxStack()` und `setMaxLocals()` berechnen die jeweiligen Werte automatisch. Es gibt auch die wahrscheinlich etwas performantere Variante, bei welcher der Programmierer die Werte direkt angibt.

In Zeile 17 wird aus dem zuvor bearbeiteten `MethodGen`-Objekt eine `Method` erzeugt. Dabei wird die Methode *finalisiert*, d.h. dass nun auch die konkreten Bytecode-Offsets berechnet und festgelegt werden. Um Speicherplatz zu sparen, wird in Zeile 18 die verwendete InstructionList wieder freigegeben.

: Manipulation der Liste von Instruktionen

```
1  InstructionList il = mg.getInstructionList();
2  InstructionHandle[] ihs = il.getInstructionHandles();
3
4  if(method_name.equals("<init>")) { // Aufrufe von 'super' oder anderen Konstruktoren
5                                        muessen am Anfang bleiben
6      for(int j=1; j < ihs.length; j++) {
7          if(ihs[j].getInstruction() instanceof INVOKESPECIAL) {
8              il.append(ihs[j], patch);
9              break;
10         }
11     }
12 } else
13     il.insert(ihs[0], patch);
14
15 mg.setMaxStack();
16 mg.setMaxLocals();
17 Method generatedMethod = mg.getMethod();
18 il.dispose();
```

Darstellung des Bytecodes

Während der Implementierung von Bytecode-Transformationen hat es sich als sehr hilfreich erwiesen, die veränderten oder erzeugten Klassen sichtbar zu machen. Außer mit dem oben erwähnten `javap`-Kommando, kann man sich den Inhalt eines class files auch mit dem BCEL-Programm *listclass* anzeigen lassen. listclass ist ein Java Programm, welches mit Hilfe der BCEL API class files analysiert und unter anderem den Constant pool, Felder, Methoden und deren Bytecode auflistet. Durch Decompilation (beispielsweise mit dem Dekompiler *jad*)der erzeugten Klassen zurück in Java Sourcecode(beispielsweise mit dem Dekompiler *jad*), lässt sich eine noch bessere Übersicht über das Ergebnis der Transformationen bekommen. Allerdings sind die Informationen aus dem class file detaillierter und dadurch für debugging Zwecke nützlicher.

2.2 Loadtime Adaption

Java unterstützt dynamisches Binden[4], d.h. erst zur Laufzeit wird bestimmt, welche Klassen tatsächlich von einem Programm benutzt werden. Um genau diese Klassen zu transformieren, muss damit also bis zum Ladezeitpunkt gewartet werden[5]. Bei normaler Ausführung eines Java-Programms werden die kompilierten Java-Klassen in die *Java Virtual Machine* (JVM) geladen und interpretiert. JMangler ([9]) ist ein Framework zur Loadtime Adaption von Java Programmen. Es klinkt sich in den Ladeprozess der JVM ein und wendet, in sogenannten *Transformern* definierte, Adaptionen auf geladene Klassen an, bevor diese ausgeführt werden.

Das Eingreifen in den Prozess des Klassenladens wird durch Austauschen der System-Klasse *ClassLoader* durch eine modifizierte Version realisiert[6]. Dadurch ist der Transformationsprozess sowohl vom Class Loader, als auch von der JVM unabhängig. Die Transformer werden immer dann aktiv, wenn eine anwendungsspezifische Klasse geladen wird, unabhängig davon mit welchem speziellen Class Loader, da diese alle von der modifizierten Superklasse erben. Ausgenommen davon sind die Java Systemklassen, die vom *bootstrap class loader* geladen werden und somit von JMangler nicht modifizierbar sind.

Binärkompatibilität

Die Bewahrung der Binärkompatibilität ist eine wichtige Bedingung für sinnvolle Bytecode Transformationen. Damit wird zugesichert, dass die Programmveränderungen kein Neukompilieren von Clients nötig machen. Würde beispielsweise die Sichtbarkeit eines Feldes von `public` auf `private` geändert, so wäre die Binärkompatibilität der Klasse verletzt, da ein vorher erlaubter Client-Zugriff auf das Feld so plötzlich nicht mehr möglich wäre.

2.2.1 JMangler Transformationen

Transformationen, die JMangler während der Ausführung eines Programms durchführt, werden in *Transformern* definiert. Dabei handelt es sich um Java Klassen, die mit der JMangler API implementiert werden und bestimmte Interfaces implementieren. Deren Methoden bekommen zur Laufzeit objektorientierte Repräsentationen aller geladenen Klassen in Form von BCEL `ClassGen` Objekten

[4] Hiermit ist nicht der dynamische Method-Dispatch-Prozess gemeint, sondern das dynamische Auffinden der zunächst nur als Strings im Constant pool bekannten Klassen.

[5] Da bereits in die JVM geladene Klassen nicht mehr transformiert werden können, kommt auch kein späterer Zeitpunkt infrage.

[6] Dies war zumindest in der JMangler Version 1.x der Fall. In der Version 2.0 wird scheinbar eine neue (geheime?) Strategie angewendet.

übergeben und können diese analysieren und die durchzuführenden Transformationen spezifizieren. Für die Implementierung der Transformationen wird die BCEL API verwendet, was bedeutet, dass hierfür eine gute Kenntnis der Struktur des Java Bytecodes erforderlich ist.

Man unterscheidet zwischen zwei Arten von Transformationen und damit auch von Transformern. Dabei handelt es sich einerseits um *Interface-Transformer*, die die Schnittstelle einer Klasse verändern (genauer: erweitern), und andererseits um *Code-Transformer,* die den Bytecode von Methoden transformieren.

Interface-Transformer

Interface-Transformationen umfassen:

- Hinzufügen von Klassen, Methoden und Feldern
- Ändern der *throws*-Definition einer Methode
- Ändern der direkten Superklasse einer Klasse, wenn die Menge der Superklassen oder Superinterfaces dabei kein Element verliert.
- Erweitern der Sichtbarkeit von Klassen, Methoden oder Feldern
- Hinzufügen oder Entfernen bestimmter Modifier von Klassen, Methoden oder Feldern

Wie man sieht, ist es nicht erlaubt Signaturelemente zu löschen oder Sichtbarkeiten einzuschränken. Die Transformationen stellen immer eine *Erweiterung* der transformierten Klassen dar. Eine Transformation, die nur Programm-Eigenschaften hinzufügt, bezeichnet man als *monoton*. Zulässige Transformationen müssen sinnvollerweise monoton sein, um die Binärkompatibilität der veränderten Klasse sicherzustellen. Insbesondere folgt daraus auch, dass Signaturen nicht geändert werden dürfen.

Interface-Transformer implementieren das Interface `InterfaceTransformerComponent`. Die durchzuführenden Transformationen werden in der Methode `ExtensionSet transformInterface(UnextendableClassSet cs)` definiert. Die Änderungen werden dabei einem `ExtensionSet` hinzugefügt und zurückgegeben. Vor der tatsächlichen Durchführung der Transformationen prüft JMangler zunächst, ob die Transformationen zulässig sind (siehe dazu 2.2.2).

Code-Transformer

Code-Transformer ändern den JVM Code, das heißt die Implentation von Methoden. Um Code-Transformationen zu definieren, wird das Interface `CodeTransformerComponent` und insbesondere die Methode `void transformCode(UnextendableClassSet cs)` implementiert. Veränderungen von Methodencode haben keinen Einfluss auf die Binärkompatibilität gegenüber Aufrufern, sondern ”nur” auf die Semantik der Methode. Code kann deshalb beliebig verändert und sogar gelöscht werden[7].

Kombinierbarkeit von Transformern

Werden mehrere Transformer unabhängig voneinander entwickelt und unantizipiert gleichzeitig auf dasselbe Programm angewendet, so stellt sich die Frage, in welcher Reihenfolge dies geschehen soll. Ideal wäre natürlich eine automatische Kombinierbarkeit beliebiger Transformer. Die Motivation für eine Trennung zwischen Interface und Code-Transformationen begründet sich in der unterschiedlichen Komponierbarkeit von Transformern der einen bzw. der anderen Art.

Das **Interface einer Klasse** kann als *ungeordnete Menge* von Namen und Signaturen betrachtet werden. Da die Reihenfolge, in welcher zusätzliche Elemente in eine Menge eingefügt werden keinen Einfluss auf die resultierende Menge hat, spielt es demnach auch keine Rolle, in welcher Reihenfolge Interface Erweiterungen durchgeführt werden. Allerdings muß darauf geachtet werden, dass jede Interface-Transformation auf jeden Teil eines Programms angewendet wird. Dies gilt insbesondere auch für Programmteile, die von anderen Interface-Transformern hinzugefügt wurden und kann ein iteratives Mehrfachanwenden von Transformern nötig machen (siehe 2.2.2).

Damit dies funktioniert, und die Transformer damit frei kombinierbar sind, dürfen Interface-Transformationen jedoch nur *positiv getriggert* werden. Triggern meint das Auslösen von Transformationen durch bestimmte Programm-Eigenschaften (wie z.B. ”die Methode ist *public*”). Positiv getriggerte Transformationen werden bei *Vorhandensein* (im Gegensatz zur Abwesenheit) einer bestimmten Eigenschaft durchgeführt.

Negativ getriggerte Transformationen können die Reihenfolgeunabhängigkeit zunichte machen, wie folgendes Beispiel zeigt:
Sei T_a eine Transformation, die nur dann ausgeführt wird, wenn eine Klasse ein bestimmtes Feld *nicht* enthält. T_b sei eine weitere Transformation, die der Klasse genau dieses Feld hinzufügt. Käme T_b zuerst zum Zuge, so wäre die Trigger-Bedingung für T_a nicht mehr erfüllt. Bei umgekehrter Anwendungsreihenfolge

[7] Es muss nur darauf geachtet werden, dass die nötigen Return-Instruktionen präsent sind.

würde T_a jedoch ausgelöst werden. Die entstehenden Klassen wären also unterschiedlich.

Der **Bytecode einer Methode** entspricht einer *geordneten Liste* von Befehlen. Unterschiedliche Reihenfolgen von Elementeinfügungen führen hierbei zu unterschiedlichen Ergebnislisten. Daraus folgt, dass die Reihenfolge, in welcher Code-Transformationen angewendet werden, einen entscheidenden Einfluss auf den resultierenden Methodencode hat, und somit nicht beliebig ist. Aus diesem Grund muss für Code-Transformer die Anwendungs-Reihenfolge explizit festgelegt werden. Dies geschieht in der anschließend beschriebenen Konfigurations-Datei.

Konfiguration der Transformer

Durch eine in XML geschriebene Konfigurations-Datei lässt sich das konkrete Verhalten von JMangler steuern. Hier wird definiert, welche Transformer-Klassen auf die geladenen Klassen angewendet werden sollen. Dabei muss auch angegeben werden, welche Transformer Code-Transformationen, und welche Interface-Transformationen durchführen. Für Code-Transformer muss zusätzlich die Anwendungsreihenfolge bestimmt werden.

Weiterhin können den Transformern in der Konfigurations-Datei Parameter übergeben werden, die diesen dann als String-Argument in ihrem Konstruktor zugänglich gemacht werden können.

Es besteht außerdem die Möglichkeit, weitere Settings vorzunehmen, wie beispielsweise die maximale Anzahl der Iterationen für die Anwendung von Interface-Transformern.
Zu Debugging-Zwecken lässt sich die Ausgabe bestimmter Informationen über den Transformationsprozess und die beteiligten Transformer einschalten. Hier kann auch festgelegt werden, ob die transformierten Klassen, die normalerweise nur die JVM zu Gesicht bekommt, in einem Extraverzeichnis als class files gespeichert werden sollen. Als Möglichkeit, das Ergebnis der Transformationen explizit sichtbar zu machen, erwies sich dies während der Entwicklung der Laufzeitumgebung von ot-java als sehr nützlich.

2.2.2 Der Transformations-Prozess

Das JMangler-System wird durch einen erweiterten Aufruf der JVM (`java`) aktiviert. Der eigentliche Transformations-Prozess von JMangler ist ebenfalls in Abbildung 2.1 dargestellt und soll im folgenden beschrieben werden.

Der **Transformation Manager** dient als Schnittstelle zwischen dem *Classloader System* und dem *Composition Algorithm*.

Der **Composition Algorithm** ist für die Aktivierung und Koordination der Transformer verantwortlich. Er führt den Transformations-Prozess, für jede geladene Klasse, in zwei Schritten durch.

Als erstes werden die von den Interface-Transformern angeforderten Transformationen betrachtet. Der Algorithmus überprüft, ob Konflikte zwischen den einzelnen Transformationen bestehen und löst diese gegebenenfalls auf. Ein Konflikt könnte beispielsweise auftreten, wenn zwei Transformer versuchen ein Feld mit demselben Namen zu einer Klasse hinzuzufügen. Außerdem wird eine Validitäts-Überprüfung durchgeführt, die sicherstellt, dass die Binärkompatibilität erhalten bleibt. Gibt es keine unlösbaren Konflikte, führt der Composition Algorithm die Transformationen in einer von ihm bestimmten Reihenfolge durch.

Wie weiter oben erwähnt, reicht es im Allgemeinen nicht aus, jede Transformation nur einmal durchzuführen. Aus diesem Grund werden alle Interface-Transformationen iterativ so lange ausgeführt, bis sich keine weiteren Änderungen mehr ergeben, also ein *Fixpunkt* erreicht ist. Das bedeutet, dies geschieht solange, bis keine Interface Modifikationen mehr anstehen. Interface-Transformationen, die *positiv getriggert* und *monoton* sind, haben die Eigenschaft, dass eine terminierende Iteration einen eindeutigen Fixpunkt liefert.

Anschließend werden die Code-Transformer aktiviert. Die in ihnen spezifizierten Transformationen werden in der genau festgelegten Reihenfolge auf die geladenen Klassen angewendet.

Multi-class Transformationen

JMangler ermöglicht es, dass ein Transformer während einer Aktivierung mehrere Klassen transformiert. Dies kann dann sinnvoll sein, wenn die Transformationen dieser Klassen von einander abhängig sind. Dafür muss ein Transformer explizit andere Klassen ”nachladen”. Über die Methode `loadClass` des `ExtensionSet`’s kann ein verfrühtes Nachladen getriggert werden. Die Klassen werden dann dem `UnextendableClassSet` der Transformations-Methoden hinzugefügt und können so mittransformiert werden.

Nach Abschluss der Transformation wird jedoch nur die ursprünglich vom Classloader System der JVM angeforderte Klasse an diese übergeben. Alle weiteren, ”künstlich” geladenen und eventuell transformierten Klassen werden (in Form von `ClassGen`-Repräsentationen) in einem `WaitingRoom` zwischengespeichert, bis sie entweder erneut von einem Transformer angefordert, oder schließlich tatsächlich in die JVM geladen werden.

Ein Beispiel für eine Multi-class Transformation wäre ein Transformer, der allen Klassen einen Vektor mit den Namen aller, diese Klasse benutzenden, Klassen hinzufügt. Wird in einer Klasse A eine Referenz auf eine Klasse B gefunden, so wird diese Information im Transfomer gespeichert und Klasse B wird geladen.

In einem *usedBy*-Vektor (der der Klasse B beim ersten Mal hinzugefügt werden muss), kann der Transformer nun den Namen der Klasse A ablegen.

Kapitel 3

Die Laufzeitumgebung für Object-Teams/Java

Ziel des praktischen Teils dieser Arbeit war es die Laufzeitumgebung für Object-Teams/Java (ot-java) bereitzustellen. Sie fungiert als *backend* für den *Object-Teams/Java Compiler* (zur Entwicklung des ot-Compiler siehe [4]).

Die ot-java Laufzeitumgebung ist in der derzeitigen Version für die Realisierung von callins, sowie der Möglichkeit zur Aktivierung mehrerer Teams verantwortlich. Entwicklung und Funktionsweise der dafür nötigen Bytecode Transformationen, sowie das Zusammenspiel mit dem ot-Compiler, sollen im folgenden dargestellt werden.

3.1 Die Schnittstelle zum Object-Teams/Java-Compiler

ot-java Sourcecode wird vom ot-Compiler übersetzt. Er ist u.a. dafür verantwortlich *callouts* in tatsächliche Methodenaufrufe der Basismethoden zu verwandeln. Dem erzeugten ”normalen” Java Bytecode sind allerdings zusätzliche Attribute, als Träger bestimmter Informationen hinzugefügt. Abbildung 3.1 gibt einen Überblick über die Verantwortlichkeiten von Compiler und Laufzeitumgebung für ot-java.

3.1.1 Typ-Lifting

Will man zu einem Basis-Objekt die zugehörige Rolle erhalten, so muss man das Basis-Objekt zu dieser *liften* (siehe *Implizite Umwandlung zwischen Rolle und Basis* in 1.2.2; S.9). Zu diesem Zweck werden vom ot-Compiler in jedem Team *lift-Methoden* für alle zum Team gehörenden Rollen erzeugt.

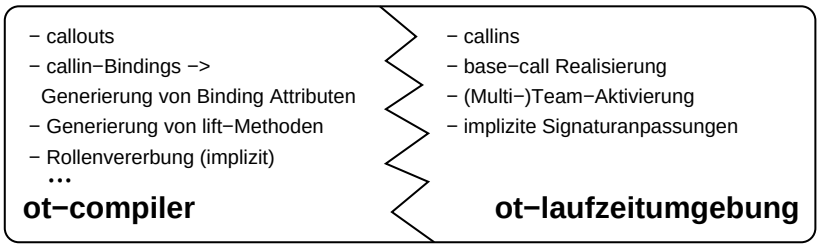

Abbildung 3.1: Verantwortlichkeiten von ot-Compiler und ot-Laufzeitumgebung

Eine lift-Methode die ein Basis-Objekt vom Typ *Base* zum Typ *Role* liftet, hat die Signatur Role *_OT$*liftToRole(Base base). Aufgrund von, zunächst nicht ersichtlichen, Problemen bei der Vererbung von Teams, wurde die Sichtbarkeit der lift-Methoden im Laufe der Entwicklung von ot-java verändert (siehe 3.4.4).

Wenn versucht wird ein Basis-Objekt zu liften, welchem noch keine Rolle zugeordnet wurde, so wird nun ein Rollen-Objekt dafür erzeugt. Paare von zusammengehörigen Basis- und Rollen-Objekten werden vom Team in einem Cache verwaltet. Falls nun erneut versucht wird dasselbe Basis-Objekt zu liften, wird keine neue Rolle erzeugt, sondern das entsprechende Objekt wird aus dem Cache geholt und zurückgegeben.

3.1.2 Informations-Transfer durch Attribute

Kompilierte Java Klassen werden, wie in 2.1.1 (S. 17) geschildert, im Java *class file format* gespeichert. Wie ebenfalls dort beschrieben, bieten *Attribute* die Möglichkeit einige Strukturen der class files mit zusätzlichen Instrumentierungen auszustatten. Im folgenden wird erklärt, wie diese Technik für ot-java eingesetzt wird.

Der ot-Compiler findet in Teamklassen callin-Bindings vor und legt diese Informationen in Attributen des Teams bzw. der Rollenklassen ab. Abbildung 3.2 gibt eine Übersicht über die generierten Attribute. Eine detaillierte Beschreibung ist im Anhang (4.2.3) zu finden.

RoleBaseBindings ordnen eine Rollenklasse einer Basisklasse zu. *MethodBindings* enthalten jeweils die Namen und Signaturen von korrespondierender Rollen- und Basismethode, sowie den dazugehörigen Callin-Modifier. *ParameterBindings* enthalten Paare von aufeinander abgebildeten Parametern von Rollenmethode und Basismethode.

Desweiteren übersetzt der Compiler den Modifier *callin* in ein entsprechendes Accessflag und codiert es so in die class-files. Dies sind die Quellen aus denen das Laufzeitsystem erfährt, welche Basismethoden ein verändertes Verhalten aufweisen sollen. Man könnte also sagen, die Attribute steuern den Transformationsprozess.

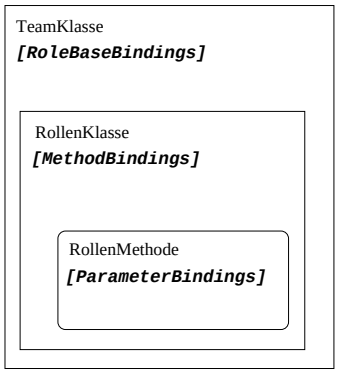

Abbildung 3.2: Callin-Attribute

Teams vor Basisklassen transformieren

Während der Laufzeittransformation werden die vom Compiler abgelegten Informationen aus den Attributen ausgelesen. Hierbei spielt die Reihenfolge der gelandenen Klassen eine entscheidende Rolle. Bevor damit begonnen wird, die Basisklassen zu transformieren, müssen die Binding-Attribute sämtlicher Team- und Rollenklassen untersucht werden, das heißt, diese müssen vor den Basisklassen geladen werden. Zu diesem Zweck generiert der ot-java Compiler in alle Klassen, die ein *Team* referenzieren das Attribut *ReferencedTeams*, welches die Namen aller benutzten Teams enthält. Dies geschieht typischerweise in Klassen mit ”main”-Funktionalität, da üblicherweise dort Teams instanziiert und verwaltet werden. Trifft die Laufzeitumgebung auf eine solche Klasse, so forciert sie das Laden der referenzierten Teams. Dafür wird die Methode `loadClass` der JMangler-Klasse `ExtensionSet` verwendet, die es erlaubt ein sofortiges Laden von Klassen zu erzwingen (siehe 2.2.2 Multi-class Transformationen).

Attribute so früh wie möglich auslesen

Die Klassen werden von mehreren Transformern transformiert. Die Informationen aus den Attributen, werden dabei ebenfalls von mehreren von ihnen benötigt. Die Reihenfolge für Interface Transformer wird in JMangler nicht explizit festgelegt. Die Attribut-Informationen müssen jedoch unbedingt von dem ersten Transformer ausgelesen werden. Da nicht klar ist, welcher Interface Transformer als erstes an die Reihe kommt, muss dies also jeder von ihnen tun können. Damit die Attribute jedoch nicht unnötigerweise mehrfach ausgelesen werden, gibt es einen *Wächter* (`org.objectteams.transformer.util.AttributeReadingGuard`) , der registriert, für welche Klassen die Attribute bereits ausgelesen worden sind. Bevor ein Transformer versucht die Attribute einer Klasse auszulesen, fragt er bei diesem Wächter nach, ob das noch nötig ist.

3.1.3 Verwaltung der Binding-Informationen

Ein vollständiges callin-Binding besteht aus einem *RoleBaseBinding*, einem *MethodBinding* und den dazugehörigen *ParameterBinding*s. Diese Teilinformationen sind jedoch nicht am Stück verfügbar, sondern werden während des Transformationsprozesses nach und nach ausgelesen.

Wird eine Team-Klasse geladen, so werden zunächst nur die darin enthaltenden RoleBaseBindings ausgelesen und müssen gespeichert werden. Erst wenn die Rollenklassen geladen werden, kann auf die dazugehörigen MethodBindings zugegriffen werden. ParameterBindings werden erst bei Untersuchung der Rollenmethoden ausgelesen. Aus diesem Grund muß bei der Verwaltung der Binding-Informationen ein schrittweises Speichern der callin-Bindings möglich sein.

Im Package org.objectteams.transformer.util wird eine Infrastruktur zum Ablegen und Auslesen der Binding-Informationen bereitgestellt. Der `CallinBindingManager` stellt Methoden zur Verfügung, mit denen zunächst ein RoleBaseBinding angelegt und später ein MethodBinding dazugefügt werden kann. Auch zum Auslesen der Bindings werden Methoden angeboten, die beispielsweise ein Binding für eine gegebene Basismethode zurückliefern. Diese Methode wird bei der Realisierung eines callins benutzt, um für Basismethoden zu prüfen, welche Bindings es gibt, welche Rollenmethoden also aufgerufen werden müssen.

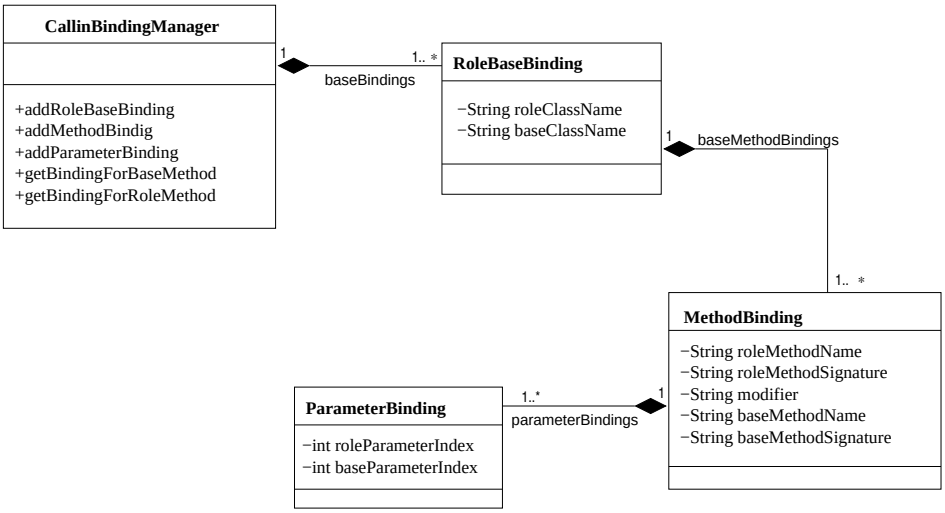

Abbildung 3.3: Klassen des Packages org.objectteams.transformer.util

Abbildung 3.3 zeigt die Klassenstruktur des Infrastruktur-Packages. Dabei sind die Verbindungen zwischen den beteiligten Klassen vereinfacht dargestellt. Tatsächlich verbergen sich hinter den Assoziationen folgende Datenstrukturen:

- baseBindings :
  HashMap(BaseClassName, HashMap(RoleClassName, RoleBaseBinding))
- baseMethodBindings :
  HashMap(BaseMethodName, HashMap(RoleMethodName, MethodBinding))
- parameterBinding :
  Vector(ParameterBinding))

3.2 Die Laufzeitumgebung

Die Laufzeitumgebung führt ihre Programm-Transformationen mit Hilfe von JMangler-Transformern (eingführt in 2.2) durch. Da JMangler Transformationen generell auf Bytecodeebene, das heißt auf den class files operieren, benötigt auch ot-java zur Anwendung von Transformationen keinen Sourcecode. Zudem sind alle Veränderungen jeweils nur in den von der JVM gelandenen Klassen wirksam. Die class-files im Dateisystem bleiben dagegen unverändert. Insbesondere die Basisklassen verbleiben dadurch im Originalzustand. Sie müssen zu keinem Zeitpunkt neu kompiliert werden und können sogar in Form von jar-files vorliegen. Dadurch wird nebenbei auch das Konfigurationsmanagement erleichtert: es muss nicht dafür gesorgt werden, dass die veränderten und die Original-Klassen nebeneinander existieren können.

Die vom ot-Laufzeitsystem durchgeführten Transformationen lassen sich nach verschiedenen Kriterien unterscheiden. Zunächst gibt es dabei zwei verschiedene Arten von Klassen, die transformiert werden: die *Basisklassen* auf der einen Seite und die *Team*- bzw. *Rollenklassen* auf der anderen.

Eher *statische* Transformationen werden hierbei dazu verwendet, um die generelle Team-Infrastruktur bereitzustellen, die das Aktivieren und Deaktivieren von Teams für Basisklassen zur Laufzeit erlaubt.

Dynamischer geht es dagegen beim Erzeugen der für die callin-Funktionalität nötigen Codefragmente zu. Hier müssen sowohl die Argument- und Rückgabetypen der betroffenen Methoden berücksichtigt werden, als auch den unterschiedlichen Ansprüchen der verschiedenen callin-Modifier Rechnung getragen werden.

Diese beiden Unterscheidungsmerkmale lassen das Problem in vier Teilbereiche zerfallen, was sich nicht zufällig in der Realisierung durch die vier Transformer des packages *org.objectteams.transformer* widerspiegelt[1].

[1] Im Laufe der Entwicklung sind inzwischen noch weitere Transformer hinzugekommen.

Im folgenden wird zunächst auf die statischen und anschließend auf die dynamischen Transformationen eingegangen.

Die hier dargestellten von den Transformern erzeugten Codefragmente treten bei regulärer Benutzung von ot-java nie explizit als Sourcecode auf. Sie existieren lediglich in Classfile-Form und können mit einem Decompiler sichtbar gemacht werden. Lediglich zur besseren Verständlichkeit werden die Codeerzeugnisse hier anhand eines solchen Decompilats vorgestellt.

Die Namen von erzeugten Feldern und Methoden haben zwecks Verhinderung von Namenskonflikten das Präfix *_OT$* bzw. einen weiteren *$* und einen Bezeichner angehängt. Laut [6, §3.8] gilt die Konvention, dass das Dollarzeichen nur in ”mechanisch” generiertem Sourcecode verwendet werden darf. Identifier, die ein $ enthalten, können so als ”künstlich” erzeugt erkannt werden. Weiterhin folgt daraus, dass derartige Methoden nie in normalem Sourcecode aufgerufen werden sollen.

3.3 Statische Transformationen

3.3.1 Teams werden *aktivierbar*

Team-Klassen erben durch die Angabe des Modifiers *team* implizit Methoden der ot-java internen Klasse *org.objectteams.Team*. Jede nicht abstrakte Team-Subklasse, die callin-Bindings für mindestens eine Basisklasse enthält, muss deshalb diese Methoden implementieren.

Die Methoden *public void activate()* und *public void deactivate()* dienen zum ein- bzw. ausschalten eines Teams. Ein Team wird aktiviert, indem es bei jeder von ihm durch callins beeinflussten Basisklasse registiert wird. Bei der Durchführung von callins werden von der Basisklasse nur jene Teams beachtet, die zum Zeitpunkt des Methodenaufrufes aktiv sind.

Basisklassen sind zur Compilezeit noch in ihrem ursprünglichen Zustand und besitzen deshalb noch nicht die für das An- und Abmelden benötigten Methoden und Datenstrukturen. Aus diesem Grund können diese Team-Methoden erst zur Laufzeit vom **TeamInterfaceImplementer** implementiert werden. Er ist darüberhinaus dafür zuständig, jeder Team-Klasse einen eindeutigen Identifier *_OT$*ID zuzuweisen, der in Form eines statischen Feldes abgelegt wird. Dieser Integerwert wird von der Klasse org.objectteams.transformer.util.TeamIdDispenser verwaltet und dient in den Basisklassen dazu, auf effiziente Weise alle aktivierten Teams abarbeiten zu können (siehe chaining-wrapper im BaseMethodTransformer 3.4, S.45). Außerdem wird die Methode *public int _ OT $ getID ()* implementiert, die es erlaubt, diese Id zu lesen.

Für die Initialisierung neu hinzugefügter statischer Felder muss allgemein beachtet werden, ob die Klasse zuvor bereits statisch initialisierte Felder hatte. Die statische Initialisierungsmethode *static void ⟨clinit⟩()* existiert dann schon und die zusätzlichen Initialisierungen können ihr hinzugefügt werden. Andernfalls muss die ⟨*clinit*⟩-Methode zunächst erzeugt werden.

3.3.2 Basisklassen werden *team*fähig

Basisklassen müssen ”wissen”, welche Teams für sie aktiviert sind, um beim Aufruf einer durch ein callin betroffenen Methode die entsprechenden Rollenmethoden all dieser Teams aufrufen zu können. Der **StaticSliceBaseTransformer** übernimmt die Funktion den Basisklassen, für die mindestens ein callin-Binding existiert, die dafür nötigen Felder und Methoden hinzuzufügen.

Er erzeugt zum Speichern der aktiven Teams, sowie deren Id’s die statischen Felder *protected static Team[] _OT$activeTeams* und *protected static int[] _OT$activeTeamIDs*. Sie werden zunächst als Arrays der Länge 0 initialisiert. Diese Felder, sowie die auf ihnen operierenden Methoden müssen statisch sein, da Teams für Basis-Klassen und nicht für konkrete Objekte aktiviert werden.

Außerdem werden die Methoden *public static void _OT$addTeam(Team team, int teamID)* sowie *public static void _OT$removeTeam(Team team)* generiert, die dazu dienen, der Basisklasse ein für sie aktives Team hinzuzufügen bzw. zu entfernen. Dazu fügen diese Methoden das entsprechende Team und seine Id den oben erwähnten Arrays hinzu, bzw. entfernen diese aus ihnen. Diese Methoden werden von *activate()* und *deactivate()* aus den Teamklassen aufgerufen, um sich bei einer Basisklasse an- und abzumelden. Die Verwendung der dabei abgelegten Informationen wird im folgenden Abschnitt erläutert.

An diesem Punkt wäre wahrscheinlich folgende Optimierung sinnvoll: Falls eine Basisklasse von einer anderen Basisklasse erbt und die Oberklasse bereits vom StaticSliceBaseTransformer transformiert wurde, werden zur Zeit in der Unterklasse trotzdem erneut die oben beschriebenen Felder und Methoden angelegt. Diese Reimplementierung ist natürlich überflüssig und verursacht unnötigen Mehraufwand. Hier könnte man eine Überprüfung einbauen, die dafür sorgt, dass eine Basisklasse nur dann in dieser Form erweitert wird, wenn es tatsächlich nötig ist.

3.3.3 Aktivierung mehrerer Teams - Aufrufreihenfolge für die Rollenmethoden

Die beiden Arrays *_OT$*teams und *_OT$*teamIDs fungieren als eine Art Listen-Datenstruktur. Der *_OT$*index, welcher bei den Methoden, die auf dieser Liste operieren stets mitgeführt wird übernimmt die Funktion eines Iterators. Diese ”Simmulation” wird aus Effizienzgründen einer ”echten” Liste mit Iterator vorgezogen.

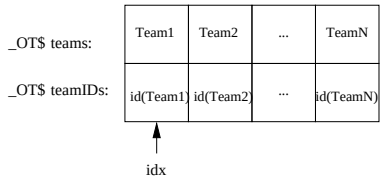

Abbildung 3.4: Datenstruktur für aktive Teams: Liste mit Iterator

Ein mit `addTeam` neu zu dieser Liste hinzugefügtes Team wird jeweils ganz vorne angehangen. Die Folge davon ist eine ”Bevorzugung” von Teams, die später aktiviert wurden. Beim callin wird dabei die dazugehörige Rollenmethode des zuletzt aktivierten Teams bei *before* als erstes und bei *after* als letztes ausgefürt. Sie hat somit das erste und das letzte Wort. Auf die Realisierung hiervon wird in 3.4.1 (S. 47) eingegangen.

Reihenfolgeveränderung durch ”Vordrängeln” eines Teams

Wenn ein Team während der Ausführung eines callins in einer `callin`-Methode aktiviert wird, drängelt es sich in der Kette der auszuführenden Rollenmethoden nach vorn.

Im Sourcecode wird dies folgendermaßen ausgedrückt:

```
callin void entryHook (final Object o) {
in (Team t) do {
    base.entryHook(o);
}
```

Der Compiler ersetzt den in .. do -Block nach

```
t.activate ();
base.entryHook(o);
t.deactivate ();
```

Damit dieser Aufruf wirklich sofort passiert, wird zur Laufzeit die Liste der aktiven Teams manipuliert. Der Iterator wird um eins zurückgesetzt[2] und das betreffende Team wird an diese Stelle gesetzt. Es ist also als nächstes an der Reihe. Dafür wird der einfache activate()-Aufruf durch einen Aufruf von activate (Team[] teams, int[] teamIDs, int idx) ersetzt, der die Listen-Manipulation durchführt.

[2] Der Index kann dabei nicht kleiner als 0 werden, da die erweiterte Rollenmethode mit einem um eins erhöhten Index aufgerufen wird, also immer größer als 0 ist.

3.4 Dynamische Transformationen

Der **BaseMethodTransformer** und der **BaseCallRedirector** realisieren gemeinsam die Veränderungen, die nötig sind, damit callins und *base-calls* (eingeführt in 1.2.2) wirksam werden.

Das Sequenzdiagramm in Abbildung 3.5 zeigt ein Beispiel für den Kontrollfluss des Aufrufs einer durch callin-Bindings adaptierten Methode *m*. Die Rollen *Role1* (aus *Team1*) und *Role2* (aus *Team2*) sind an eine Basisklasse *Base* gebunden. Die Rollenmethoden *rm1* (replace-Binding), *rm2* (before-Binding) und *rm3* (after-Binding) sind allesamt an die Basismethode *m* gebunden. Das UFA-Diagramm zu diesem Szenario ist in Abbildung 3.6 zu finden. Wurde zuerst *Team1* und dann *Team2* aktiviert, bewirkt der Aufruf der Methode *m* den in Abb. 3.5 dargestellten Kontrollfluss, der im Laufe des nächsten Abschnitts näher erläutert wird.

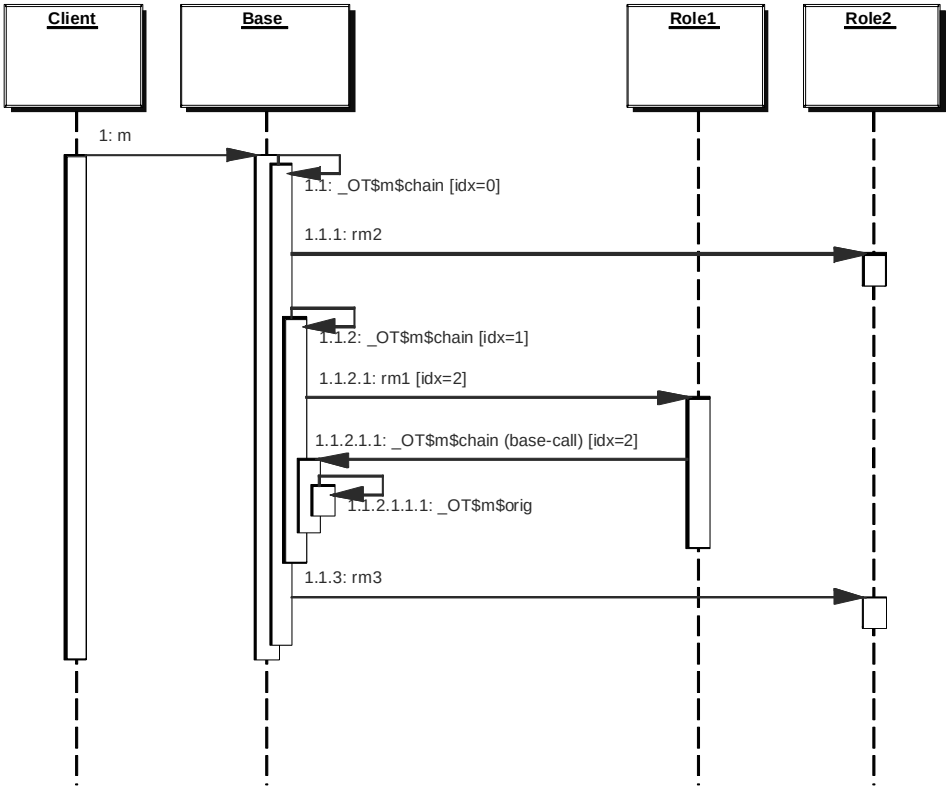

Abbildung 3.5: Ablauf eines callins

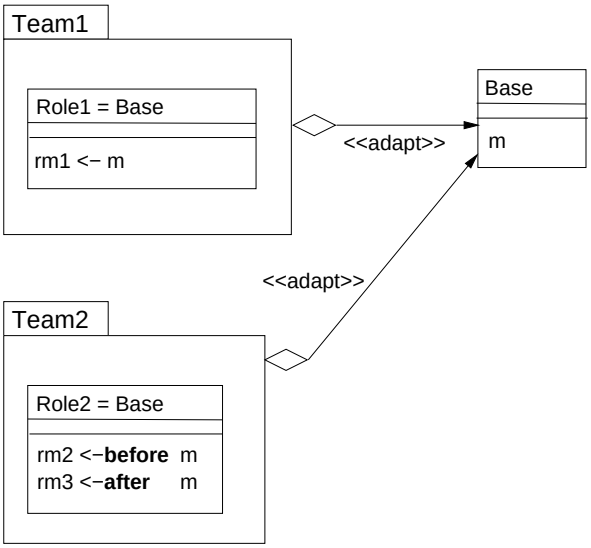

Abbildung 3.6: UFA-Diagramm

3.4.1 callins: Basismethoden zeigen ein verändertes Verhalten

Der BaseMethodTransformer verändert Basisklassen, für die mindestens ein callin existiert. Genauer gesagt, wird jede Basismethode transformiert, für die es ein callin-Binding an eine Rollenmethode gibt. Hierbei kann es sich auch um mehrere Rollenmethoden aus jeweils unterschiedlichen Teams handeln. Die Rollenmethoden werden dann in umgekehrter Reihenfolge der Aktivierung ihrer Teams aufgerufen.

Im weiteren Verlauf werden formale Darstellungen des erzeugten Codes vorgestellt, in welchen sowohl von den Signaturen der beteiligten Methoden, als auch von den deklarierten callin-Bindings abstrahiert wird. Die Listings stellen also allgemeine Muster für das Aussehen des tatsächlich erzeugten Codes dar. Vom Rückgabe-Typ der Methoden wird als *RType* abstrahiert. Falls eine Methode keinen Rückgabewert besitzt, also RType=void ist, so fallen natürlich die Deklaration des `results` sowie Zuweisung daran und Rückgabe weg.

Für die Realisierung von callins wird zunächst jede betroffene Basismethode kopiert. Sei *RType m(a1, ..., aN)* die Signatur der ursprünglichen Methode, so wird eine Methode mit gleicher Signatur und verändertem Namen erzeugt, die eine Kopie der ursprünglichen Methode *m* darstellt:

```
RType _OT$m$orig(a1  , ...,   aN) {
    //Code von m
}
```

Die Originalmethode wird aufgerufen, wenn keine Teams aktiv sind, sowie bei *after*- und *before*-Callins und bei einem *base-call*.

Die Mehode mit dem Originalnamen wird nun überschrieben. Sie dient als *initialer Wrapper*, der von Clients aufgerufen wird, die nicht das geringste von den Transformationen des ot-Laufzeitsystems wissen. Er hat die Form:

```
RType m(AType1 a1 , ..., ATypeN aN) {
    Team[] _OT$teams = (Team[])_OT$activeTeams.clone();
    int [] _OT$ teamIDs = (int [])_OT$ activeTeamIDs.clone();
    return _OT$m$chain(_OT$teams, _OT$teamIDs , 0, a1 , ..., aN);
}
```

Von hier aus wird der sogenannte *chaining-wrapper* aufgerufen[3], der die eigentlichen Aufrufe in die Rollenklassen realisiert. Er hat die Signatur:

```
RType _OT$m$chain(Team[] _OT$teams, int [] _OT$ teamIDs, int _OT$idx , AType1 a1 , ..., ATypeN aN)
```

Bei den ersten beiden Argumenten handelt es sich um Arrays, die die aktiven Teams, sowie deren Id's enthalten. *idx* ist der Index des aktuell zu behandelnden Teams. Er spielt, wie anschließend erklärt, eine Rolle bei der rekursiven Abarbeitung der aktiven Teams. Der intitiale Wrapper ruft diese Methode mit geklonten Versionen[4], der oben erwähnten (vom StaticSliceBaseTransformer hinzugefügten) Arrays, sowie dem Index 0 und den ursprünglich an ihn übergebenen Argumenten auf.

Seien BCONNECTORS = {Connector1, ..., ConnectorP} Konnektoren mit Bindings für die zu transformierende Methode *m* der Basisklasse *B* und
BEFORE = {c ∈ BCONNECTORS | c definiert before-Binding für B}
REPLACE = {c ∈ BCONNECTORS | c definiert replace-Binding für B}
AFTER = {c ∈ BCONNECTORS | c definiert after-Binding für B}
sowie *R* eine Rolle von B und *rm* eine an m gebundene Rollenmethode.

Allgemein hat der chaining-wrapper dann die auf der folgenden Seite dargestellte Form.

[3] Dabei werden u.a. auch die Argumente, die an den initialen Wrapper übergeben wurden, an den chainig-wapper durchgereicht. Das Laden der Argumente geschieht ananlog zu dem BCEL-Beispiel in 2.1.2.

[4] Auf diese Weise wird die Liste der aktiven Teams im Moment des Methoden-Aufrufs "eingefroren". Aktivierungen und Deaktivierungen, die anschließend erfolgen, haben auf die aktuelle Ausführung also keinen Einfluss mehr.

```
1  RType _OT$m$chain(Team[] _OT$teams, int [] _OT$ teamIDs, int _OT$idx , AType1 a1 , ..., ATypeN aN) {
2      RType _OT$result = null;
3      Team _OT$team;
4      int _OT$teamID;
5      if (_OT$ idx >= _OT$teams.length ) {
6          _OT$ result = _OT$m$orig(a1 , ..., aN);
7          return _OT$ result ;
8      }
9      _OT$team = _OT$teams[_OT$idx ];
10     _OT$teamID = _OT$teamIDs[_OT$idx ];
11
12     // if BEFORE ≠ ∅:
13
14     Before-Code
15
16     // if REPLACE ≠ ∅:
17
18     Replace-Code
19
20     // if REPLACE = ∅ → put recursive call here:
21         _OT$ result = _OT$m$chain(_OT$teams, _OT$teamIDs, _OT$idx+1 , a1 , ..., aN);
22
23     // if AFTER ≠ ∅:
24
25     After-Code
26
27     return _OT$ result ;
28 }
```

Zur besseren Übersichtlichkeit und weil der Code für den after- und den before-Teil praktisch identisch ist, wurden diese Codestücke oben ausgelassen und folgen erst jetzt:

: Code für die Ausführung von **before**- bzw. **after**-Callins

```
switch(_OT$teamID) {
// ∀C ∈ BEFORE bzw. ∀C ∈ AFTER:
    case C._OT$ID:
    {// [C:R↔B, rm↔m]
        C.R._OT$ role ;
        _OT$role = ((C)_OT$team)._OT$liftToR(this);
        _OT$role.rm(a1 , ..., aN); // lifted args if necessary
        break;
    }
}
```

: Code für die Ausführung von **replace**-callins

```
switch(_OT$teamID) {
// ∀C ∈ REPLACE:
    case C._OT$ID:
    {// [C:R↔B, rm↔m]
        C.R. _OT$ role ;
        _OT$role = (( C)_OT$team)._OT$liftToR ( this ) ;
        //————————- if rm has a Role-Returntype lowering is necessary:
        RTypeR _OT$resultR ;
        _OT$resultR = _OT$ role .rm(_OT$teams, _OT$teamIDs, _OT$idx+1 , a1 , ..., aN);
        // lifted args if necessary
        _OT$ result = _OT$ resultR .base;
        //————————- else:
        _OT$ result = _OT$ role .rm(_OT$teams, _OT$teamIDs, _OT$idx+1 , a1 , ..., aN);
        // lifted args if necessary
        //————————-
        break;
    }
    default :
    { // recursive call:
        _OT$ result = _OT$m$chain(_OT$teams, _OT$teamIDs, _OT$idx+1 , a1 , ..., aN);
    }
}
```

Wenn der *Iterator* das Ende der Liste erreicht hat, wird aus der Rekursion gesprungen und die Original-Methode wird aufgerufen (Zeilen 5-8). Ansonsten wird der before-Fall aufgerufen, der rekursive Aufruf durchgeführt und anschließend der after-Fall ausgeführt.

replace-callins verändern dieses Verhalten wie folgt: Alle Teamaktivierungen, die vor dem zuletzt aktivierten Team mit replace-callin für die aktuelle Methode stattgefunden haben werden sozusagen ungültig. Das letzte replace-callin ersetzt also alles vorher da gewesene (incl. des Aufrufs der Original-Methode). before- und after-callins aus später aktivierten Teams werden normal ausgeführt. Wird in der Rollenmethode ein base-call getätigt, so wird die Ausführungskette jedoch fortgesetzt.

Der Rückgabewert des chaining-wrappers und damit auch des Aufrufs der Basismethode entspricht dem Rückgabewert der Original-Methode. Existieren jedoch aktive replace-callins, so wird er ersetzt durch den Rückgabewert der zuletzt aktivierten replace-Methode.

Der rekursive Aufruf führt den Algorithmus bis ans letzte Element der Liste, welches dem zuerst aktivierten Team entspricht. Danach werden die after-callins mit diesem beginnend in umgekehrter Listen-Reihenfolge ausgeführt. Die zuständige Rollenmethode des zuletzt aktivierten Teams wird also ganz am Ende aufgerufen.

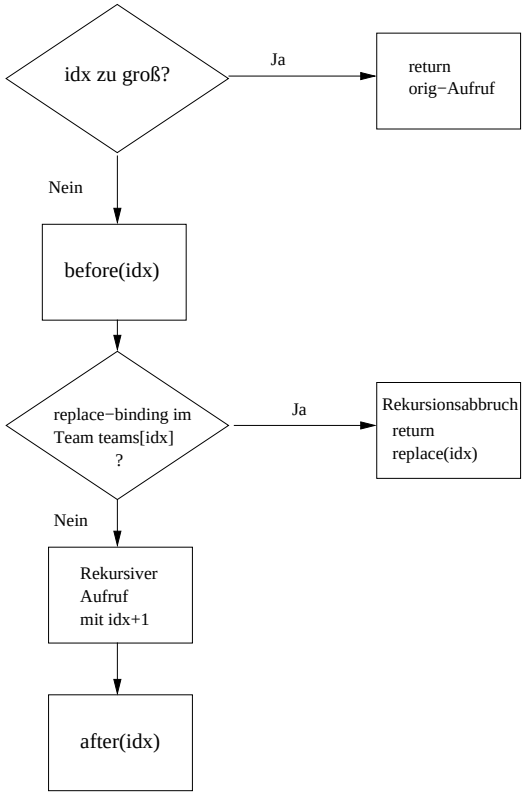

Abbildung 3.7: Kontrollfluss des chaining-wrappers

Wie in Abbildung 3.7 erkennbar gibt es zwei Möglichkeiten für einen Rekursions-Abbruch:

1. Der Iterator hat das Ende der Liste erreicht, das heißt die Rollenmethoden mit callins (after oder before) für sämtliche aktive Teams wurden ausgefürt. Nun wird lediglich noch die Original-Methode ausgeführt.

2. Ein replace-callin bewirkt einen vorzeitigen Abbruch der Rekursion. Durch einen base-call wird die Rekursion jedoch wieder aufgenommen und fortgesetzt.

Auch wenn kein Team aktiv ist, also auch kein callin durchgeführt werden muss, geht der Kontrollfluss über den initialen Wrapper zum chaining-wrapper und von da aus zur Original-Basismethode. An dieser Stelle könnte man aus Effizienzgründen im initialen Wrapper überprüfen, ob die Liste der aktiven Teams leer ist. So könnte man diesen Fall abfangen und die Originalmethode direkt aus dem initialen Wrapper heraus aufrufen.

3.4.2 Realisierung des base-calls

In einer *callin*-Rollenmethode besteht wie in 1.2.2 (S. 8) beschrieben die Möglichkeit mit Hilfe eines *base-calls* die dazugehörige Basismethode aufzurufen. Der ot-Compiler ersetzt `base`-Aufrufe in callin-Rollenmethoden durch rekursive Aufrufe derselben. Letztendlich soll an diesen Stellen die dazugehörige Basismethode aufgerufen werden. Damit jedoch die Rekursion des chaining-wrappers fortgesetzt werden kann, muss genaugenommen dieser aufgerufen werden. Da der chaining-wrapper zur Compilezeit jedoch noch nicht existiert, wird der korrekte Aufruf erst zur Laufzeit erzeugt.

Beim Aufruf von *callin*-Rollenmethoden aus dem chaining-wrapper müssen auch die zusätzlichen Team-Parameter an die Rollenmethode übergeben werden. Die Signatur der Methode muss deshalb erweitert werden. Der BaseCallRedirector transformiert zu diesem Zweck alle callin-Methoden. Da es (wie in 2.2.1, S. 29 motiviert) nicht erlaubt ist, Methoden aus dem classfile zu löschen wird hier genaugenommen zusätzlich eine erweiterte Version erzeugt.
Diese hat die drei bekannten Argumente zur Abarbeitung von callins am Anfang ihrer Argumentliste. Im kopierten Code der ursprünglichen Methode wird jeder rekursive Aufruf durch den Aufruf der entsprechenden Basismethode (bzw. deren chaining-wrappers) ersetzt. Durch die Einführung der drei neuen Argumente wird es (wie in 2.1.2 erläutert) zusätzlich nötig, die Variabeln-Indizes aller Lade- und Speicherinstruktionen anzupassen (genauer: um drei zu erhöhen).

Für eine Rollenmethode

```
callin RType rm(AType1 a1 , ..., ATypeN aN) {
    ...
    rm(a1' , ..., aN' ); //im Sourcecode: base.rm(a1', ..., aN');
    ...
}
```

wird folgende Erweiterung generiert:

```
RType' rm(Team[] _OT$teams, int [] _OT$ teamIDs, int _OT$idx , AType1 a1 , ..., ATypeN aN) {
    ...
    _OT$base._OT$m$chain(_OT$teams, _OT$teamIDs, _OT$idx , a1 , ..., aN);
    // lifted result, if necessary
    ...
}
```

Dabei ist m die an rm gebundene Basismethode. Beim Aufruf der Basismethode kann es nötig sein, dass Argumente gecastet werden, falls Rollenargumente von einem Supertyp sind.

Das Sequenzdiagramm (Abb. 3.5) verdeutlicht den verallgemeinert dargestellten Ablauf an einem Beispiel. Der Client ruft die Methode *m* der Klasse *Base* auf, wie er es auch getan hätte, wenn *Base* nicht durch Teams adaptiert worden wäre. Der Unterschied ist für ihn nicht erkennbar. Tatsächlich ist die aufgerufene Methode jedoch der *initiale* Wrapper, der den *chaining-wrapper* mit Index `idx`=0 aufruft. Der Wert dieses Indexs während der nun folgenden Aufruf-Kette ist im Sequenzdiagramm mit angegeben. Da *Team2* als letztes aktiviert wurde, werden seine before- und after-callins als erstes, bzw. als letztes von dem äußersten Aufruf des chaining-wrappers durchgeführt. Dazwischen erfolgt der rekursive Aufruf, der für callins von *Team1* (idx=1) zuständig ist. Hier existiert nur ein replace-callin zur Methode *rm1*. Der Aufruf 1.1.2.1 geht an die *erweiterte* Rollenmethode und erfolgt ebenfalls mit einem um eins erhöhten Index (idx=2). Als nächstes folgt der *base-call*, der dafür sorgt, dass die Rekursion fortgesetzt wird. Da nun jedoch der Iterator das Ende der Liste erreicht hat ($[idx = 2] >= [teams.length = 2]$), wird die Original-Methode _OT$m$orig aufgerufen.

3.4.3 Realisierung von Signatur-Anpassungen

Der Rückgabetyp der *erweiterten* Rollenmethode *RType'* unterscheidet sich vom Rückgabetyp der ursprünglichen Rollenmethode *RType*, falls das in folgendem Abschnitt beschriebene Szenario eintritt. Es ist nur für replace-callins relevant, da bei after- und before-callins die Rollenmethoden-Resultate in der Basis nicht weiter verwendet werden.

unused result-Szenario

Sei *rm* eine Rollenmethode mit Rückgabetyp void. Angenommen die an rm gebundene Basismethode hat einen Rückgabewert. Bei einem base-call muss die Rollenmethode jedoch in der Lage sein, dessen Ergebnis an die aufrufende Basismethode zurückzugeben.

Deshalb wird in einem solchen Fall der Rückgabetyp auf *Object* geändert um das Resultat "anonym" durchreichen zu können. Handelt es sich bei dem Rückgabetyp der Basismethode um einen Grundtyp (wie int, char usw.) so muss das Ergebnis zunächst in ein Objekt verpackt werden.

Bei diesem *boxing* wird für ein int zum Beispiel ein umschließendes Integer Objekt erzeugt und zurückgegeben. Objekttypen werden unverändert resultiert. Beim Aufrufer, der Basismethode muss das Ergebnis wieder ausgepackt werden. Für Grundtypen wird per *unboxing* wieder auf den eigentlichen Rückgabetyp zugegriffen. Objekttypen müssen lediglich auf ihren Originaltyp zurück gecastet werden. ("boxing" und "unboxing" bedeuten (automatisches) Ein- und Auspacken eines primitiven Type in ein Wrapper-Objekt.)

unused arguments-Szenario

Ähnlich wird mit überzähligen, in der Rollenmethode nicht benötigten Argumenten verfahren. Die erweiterte Rollenmethode erhält dafür einen zusätzlichen Parameter vom Typ Object[]. In diesem Array werden alle nicht in der Rollensignatur enthaltenen Argumente übergeben. Grundtypen werden, genau wie beim unused result in Wrapper-Objekte vergepackt. Die Rollenmethode braucht auch diese zusätzlichen Argumente, die nicht in ihrer Signatur auftauchen, wenn sie einen base-call durchführt. Dafür wird das `unusedArgs`-Array wieder ausgepackt und die Basismethode bekommt auch diese Argumente übergeben.

Für callin-Rollenmethoden (replace) werden erweiterte Rollenmethoden generiert. Außer der Signaturerweiterung (Team[], int[], int) gilt:

- das letzte Argument ist ein 'Object[]', welches eventuelle Mehrargumente der Basismethode enthält
- der Rückgabetyp ist 'Object' (und enthält das Resultat des base-calls), falls die Basismethode einen Rückgabewert hat, die Rollenmethode jedoch nicht.

Bei after- und before-callins können die Rollenmethoden nicht in die Basis "zurückrufen". Es ist in diesem Fall also nicht nötig, dass in der Rolle alle Basismethoden-Parameter bekannt sind.

3.4.4 Revidierte Design Entscheidungen

Privatsphäre der lift-Methoden wiederhergestellt!

Zunächst war geplant die lift-Methoden der Teams public zu deklarieren. Basis-Objekte konnten so von überall zu ihren Rollen geliftet werden. Dies geschah unter anderem in den chaining-wrappern der Basisklassen. Sollte beim callin eine Rollenmethode aufgerufen werden, so wurde die Basis zur Rolle geliftet, an welcher dann die gewünschte Methode aufgerufen werden konnte. Weiterhin wurden die lift-Methoden auch verwendet, um für diese Methodenaufrufe Basis-Typ-Argumente zum entsprechenden Rollen-Typ zu liften. Praktisch!

Die Tatsache, dass die lift-Methoden public waren stellte sich jedoch als problematisch heraus, wie folgendes Szenario deutlich macht. Seien *TeamB* ein Subteam (also eine Unterklasse) von *TeamA* und *Role* eine zu beiden Teams gehörende Rolle. In *TeamA* generiert der Compiler für die Rolle *Role* eine lift-Methode mit dem Rückgabetyp *TeamA.Role*. Im abgeleiteten *TeamB* muss es nun eine entsprechende lift-Methode geben, die jedoch *TeamB.Role* zurückliefert. Da *Role* eine InnerClass der Teams ist, besteht jedoch keine Subclass-Beziehung zwischen diesen beiden Rückgabetypen. Da Java es verbietet eine public-Methode mit nicht

polymorph-passendem Rückgabetyp zu redefinieren, wurde beschlossen die lift-Methoden privat zu machen. Sie können nun durch *shadowing* redefiniert, d.h. in jedem erbenden Team mit neuer Signatur implementiert werden.

Um den Basisklassen weiterhin callins zu ermöglichen, werden nun für alle callin-gebundenen Rollenmethoden Wrapper-Methoden generiert. Sie bekommen als zusätzlichen Parameter das aufrufende Basisobjekt und sind sowohl für dessen lifting, als auch für eventuell notwendiges Parameterlifting verantwortlich. Außerdem führen sie, falls nötig vor Ergebnisrückgabe ein lowering durch. Weiterhin sind diese Wrapper-Methoden auch für die Realisierung der Parameter-Mappings zuständig.

Zum Beispiel wird für

```
void ci (int i, RoleY y) <- void bm (Base b, Integer zahl) with
    { i <- zahl.intValue (), y <- b }
```

folgender Wrapper erzeugt:

```
void _OT$RoleX$ci (BaseX bx, Base b, Integer in) {
        _OT$liftToRoleX(bx).ci(in.intValue (), _OT$ liftToRoleY(b));
}
```

Allgemein generiert der Compiler für callin-gebundene Rollenmethode

```
RType rm(ArgType1 a1 , ..., ArgTypeN) {...}
```

einer Rollen-Klasse *Role* mit dazugehöriger Basis-Klasse *Base* im umschließenden Team den Wrapper:

```
public void _OT$Role$rm(Base base, AType1 a1 , ..., ATypeN aN) {
    _OT$ liftToRole (base).rm(a1 , ..., aN); // lifted args if necessary
}
```

Für die Signatur dieses Wrappers gilt:

- RollenTypen der Rollenmethode sind durch ihren Basistyp ersetzt
- Der Rückgabetyp des Wrappers entspricht dem der Rollenmethode, es sei denn es handelt sich um einen Rollentyp. Dann wird das Ergebnis der Rollenmethode automatisch gelowert und der Wrapper gibt den Basistyp zurück.
- gibt es Parametermappings, so entspricht die Reihenfolge der der Basis-Signatur; überzählige Basis-Argumente werden abgeschnitten.

Für callin-Rollenmethoden ist dies jedoch nicht ausreichend, da für sie wie in 3.4.2 beschrieben, neue Versionen mit erweiterten Signaturen generiert werden. Das bedeutet, dass für sie zusätzliche Wrapper mit erweiterter Signatur angefertigt werden müssen. Dies kann jedoch noch nicht zur Compilezeit geschehen, da

dort die aufzurufenen erweiterten Rollenmethoden noch gar nicht existieren. Deswegen wird diese Aufgabe von einem weiteren Transformer, dem **CallinWrapperEnhancer** übernommen. Er generiert für alle Wrapper von replace-gebundenen Rollenmethoden eine erweiterte Version:

```
public RType _OT$Role$rm(Base base, Team[] _OT$teams, int [] _OT$ teamIDs, int _OT$idx,
                                    AType1 a1 , ..., ATypeN aN){
    return _OT$ liftToRole (base).rm(_OT$teams, _OT$teamIDs,_OT$idx , a1 , ..., aN);
    // lifted args if necessary
}
```

Dabei entspricht RType dem Rückgabetyp der erweiterten Rollenmethode.

Für die Signatur des erweiterten Wrappers gilt folgendes:

- das erste Argument ist vom Typ der aufrufenden Basis, die sich darin selbst übergibt
- die drei folgenden Argumente sind die altbekannten Listen-Parameter für die aktivierten Teams (Team[], int[], int)
- anschließend folgen die anderen Argumente
- als letztes Argument wird ein 'Object[]' angehangen, welches eventuelle Mehrargumente der Basismethode enthält
- hat die Rollenmethode den Rückgabetyp `void`, die Basismethode hat jedoch ein Resultat, so ist der Rückgabetyp 'Object'; sonst bleibt er wie im einfachen Wrapper

Da lifting, lowering und Parameter-Mapping im erweiterten Wrapper genauso benötigt werden, wird dieses aus dem einfachen Wrapper übernommen. Zu diesem Zweck wird der Code aus dem einfachen Wrapper wiederverwendet. Er muss allerdings für den erweiterten Wrapper angepasst werden.

Im Code des einfachen Wrappers wird der Aufruf der Rollenmethode durch den Aufruf der erweiterten Rollenmethode ersetezt. Dabei

- müssen die Indizes der lokalen Variablen für alle Lade- und Speicher-Operationen um drei erhöht werden; ausgenommen davon sind die mit den Indizes 0 (this) und 1 (der erste Parameter bleibt vorn)
- müssen die zusätzlichen Parameter durchgereicht werden
- muß darauf geachtet werden, dass die Instruktionen, welche lifting, lowering und Parameter-Mapping realisieren korrekt erhalten bleiben.

Das lowern des Ergebnisses geschieht nach dem Rollenmethoden-Aufruf und zwar nur in dem Fall, wo die Rollenmethode einen (Rollen-)Rückgabewert hat. Der Rückgabewert des erweiterten Wrappers unterscheidet sich in dem Fall nicht von diesem. Eventuelle lowering-Instruktionen können also bleiben, wo sie waren.

Lifting und Parameter-Mappings passieren während des Ladens der Argumente für den Rollenmethoden-Aufruf. Damit dieser Block von Instruktionen unverändert bleibt, ist es nötig seinen Anfang und sein Ende zu bestimmen. Um die Stelle zu finden, an welcher mit dem Laden der Parameter angefangen wird folgende Annahme gemacht: Das erste Argument (die Basis) im Wrapper-Code wird das erste Mal geladen, um für den Aufruf der Rollenmethode zur Rolle geliftet zu werden. Die Stelle nach dem anschließenden lift-Aufruf markiert den Anfang des gesuchten Blocks. Hier muss nun das Laden der drei neu hinzugefügten Listen-Parameter eingeschoben werden.

Das Ende des Blockes wird anhand der Anzahl der ursprünglich geladenen Argumente und deren Größe berechnet. Hier muss zusätzlich das Object-Array der überzähligen Basis-Argumente geladen werden. Im Anschluss daran kann dann die folgende Methodenaufruf-Instruktion durch einen Aufruf der erweiterten Rollenmethode ersetzt werden.

Falls ein *unused result Szenario* vorliegt, muss nun noch die Rückgabe-Instruktion an den neuen Rückgabetyp (Object) angepasst werden.

Als Folge dieser Änderungen entfällt im BaseMethodTransformer das Erzeugen von lift-Aufrufen sowie das lowering von Rückgabewerten. Beim Aufruf der neuen Wrapper ist nun darauf zu achten, dass (abgesehen von den zusätzlichen Argumenten base, teams, teamIDs und idx):

- die übergebenen Argumente in ihrer Anzahl denen der Rollenmethode entsprechen
- wo die Rollenmethode ein Rollen-Argument erwartet, das ungeliftete Basis-Objekt übergeben wird
- falls die Rollenmethode es so erwartet, Argumente von einem Supertyp geliefert werden

Probleme mit der Privatisierung der lift-Methoden

Private Methoden einer Klasse können nur aus der Klasse selbst heraus aufgerufen werden. Auf Bytecode-Ebene[5] ist selbst einer InnerClass die Benutzung solcher Methoden untersagt. Das bedeutet, dass auch die Methoden der Rollenklassen die privaten lift-Methoden nicht mehr aufrufen können. Rollenmethoden, die base-calls durchführen müssen jedoch gegebenenfalls dessen Ergebniss zum entsprechenden Rollentyp liften können. Dies ist zumindest dann nötig, wenn die Rollenmethode das base-call-Ergebnis nicht direkt zurückgibt, sondern zunächst selbst noch Berechnungen damit anstellt.

Aus diesem Grund hat sich der Ansatz die lift-Methoden privat zu machen als unpraktikabel erwiesen. Stattdessen werden nun statische lift-Methoden in den Teams eingeführt. Sie haben die Signatur

```
static TeamX.Role _OT$liftToRole(TeamX t, Base base)
```

In jedem Subteam kann diese Methode nun neu implementiert werden, wobei TeamX dem Typ des neuen Teams entspricht. Da die Signatur der Subteam-Methode verändert ist, handelt es sich lediglich um overloading.

Abbildung 3.8 zeigt eine Zusammenfassung, der entwickelten Transformer, sowie die von ihnen transformierten Programmteile.

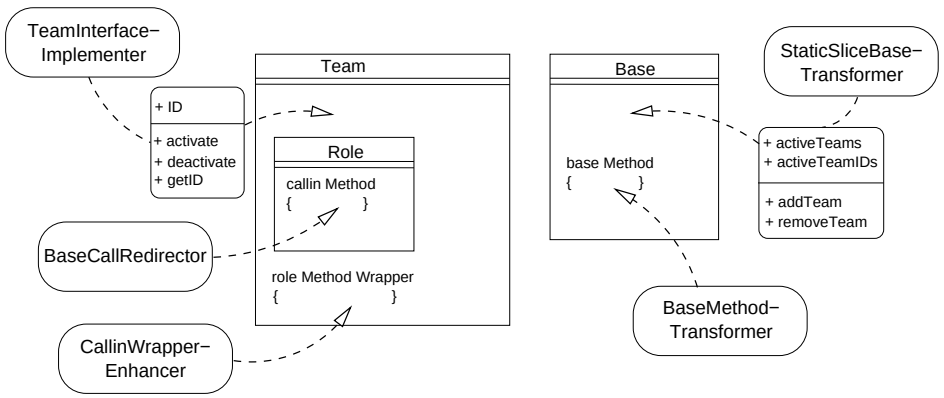

Abbildung 3.8: Einflussbereiche der Transformer

[5] Aus dem Sourcecode heraus ist dies möglich, da der Java Compiler intern statische Zugriffsmethoden generiert und die Aufrufe privater Outerclass-Methoden an diese umlenkt.

Kapitel 4

Zusammenfassung und Ausblick

Die Programmiersprache ot-java ist lauffähig und bereit eingesetzt zu werden. Wenngleich noch nicht alle Object-Teams-Features umgesetzt wurden, ist ein Großteil schon funktionstüchtig. Im folgenden soll die Softwareentwicklung mit Object Teams unter Verwendung von ot-java beleuchtet und auf die besonderen Umstände, die sich daraus ergeben hingewiesen werden. Ausserdem wird auf den aktuellen Stand der Entwicklung und einige Punkte, an denen noch Entwicklungsbedarf besteht, eingegangen.

4.1 Softwareentwicklung mit ot-java

Die eingangs vorgestellten Sprachfeatures des Programmiermodells *Object Teams* konnten größtenteils schon umgesetzt werden und mit der jetzigen Version von ot-java können bereits Anwendungen gebaut werden, die viele der Vorteile von Object Teams nutzen können.

Für das Design komplett neuer Anwendungen, wie auch einzelner Teams bietet es sich prinzipiel an, wie bei der Entwicklung ”normaler” objektorientierter Systeme, die Darstellungsmittel der UML zu verwenden.

Für Klassendiagramme, die Binding-Deklarationen und Adaptions-Relationen enthalten, kann die (in Kapitel 1, S.9 sowie in Kapitel 3, S.44 benutzte) UML-Erweiterung *UFA* [7] verwendet werden. Dabei werden Teams auf Packages abgebildet, die zusätzlich die Eigenschaft haben, dass sie instanziierbar sind und Felder und Methoden haben können (sie stellen also *first class citizens* dar). Die Adaption eines Basis-Packages durch ein Team, wird durch die neu eingeführte $<< adapt >>$-Assoziation dargestellt. In Teams, die als *Konnektoren* (siehe 1.2.2, S.9) fungieren, können Bindungen von *Rollen* an *Basisklassen* sowie Methoden-Bindings deklariert werden.

Während der Anforderungsanalyse werden oft geforderte Programm-Eigenschaften als einzelne Punkte herausgestellt, die in späteren Designphasen jedoch nicht mehr gekapselt sind, sondern über verschiedene Klassen des Systems verstreut (*scattering*, siehe 1.1.1) realisiert werden. Aus solchen Anforderungen[1] lassen sich möglicherweise Hinweise auf Aspekte, die in Teams gekapselt werden könnten, erhalten.

Einzelne Aspekte einer Anwendung können durch eigenständige Teams implementiert werden. Dies kann relativ unabhängig und damit auch von verschiedenen Entwicklern, die theoretisch nicht einmal von der gegenseitigen Existenz wissen müssen, getan werden.

4.1.1 Design der Basis-Anwendung

Wird eine Anwendung von Grund auf mit Object Teams entwickelt, so wird auch die Basis-Anwendung zumindest mit der Kenntnis einiger, ihr Funktionalität hinzufügenden, Teams entworfen. Sie kann dann so ”schlank” wie möglich gehalten werden, das heißt sie muss wirklich nur *Kernfunktionalität* enthalten. Alle zusätzlichen *Concerns* können in Teams gekapselt werden.

”Weiß” eine Anwendung davon, dass sie als Basis-Anwendung fungiert, so wird sich diese Tatsache wahrscheinlich auch positiv auf die Adäquatheit ihrer Schnittstelle für diese Teams auswirken. Aber auch dann muss natürlich davon ausgegangen werden, dass möglicherweise im Nachhinein unvorhergesehene weitere Adaptionen durch ”unbekannte” Teams hinzukommen. Letztendlich muss die Basis-Anwendung also als weitgehend unabhängige Einheit betrachtet werden.

Trotzdem können bei der Entwicklung einer Anwendung einige allgemeine Richtlinien beachtet werden, die sie zu einer ”besseren” potentialen Basis machen. Die Adaption eines (Basis-)Packages durch ein Team geht letzendlich auf Methoden-Ebene von statten. Deswegen hat speziell das Design der Methoden entscheidenden Einfluss auf den Grad der Adaptierbarkeit. Wenn Methoden nur eine ganz bestimmte Funktionalität kapseln und alles weitere an andere Methoden ”deligieren”, dann gibt es viele ”Angriffspunkte” (joinpoints; siehe 1.1.3, S.5) für Bindungen an Rollenmethoden. Wenn beispielsweise alle Zugriffe auf Felder über Zugriffsmethoden laufen, so hat ein Team die Möglichkeit diese Zugriffe zu überwachen oder zu verändern. Würde direkt auf die Felder zugegriffen, so hätte man von aussen keinen direkten Einfluss darauf, insbesondere nicht auf die Werte, die dabei verwendet werden. ”Kleine” Methoden (und auch Klassen) haben ausserdem den Vorteil, dass die Rolle besser ein ganz bestimmtes Verhalten zur Adaption auswählen kann.

[1] z.B. ”alle Lesezugriffe müssen geloggt werden”

4.1.2 Adaption bestehender Anwendungen

Dadurch, dass das Hineinweben der durch die Rollen spezifizierten Aspekte auf Bytecode-Ebene geschieht, ist es unnötig, dass die Basis-Anwendung in Sourcecode-Form vorliegt. Stattdessen kann sie sogar nur in Form eines jar-Files vorhanden sein. Das bedeutet, dass sich ot-java auch dafür eignet, ”gekaufte” Anwendungen nach speziellen eigenen Wünschen zu adaptieren.

Auch wenn der Sourcecode nicht vorliegt, so ist es natürlich trotzdem nötig, Kenntnis von der Klassenstruktur der Anwendung, sowie der Methoden, an die man Rollenmethoden binden möchte zu haben. Eine Schnittstellenbeschreibung, welche die Klassen, ihre Methoden sowie deren (möglichst präzise) Spezifikation enthällt, wäre jedoch alles, was benötigt wird, um eine Anwendug durch die Funktionalität eines Teams zu adaptieren. Je ausführlicher und präziser eine solche Schnittstellenbeschreibung ist, desto gezielter und ”sauberer” läßt sich die dazugehörige Anwendung durch ein Team adaptieren. Werden für Methoden beispielsweise (formale) Zusicherungen in der Art von *Design by Contract* festgehalten, so hilft dies bei einer erfolgreichen Benutzung im Allgemeinen und einer sinnvollen Adaption im Speziellen. Ein adaptierendes Team könnte dann dafür sorgen, dass die Vorbedingungen für Basismethoden-Aufrufe eingehalten werden und Methoden auch nach einer Adaptierung ihre Nachbedingungen erfüllen.

Sind keine Informationen über die Basis-Schnittstelle zugänglich, so könnte man versuchen durch ein entsprechendes Analyse-Tool die Namen der Klassen und Methoden einer Anwendung herausfinden. Ohne ein relativ gutes Verständnis von dem, was die Methoden tun, scheint es jedoch wenig erfolgversprechend, sinnvolle Verbindungsstellen zwischen dem Team und der Basis-Anwendung zu finden. Ausgenommen hiervon sind eventuell Methoden mit ”sprechenden” Namen, wie ”`getXY`”, von deren Existenz aber leider nicht ausgegangen werden kann. Dies ist ein gutes Beispiel dafür, dass die Einhaltung bestimmter Namens-Konventionen der Softwareentwicklung sehr entgegen kommen würde.

Wäre ein Tool in der Lage, zumindest in begrenztem Ausmaß, auch die Semantik der unbekannten Basismethoden zu analysieren, könnte man die Menge potentieller *Basis*-Packages wahrscheinlich drastisch erhöhen. Es ist schon möglich anhand des Sourcecodes Sequenzdiagramme für bestimmte Methoden-Aufrufe generieren zu lassen (z.B. die IDE *Together* ermöglicht dies). Sequenzdiagramme der Basismethoden wären jedenfalls ein erster Schritt in Richtung des Verständnisses ihrer Funktionalität.

4.1.3 Die Entwicklungsumgebung

Für Softwareentwicklung mit ot-java wird es in der Praxis nötig sein, eine adäquate Entwicklungsumgebung zur Verfügung zu haben. Sie sollte die Benutzung des ot-java-Compilers unterstützen, sowie eine Programmausführung mit dem durch JMangler ”gewrappten” java-Aufruf ermöglichen. Auf diese Weise könnte auch ein Debugger benutzt werden, welcher JMangler verwendet.

Weiterhin sollte eine Form von *Aspekt-Browser* in die Entwicklungsumgebung integriert sein. Steht der Sourcecode der Basis-Anwendung zur Verfügung, so könnte man dann von einer Rollenmethode zu der an sie gebundenen Basismethode navigieren, bzw. in der Basismethode ein Hinweis auf eine Adaption durch eine Rollenmethode finden. Gerade wenn mehrere Teams involviert sind, würde dieses Navigieren entlang der Bindings sicherlich dem Programmverständnis sehr entgegenkommen. Falls der Basis-Code nicht verfügbar ist, könnte man sich immerhin noch vorstellen in einer, in die Entwicklungsumgebung eingebundenen, Schnittstellen-Beschreibung zu browsen.

Eine laufende Diplomarbeit beschäftigt sich mit der Integration von ot-java in die eclipse-Plattform [1]. Zu diesem Zweck soll u.a. ein neuer *inkrementeller* ot-Compiler entwickelt werden. Dabei erweist es sich als vorteilhaft, dass die gewählte Schnittstelle zwischen ot-Compiler und ot-Laufzeitumgebung (siehe 3.1) eine relativ grosse Unabhängigkeit dieser beiden Komponenten gewährleistet. Der neue Compiler könnte seinen Teil der Aufgaben zur Realisierung von Object-Teams auf völlig andere Weise lösen, als der aktuelle Compiler. Solange nur gewährleistet ist, dass die Schnittstelle zur Laufzeitumgebung gleich bleibt, ist die ot-Laufzeitumgebung ohne Änderungen dazu kompatibel. Der Compiler muss dazu also die benötigten Informationen durch Attributen und Modifier kodierern sowie die lift-Methoden und die Rollenmethoden-Wrapper wie gehabt erzeugen.

4.1.4 Programmm-Ausführung

Jeder Benutzer einer durch ein Team adaptierten Anwendung, muss die JMangler-Erweiterung des JVM-Aufrufs benutzen um die Änderungen mitzubekommen. Das heißt auch, es ist nicht möglich jemandem, der eine Anwendung benutzt, die Adaptionen ”unterzuschieben”. Möglicherweise wäre jedoch in bestimmten Fällen genau das beabsichtigt. Man stelle sich eine Web-Anwendung vor, bei der bestimmte Java-Klassen über das Netz auf einen lokalen Rechner übertragen werden, um von der dortigen JVM ausgeführt zu werden. Wird dazu ”gewöhnliches” Java benutzt, so kann die ot-java-Laufzeitumgebung nicht zum Zuge kommen.

Ist eine Anwendungsklasse mit dem ot-java-Compiler übersetzt worden, so ist es zumindest immernoch möglich, sie mit normalem java auszuführen, da die Klas-

sen der Basis-Anwendung erst von der Laufzeitumgebung transformiert[2] werden. Der Benutzer bekäme also das unadaptierte Verhalten der Basis-Anwendung zu sehen. Dies gilt allerdings nur, falls die ”main”-Funktionalität in der Basis liegt.

Eine andere Möglichkeit ist, dass die Basisklassen lediglich eine Art Bibliothek darstellen, deren Funktionalität durch Teams erweitert wird. Gibt es in diesem Fall lediglich callout-Bindings und liegt der Programm-Einstiegspunkt auf Team-Seite, so ist das erwartete Verhalten trotzdem sichergestellt, da callouts vollständig vom Compiler realisiert werden und unabhängig von der Laufzeitumgebung funktionieren.

Lediglich in dem Fall, dass callin-Bindings existieren und die ”main”-Funktionalität in einem Team, bzw. in einer Klasse, die ein Team explizit benutzt liegt, kann nicht mehr davon ausgegangen werden, dass die Anwendung im Sinne des Erfinders lauffähig ist, falls der Benutzer sie ohne JMangler, bzw. die ot-java Laufzeitumgebung startet. Dann würde nämlich versucht werden unvollständige Klassen zu benutzen. Callouts würden zwar durchgeführt werden, Teamaktivierung, und damit alle callins, hätten jedoch keine Wirkung. Diese Situation ist in der untenstehenden Tabelle durch ein ? gekennzeichnet.

Eine Zusammenfassung der Auswirkungen der verschiedenen Client-Konstallationen ist in der folgenden Tabelle zu finden.

	Adaptions-Richtung		
Einstiegspunkt	**nur callout**	**nur callin**	**callout & callin**
Basis	B* (B)	A (B)	A (B)
Team	A (A)	B* (B)	A (?)

Hierbei steht *B* für die ursprüngliche nicht-adaptierte (Basis-)Applikation und *A* für die durch Adaptionen verändete Applikation. Angegeben ist die Form des jeweils ausgeführten Programms mit ot-java. In Klammern dahinter steht, welche Version bei Benutzung des normalen java-Aufrufs verwendet werden würde.

Liegt der Einstiegspunkt (die ”main”-Funktionalität) auf Basis-Seite, so kann ein statisches Team benutzt werden, um mit einem statischen callin in einer initialen Basismethode den Aufruf einer Rollenmethode zu triggern, die die gewünschten Teams aktiviert.

Startet der Kontrollfluss auf Team-Seite, so kann die Basis-Funktionalität über callouts ins System ”geholt”. Werden dabei callins benutzt, so kann durch eine explizite Team-Referenzieung und -Aktivierung die Basis adaptiert werden. Deren Funktionalität ist dann jedoch nur über callouts benutzbar.

[2]Der Compiler unterzieht nur Team- und Rollenklassen einer *ot-spezifischen* Behandlung. Mit Basisklassen, wie mit allen anderen ”normalen” Java-Klassen verfährt er genauso, wie ein gewöhnlicher Java-Compiler.

Die in der Tabelle mit * gekennzeichneten Einträge, stellen Situationen dar, in denen es garnicht möglich ist, ein adaptiertes Verhalten zu erreichen, da die Adaptionsrichtung keinen Einstieg in das Team-Verhalten erlaubt.

In den dargestellten Szenarien wird davon ausgegangen, dass das main-Programm jeweils *entweder* das Team, *oder* das Basis-Package ”kennt” (referenziert). Bei einer Sichtbarkeit ”beider Seiten” ergeben sich noch weitere Möglichkeiten, auf die hier nicht näher eingegangen werden soll.

Eine weiter Möglichkeit, Teams in eine Applikation zu bringen wäre eventuell diese in einer zusätzlichen Konfigurationsdatei zu spezifizieren, dann wäre allerdings noch immer nicht die Frage ihres Aktivierungszeitpunktes geklärt.

4.2 Zukunftsmusik

4.2.1 Feststellung der ungenutzten Basis-Argumente im Wrapper

Zur Zeit werden die von der Rollenmethode nicht benötigten Argumente in der Basismethode festgestellt und im unusedArgs-Array weitergereicht. Dabei dient die Rollenmethoden-Signatur als Orientierung. Dem Rollenmethoden-Wrapper werden soviele Basismethoden-Argumente übergeben, wie die Rollenmethode an Argumenten erwartet, d.h. *unused* sind alle Argumente, die die Basismethode mehr hat, als die Rollenmethode. Solange es für ein Methoden-Binding keine Parameter-Mappings gibt, ist dies völlig in Ordnung und spiegelt das erwünschte Verhalten wider.

Gibt es jedoch Parameter-Mappings, so wird es letztlich nötig sein, die nicht benutzten Argumente erst im Rollenmethoden-Wrapper zu ”verpacken”. Aufgrund von Mappings ist es nämlich eigentlich erst dort möglich zu entscheiden, welche Argumente die Rollenmethode bekommt, und welche tatsächlich *unused* sind.

Aus diesem Grund wurde entschieden das Verfahren folgendermaßen abzuwandeln: Gibt es keine Parameter-Mappings, so bleibt alles wie bisher (siehe oben). Gibt es welche, so bekommt der Wrapper alle Basis-Argumente und ein leeres unusedArgs-Array übergeben. Dort können dann die Mappings durchgeführt werden und unbenutzte Argumente über das Array an die Rollenmethode durchgereicht werden. An diesem Punkt fehlt in der aktuellen Version noch Folgendes: Parameter-Mappings können Argumente umsortieren oder zwischendurch Basis-Argumente nicht benutzen. Abbildung 4.1 demonstriert ein solches Beispiel.

Die dazugehörige Binding-Deklaration in ot-java sieht folgendermaßen aus:

```
rm(r1 , r2 , r3 ) <− bm(b1, b2, b3, b4 , b5 ) with {
    r1 <− b1,
    r2 <− b2,
    r3 <− b4
}
```

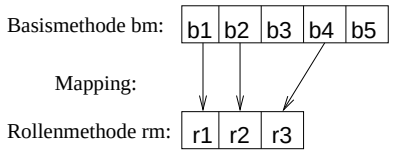

Abbildung 4.1: ”Verstreuendes” Parameter-Mapping

Hier werden die Basis-Argumente b3 und b5 nicht benutzt. Sie müssten demnach in das unusedArgs-Array gepackt werden. Dabei stellt sich die Frage, wie beim Auspacken rekonstruiert werden kann, um welche Basismethoden-Argumente es sich dabei handelt. Wahrscheinlich ist es dafür nötig, zusätzliche Informationen über die nicht benutzten Argumente mit zu übergeben. Vorstellbar wäre z.B. unbenutzte Argumente gemeinsam mit ihrer Position in der Basis-Signatur aufzubewahren. Aber auch für Argumente, die in der Rollensignatur einfach nur in anderer Reihenfolge auftreten als in der Basismethode, müsste es möglich sein die Original-Reihenfolge für den Basismethoden-Aufruf (base-call) wiederherzustellen.

Da Parameter-Mappings nun vom Compiler (im Rollenmethoden-Wrapper) realisiert werden, ist es unnötig geworden, die Informationen, welcher Basis-Parameter auf welchen Rollen-Parameter abgebildet werden soll, über Attribute dem Laufzeitsystem zugänglich zu manchen. Um das oben beschriebene Verhalten realisieren zu können, ist es lediglich noch nötig, dass festgestellt werden kann, ob Parameter-Mappings vorliegen, oder nicht. Dazu wurde das Attribut *CallinParam-Mappings* (siehe 4.3) verändert. Es ist jetzt nur noch ein Flag, dessen Vorhandensein anzeigt, dass es für eine Methode Parameter-Mappings gibt.

4.2.2 Bindung mehrerer Basismethoden an eine Rollenmethode

In der aktuellen Ausbaustufe der ot-Laufzeitumgebung ist jeder Rollenklasse eine eindeutige Basisklasse zugeordnet. Ausserdem wird bei einem base-call in einer callin-Rollenmethode davon ausgegangen, dass eindeutig ist, welche Basismethode diesen Aufruf empfangen soll. Es muss also immer genau eine Basismethode an eine Rollenmethode gebunden sein.
Konzeptionell spricht jedoch nichts gegen eine Bindung mehrerer Basismethoden an eine Rollenmethode. Deshalb ist für eine weitere Ausbaustufe eine Unterstützung hierfür geplant.

Geht man davon aus, dass die Basis zu einer Rolle eindeutig ist, so kommen die verschiedenen Basismethoden zumindest alle aus der selben Basisklasse. Für jedes einzelne Bindings wird ein spezieller Rollenmethoden-Wrapper benötigt. Die Wrapper sind schließlich an die jeweilige Signatur der Basismethode angepasst und führen das individuelle Parameter-Mapping für eine bestimmte Basismethode durch. Im Fall von after- und before-Binding wird von den einfachen Wrappern jeweils die unveränderte Rollenmethode mit den jeweiligen Argumenten

aufgerufen. Für jedes replace-Binding muss nun ein erweiterter Rollenmethoden-Wrapper generiert werden. Er ruft die erweiterte Rollenmethode auf, für die jetzt jedoch folgendes beachtet werden muss: Bei einem base-call muss sichergestellt werden, dass die ”richtige” Basismethode, d.h. die jenige, von welcher der Rollenmethoden-Aufruf ausging, aufgerufen wird.

Zwei denkbare Lösungsansätze sind folgende:

1. Eindeutige ID für Basismethoden:
   Beim Aufruf einer Rollenmethode (bzw. deren Wrapper aus dem umschliessenden Team) könnte die Basismethode eine ihr hinzugenerierte eindeutige ID mit übergeben. Anhand dieser wäre es dann möglich beim base-call in der Rollenmethode zu entscheiden, welche Basismethode aufgerufen werden muss. Dafür müsste ein switch-Block über die möglichen Basismethoden-ID’s in die (erweiterte) Rollenmethode generiert werden.

2. Generierung eigener Methoden für jede Möglichkeit:
   Eine andere Möglichkeit besteht darin, für jedes replace-Binding (also pro erweitertem Wrapper) eine spezifische erweiterte Rollenmethode zu generieren, die den base-call an die ”richtige” Basismethode stellt. Der Wrapper müsste dann die für ”seine” Basismethode zuständige Rollenmethode aufrufen.

Zusätzlich dazu muss die Infrastruktur zur Verwaltung der Bindings angepasst werden. Im Moment liefert die Methode *getBindingForRoleMethod* des CallinBindingManagers jeweils nur ein *MethodBinding* zurück. Existieren mehrere, so gibt er das *MethodBinding* der ersten Basismethode zurück, welches die entsprechende Rollenmethode bindet.

4.2.3 Weitere Baustellen

Folgende in Kapitel 1 vorgestellte Sprachkonzepte von Object Teams sind noch in der Entwicklungsphase und stehen in der aktuellen Version von ot-java deswegen noch nicht, bzw. eingeschränkt zur Verfügung:

1. Die implizite Team-Aktivierung bei Benutzung von Team-Level Features wurde noch nicht umgesetzt.

2. Teams können noch nicht einfach durch die Angabe des `static`-Modifiers permanent auf die Anwendung einwirken. Stattdessen müssen solche generell aktiven Teams durch Teams, welche ihre eigene `activate()`-Methode aufrufen, sich also selbst aktivieren, simmuliert werden.

Anhang

4.3 Callin-Attribute

Der ot-java Compiler generiert die folgenden Attribute, wenn er auf ein *Team* trifft, in dem Callin Bindings definiert sind. Jedes Element, dessen Name mit ”_index” endet, ist eine Referenz in den Constant Pool der jeweiligen Klasse. Die ”_count” endenden Einträge geben die Anzahl der Elemente in dem darauffolgenden Array an.

- Attribut *CallinRoleBaseBindings* der Team Klasse:
  CallinRoleBaseBindings {
  u2 attribute_name_index;
  u2 callin_bindings_count;
  CallinBinding callin_bindings[callin_bindings_count];
  }

  CallinBinding {
  u2 role_name_index;
  u2 base_name_index;
  }
  Ein CallinBinding entspricht einem `RoleBaseBinding` in der Laufzeitumgebung.

- Attribut *CallinMethodMappings* der Rollen Klasse (inner class des Teams)
  CallinMethodMappings {
  u2 attribute_name_index;
  u2 method_mappings_count;
  CallinMethodMapping method_mappings[method_mappings_count];
  }

  CallinMethodMapping {
  u2 role_method_name_index;
  u2 role_method_signature_index;

u2 binding_modifier_index;
u2 base_method_name_index;
u2 base_method_signature_index;
}
Ein CallinMethodMapping entspricht einem `MethodBinding` in der Laufzeitumgebung.

- Attribut *CallinParamMappings* der Rollenmethode:
  CallinParamMappings {
  u2 attribute_name_index;
  u2 param_mappings_count;
  CallinParamMapping param_mappings[param_mappings_count];
  }

  CallinParamMapping {
  u2 pos_role_param_index;
  u2 pos_base_param_index;
  }
  Ein CallinParamMapping entspricht einem `ParameterBinding` in der Laufzeitumgebung.

Das nachfolgende Attribut, wird für Klassen generiert, die Team-Klassen referenzieren.

- Attribut *ReferencedTeams* einer Klasse, die Teams referenziert:
  ReferencedTeams {
  u2 attribute_name_index;
  u2 referenced_teams_count;
  ReferencedTeam referenced_teams[referenced_teams_count];
  }

  ReferencedTeam {
  u2 referenced_team_index; }

4.4 BCEL-Beispiel

```
private Method insertParameterDumping(Method m, String class_name, ConstantPoolGen cpg) {
    MethodGen mg = new MethodGen(m, class_name, cpg);
    String method_name = m.getName();
    Type[] argTypes = mg.getArgumentTypes();
    ObjectType p_stream = new ObjectType("java.io.PrintStream");

    InstructionList patch = new InstructionList();
    int index;
    if (m.isStatic())
        index = 0;
    else
        index = 1; // nicht statische Methoden haben 'this' an Index 0
    for (int i=0; i<argTypes.length; i++) {

        patch.append(factory.createFieldAccess("java.lang.System", "out", p_stream, Constants.GETSTATIC));
        patch.append(factory.createLoad(argTypes[i], index));

        Type printArgType = argTypes[i];
        if (!(printArgType instanceof BasicType))
            printArgType = new ObjectType("java.lang.Object");

        patch.append(factory.createInvoke("java.io.PrintStream",
                                          "print",
                                          Type.VOID,
                                          new Type[] { printArgType },
                                          Constants.INVOKEVIRTUAL));
        index += argTypes[i].getSize();
    }

    InstructionList il = mg.getInstructionList();
    InstructionHandle[] ihs = il.getInstructionHandles();

    if (method_name.equals("<init>")) { // Aufrufe von 'super' oder anderen Konstruktoren
                                        m"ussen am Anfang bleiben
        for (int j=1; j < ihs.length; j++) {
            if (ihs[j].getInstruction() instanceof INVOKESPECIAL) {
                il.append(ihs[j], patch);
                break;
            }
        }
    } else
        il.insert(ihs[0], patch);

    mg.setMaxStack();
    mg.setMaxLocals();
    Method generatedMethod = mg.getMethod();
    il.dispose();
    return generatedMethod;
}
```

Literaturverzeichnis

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

[2] Homepage des Darwin Projekts: http://javalab.cs.uni-bonn.de/research/darwin/.

[3] Object Teams Homepage: http://www.objectteams.org.

[4] Christof Binder. *Aspectual Collaborations: Erweiterung des Java-Compilers für verbesserte Modularität durch aspekt-orientierte Techniken*. Diplomarbeit, Technische Universität Berlin, 2002.

[5] Markus Dahm. Byte Code Engineering with the BCEL API. Technischer Bericht, Freie Universität Berlin, 2001.

[6] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java Language Specification*. Java Series. Addison-Wesley, zweite Auflage, 2000.

[7] Stephan Herrmann. Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML at 1 st Intl. Conference on Aspect Oriented Software Development*. 2002.

[8] Stephan Herrmann. Object teams: Improving Modularity for Crosscutting Collaborations. 2002.

[9] Günter Kniesel, Pascal Costanza und Michael Austermann. JMangler - A Framework for Load-Time Transformation of Java Class Files. November 2001.

[10] Tim Lindholm und Frank Yellin. *The Java Virtual Machine Specification*. Java Series. Addison-Wesley, zweite Auflage, 1999.