Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik

Aspectual Collaborations: Erweiterung des Java-Compilers für verbesserte Modularität durch aspekt-orientierte Techniken

Diplomarbeit von:

Christof Binder
Dieffenbachstraße 33
10967 Berlin
Matrikelnummer: 16 67 65

Angefertigt unter der Leitung und Betreuung von Dipl. Infor. Stephan Herrmann
Berlin, 2002

Die selbständige und eigenhändige Anfertigung versichere ich an Eides Statt.

Berlin, den 7.8. 2002

Unterschrift:

Inhaltsverzeichnis:

Einleitung ... 7

Erfassung von dynamischen Artefakten ... 7

Erfassung von verstreuter Funktionalität ... 8

Aspektbegriff und aspekt-orientierte Ansätze ... 8

AspectJ ... 9

HyperJ ... 10

Vergleich der beiden Ansätze ... 11

Aspectual Collaborations ... 11

Vorläufermodelle ... 11

Von *Aspectual Collaborations* zu Object Teams ... 12

Object Teams ... 12

Die Kollaboration als Modul ... 13

Aspekt-Funktionalität ... 15

Rollen und Vererbung ... 16

Hinter den Kulissen ... 20

Zuordnung von Rollen- zu Basisobjekt ... 20

Umwandlung von Rollen- zu Basisobjekt und umgekehrt ... 20

Kontextverwaltung ... 22

Object Teams und Java ... 22

Realisierungskonzept von Object Teams ... 22

Die Programmiersprache Java für Object Teams ... 23

Ausgewählte Eigenschaften der Programmiersprache Java ... 24

Der Object Teams Compiler ... 25

Genereller Compiliervorgang ... 25

Analyse des Compilers ... 25

Datenstrukturen des *javac* ... 26

Die Phasen des *javac* ... 28

Lexikalische und syntaktische Analyse ... 28

Attribuierungsphase ... 29

Flusskontrollphase ... 29

Sourcecode-Transformationsphase ... 30

Bytecode-Generierungsphase 30
Compiler-Modifikationen für Object Teams 31
Spracherweiterung 31
Erweiterung der Datenstrukturen des *javac* 33
Erweiterung der lexikalischen und syntaktischen Analyse 37
Object Teams-Codegenerierung 37
callin: die Ersetzung der *base calls* (1) 38
Erzeugung der Rollen-Konstruktoren und `liftTo`-Methoden (2) 39
callin: die Erzeugung der Team-Konstruktoren (3) 39
callout: die Erzeugung der Rollenmethoden (4) 39
Kopieren der Rollen / Merging der Features (5) 42
Erweiterung der Attribuierungsphase 43
Erweiterung der Flusskontroll- und der Sourcecode-Transformationsphase 43
Erweiterung der Bytecode-Generierungsphase 43
Kopieren / Anpassen des Codes geerbter Rollen-und Team-Methoden (6) 43
Generieren von Attributen im Fall von *callin*-Bindings (7) 46
Zusammenfassung der Compilermodifikationen 47
Die Auswertung der Attribute durch das Object Teams-RE 48
Ausstehende Arbeiten am Compiler 48
Vervollständigung der elementaren Object Teams-Funktionalität 48
Type-Checking 50
Parameterabbildungen bei *callin*-Bindings 51
Aussagekräftige Fehlermeldungen des Compilers 51
Integration in eine Software-Entwicklungsumgebung 52
Ausblick 52
Kritikpunkte am *javac* 53
Nebeneffekte von Methoden 53
Kombinierte Verwendung von Shift- und Bit-Operatoren 53
Unpassende Feature-Namen 53
Bad smells 53
Fehlende Polymorphie 53
Weitere Optimierungen 54

Problematisches *Visitor Pattern* 54
Mögliche Compiler-Weiterentwicklungen 54
Analyse der Kapselbarkeit von Object Teams-Code 56
Neu eingeführte Object Teams-Klassen 56
Object Teams-spezifische Methodenanpassungen 57
Neue Object Teams-Features 57
Neu eingeführte innere Klassen 57
Dokumentation der eigenen Arbeitsmethodik 59
Analyse-Prozess 60
Konzepterstellungsprozess 62
Implementationsprozess 63
Prozessübergreifende Erfahrungen 65
Literatur und Quellen 67

Einleitung

Seit dem Beginn der Softwareentwicklung sind die beiden Ziele Flexibilität und Wiederverwendung von Software immer stärker in den Vordergrund gerückt. Forciert wurde dieser Prozess u.a. durch verschiedene „Software-Krisen“, in denen die fehlende Umsetzung der beiden oben genannten Anforderungen die Beteiligten zwang, Software völlig neu zu entwickeln und bestehende Software nicht integrieren zu können. Verschiedene Ursachen können dafür herangezogen werden, Software nicht anpassen oder wiederverwenden zu können. Eine Ursache gibt Meyer in [1] indirekt an: das Fehlen von Modularität der Software. Software, die aus Modulen aufgebaut ist, könne nach Meyer am ehesten flexibel und wiederverwendbar sein. Eine einfache Definition von Modularität liefert Meyer nicht, vielmehr listet er fünf Kriterien, fünf Regeln und fünf Prinzipien auf, die, wenn angewandt und befolgt, zu modularer Software führen. Stark verkürzt könnte man ein Modul als einen funktional in sich abgeschlossenen und stabilen Teil eines Softwaresystems bezeichnen. Im objektorientierten Ansatz, Software zu entwickeln, wurden diese Kriterien, Regeln und Prinzipien zur Modulentwicklung berücksichtigt bzw. die Grundlagen für deren Berücksichtigung geschaffen. Das Konzept des Moduls wurde in der *Klasse* realisiert.

Nachdem man den objektorientierten Ansatz heute als *state of the art* sowohl in der industriellen Softwareerstellung als auch – wenn auch schon länger – an Hochschulen bezeichnen kann, so ist festzustellen, dass dieser Ansatz in seiner ursprünglichen und puren Form zwei grundsätzlichen Anforderungen nicht Rechnung trägt: einerseits der Möglichkeit, dynamische Analyse- und Designartefakte in einer objektorientierten Sprache direkt umzusetzen und wiederzuverwenden, und andererseits die Funktionalität einer Applikation, die sich über mehrere Klassen und damit Module verteilt, modular zu erfassen. Diese Anforderungen werden im folgenden näher erläutert.

Erfassung von dynamischen Artefakten

In der Analyse- und Designphase des Softwareerstellungsprozesses können nicht nur die grundlegenden Elemente eines Systems, deren Eigenschaften und Operationen und deren Beziehungen untereinander modelliert werden, sondern auch Interaktionsmuster zwischen den Elementen. Die Unified Modeling Language (UML, [2]) z.B. bietet die Möglichkeit, durch Klassendiagramme die Elementstruktur – den statischen Teil eines Systems – zu erfassen und durch eine Reihe weiterer Diagrammtypen wie z.B. *Sequence Charts*, *Collaborations*, *Activity Diagrams* etc. die Interaktion der Elemente – den dynamischen Teil – darzustellen. Wird nun versucht, diese Artefakte auf eine Programmiersprache abzubilden, z.B. durch direkte Überführung in Quellcode, so ist dies nur bedingt möglich, eine direkte Entsprechung liegt nur für das Klassenmodell vor – eben die daraus entstehenden Klassen einer ojektorientierten Sprache. Ein dynamisches Diagramm wie z.B. das *Collaboration Diagram* kann nicht eins-zu-eins auf eine objektorientierte Sprache abgebildet werden, sondern findet sich im Code auf viele Klassen verteilt wieder. Die bedeutet

auch, dass dynamische Modellierungselemente nicht modular abgebildet werden können und somit schwer anpassbar und wiederverwendbar sind.

Erfassung von verstreuter Funktionalität

Ist ein Softwaresystem implementiert, so erfolgt nach bestimmten Zwischenschritten (z.B. nach dem Abnahmetest) der Einsatz, die Inbetriebnahme des Systems, woran sich wiederum Wartungsarbeiten anschliessen können. Wartung bedeutet u.U. die Reaktion auf neu aufgekommene Anforderungen an das System, z.B. die Unterstützung einer neuen Währung. Die Umsetzung einer solchen Anforderung bedeutet oft, das System an vielen Code-Stellen anzupassen, falls überhaupt offensichtlich ist, welche Stellen angepasst werden müssen. Die Funktionalität, die von einer Anforderung betroffen ist, ist oft über das gesamte System, also über verschiedene Klassen und damit Module verteilt, und ist somit keine modulare, in sich abgeschlossene Einheit, sondern von vielen anderen Einheiten abhängig.

Die vorliegende Arbeit hat die partielle Umsetzung eines Programmiermodells namens *Object Teams* zum Thema, das die beiden beschriebenen Probleme adressiert. Bevor auf dieses Modell detailliert eingegangen wird, werden der Begriff „Aspekt“ und damit verwandte Begriffe geklärt und bereits existierende aspekt-orientierte Ansätze vorgestellt.

Aspektbegriff und aspekt-orientierte Ansätze

Eine oben beschriebene Funktionalität, die über viele Klassen verstreut und somit nicht mehr modular ist, kann man als Aspekt einer Software bezeichnen. Man spricht im englischen Sprachgebrauch auch von *crosscutting aspects*, also von Aspekten, die nicht nur passiver Weise nicht modular sind, sondern dadurch die Module, in denen sie auftauchen, aktiv aufbrechen und „zerschneiden“. Ein Aspekt im Sinne einer Systemfunktionalität ist nicht notwendigerweise über verschiedene Module verstreut, wenn er allerdings nicht verstreut ist, so ist er in einer Klasse gekapselt und stellt somit kein „Modul-Problem“ dar. Unter einem Aspekt ist im folgenden stets ein über Module verstreuter Aspekt zu verstehen. Ebenfalls gebräuchliche oder zumindest verwandte Begriffe sind *Concern* [3] und *Dimension* [4].

Ein viel zitiertes Beispiel für einen Aspekt ist die Logging-Funktionalität einer Software. Ein *Logger* protokolliert zur Laufzeit vorher ausgewählte Aktionen der Software und gibt das Protokoll auf dem Bildschirm aus, schreibt es in eine Datei oder in einen anderen Datenstrom. Die Verstreutheit dieses Aspekts ist unmittelbar einsichtig: Aktionen entsprechen den in Modulen definierten Operationen. Wenn diese protokolliert werden sollen, so muss an allen Operationen, die für das Protokoll von Interesse sind, eine Protokollierungsaufruf eingefügt werden. Alle von diesem Aspekt betroffenen Module sind *tangled*, also „verwickelt“ mit diesem Aspekt, aus der Sicht des Aspekts kann dieser als *scattered*, also verstreut über viele Module angesehen werde. Ein weiteres Beispiel für einen Aspekt ist die (eventuell nachträgliche und während der Wartung auftauchende) Entscheidung, ein Softwaresystem verteilt arbeiten zu lassen; z.B. könnten die Anfragen an einen

Server dramatisch gestiegen sein und man möchte durch mehrere parallele Server eine Lastenverteilung erreichen. Auch für die Realisierung dieses Beispiels sind an zahlreichen Quellcode-Stellen Modifikationen zu erstellen, im Falle von z.B. CORBA müsste für jeden verteilten Objektgebrauch die gesamte CORBA-Infrastruktur (ORB, Namensdienst etc.) berücksichtigt werden.

Zu den bekanntesten aspekt-orientierten Ansätzen zählen AspectJ [5] und HyperJ [4], die nun kurz vorgestellt werden.

AspectJ

Dieser Ansatz stellt eine Erweiterung der Programmiersprache Java dar und wurde unter einer *Open Source License* [6] entwickelt. Diese Erweiterung ermöglicht die Modularisierung von Aspekten, die mehrere Java Klassen betreffen, in eigenen Dateien, in denen ein Aspekt definiert werden kann. Drei Elemente des AspectJ-Modells sind von zentraler Bedeutung: *Joint Point*, *Pointcut* und *Advice*. Ein Joint Point ist eine Sourcecode-Stelle einer Software, an der zusätzlicher Aspekt-Code eingefügt werden soll. Dies kann ein Methodenaufruf oder auch die Behandlung einer Ausnahme sein. Ein Pointcut stellt eine Menge von Joint Points dar, wobei diese Menge deklarativ erfasst wird. Sind z.B. alle *public* Methodenaufrufe einer Klasse `A` für einen Aspekt relevant, so kann der Pointcut wie folgt definiert werden:

```
pointcut methods(A a): target(a) && call(public * *(..))
```

Der Pointcut `methods` erhält ein Parameter `a` vom Typ `A`, auf den in der Definition (rechts des Doppelpunktes) Bezug genommen werden kann, hier in der Zielangabe eines Methodenaufrufs `target(a)`. Dieses Ziel wird mit der Aufrufdefinition `call(public * * (..))` durch ein logisches UND verknüpft. Die Aufrufdefinition betrifft alle *public* Methoden der Klasse `A` – unabhängig von Namen, Rückgabewert, Paramteranzahl- und typen. Durch eine solche Definition können mehrere Joint Points und damit Code, der an sehr vielen Stellen verwendet wird, in einer Zeile erfasst werden.

Nach der Definition der relevanten Quellcode-Stellen muss die eigentliche Funktion des Aspekts durch einen Advice definiert werden. Hier ein Beispiel:

```
before(A a): methods(a) {
    if (!a.getPasswd().equals(„root“)) {
        throw new AuthenticationException();
    }
}
```

Dieser Advice besitzt einen *Modifier* namens `before`, der besagt, dass er jeweils vor dem jeweiligen Point Cut (hier `methods`) aktiviert werden soll. Der Pointcut definiert also, wann eine aspekt-relevante Modifikation erfolgen soll, der Advice definiert die eigentliche Modifikation. In Kombination mit dem definierten Pointcut `methods` prüft der Advice immer vor der Ausführung einer *public*-Methode eines Objekts vom Typ `A`, ob das Passwort-Feld des Objekts dem Inhalt „root“ entspricht. Ist dies nicht der Fall, wird eine Ausnahme geworfen.

Nachdem ein Aspekt in AspectJ definiert wurde, müssen dieser und die Klassen, die davon betroffen sind, neu compiliert werden. Dabei parst ein Prä-Prozessor die Aspektdefinitionen und generiert daraus konventionellen Java-Quellcode, der anschliessend von einem Java-Compiler compiliert wird.

HyperJ

Ebenfalls für die Sprache Java in einer IBM Forschungseinrichtung entwickelt verfolgt HyperJ einen Aspekt-Ansatz, der im Vergleich zu AspectJ als feingranularer bezeichnet werden kann. Während in AspectJ der Schwerpunkt auf Klassen und deren Features liegt, so sieht das HyperJ-Modell einen Aspekt allgemeiner, wodurch theoretisch auch einzelne Anweisungen eines Methodenrumpfs für einen Aspekt relevant sein können, während andere desselben Methodenrumpfs das nicht sein müssen. Methodenanweisungen als Aspekt-Element sind jedoch z.Z. noch nicht implementiert, aktuell sieht die HyperJ-Implementierung Methoden und Variablen als feinste Aspekt-Elemente vor. Solche feingranularen Einheiten werden *Units* genannt. Neben Sourcecode-Elementen können auch größere Informationsblöcke wie z.B. Modellierungsdiagramme oder Dokumentationsdokumente Units darstellen. Diese Units stellen eine multi-dimensionale Matrix da, in der jede Achse eine Dimension eines Aspekts darstellt. Unter einer Aspektdimension ist eine Sicht auf einen Aspekt zu verstehen. Die funktionale und strukturelle Dekomposition von Software, die ein Ergebnis des Designs sein kann, ist ein Beispiel für eine Dimension, der Code einer Software ein weiteres. Eine konkrete Software besteht aus mindestens einer Schicht, einem *Hyperslice* innerhalb dieser Matrix, die alle relevanten Units für die Ausführung der Software zusammenfasst. Hyperslices können wiederum zu *Hypermodules* zusammengefasst werden. Die gröbste Sicht, sozusagen die oberste Hierarchie-Stufe der HyperJ-Konstrukte, stellt der *Hyperspace* dar, der Hypermodules integriert.

Werden Units zu einer Softwaredimension zusammengestellt, so referenziert das enstandene Hyperslice eventuell eine Vielzahl von Units, z.B. Klassen und Methoden. Dies würde die Schicht stark anwachsen lassen, wenn diese referenzierten Elemente komplett mit in die Schicht integriert würden. Dies ist jedoch meist nicht im Sinne des Hyperslice-Designers, der durch die Schicht einen bestimmten Teil-Aspekt der Software kapseln will, der zwar von anderen Teil-Aspekten abhängen kann, trotzdem aber in sich abgeschlossen sein soll, ohne alle referenzierten Units komplett zu enthalten. Bei der Erzeugung eines Hyperslice kann deshalb die Tiefe angegeben werden, mit der innerhalb von Methodenaufrufen Referenzen verfolgt und referenzierte Units integriert werden. Beim Erreichen der maximalen Tiefe bricht der Integrationsalgorithmus ab. Damit das Hyperslice trotzdem compilierbar bleibt, werden alle Klassen und Methoden, die nicht integriert aber trotzdem referenziert werden, in diesem Hyperslice zusätzlich deklariert, das Hyperslice wird somit deklarativ vollständig.

Auch in HyperJ erfolgt die Umsetzung der Aspektdefinitionen per Prä-Prozessor, Ergebnis ist konventioneller Java-Bytecode.

Vergleich der beiden Ansätze

Im Vergleich zu AspectJ ist HyperJ sehr viel mächtiger, was die Zusammenstellung von Code-Teilen zu Aspekten betrifft. Dafür ist die Benutzung bei HyperJ komplizierter und verlangt mehr Planung und Konfiguration bei der Definition der Aspekte bzw. deren Dimensionen, während durch AspectJ besser auf ungeplante, während des Softwareentwicklungsprozesses spontan auftauchende Anforderungen mittels „ad-hoc"-Aspekten reagiert werden kann.

Während AspectJ nur die Forderung nach Modularisierung von Aspekten einlöst, ist mit HyperJ auch die in der Einleitung erwähnte erste Forderung teilweise einlösbar: die Möglichkeit, dynamische Elemente des Designs wie *Collaborations* zu definieren und diese Definitionen wiederzuverwenden. Allerdings sind die HyperJ-Definitionen nicht Bestandteil der genutzten Programmiersprache Java, sondern eine Konfigurationsmöglichkeit ausserhalb der Sprache.

Das im nächsten Kapitel vorgestellte Modell *Aspectual Collaborations* hat ebenfalls den Anspruch, sowohl die Modularisierung von *crosscutting concerns* zu ermöglichen als auch größere Einheiten wie Objekt-Interaktionsmuster modular zu erfassen. Als zwei von mehreren Unterschieden zwischen diesem und den zuletzt vorgestellten Modellen seien jetzt schon die Integration des Modells in die Sprache Java selbst und die fehlende Notwendigkeit einer Neu-Compilierung der beteiligten Klassen erwähnt.

Aspectual Collaborations

Dieses Programmiermodell hat verschiedene Vorläufermodelle, die alle die Trennung von *Concerns*, also von verschiedenen Aspekten einer Software zum Thema hatten. Diese werden nun kurz vorgestellt.

Vorläufermodelle

Als Ausgangsmodell können die *Adaptive Plug&Play Components* (AP&PC) [7] bezeichnet werden. Adaptiv sind die vorgestellten Komponenten, da sie Kollaborationen erfassen, die an eine Vielzahl von konkreten Applikationen (*class graphs*) angepasst werden können. Die Applikation kann als formaler Parameter für eine AP&PC gesehen werden. Als *Plug&Play* werden sie bezeichnet, da für ihre Anwendung keine Implementierungsdetails der Applikationen bekannt sein müssen, sondern diese als *black box* verwendet werden können. Eine Grundlage für AP&PC sind *Templates*, also mit Freiheitsgraden versehener Code, die durch eine Applikation und deren Klassen gefüllt werden können.

Das Nachfolgemodell der AP&PC ist das *Aspectual Components*-Modell [8], das die Applikation und deren Kompatibilität mit denen in den Komponenten erwarteten Typen durch das sogenannte *expected interface* betont. Wie auch bei HyperJ ist hier der Begriff der deklarativen Vollständigkeit innerhalb einer Komponente zu erwähnen, wonach alle benötigten Features innerhalb der Komponente deklariert wurden und die Komponente somit als eigenständiges Modul verwendet werden kann. Desweiteren führen *Aspectual Components* auch erstmals innerhalb dieser

Vorgängermodelle Aspektfunktionalität durch das *replace*-Konstrukt ein, wodurch ein Aufruf einer Komponentenmethode aus einer Anwendung heraus ermöglicht wird.

Das *Dynamic View Connector*-Modell (DVC) [9] wurde parallel zur Implementierung einer Softwareentwicklungsumgebung entwickelt, ist stark mit diesem Anwendungsbeispiel verknüpft und führt im Vergleich mit den Vorgängermodellen eine Reihe weiterer Features ein, was die Definition einer Kollaboration betrifft. Der DVC kann dabei eine Sicht auf eine objektorientierte Datenbank darstellen, da neben Methoden- auch Attributabbildungen einer Kollaboration auf die Anwendung, also z.B. auf Datenbankelemente wie Spalten unterstützt werden. Diese Attributabbildungen können mit *filter*- und *redirect*-Konstrukten angepasst werden.

Der direkte Vorgänger zu dem in dieser Arbeit behandelten Modell ist *LUA Aspectual Components* (LAC) [10], was wiederum hauptsächlich auf *Aspectual Components* und teilweise auf DVC aufbaut. Dabei werden die zwei elementaren Konstrukte einer angewendeten Kollaboration, die *Collaboration* selbst und der *Connector* einführt. Der *Connector* bildet dabei die Features der Kollaboration auf die Anwendung ab. Er kann dabei statisch und damit stets aktiv oder dynamisch und damit instantiierbar sein. Im Vergleich zu *Dynamic View Connector* ist LAC, was die verwendeten Features betrifft, bescheidener, aber dafür übersichtlicher und verständlicher. Dieses Modell wurde in der erweiterbaren Interpretersprache LUA [11] implementiert.

Das Modell, das Thema dieser Arbeit ist, heisst *Aspectual Collaborations* und kann als direkter Nachfolger von LAC angesehen werden. Hier ist die strikte Trennung zwischen Kollaboration und Konnektor wieder aufgehoben, letzterer ist schlicht eine Spezialisierung der ersteren, Zuordnungen von Kollaborations- zu Applikations-Elementen können sich somit auch über mehrere, die Kollaboration erweiternde Konnektoren verteilen.

Von *Aspectual Collaborations* zu Object Teams

Obwohl der Name *Aspectual Collaborations* die beiden Hauptmerkmale des Modells, nämlich die Adressierung von Kollaborationen und Aspekten, erkennen läßt, wurde im Laufe der Bearbeitung dieser Diplomarbeitsaufgabe ein neuer Name für dieses Modell eingeführt, der auch im folgenden benutzt wird: *Object Teams*.

Object Teams

Der Name *Team* zeigt bereits an, dass nicht Klassen, sondern eine größere Einheit, nämlich ein Team, eine Kollaboration von Objekten modular erfasst wird. Dabei kann ein Team aus beliebig vielen *Rollen* bestehen, die zusammen eine bestimmte Funktionalität zur Verfügung stellen. Der zweite wichtige Bestandteil dieses Modells sind *Bindings*, mit denen aspektorientiertes Verhalten erreicht werden kann. Durch ein Binding wird festgelegt, welche *Basis* eine bestimmte Rolle spielt und welche Methoden dabei aufeinander abgebildet werden. Ein genereller Unterschied zwischen einem Aspekt aus AspectJ oder einer Hyperslice aus HyperJ und einem Team mit Bindings ist, dass dieses instantiierbar und somit ein „Objekt erster Ordnung“ und damit genauso wie „normale“ Objekte benutzbar ist, also z.B.

polymorph verwendet werden kann. Sowohl die Konzepte der modularen Kollaboration als auch des Bindings lassen sich am besten an Beispielen erläutern.

Die Kollaboration als Modul

```
public abstract team class GetAndSet {
    public abstract class Role1 {
        public abstract Role2 createRole2();
        public void collaborate() {
            Role2 r2 = createRole2();
            r2.set(this);
            Role1 r1 = r2.get();
            if (this != r1) {
                throw new RuntimeException();
            }
            System.out.println("collaboration worked ok");
        }
    }
    public abstract class Role2 {
        // getter
        abstract public void set(Role1 r1);
        // settter
        abstract public Role1 get();
    }
}
```

Das Team `GetAndSet` enthält zwei Rollen, `Role1` und `Role2`, deren Verhalten teilweise noch nicht implementiert ist (die entsprechenden Methoden sind als `abstract` deklariert). Die Rolle `Role1` enthält die Methode `collaborate`, die das Zusammenspiel der beiden Rollen deutlich macht. Dabei fungieren die beiden Methoden der Rolle `Role2` `set` und `get` als *setter*- und *getter*-Methoden für ein – nicht implementiertes - Feld der Rolle `Role2`. Dieses Team ist nicht ausführbar, da es abstrakte Elemente enthält. Es kapselt jedoch ein ganz bestimmtes Verhalten, nämlich das Schreiben und Lesen eines Feldes und das Testen dieser Aktionen. Um dieses Verhalten auch durchführen zu können, müssen die Rollen von konkreten Basisklassen „gespielt“ werden. Diese Abbildung wird durch ein Binding erreicht:

```
public open team class ConnectorA extends GetAndSet {
    open class Role1 playedBy Base1 {
        createRole2 -> createBase2;
    }
    open class Role2 playedBy Base2 {
        set -> setBase1;
        get -> getBase1;
    }
    public void perform() {
        Role1 r1 = new Role1();
        r1.collaborate();
    }
}
```

Innerhalb der Klasse `ConnectorA` sind zwei Rollen-Bindings definiert, wodurch einer Rolle eine Basis zugeordnet wird. Ein Rollen-Binding wird durch die `playedBy`-Klausel erreicht. In jeder Rolle erfolgt die Abbildung von Rollenmethoden auf

Basismethoden. Die Abbildungsrichtung ist durch das Symbol `->` festgelegt und wird als *callout* bezeichnet. Voraussetzung für dieses Binding ist die Existenz der beiden Basisklassen `Base1` und `Base2` mit konkreten Methoden, die den Namen aus den Methodenabbildungen entsprechen und darüber hinaus zu ihren Rollenmethoden hinsichtlich ihrer Signatur „passen“ (unter welchen Umständen Methodensignaturen zueinander passen wird später erläutert). Da dies in diesem Beispiel der Fall ist, können die Methodensignaturen bei ihrer Abbildung vernachlässigt werden. Was noch fehlt sind die Basisklassen `Base1` und `Base2`.

```
public class Base1 {
    public Base2 createBase2() {
        return new Base2();
    }
}

public class Base2 {
    private Base1 b1;
    public void setBase1(Base1 b1) {
        this.b1 = b1;
    }
    public Base1 getBase1() {
        return b1;
    }
}
```

Die Klasse `ConnectorA` kann nun instantiiert und auf dem `ConnectorA`-Objekt kann die Methode `perform` aufgerufen werden, mit dem Ergebnis, dass die beiden Basisklassen nach dem Interaktionsmuster in `GetAndSet` interagieren. Dabei werden Aufrufe an Rollenobjekte an deren Basisobjekte weitergeleitet. Die Klasse `GetAndSet` ist unabhängig von den konkreten Basisklassen definiert, compilierbar und somit auch mit anderen Basisklassen wiederverwendbar. Desweiteren ist die Objektinteraktion in einem eigenen Modul, dem Team `GetAndSet`, erfasst.

Im obigen Beispiel ist für die Bindungsrichtung *callout* (`->`) gewählt worden, da die Methoden der Rollen des Teams `GetAndSet`, die im Binding des Teams `ConnectorA` gebunden werden, alle abstrakt sind. Es besteht ebenfalls die Möglichkeit, konkrete, also bereits implementierte Rollenmethoden in einem Binding an Basismethoden zu binden, ihnen also eine neue Implementierung zuzuweisen. Dafür ist eine eigene Bindungsrichtung vorgesehen, da dies semantisch einem Überschreiben der ursprünglichen Rollenmethode gleichkommt. Diese Bindungsrichtung heisst *callout override* und wird durch das Symbol `=>` ausgedrückt. Dieser Fall wird im Rest dieser Arbeit stets als Spezialfall von *callout* aufgefasst. Nur wenn nötig wird er gesondert betrachtet.

Nachdem die Modularisierung der Kollaboration von Objekten im vorherigen Beispiel gezeigt wurde, zeigt ein weiteres Beispiel, wie Aspekte im Sinne von AspectJ und HyperJ durch Object Teams realisiert werden können.

Aspekt-Funktionalität

Das folgende Beispiel soll die Implementierung eines Aspekts illustrieren, der für die Überprüfung bestimmter Eingabewerte verantwortlich ist. Dazu sei zuerst das Team `ConnectorB` aufgezeigt:

```
public open team class ConnectorB extends CheckAndLog {
    open class Role1 playedBy AccessGranter {
        checkCredentials <- replace createAccess;
    }
    open class Role2 playedBy Access {
        log <- after enter;
    }
    public void perform() {
        Base1 b1 = new Base1();
        Base2 b2 = b1.m1("admin", "passwd");
        b2.m2();
    }
}
```

Ein augenfälliger Unterschied zwischen dem ersten Konnektor und diesem ist die Richtung der Methodenabbildung, die sich von *callout* (`->`) zu *callin* (`<-`) geändert hat. Die Richtung *callin* zeigt an, dass die Basis in die Rolle „hineinruft", die Rollenmethode wird somit von der Basis aufgerufen, während bei der Richtung *callout*, die im `ConnectorA` verwendet wurde, die Rolle die Methode der Basis aufruft. Der Modifier nach dem Richtungspfeil zeigt an, wie dieser Aufruf erfolgt. Bei *replace* wird die Rollenmethode anstelle der Basismethode aufgerufen, bei *after* nach Beendigung der Basismethode und bei *before* (nicht in `ConnectorB` verwendet) vor der Basismethode. Damit die Rollenmethoden von den Basisklassen benutzbar sind, müssen diese – im Gegensatz zu denen der Rollen des Teams `GetAndSet` – konkret sein, wie in der folgenden Teamdefinition `CheckAndLog`.

```
public open team class CheckAndLog {
    public open class Role1 {
        public callin Role2 checkCredentials(
            String id, String passwd) {
            if (id.equals("admin") && passwd.equals("passwd")) {
                return base(id, passwd);
            } else {
                throw new RuntimeException(
                    "Authorization Error");
            }
        }
    }
    public class Role2 {
        public void log() {
            System.out.println("logged by Role2");
        }
    }
}
```

Der Modifier `open` zeigt an, dass diese Klasse ein Binding benötigt, um benutzbar zu sein (trotzdem ist sie ohne Binding compilierbar). Der Grund hierfür liegt in der Methode `checkCredentials` der Rolle `Role1`, die durch die Verwendung des

Schlüsselworts `base` die Originalmethode der Basis aufruft, wofür ein Binding notwendig ist. Die Verwendung des Schlüsselworts `base` wurde deshalb eingeführt und ist nicht durch den eigentlichen Namen der Basismethode zu ersetzen, da das Team `CheckAndLog` unabhängig von anderen Teams compilierbar und darüberhinaus mit verschiedenen Basis-Objekten und deren Methoden verwendbar sein soll. Um anzuzeigen, dass die Methode `checkCredentials` von einer Basis aufgerufen werden muss, ist sie mit dem Modifier *callin* ausgestattet.

Die Basisklassen, deren Verhalten durch die Kollaboration und das Binding verändert werden, sind die folgenden:

```
public class AccessGranter {
    private Map map = new HashMap();
    public Access createAccess(String id, String passwd) {
        map.put(id, passwd);
        return new Access();
    }
}

public class Access {
    public void enter() {
        System.out.println("entered");
    }
}
```

Der Effekt der Teams `CheckAndLog` und `ConnectorB` auf die Klassen `Access` und `AccessGranter` ist folgender: wird die Methode `createAccess` auf ein `AccessGranter`-Objekt aufgerufen, dann wird zuerst durch die Methode `checkCredentials` der Rolle `Role1` im Team `CheckAndLog` überprüft, ob die aktuellen Parameter von `createAccess` bestimmten Anforderungen genügen. Wenn dem so ist, dann ruft `checkCredentials` seine Basis-Methode, also `createAccess` auf. Durch das zweite Binding im Team `ConnectorB` wird erreicht, dass nach einem Aufruf `enter` auf ein `Access`-Objekt dieser Aufruf durch die Methode `log` der Rolle `Role2` des Teams `CheckAndLog` protokolliert wird.

Mit den beiden vorgestellten Beispielen wurden die Object Teams-Features Kollaborationsmodularität und Aspektfunktionalität illustriert. Mit diesen Features können bereits viele Aufgaben sinnvoll bearbeitet werden. Als Klassen im objektorientierten Sinn müssen Rollen jedoch auch Vererbung unterstützen. In Object Teams gilt dabei eine besondere Vererbungssemantik, wobei die erbende Rolle zwar die Funktionalität der beerbten Rolle übernimmt, jedoch keinen Subtyp der beerbten Rolle darstellt, wenn beide Rollen verschiedenen Teams angehören. Das Erben von Rollen, die im Super-Team definiert sind, geschieht automatisch und wird deshalb als *implicit inheritance* bezeichnet. Diese Vererbungssemantik wird im folgenden Abschnitt detailliert beschrieben.

Rollen und Vererbung

Es gibt drei Möglichkeiten, wie eine Rolle Verhalten von anderen Rollen erben kann. Die erste Möglichkeit wurde bereits beschrieben: indem abstrakte Rollenmethoden per *callout* an Basismethoden gebunden werden, erbt die Rolle das Verhalten der

Basis. Dieser Mechanismus kann als Delegation, präziser als *Forwarding* bezeichnet werden: der Aufruf an die Rolle wird an die Basis weitergeleitet, weitere Aufrufe an *self* werden an das Basisobjekt gerichtet. Die zwei weiteren Vererbungsmöglichkeiten seien an einem Beispiel erklärt.

```
public open team class CheckAndLog2 extends CheckAndLog {
    public open class Role1 {
        Map logMap = new HashMap();
        public callin Role2 checkCredentials(
            String id, String passwd) {
            logMap.put(id, passwd);
            return tsuper.checkCredentials(id, passwd);
        }
    }
    public open class RoleX extends Role1 {
        public callin Role2 checkCredentials(
            String id, String passwd) {
            if (logMap.get(id) == null) {
                return super.checkCredentials(id, passwd);
            } else {
                System.out.println(„allready logged in");
                return super.checkCredentials(null, null);
            }
        }
    }
}
```

Das Team `CheckAndLog2` erbt vom Team `CheckAndLog`. In Object Teams erbt es dabei automatisch alle Rollen, die in `CheckAndLog` definiert wurden, also `Role1` und `Role2` (*implicit inheritance*). Wäre dies nicht der Fall, so könnte die Methode `checkCredentials` der Rolle `CheckAndLog2.Role1` keinen Rückgabewert vom Typ `Role2` besitzen, da dieser in `CheckAndLog2` nicht explizit definiert ist.

Sowohl `CheckAndLog` als auch `CheckAndLog2` besitzen eine Rolle mit dem Namen `Role1`. In Object Teams erbt die Rolle `CheckAndLog2.Role1` damit implizit alle Features der Rolle `CheckAndLog.Role1`, die Vererbung wird per *name matching* erreicht. Im obigen Beispiel überschreibt die Rolle `CheckAndLog2.Role1` die Methode `checkCredentials`. Im Methodenrumpf der überschreibenden Methode kann die Implementierung der überschriebenen Methode weiterhin aufgerufen werden, wie dies auch bei konventioneller Vererbungssemantik möglich wäre. Da dies im Beispiel über Team-Grenzen hinweg geschieht und um den Unterschied zur konventionellen Vererbungssemantik deutlich zu machen – `CheckAndLog2.Role1` ist kein Subtyp von `CheckAndLog.Role1` –, wurde das neue Schlüsselwort `tsuper` eingeführt.

Vererbung zwischen Rollen innerhalb eines Teams wird durch die Rolle `RoleX` demonstriert. Sie überschreibt erneut die Methode `checkCredentials` und benutzt dabei die Implementierung der überschriebenen Methode aus `CheckAndLog2.Role1`. Dabei benutzt sie das Schlüsselwort `super`, das die Standard-Vererbungssemantik zum Ausdruck bringt: `CheckAndLog2.RoleX` stellt einen Subtyp von `CheckAndLog2.Role1` dar. In einer Team-Methode wären damit Objekte vom Typ

CheckAndLog2.Role1 durch Objekte vom Typ CheckAndLog2.RoleX ersetzbar und Referenzen vom Typ CheckAndLog2.Role1 polymorph.

Allerdings kann ein Team in seinen Methoden nicht auf Rollen aus anderen Teams zurückgreifen, wenn diese Rollen im Team überschrieben wurden, obwohl es von diesen Teams erbt. Gegeben sei die folgende Konstellation:

```
public open team class CheckAndLog {
    // gleiche Definitionen wie im vorangegangenen Beispiel
    ...
    public Role1 createRole1() {
        return new Role1();
    }
}
public open team class CheckAndLog2 extends CheckAndLog {
    // gleiche Definitionen wie im vorangegangenen Beispiel
    ...
    public void team_method() {
        Role1 r1 = createRole1();
        r1.logMap = new WeakHashMap();
    }
}
```

Das Team CheckAndLog definiert die Methode createRole1, die eine Rolle vom Typ CheckAndLog.Role1 zurückgibt. Das Team CheckAndLog2 erbt diese Methode und benutzt sie in ihrer neuen Methode team_method. Obwohl die Methode createRole1 nach ihrer Signatur eine Rolle vom Typ CheckAndLog.Role1 zurückgibt, wird diese Signatur durch die Benutzung der Methode in CheckAndLog2 zu CheckAndLog2.Role1 createRole1() implizit angepasst. Damit wird der Zugriff auf das Feld logMap erst ermöglicht, da dieses erst in CheckAndLog2.Role1 definiert wurde. Wenn also der Rückgabewert einer Team-Methode oder einer ihrer formalen Parameter einen Rollen-Typ darstellt, so wird dieser Typ nach den Rollen-Typen aufgelöst, die im dynamischen Typ der aktuellen Team-Instanz verfügbar sind. Dies gilt nicht nur für Team- sondern auch für Rollenmethoden, deren Signatur Rollentypen beinhalten. Rollen-Typen in Signaturen beziehen sich somit stets auf die Rollen-Typen des Teams, in dem sie benutzt werden. Obwohl in diesem Kapitel das Konzept der Vererbung von Object Teams im Vordergrund steht, so soll trotzdem bereits darauf hingewiesen werden, dass nicht alleine die Signatur angepasst wird. Im vorgestellten Beispiel muss ebenfalls die Rollenerzeugung new Role1() im Rumpf der Methode CheckAndLog.createRole1 angepasst werden: statt einem Objekt vom Typ CheckAndLog.Role1 muss ein Objekt vom Typ CheckAndLog2.Role1 erzeugt werden. Weitere Konsequenzen der speziellen Vererbungssemantik von Object Teams werden später behandelt.

Der Grund für die besondere Vererbungssemantik liegt in der verfolgten Absicht, Teamfunktionalität schrittweise verfeinern zu können. Dabei wird sichergestellt, dass die Funktionalität eines Teams sich immer auf die in diesem Team befindlichen Rollen bezieht, sowohl was die Features des Teams als auch was die Features der Rollen betrifft. Beispielsweise könnte ein Team TeamA eine Methode mit der Signatur RoleX cloneRole(RoleX role) besitzen. Ein Team TeamB erbt von TeamA und

damit auch die Methode `cloneRole`. Da bei dieser sowohl das Ergebnis als auch der formale Parameter vom Typ `TeamA.RoleX` sind, wäre sie in `TeamB` – in Sinne einer verfeinerten Teamfunktionalität – nur dann sinnvoll einsetzbar, wenn Ergebnis und formaler Parameter spezieller, also vom Typ `TeamB.RoleX` wären und sich damit auf die Rolle bezöge, die in `TeamB` definiert (und eventuell verfeinert) wurde. Das Ergebnis wäre eine kovariante Signatur-Redefinition von `cloneRole`. Abbildung 1 veranschaulicht dies und die Vererbungsbeziehung.

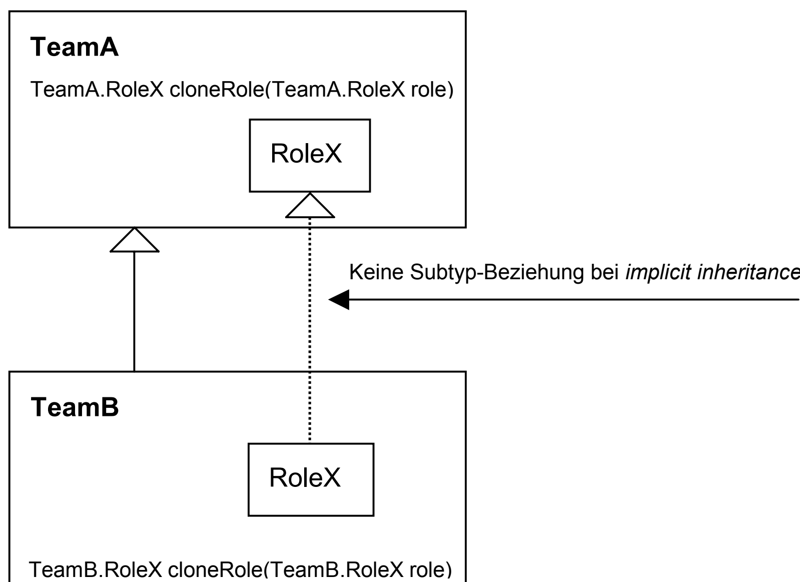


Abbildung 1: Beispiel für *implicit inheritance*

In [1] ausführlich dargestelt, durch andere Quellen dokumentiert und auch durch eigene Beispiel nachvollziehbar ist die Tatsache, dass kovariante Redefinition nicht sicher ist, wenn Objekte innerhalb der Vererbungshierarchie in einer Subtyp-Beziehung stehen und Referenzen auf diese Objekte polymorph verwendet werden (sicher ist neben der Signatur-Beibehaltung die konforme Redefinition). Deshalb stehen in Object Teams Rollen über Teamgrenzen hinweg nicht in einem Subtyp-Verhältnis, können dabei jedoch trotzdem Features der Super-Rollen – nach Object Teams-Terminologie Features der TSuper-Rollen – erben. Weiterhin erlaubt die Anpassung von Team-Features deren Wiederverwendung über Teamgrenzen hinweg.

Durch den Verzicht auf die Subtyp-Beziehung von „Inter-Team“-Rollen und damit auf die Substituierbarkeit von „Inter-Team“-Rolleninstanzen wird statische Typsicherheit erreicht, d.h. es kann trotz kovarianter Redefinitionen zur Compilezeit sichergestellt werden, dass zur Laufzeit niemals versucht wird, über eine Referenz auf eine Rolleninstanz Features dieser Rolle abzurufen, die der Typ dieser Rolle nicht besitzt.

Die Beispiele zur Modularisierung von Kollaborationen, zur Realisierung von Aspekten und zur Vererbungssemantik bei Rollen sollten einen Einblick in das Programmiermodell und in die Konzepte von Object Teams gegeben haben. Für eine detaillierte Beschreibung sei auf [12] verwiesen, teilweise werden weitere Features

des Modells bei der Beschreibung des Object Teams-Compiler auch in dieser Arbeit eingeführt.

Hinter den Kulissen

Damit die beschriebenen Beispiele implementier- und ausführbar sind, ist neben einem die Konstrukte akzeptierenden und übersetzenden Compiler und einer Laufzeitumgebung eine Object Teams-Infrastruktur nötig, die das Verhältnis von Rollen- zu Basisobjekt und umgekehrt organisiert. Dabei ist die Zuordnung dieser beiden Typen und deren „Umwandlung“ in den jeweils anderen Typ wichtig. Desweiteren wurde bereits auf die Möglichkeit der Aktivierung und Deaktivierung von Aspekten verwiesen, wodurch zur Laufzeit sowohl das Orginalverhalten als auch das durch den Aspekt modifizierte Verhalten verfügbar sein muss. Es muss also bei einem Methodenaufruf entscheidbar sein, ob für das Zielobjekt ein Aspekt-Kontext aktiv ist oder nicht.

Zuordnung von Rollen- zu Basisobjekt

Jedem Rollenobjekt muss genau ein Basisobjekt zugewiesen werden. Der Grund für diese Forderung kann man am ersten gegebenen Beispiel erkennen. Dabei wird `Role2` von `Base2` gespielt, wobei `Base2` ein Feld `b1` vom Typ `Base1` besitzt. Wenn in der Kollaboration ein `Role2`-Objekt verwendet und darauf die `get`-Methode aufgerufen wird, so muss dieser Aufruf immer an dasselbe `Base2`-Objekt delegiert werden, damit letzteres den Inhalt seines Feldes `b1` zurückgibt. Wäre dies nicht der Fall, so hätte die Methode `get` eines `Role2`-Objekts kein deterministisches Verhalten. Damit die Zuordnung von Rollenobjekt zu Basisobjekt gewährleistet ist, muss ein Team einerseits diese Zuordnung speichern, z.B. in einer Hash-Datenstruktur, andererseits muss bereits bei der Erzeugung von Rollenobjekten die Zuordnung zu einem Basisobjekt erfolgen, was die Anpassung der bzw. Erzeugung von Rollenkonstruktoren zur Folge hat.

Umwandlung von Rollen- zu Basisobjekt und umgekehrt

Wird im Zusammenhang mit dem ersten Beispiel auf ein Rollenobjekt vom Typ `Role2` die Methode `get` aufgerufen, so wird dieser Aufruf im *callout*-Fall an die Methode `getBase1` eines Objekts vom Typ `Base2` weitergeleitet, die ein Objekt vom Typ `Base1` liefert. Innerhalb der Kollaboration wird jedoch ein Objekt vom Typ `Role1` erwartet. Für das `Base1`-Objekt muss also sein `Role1`-Objekt ermittelt werden. Diese Aufgabe bearbeiten die sogenannten `liftTo`-Methoden, die als Argument ein Basisobjekt erhalten und das zugeordnete Rollenobjekt zurückliefern. Für beide Bindungsrichtungen (*callout* und *callin*) kann eine `liftTo`-Methode aus einer Datenstruktur die Rolle herausfinden, die einer Basis zugeordnet ist, wenn die Rolle bereits erzeugt und zugeordnet wurde. Ansonsten muss diese Rolle erst erzeugt, ihre Zuordnung zur Basis erstellt und anschliessend von der `liftTo`-Methode zurückgegeben werden. Dieser Vorgang wird als *lifting* bezeichnet. Die Auswahl eines Rollen-Objekts anhand eines Basis-Objekts wird komplexer, wenn sowohl Rollen- als auch Basisklassen in Vererbungsverhältnissen stehen und gleichzeitig mit

Object Teams aufeinander abgebildet werden. Abbildung 2 zeigt ein Beispiel für eine solche Situation:

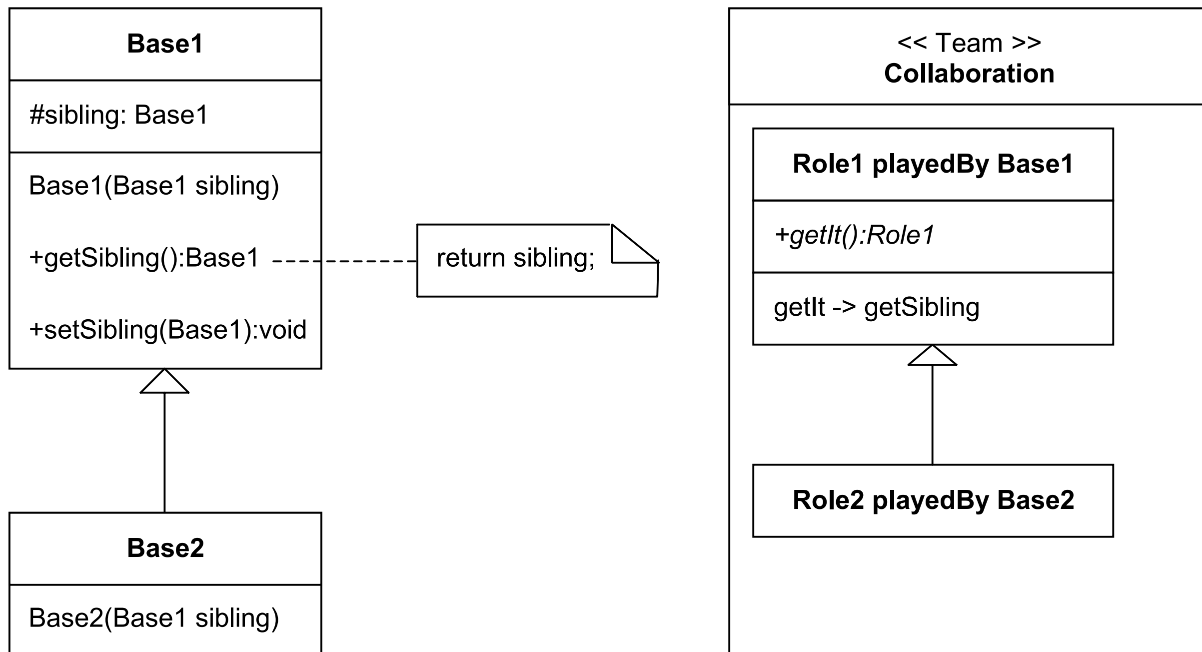


Abbildung 2: Vererbungshierarchie und Bindings

Für `Role2` muss die geerbte Methode `getIt` nicht mehr gebunden werden, da sie das Methoden-Binding von `Role1` erbt. Bei den Basisklassen ist es wichtig zu erkennen, dass das Feld `sibling` auf `Base1` typisiert ist, dynamisch jedoch ein Objekt vom spezielleren Typ `Base2` referenzieren kann. Wird an ein `Role1`-Objekt die Nachricht `getIt` geschickt, so wird diese durch das das Binding `getIt -> getSibling` an das zugehörige Basisobjekt weitergeleitet. Dabei könnte `getSibling` aufgrund des polymorphen Feldes `sibling` ein Objekt der Klasse `Base2` liefern, das zum Argument der `liftTo`-Methode `Role1 liftToRole1(Base1 b1)` wird. Es wird dabei nicht die ebenfalls verfügbare Methode `Role2 liftToRole2(Base2 b2)` ausgewählt, da der ursprüngliche Adressat der Nachricht vom Typ `Role1` war. Die `liftToRole1`-Methode muss nun erkennen, dass der aktuelle Parameter spezieller ist als der formale Parameter der Signatur und, gemäß dem Binding `Role2 playedBy Base2`, ein Objekt vom Typ `Base2` zurückliefern. Dieser Vorgang wird als *smart lifting* bezeichnet und stellt sicher, dass für ein gegebenes Basis-Objekt und gemäss der definierten Bindings stets das speziellst mögliche Rollenobjekt zurückgegeben wird.

Die entgegengesetzte Abbildungsrichtung, also für ein Rollenobjekt das dazugehörende Basisobjekt zu erhalten, ist trivial, da jede Rolle ihre Basis dadurch kennt, dass für jede Rolle ein Basis-Feld deklariert wird, welches in ihrem Konstruktor initialisiert wird. Sie entspricht der Dereferenzierung dieses Basis-Feldes und wird als *lowering* bezeichnet. Sowohl die `liftTo`-Methoden als auch das Basis-Feld einer

Rolle werden von Compiler erzeugt und sind in ihrer Verwendung für den Benutzer unsichtbar.

Kontextverwaltung

Jede Kollaboration erbt von der von Object Teams vordefinierten Klasse `Team`, wodurch sie die in `Team` definierten Methoden `activate` und `deactivate` erbt. Diese Methoden müssen bei der Aktivierung eines Teams dessen Existenz den Basisklassen im *callin*-Falle bekannt machen, damit der Aufruf einer Basismethode dem Modifier (*before*, *after*, *replace*) entsprechend umgeleitet werden kann. Dies wird dadurch erreicht, dass die Basisklassen zur Laufzeit dahingehend modifiziert werden, dass sie um eine Datenstruktur erweitert werden, welche die für sie aktiven Teams aufnehmen kann. Weiterhin muss sowohl das Originalverhalten als auch das modifizierte Verhalten verfügbar sein, was erneut ein Cashing-Verfahren nahelegt. Durch `deactivate` wird das jeweilige Team aus dieser Datenstruktur entfernt. Weitere Details der Kontextverwaltung sind Bestandteil einer Diplomarbeit, die z.Z. entsteht und können nach deren Anfertigung dort in Erfahrung gebracht werden.

Das folgende Kapitel beschreibt das generelle Konzept, das zur Implementierung des Object Teams-Modells erstellt wurde, und die Programmiersprache Java, die als Zielsprache ausgesucht wurde.

Object Teams und Java

Das Programmiermodell von Object Teams wurde bereits in den Interpreter-Sprachen LUA [11] und Ruby [25] realisiert. Als nächster Schritt sollte dies in einer Compiler-Sprache geschehen, wobei die Wahl auf die Sprache Java fiel. Die Gründe für diese Entscheidung werden deutlich, nachdem das Konzept für die technische Realisierung dargelegt ist.

Realisierungskonzept von Object Teams

Wie in den bisherigen Beispielen bereits gezeigt wurde, gibt es die zwei grundsätzlichen Bindungsrichtungen *callout* und *callin*. Im Falle von *callout* sind die Rollenmethoden des Teams, die gebunden werden sollen, typischerweise abstrakt, die jeweiligen Basismethoden konkret. Wenn ein Binding für die abstrakten Rollen vorliegt, dann kann das Team instantiiert werden, da den abstrakten Rollenmethoden durch das Binding die Implementierungen der Basismethoden zugewiesen wurden. Für den Compiler bedeutet dies, dass er für die – gegebenenfalls per *implicit inheritance* geerbten – Rollen des Teams, das die Bindings enthält, neue Methoden erzeugen kann. Diese sind nun nicht mehr abstrakt, sondern enhalten in ihrem Rumpf jetzt die jeweiligen Aufrufe an die Basismethoden. Wird dieses Konnektor-Team instantiiert, so werden die neu generierten Rollenmethoden benutzt. Der *callout*-Fall ist somit zur Compile-Zeit vollständig umsetzbar.

Der *callin*-Fall entspricht in seiner Zielsetzung der von AspectJ, nämlich die Modifikation bereits implementierten Verhaltens. Der AspectJ-Compiler spielt dabei die Rolle eines Prä-Prozessors, der AspectJ-Anweisungen interpretiert und die jeweiligen Klassen – im Sinne von Object Teams sind dies die Basisklassen –

anpasst. Man nennt diesen Vorgang auch *weaving*, im Sinne von „Code-Fragmente miteinander verweben". Das Ergebnis einer AspectJ-Compilierung ist somit veränderter Sourcecode. Dies bedeutet, dass für die Realisierung eines Aspekts bei AspectJ stets die Neu-Compilierung der Klassen (per *javac* oder eines anderen Java-Compilers) notwendig ist, da das *weaving* auf Sourcecode-Ebene, nicht auf Bytecode-Ebene stattfindet. Object Teams verfolgt einen anderen Ansatz: der Compiler erzeugt die für die Realisierung der Aspekte notwendigen Informationen im Java-Bytecode, das Laufzeitsystem kümmert sich um die Interpretation dieser Informationen und realisiert Aspektverhalten. Der Vorteil hierbei ist einerseits, dass bestehende Klassen (die auch in Bibliotheken von Drittanbietern vorkommen können), nicht neu compiliert werden müssen, andererseits, dass Aspekte selbst zur Laufzeit verändert, z.B. aktiviert und deaktiviert, werden können.

Die Programmiersprache Java für Object Teams

Aus dem vorherigen Absatz geht hervor, dass es zwei wesentliche Anforderungen an eine bestehende Programmiersprache gibt, damit diese als „Object Teams-kompatibel" gelten kann: einerseits muss der Compiler erweiterbar sein, also als Quellcode zur Verfügung stehen (für *callout*-Fall), andererseits muss das Laufzeit-system Eingriffsmöglichkeiten anbieten, die es erlauben, die vom Compiler zusätzlich generierten Binding-Informationen auszuwerten (für den *callin*-Fall). Eine weitere Anforderung bestand darin, das Modell möglichst in einer industrienahen Sprache zu testen, um auch reale Anwendungsszenarien durchspielen zu können.

Alle Anforderungen werden von der Sprache Java erfüllt:

- Der Standard-Java-Compiler (*javac*), der Bestandteil eines *Java Development Kit* (JDK) ist, liegt als Quellcode vor (nach Registrierung beziehbar unter [13]), er ist selbst in Java geschrieben.
- Das Laufzeitsystem von Java (*Java Runtime Environment,* JRE) erlaubt den Austausch des *System Class Loaders*, der die Klassen lädt, bevor sie vom JRE benutzt werden. Ein eigener *System Class Loader* kann dadurch die vom Compiler generierten *callin*-Informationen auslesen und entsprechend agieren.
- Java ist augenblicklich eine der meistgenutzten Programmiersprachen sowohl in der industriellen Softwareentwicklung als auch an Hochschulen.

Desweiteren musste für die Einbindung in das JRE kein eigener Code entwickelt werden, sondern man konnte auf *JMangler* [14], ein Framework zur Modifikation von Java-Bytecode zur Laufzeit, zurückgreifen. Schliesslich läßt es Java zu, Klassen als Features von Klassen zu definieren – sogenannte *inner classes* –, was eine direkte Abbildung von Teams auf Klassen und Rollen auf innere Klassen ermöglicht.

Bevor der *javac* und die für Object Teams notwendigen Modifikationen und Erweiterungen vorgestellt werden, erfolgt eine kurze Beschreibung der Sprache Java und deren Laufzeitumgebung.

Ausgewählte Eigenschaften der Programmiersprache Java

Java ist eine Programmiersprache, die von ihrem Hersteller Sun als „einfach, objektorientiert, verteilt, interpretiert, robust, sicher, architektur-neutral, portabel, schnell, nebenläufig und dynamisch“ [15] bezeichnet wird. Aus dieser reichhaltigen Liste sind für Object Teams vor allem die Attribute „objektorientiert“, „interpretiert“ und „dynamisch“ relevant. Objektorientiert bedeutet, dass Java das Konzept von Objekten als Instanzen von Klassen und Vererbung und damit Polymorphie unterstützt. Der Quellcode wird vom *javac* in ein Zwischenformat, den sogenannten *Bytecode*, übersetzt, der zwar einem plattformabhängigen Assemblercode ähnelt, allerdings von einer weiteren Software, der *Java Virtual Machine* (JVM), interpretiert wird. Dadurch wird die Architektur-Neutralität des Bytecodes erreicht, für jede konkrete Hardwareumgebung existiert jeweils eine plattformabhängige JVM.

Dynamisch an Java ist der Vorgang, wie benötigte Klassen in das ablaufende Programm integriert werden. Bei Sprachen wie C++ werden alle Elemente, die zum Programmablauf benötigt werden, zur Compile-Zeit statisch an das Programm gebunden (*gelinkt*), alle Sprungadressen sind somit bereits festgelegt. Java dagegen vertritt hier die sogenannte *open world assumption*, d.h. alle benötigten Elemente werden zur Laufzeit *dynamisch* geladen (z.B. auch über das Netz, deshalb auch das Attribut „verteilt“), und zwar erst dann, wenn sie tatsächlich benötigt werden. Dazu wurde in Java das Konzept des *Class Loaders* eingeführt.

Wird ein Java-Programm gestartet, so wird zuerst die JVM gestartet, die per *Class Loader* die erste Klasse, die das Java-Programm benötigt, lädt. Anschliessend führt die JVM die Anweisungen im Bytecode aus. Die JVM ist eine virtuelle Stack-Maschine mit den wiederum virtuellen Elementen CPU, dem Variablenstack, dem Javastack (Stack für einzelne Anweisungen), dem c*onstant pool* (ein Register für alle Konstanten, ähnlich einer Symboltabelle), einem Codesegment mit *program counter* und dem *heap* (Objekt-Pool). Stößt sie dabei auf Features unbekannten Typs, so lädt sie per Class Loader die Klasse dieses Typs nach.

Ein weiteres Charakteristikum, das in der obigen Aufzählung nicht vorkommt, für Object Teams jedoch immanent wichtig ist, könnte man als „erweiterbar“ bezeichnen. Java ist erweiterbar, da wie bereits erwähnt der Compiler selbst als Quellcode vorliegt und damit neue Sprachelemente verarbeitet werden können, andererseits bietet der Bytecode über die sogenannten *attributes* Erweiterungsmöglichkeiten, die von einer *Java Virtual Machine* ausgewertet werden können. Ein *attribute* kann eine benutzerspezifische Information sein, die einem Bytecode-Element zugewiesen werden kann. Zum Beispiel erzeugt der *javac* standardmässig ein s*ourcefile attribute*, das zu einer compilierten Klasse gehört. Enwicklungsumgebungen und die darin integrierten Debugger können dieses Attribut auswerten und damit den zugehörigen Sourcecode beim Ausführen einer Klasse anzeigen.

Weitere Details zur Spezifikation der JVM findet man unter [20] und [21].

Der Object Teams Compiler

Als Einleitung zu diesem Kapitel sei kurz daran erinnert, welche grundsätzlichen Elemente Teil eines Compilers sind. Detalliert und erschöpfend werden die Bestandteile und Compiler-Phasen im klassischen Compiler-Drachenbuch [16] beschrieben und erklärt.

Genereller Compiliervorgang

Am Anfang des Compiliervorgangs steht die lexikalische Analyse des Quellcodes. Dabei wird der einfache Text in sogenannte Tokens umgewandelt, wodurch z.B. von Leerzeichen und Einrückungen abstrahiert werden kann. Diese erste Phase wird vom *Scanner* durchgeführt. Die vom Scanner generierten Tokens erhält der *Parser* als Eingabe und prüft, ob der Quellcode syntaktisch korrekt ist. Grundlage dieser Prüfung ist die Syntax einer Programmiersprache. Der Parser prüft somit, ob das Programm als Wort aus dieser Syntax erzeugt werden kann. Resultat des Parsens ist ein abstrakter Syntaxbaum (*abstract syntax tree*, AST), in dem die Elemente des Programms baumartig angeordnet und somit für die folgenden Phasen einfacher zu verarbeiten sind. Es folgt die Attribuierungsphase, in der parallel zu Typüberprüfungen alle zur Codeerzeugung wichtigen Informationen gesammelt und an die Elemente des AST angehängt werden. Danach wird in der Flusskontroll-Phase versucht, ineffizienten Code zu erkennen und zu optimieren (z.B. *dead code*). Anschliessend erfolgt die Maschinencode-Generierungsphase und typischerweise das Schreiben dieses Codes in eine Datei.

Zwischen diesen Phasen können weitere Phasen eingeschoben sein, die von der zu compilierenden Sprache abhängen. Im Falle des *javac* sind dies die Umwandlung von Klassen mit generischen Elementen (vgl. *Generic Java*, [27]) zu konventionellen Java-Klassen und von inneren zu flachen Klassen, für die schliesslich ein jeweils eigenes *class file* erzeugt wird. Diese beiden Phasen schliesen sich im *javac* an die Flusskontroll-Phase an. Die Code-Generierungsphase wird im javac ersetzt durch eine Bytecode-Generierungsphase dar, in der der architektur-neutrale Bytecode generiert und in eine Datei geschrieben wird.

Analyse des Compilers

Der Java-Compiler *javac*, der für Object Teams angepasst wurde, ist Teil der Quellen des *Java Development Kit* (JDK) 1.3.0. Es handelt sich hierbei eigentlich um den Compiler *gjc* von Martin Odersky, der neben „konventionellen" Datentypen auch generische, parametrisierbare Datentypen unterstützt. Diese Funktionalität ist allerdings von Sun nachträglich „ausgeschaltet" worden und nicht mehr verfügbar. In der Beschreibung der Compiler-Phase „Sourcecode-Transformationsphase" wird auf das Thema Generizität noch einmal eingegangen.

Die erhältichen Quellen stellen alle Klassen des JDK dar, die Klassen des Compilers sind jedoch in ausschliesslich diesen Paketen enthalten:

```
com.sun.tools.javac.v8
com.sun.tools.javac.v8.code
com.sun.tools.javac.v8.comp
com.sun.tools.javac.v8.parser
com.sun.tools.javac.v8.tree
com.sun.tools.javac.v8.util
```

Um den Compiler für Object Teams erweitern zu können, musste dieser zuerst in seiner Struktur und Funktionalität verstanden werden. Da zu diesen sechs Paketen keinerlei Dokumentation vorlag, beschränkte sich die Grundlage der Analyse des Compilers auf die Quellen selbst. Dabei waren zwei Tools von enscheidender Bedeutung: ein Debugger und ein Decompiler. Durch den Debugger – der eher im Sinne eines „Code-Stepper“ und weniger als Korrekturwerkzeug verwendet wurde – ist es möglich, den Compilierungsvorgang an beliebigen Stellen anzuhalten, Variablenwerte auszulesen und Schritt für Schritt den Compilierungsvorgang fortzusetzen. Der Decompiler ermöglicht es, ausgehend von einer *class*-Datei den Compilierungsvorgang umzukehren. Dabei kann aus der compilierten Klasse wieder die Quelldatei gewonnen und dabei die Auswirkungen von eigenen Anpassungen am Compiler direkt eingesehen und kontrolliert werden. Es konnte somit z.B. überprüft werden, ob Object Teams-relevanter Code korrekt erzeugt wurde. Darüberhinaus bieten manche Decompiler als Feature auch eine Bytecode-Ansicht einer *class*-Datei an. Als Entwicklungsumgebung für den Object Teams-Compiler wurde *Eclipse* [17] (Build 20020321) verwendet, worin ein geeigneter Debugger integriert ist, als Decompiler kam *jad* als Decompiler-Engine [18] und *DJ Java Decompiler* [19] als graphisches Frontend von *jad* zum Einsatz. Durch u.a. diese Tools konnten die einzelnen Phasen des Compilers und die von ihm verwendeten Datenstrukturen identifiziert werden, die in den folgenden Abschnitten näher beschrieben sind.

Datenstrukturen des *javac*

Das Paket `com.sun.tools.javac.v8.tree` enthält Klassen, die die Datenstrukturen und Utility-Methoden für die vom Parser erzeugten AST bereitstellen. Die abstrakten Syntaxbäume werden nach ihrer Erzeugung in nahezu allen weiteren Compilierungsphasen wiederverwendet und bearbeitet. Damit stellen die Klassen im obigen Paket die Hauptdatenstruktur des Compilers dar. Die Klasse `Tree` spielt dabei eine zentrale Rolle, da sie als innere statische Klassen alle benötigten Datenstrukturen für die AST enthält. Darunter fallen Syntax-Bäume für z.B. eine Klassendefinition (`Tree.ClassDef`) , für eine For-Schleife (`Tree.ForLoop`) oder auch eine allgemeine Anweisung (`Tree.Exec`). Das hier verwendete *Design Pattern* – nach [26] – ist das *Composite Pattern*. Dabei ist ein *Leaf* ein primitiver Wert, der im AST enhalten ist, z.B. ein Name oder ein Integer-Wert. Das *Composite* kann eine der in `Tree` enthaltenen inneren Klassen sein, die sämtlich Subklassen von `Tree` sind. Das *Composite* wiederum aggregiert weitere *Components*, die wiederum entweder primitive Typen oder Subtypen von `Tree` sind. Der Grund der/des Entwickler(s), statische innere Klassen zu verwenden, kann die dadurch gewonnene Übersichtlichkeit gewesen sein, da es der inneren Klassen vierzig sind, die sich ansonsten alle in eigenen Dateien befinden müßten. Statisch sind diese inneren

Klassen deshalb, damit sie unabhängig von einer äußeren Tree-Instanz verwendbar, d.h. instantiierbar sind. *Static inner classes* werden im *javac* häufig verwendet. Auffallend am AST-Design des *javac* ist jedoch, dass eine abstrakte *Expression*-Klasse als *inner class* mit dem Namen `Tree.Expresion` fehlt, die Superklasse von z.B. `Tree.Exec`, `Tree.ForLoop` etc. sein könnte.

Die AST werden im weiteren Compilierungsverlauf mit relevanten Informationen augestattet, welche für die Erzeugung des Bytecodes wichtig sind. Das Paket `com.sun.tools.javac.v8.code` umfasst Klassen, die Elemente des Bytecodes auf Java-Klassen abbilden. Die wichtigsten sind diese: `Flags`, die alle möglichen *Modifier* von Java-Elementen definiert; `Pool`, die das Java-Element eines Bytecode-Constant Pool darstellt; `Symbol`, die Default-Implementierungen für ihre statischen inneren Klassen `ClassSymbol`, `MethodSymbol` etc. bereitstellt; `Scope`, die für ein Symbol alle in dessen Gültigkeitsbereich liegenden Symbole auflistet; `Type`, die alle für einen Typ relevanten Informationen kapselt. Desweiteren enthält das Paket die Klassen `ClassReader` und `ClassWriter`, mit denen sowohl bereits compilierte als auch uncompilierte Klassen eingelesen bzw. compilierte Klassen in eine Datei geschrieben werden können.

Die Phasen des *javac*

Abbildung 3 stellt die Phasen eines Compilierungsvorgangs mit dem *javac* als graphische Übersicht dar. Dabei repräsentieren die Rechtecke die einzelnen Phasen und enhalten im oberen Bereich deren Aufgabe und im unteren Bereich die zugehörige Klasse, die die Aufgabe der Phase hauptsächlich bearbeitet.

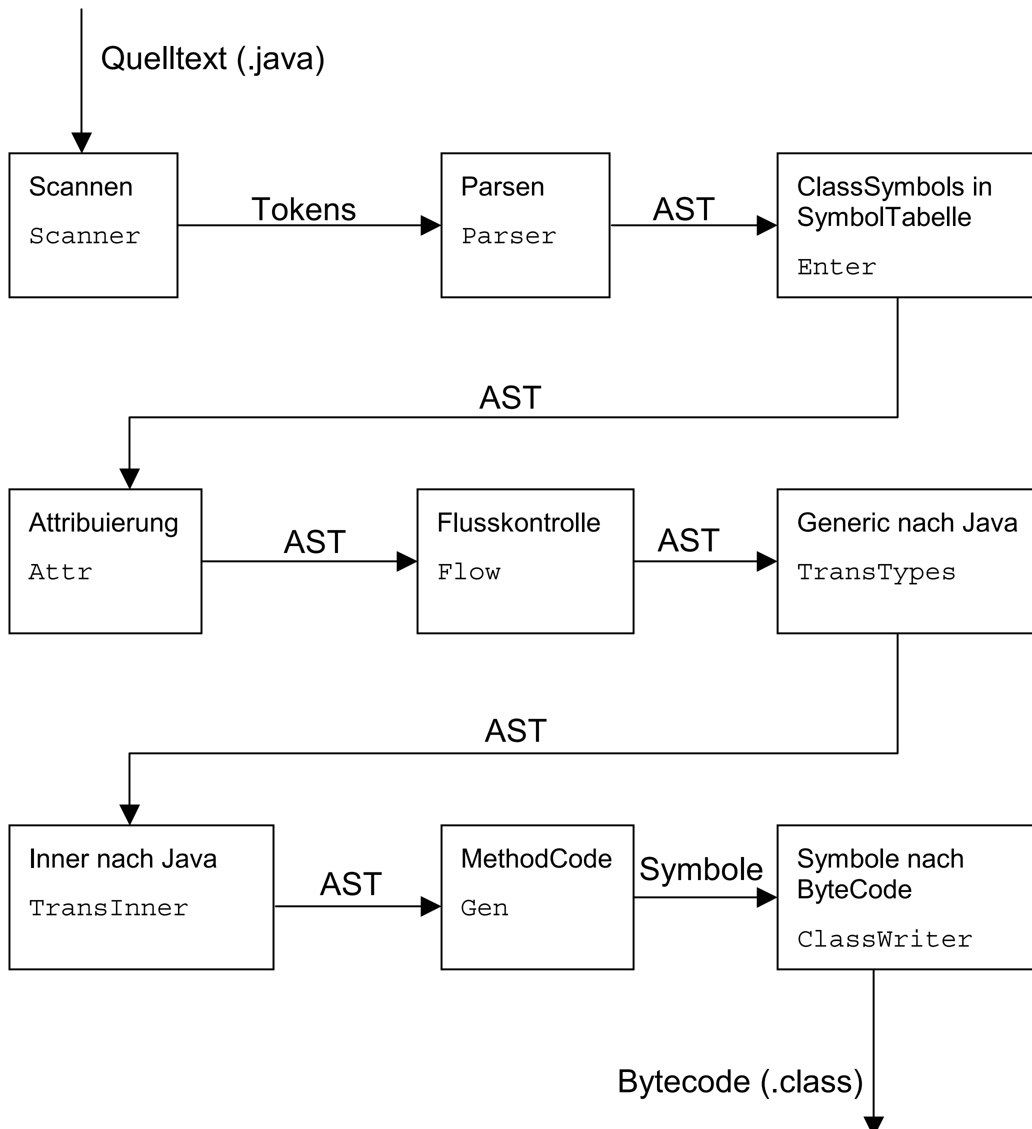


Abbildung 3: die Phasen des *javac*

Die einzelnen Phasen werden nun genauer beschrieben.

Lexikalische und syntaktische Analyse

Die Erzeugung der AST wird an eine *Factory* delegiert, deren Interface (`Factory`) ebenfalls in der Klasse `Tree` deklariert ist. Dieses Interface wird von der Klasse `TreeMaker` implementiert. Mit den in `Tree` definierten Typen und der Factory kann der Parser aus dem Quellcode die AST erzeugen. Voraussetzung ist die Umwandlung des Quelltextes in Tokens, die die Klasse `Scanner` durchführt. Sowohl

die Klasse `Scanner` als auch die Klasse `Parser` sind im Paket `com.sun.tools.javac.v8.parser` enthalten. Der Scanner benutzt zur lexikalischen Analyse das Interface `Tokens`, in dem alle Tokens als Integer-Konstanten enthalten sind. Im Scanner selbst werden sämtliche Schlüsselwörter in einem *static* Block definiert. Der Parser benutzt den Scanner, um sukzessive Tokens zu lesen. Dabei verfolgt er das Prinzip des rekursiven Abstiegs, wobei seine Methoden – mehr oder weniger genau – den in der Java Syntax ([20]) definierten Produktionsregeln entsprechen.

Nach der lexikalischen Analyse und dem Erzeugen der AST werden diese AST weiteren Bearbeitungsphasen unterzogen. Diese Phasen finden ihre Entsprechung in Klassen des Pakets `com.sun.tools.javac.v8.comp`. Zunächst werden die AST der Klasse `Enter` übergeben, um für alle AST-Elemente vom Typ `Tree.ClassDef`, also für alle Klassendefinition-Composite-Elemente des AST, ein Klassensymbol zu erzeugen, dieses in eine Symboltabelle einzutragen und das `ClassDef`-Composite mit diesem Klassensymbol auszustatten. In der anschliessenden Attribuierungsphase werden alle Elemente des AST mit weiteren Symbolen und anderen Informationen ausgestattet, welche im nächsten Absatz erläutert wird. Dabei wird zum ersten Mal vom einem weiteren, im *javac* neben *Composite* zentralen *Design Pattern* Gebrauch gemacht, dem *Visitor Pattern.* Kennzeichnend für dieses Muster ist die Möglichkeit, Definition und Bearbeitung einer Datenstruktur voneinander zu trennen. Die zu bearbeitende Datenstruktur stellt im *javac* der AST dar, ein im *javac* verwendeter Visitor ist die Klasse `Attr`.

Attribuierungsphase

In der Attribuierungsphase, die von der Klasse `Attr` durchgeführt wird, sind folgende Unteraufgaben enthalten: Namensauflösung von Elementen anhand ihres Gültigkeitbereiches, Typüberprüfungen aller Ausdrücke und Konstantenersetzung. Nebeneffekt dieser Aufgaben ist die Ausstattung des AST mit weiteren Symbolen für Methoden und Variablen und mit Typinformationen. Das Traversieren des AST wird durch das Erben von der abstrakten Klasse `Tree.Visitor` erreicht. Die Methode, die ein AST innerhalb seiner Visit-Methode auf dem übergebenen Visitor – in diesem Fall auf einer Instanz der Klasse `Attr` – aufruft, heisst immer `_case`, sowohl in der Klasse `Attr` als auch in allen weiteren Klassen, die von der Klasse `Tree.Visitor` erben. Die `_case`-Methoden stellen gemäss dem *Visitor Pattern* die `accept`-Methoden dar und werden für jeden AST-Typ innerhalb der Klasse `Tree.Visitor` bereitgestellt.

Flusskontrollphase

Die sich anschliessende Flußkontrollphase wird von der Klasse `Flow` durchgeführt, welche ebenfalls von `Tree.Visitor` erbt und zwei Hauptaufgaben implementiert: erstens die Überprüfung von Codezeilen auf Kontrollfluss-Erreichbarkeit, zweitens das Finden von Optimierungsmöglichkeiten. Ein *catch block* in der Sprache Java ist z.B. nicht erreichbar, wenn nicht wenigstens eine Anweisung im vorherigen *try block* die entsprechende *Exception* werfen kann. Eine mögliche Optimierung ist das Eliminieren von *dead code*, also von Anweisungen, die niemals durchlaufen werden. Ein Beispiel hierfür ist der *then part* einer Konditional-Anweisung, wenn der

Wächterausdruck immer nur zu *false* ausgewertet werden kann und kein Element des Wächterausdrucks mit dem Modifier *volatile* versehen ist. Dies ist im Zusammenhang mit Threads von Bedeutung, wenn diese auf gemeinsame Variablen zugreifen. Ist eine gemeinsame Variable z.B. ein boolsches Flag, das im Laufe einer Thread-Methode auf *false* gesetzt wird und in der gleichen Methode nach der *false*-Zuweisung einen Wächterausdruck darstellt (im Sinne von `if (flag) {...}`), so würde die Flusskontrolle die *if*-Anweisung mit dem *then*-Teil eliminieren und nur den *else*-Teil übrig lassen, da der Compiler nicht erkennt, dass `flag` von mehreren Threads nebenläufig veränderbar ist. Der Modifier *volatile* zeigt dies dem Compiler explizit an und die Code-Eliminierung durch die Klasse `Flow` wird unterbunden.

Sourcecode-Transformationsphase

Zwei Konstrukte der Sprache Java müssen in „konventionellen“ bzw. „flachen“ Javacode umgewandelt werden, bevor daraus Bytecode generiert werden kann: *generic Java* und *inner classes*.

Wie bereits erwähnt ist der *javac* auf gererische Typen, also parametrisierbare Datentypen vorbereitet. Nach augenblicklichem Stand wird dieses Feature für das JDK 1.5 im Jahr 2003 von Sun „freigeschaltet“. Es ist damit möglich, nach einer Deklaration `Vector<String> v` in Anweisungen auf Typecasts wie in `String s = (String)v.firstElement()` zu verzichten. Der Compiler prüft durch die `Tree.Visitor`-Subklasse `TransTypes` die Typkorrektheit der Anweisungen und wandelt sie dabei in konventionellen Javacode um, z.B. fügt sie die ensprechenden Typecasts in den Code ein.

Eine Javaklasse kann neben Methoden und Feldern auch innere Klassen als Features besitzen. Diese inneren Klassen werden in einer eigenen Phase in konventionelle Klassen umgewandelt, die einen Verweis auf ihre äußere Klasse enthalten. Eine Konsequenz daraus ist, dass auch innere Klassen letztlich in einer eigenen Datei definiert sind. Der Visitor `TransInner` führt diese Transformation durch.

Bytecode-Generierungsphase

Die letzten beiden Phasen sind die Generierung des Code-Attributs einer Methode und das Schreiben des Bytecodes in eine *class*-Datei. Ein Attribut ist ein Element des Bytecodes, wobei Klassen, Methoden und Felder Attribute besitzen können. Im Zusammenhang mit der *callin*-Funktionalität von Object Teams werden Attribute später genauer erklärt, hier sei nur das Code-Attribut einer Methode erwähnt, das die Anweisungen dieser Methode als Array von Bytecodes enthält. Dieses Array wird in einem ersten Schritt durch die Visitorklasse `Gen` erzeugt, schliesslich erzeugt eine Instanz der Klasse `ClassWriter` den Bytecode für Variablen, den Constant Pool und alle anderen Elemente und schreibt diesen in eine Datei.

Es werden nun die für Object Teams relevanten Modifikationen am Compiler beschrieben, die nach der Analyse der bestehenden Funktionalität möglich wurden. Dabei waren Compiler-Analyse und –Modifikation jedoch keine zeitlich scharf zu trennenden Phasen, vielmehr stellten sie sich als sich abwechselnde Perioden dar,

wobei durch Modifkationen und deren Auswirkungen das Bestehende besser verstanden werden konnte, was wiederum zu weiteren Modifkationen führte, usw.

Compiler-Modifikationen für Object Teams

Die Modifikationen betrafen alle Phasen des Compiliervorgangs mit Ausnahme der Flusskontrollphase. Bevor der Compiler selbst Ziel der Anpassung wurde, mussten die Ausdrucksmöglichkeiten des Object Teams-Programmiermodells formal erfasst und für sie eine Beschreibung in Form einer Grammatik erzeugt werden.

Spracherweiterung

Da das Object Teams-Programmiermodell in die Sprache Java integriert werden sollte – und damit nicht als separater Programmteil wie z.B. AspectJ-Elemente von einem Prä-Prozessor ausgewertet werden sollte – mussten die Object Teams-Sprachkonstrukte direkt in die Java Syntax integriert werden, d.h. das Programmiermodell sollte eine Erweiterung der Sprache Java darstellen. Deshalb wurden der existierenden LL(1) Java Syntax, die unter [20] einsehbar ist, neue Produktionsregeln hinzugefügt und bestehende Regeln angepasst und erweitert. In Abbildung 4 sind die Syntaxmodifikationen aufgelistet. Dabei sind folgende Punkte zu beachten:

- „...“ zeigt an, dass die umgebende Regel Teil der Java Syntax ist und um Object Teams-Elemente erweitert wurde.
- Die einzelnen Zeilen einer Regel sind implizit mit ODER logisch verknüpft, ausser sie sind eingerückt, dann sind sie als Fortsetzung der darüber begonnenen Alternative zu verstehen.
- Elemente mit grossem Anfangsbuchstaben stellen Nonterminal-Symbole dar, kleingeschriebene Elemente sind terminal.
- Geschweifte Klammern wurden in Anführungszeichen gesetzt, wenn sie als Elemente der Menge der Terminal-Symbole verwendet wurden, ansonsten sind sie Teil der Syntax-Beschreibungssprache und zeigen an, dass das geklammerte Element null-, einmal oder mehrmals vorkommen kann.
- Symbole, die mit eckigen Klammern umgeben sind, sind optional.
- Fett gedruckte Regeln sind neue Object Teams-Regeln.

```
Modifier:
  ...
  team
  open
  callin
ClassDeclaration:
  class Identifier [extends Type] [implements TypeList]
    [playedBy Identifier] ClassBody
ClassBody:
  { '{' ClassBodyDeclaration '}' }
ClassBodyDeclaration:
  ...
  [BindingDeployment]
FormalParameter:
  [final] Type [LiftToRoleFormalParameter] VariableDeclaratorId
LiftToRoleFormalParameter:
  as Type
BindingDeployment:
  BoundRoleMethods BindingDirection [BindingModifier]
    BoundBaseMethods [BindingParameterMappingsClause] ;
BindingModifier:
  replace
  after
  before
BoundRoleMethods:
  BoundMethods
BoundBaseMethods:
  BoundMethods
BoundMethods:
  Identifier [BindingFormalParameters]
    '{' Identifier [BindingFormalParameters]
      {,Identifier [BindingFormalParameters]} '}'
BindingFormalParameters:
  ( FormalParameter {,FormalParameter} )
BindingParameterMappingsClause:
  with '{' BindingParamterMapping {,BindingParamterMapping} '}'
BindingParamterMapping:
  Expression -> Identifier
  Identifier <- Expression
BindingDirection:
  -> | => | <-
Statement :
  ...
  InClause
InClause:
  in ( Identifier ) do Block
```

Abbildung 4: Object Teams-Erweiterungen der Java Syntax

Nach der Syntaxerweiterung konnte die eigentliche Software-Erweiterung beginnen. Dabei wurden als Compiler-Input zwei Mengen von Beispiel-Klassen verwendet, wobei jede Menge ein Object Teams-Anwendungsszenario darstellte, ein *callout*-Szenario und ein *callin*-Szenario. Durch die dabei auftretenden Compiler-Fehler konnten sukzessiv Erweiterungen und Anpassungen vorgenommen werden, bis im ersten Schritt keine Fehler seitens des Compilers mehr gemeldet wurden und im zweiten Schritt der entsprechende Bytecode generiert wurde. Die „Hauptroutine“ des Compilers stellte hierbei ein Orientierungshilfe dar, wie weit der Compilierungsvorgang fortgeschritten ist und welche Phasen noch zu durchlaufen sind. Diese Routine stellt die Methode `public List compile(List filenames) throws Throwable` der Klasse `JavaCompiler` dar. Diese Klasse befindet sich im Paket `com.sun.tools.javac.v8`, das weitere mögliche Haupteinstiegspunkte zur Benutzung des Compilers als eigene Klassen enthält.

Da am Anfang die syntaktische und lexikalische Analyse steht, mußten Scanner und Parser bearbeitet werden. Da jedoch beide auf bestehenden Datenstrukturen aufbauen – Tokens und AST – mußten diese Datenstrukturen zu allererst entsprechend den Object Teams-Anforderungen angepasst und erweitert werden.

Erweiterung der Datenstrukturen des *javac*

Wie oben beschrieben enthält die Klasse `Tree` als innere Klassen sämtliche AST. Für Object Teams mussten die folgenden Subklassen hinzugefügt werden: `Tree.BindingDeployment`, `Tree.BindingParameterMapping`, `Tree.BoundMethod` und `Tree.In`. Beispielhaft soll das Konstrukt `Tree.BindingDeployment` vorgestellt werden:

```
public static class BindingDeployment extends Tree {
    // Liste von auf Rollenseite gebundenen Methoden
    // List of Tree.BoundMethod
    public List r_methods;
    public int b_dir;
    public Name b_mod;
    // Liste von auf Basisseite gebundenen Methoden
    // List of Tree.BoundMethod
    public List b_methods;
    // Liste von Parameterabbildungen
    // List of Tree.BindingParameterMapping
    public List param_mappings;

    public BindingDeployment(List r_methods, int b_dir,
                             Name b_mod, List b_methods,
                             List param_mappings) {
        super(BINDING_DEPLOYMENT);
        // Zuweisung der Parameter an die jeweiligen Felder
        this.r_methods = r_methods;
        ...
        this.param_mappings = param_mappings;
    }
    public void visit(Visitor v) {
        v._case(this);
    }
}
```

Ein Objekt der Klasse Tree.BindingDeployment erfasst beispielsweise die Informationen der folgenden Codezeile:

```
r_m1(int i) <- before {b_m1(int i1,int i2), b_m2(int i1,int i2)}
    with {
        i <- i1;
    }
```

Für die beim Binding-Deployment beteiligten Rollen- und Basismethoden (im Beispiel r_m1, b_m1 und b_m2) stellt die Klasse Tree.BindingDeployment die Felder r_methods und b_methods zur Verfügung. Das Feld b_dir erfasst die Bindungsrichtung (*callout* oder *callin*, im Beispiel <-, also *callin*), b_mod den Bindings-Modifier (before). Die Liste param_mappings nimmt Objekte der Klasse Tree.BindingParameterMapping auf, die Parameterabbildungen wie i auf i1 erfassen können. Desweiteren ist neben dem Konstruktor die von den meisten Compilerphasen benutzte visit-Methode definiert.

Für die neuen AST-Klassen wurden die Schnittstellen Tree.Visitor und Tree.Factory erweitert und die entsprechenden Erzeugungsmethoden in der Factory TreeMaker hinzugefügt.

Für alle weiteren Datenstruktur-Anpassungen war weniger Aufwand nötig, es sind die folgenden:

- In der Klasse Tokens mußten die Object Teams-relevanten Tokens wie z.B. TEAM oder PLAYEDBY hinzugefügt werden.
- Durch die Methode enterKeyword der Klasse Scanner konnten alle Object Teams-Schlüsselwörter hinzugefügt werden.
- In der Klasse Flags mussten die Object Teams-Modifier wie team oder replace hinzugefügt werden. Dabei war zu beachten, dass die Modifier mit Integer-Konstanten belegt wurden, die Zweier-Potenzen darstellen, da im Compiler oft von Bit-Operatoren Gebrauch gemacht wird, wodurch effizient auf das Vorhandensein eines Operators geprüft werden kann. So prüft z.B. der Ausdruck `methodSymbol.flags & OPEN != 0`, ob das Methodensymbol methodSymbol mit dem Modifier open versehen ist.
- In der Klasse Symbol wurden die inneren Klassen Symbol.CallinBinding, Symbol.CallinMethodMapping und Symbol.CallinParamMapping hinzugefügt. Objekte dieser Klassen stellen im ersten Fall ein Feld eines Teams, also eines Klassensymbols, im zweiten Fall ein Feld einer Rolle, d.h. eines Symbols einer inneren Klasse (also ebenfalls eines Klassensymbols) und im dritten Fall ein Feld eines Methodensymbols dar. Falls *callin*-Bindings definiert werden, dann kapseln diese Objekte alle Informationen, die für die Auswertung zur Laufzeit notwenig sind. In einer späteren Phase werden diese Informationen als Attribute zu Bytecode transformiert. Falls keine *callin*-Bindings für das jeweilige Symbol definiert werden, sind diese Felder null.

Durch diese Erweiterungen der Datenstrukturen waren die Voraussetzungen für die Modifikation der Klassen, die bestimmte Compilerphasen implementieren, erfüllt.

Abbildung 5 stellt die bereits bekannten *javac*-Compilerphasen plus die durch die Object Teams-Modifikationen zusätzlich enstandenen Phasen dar, wobei letztere mit einem breiteren Rand versehen sind. Die erste Zeile einer Object Teams-Phase zeigt an, ob sie bei einer Teamdefinition unabhängig von Bindings (Team), bei einer Teamdefinition mit *callout*-Bindings (Callout) oder bei einer solchen mit *callin*-Bindings (Callin) durchlaufen wird. Ab der zweiten Zeile wird kurz die Aufgabe der Phase umschrieben, in der letzten wird die Klasse genannt, die die Phase hauptsächlich implementiert.

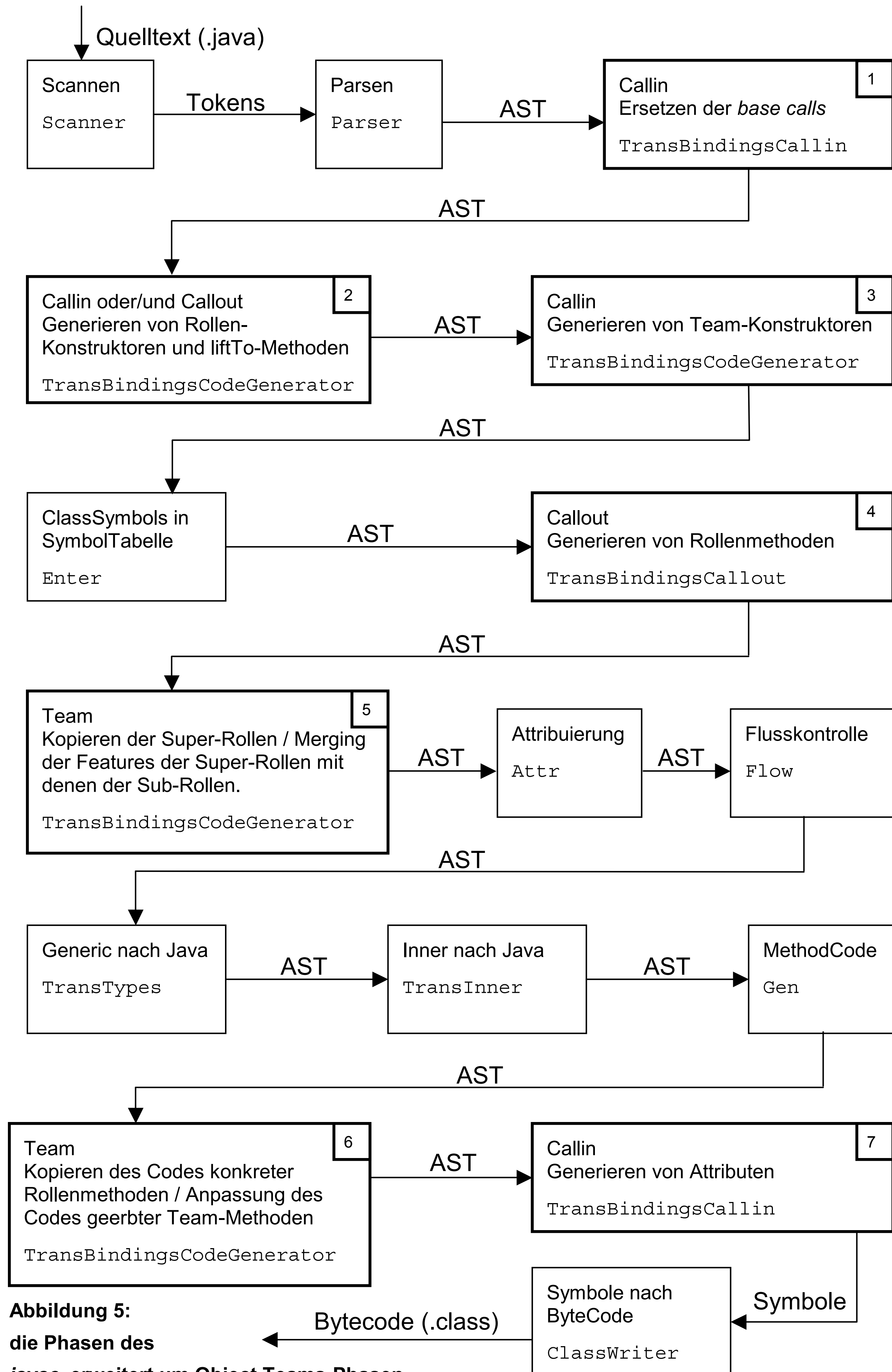


Abbildung 5:
die Phasen des
***javac*, erweitert um Object Teams-Phasen**

Als erste Phase wurde das Parsen angepasst.

Erweiterung der lexikalischen und syntaktischen Analyse

Wie bereits erwähnt nutzt der Parser den Scanner, um einzelne Tokens zu lesen. Ausser den oben beschriebenen Erweiterungen für die Klassen `Tokens` und `Scanner` war für die lexikalische Analyse nichts anzupassen. Für die syntaktische Analyse waren die oben beschriebenen Syntaxerweiterungen in den Methoden der Klasse `Parser` umzusetzten. Dazu mussten neue Parser-Methoden implementiert und die Aufrufe dieser neuen Methoden in bestehende Methoden eingefügt werden. Dies war relativ trivial, wenn die Stelle des Aufrufs sich als unabhängig von Vorbedingungen herausstellte, die im Rumpf der bestehenden Methode eventuell definiert waren. Dies war der Fall, wenn der Aufruf im Rumpf der bestehenden Methode als einfache Anweisung implementiert werden konnte; es war nicht der Fall, wenn sich die Aufrufstelle in einem „Wald" von `if`- und `switch`-Anweisungen befand. Diese Stelle war durch Debugging und entsprechende Beispiel-Input-Dateien schwieriger zu finden. Letzteres war allerdings seltener der Fall, da die Syntax relativ direkt auf die Parser-Methoden abgebildet wurde und Fallunterscheidungen meist durch explizit-optionale Regeln realisiert wurden. Dabei kann eine if-Anweisung auf Syntax-Seite durch einen optionalen Regelbestandteil ausgedrückt werden, auf Parser-Seite wird auf jeden Fall die entsprechende Methode ausgeführt, und erst in dieser erfolgt die Fallunterscheidung per `if`-Anweisung.

In den jeweiligen Parser-Methoden findet immer dann, wenn genügend Informationen für einen AST zusammengetragen wurde, der Aufruf an die *Tree Factory* statt, um eine AST-Instanz zu erzeugen. Ist dieser Vorgang für den gesamten Quelltext durchgeführt und hat der Parser das Programm als Wort seiner Grammatik akzeptiert, so werden im *javac* die AST direkt an eine Instanz der Attribuierungsklasse `Attr` übergeben. Aufgrund der Object Teams-Anforderungen mussten verschiedene Code-Generierungsaktivitäten eingeschoben werden, die Thema der nächsten Abschnitte sind.

Object Teams-Codegenerierung

Es gab bei der Generierung von zusätzlichem bzw. Anpassung von existierendem Sourcecode fünf Hauptaufgaben:

- bei Teams mit *callin bindings* die Ersetzung aller im Quelltext auftauchenden *base calls* (1),
- bei Teams mit *callin* und/oder *callout bindings* die Erzeugung der Rollen-Konstruktoren und `liftTo`-Methoden (2),
- bei Teams mit *callin bindings* die Erzeugung der Team-Konstruktoren (3),
- bei Teams mit *callout bindings* die Erzeugung der dem Binding-Deployment entsprechenden neuen Rollenmethoden (4),

- bei Teams mit oder ohne Bindings das Kopieren aller Rollen des Super-Teams in das zu compilierende Team bzw. das „Zusammenmischen“ aller Features der Rollen des Super-Teams mit den Features der Rollen des zu compilierenden Teams, was in Abbildung 5 als *Merging* bezeichnet wird (5).

Die meisten dieser Erzeugungen mussten zu diesem Zeitpunkt geschehen, da die Anzahl und Komplexität der dafür notwendingen Elemente, die ja sämtlich „per Hand“ konstruiert werden mussten, in dieser Phase noch überschaubar ist. So musste in den meisten der obigen Punkte ein AST, also eine Subklasse von `Tree` nach dem *Composite Design Pattern* aufgebaut werden. Anhand einer Instanz von `Tree.MethodDef` sei dies veranschaulicht. Diese benötigt u.a. einen Namen, einen Rückgabewert, eine Liste von Parametertypen und einen Methodenblock. Die letzten drei Elemente sind nach dem *Composite* Muster *Composite*-Elemente, also zusammengesetzt. Jedes Element der Parameterliste ist vom Typ `Tree.VarDef`, das wiederum dem *Composite* Muster enspricht. Die Komplexität des Methodenblocks, der vom Typ `Tree.Block` ist, hängt gänzlich von den im Block vorkommenden Anweisungen ab, die ebenfalls dem *Composite* Muster entsprechen. Die Erzeugung eines AST für z.B. diese Methode

```
public Role1 liftToRole1(Base1 base1) {
    Role1 role1;
    if(!cache.containsKey(base1)) {
        role1 = new Role1(base1);
        cache.put(base1, role1);
    } else {
        role1 = (Role1)cache.get(base1);
    }
    return role1;
}
```

entspricht 92 Zeilen Object Teams-Compilercode. Der entsprechende AST enthält dabei noch keine detaillierten Typinformationen, auch ist er noch nicht komplett mit Symbolen ausgestattet (dies findet erst in der Attribuierungsphase statt). Dieses Beispiel sollte die Notwendigkeit, das Generieren neuer Object Teams-Elemente möglichst frühzeitig durchzuführen, deutlich gemacht haben. Die AST-Elemente, die notwendigerweise nach dem Parsen vorhanden sein müssen, wurden „per Hand“ erzeugt, die restlichen Informationen konnten vom Compiler meist selbst erzeugt und inferiert werden.

Die oben aufgelisteten Aufgaben werden nun im Detail erläutert.

callin: die Ersetzung der *base calls* (1)

Das bereits zitierte Object Teams-*callin*-Beispiel enthält die Methode `callin Role2 checkCredentials(String id, String passwd)`, diese wiederum in der vierten Zeile einen *base call*, also einen Aufruf der Methode des Basis-Objekts, die über das Binding an diese Rollenmethode gebunden ist. Dieser *base call* wird vom Object Teams-Compiler durch einen rekursiven Aufruf ersetzt, in diesem Beispiel wird `base(id, passwd)` durch `checkCredentials(id, passwd)` ersetzt. Damit dies zur Laufzeit nicht in einer Endlosschleife resultiert, muss das Object Teams-RE (Runtime

Environment) diesen Aufruf entsprechend des definierten Bindings durch `base.createAccess(id, passwd)` ersetzen. Diese spezielle Kodierung eines *base call* ist dadurch begründet, dass der Compiler den Aufruf an das Basisobjekt noch nicht auflösen kann, da ein Team unabhängig vom Binding compiliert werden kann. Der rekursive Aufruf dagegen wird vom Compiler akzeptiert.

Erzeugung der Rollen-Konstruktoren und `liftTo`-Methoden (2)

Werden in einer Klasse, die mit dem Modifier `team` ausgestattet ist, Rollen definiert, die ihrerseits Bindings definieren, so erhält das Team eine Cache-Definition (aktuell vom Typ `java.util.HashMap`), die alle Basis-Rollen-Instanz-Paare aufnimmt und damit sicherstellt, dass jeder Rolle nur eine und stets dieselbe Basis zugeordnet wird. Für jede Rolle werden zusätzlich – falls noch nicht existierend – Konstruktoren nach den Mustern `<RoleName>()` und `<RoleName>(<BaseType> b)` erzeugt, der erste Konstruktor allerdings nur dann, wenn auch die Basis über einen parameterlosen Konstruktor verfügt. Schliesslich werden `liftTo`-Methoden erzeugt, die für eine Basis die zugehörige Rolle liefern, sie haben die allgemeine Signatur `<RoleType> liftToRole(<BaseType> b)`.

callin: die Erzeugung der Team-Konstruktoren (3)

Ein wichtiges Feature von Object Teams ist die Aktivierung von Bindings – und damit das Aktivieren und Deaktivieren von Aspekten – zur Laufzeit der Software. Zur Realisierung dieses Features wurde dessen Implementierung zuerst im Konnektor realisiert. Dabei wurde von folgender Überlegung ausgegangen: wenn ein Konnektor instantiiert wird, muss er im Falle von *callin* die Informationen tragen, welche Rollen- und Basisklassen gebunden wurden, damit das Object Teams-RE die entsprechenden Code-Transformationen durchführen kann. Diese Informationen werden dem generierten Konstruktor mitgegeben und im *mapping cache* hinterlegt, der in der Klasse `Team` definiert wurde (von der alle Teams erben), wobei gemäss der Binding-Definition einer Rollenklasse eine Basisklasse zugewiesen wird. Anhand dieses Caches kann das Object Teams-RE bei *callin* entscheiden, ob für eine Klasse ein aktiver Konnektor vorliegt und damit, ob das Verhalten dieser Klasse modifiziert werden muss.

Dieser Ansatz wurde jedoch verworfen. Der neue Ansatz sieht vor, die Verwaltung der für eine Basis aktiven Teams der Basis selbst zu überlassen, womit das Object Teams-RE die dafür nötige Infrastruktur selbst generieren muss. Somit ist davon auszugehen, dass diese Object Teams-Phase komplett wegfallen wird.

callout: die Erzeugung der Rollenmethoden (4)

Im *callout*-Fall sind die Rollenmethoden einer Kollaboration typischerweise abstrakt, die Kollaboration somit erst instantiierbar, nachdem diese abstrakten Methoden an konkrete Basis-Methoden gebunden wurden. Das definierte Binding wird vom Object Teams-Compiler folgendermassen umgesetzt: in einem vorherigen Schritt wurden bereits den Rollen in der Teamklasse, in der das Binding definiert wurde, neue Features hinzugefügt, insbesondere die `liftTo`-Methoden (siehe (2)). Die geerbten

abstrakten Rollenmethoden werden nun gemäss des Bindings implementiert. Wurde z.B. für eine Rolle `Role1` eine abstrakte Methode folgendermassen deklariert:

```
abstract public Role1 m(Role2 r2);
```

und liegt das folgende Binding vor:

```
...
Role1 playedBy Base1 {
    m -> concr_m;
    ...
}
Role2 playedBy Base2 {...}
...
```

so resultiert dies bei einer Signatur der Methode `concr_m` der Basis `Base1`:

```
Base1 concr_m(Base2 b2)
```

in der folgenden vom Compiler generierten Implementierung:

```
public Role1 m(Role2 r2) {
    return liftToRole1(base.concr_m(r2.base));
}
```

Diese Implementierung hat den Effekt, dass zuerst der aktuelle Rollen-Parameter `r2` „gelowered“ wird, es wird dabei dessen Basis-Feld dereferenziert. Anschliessend wird diese Basis als Parameter für die Basis-Methode verwendet, an die `m` gebunden wurde, also für `concr_m`. Da `concr_m` ein Basis-Objekt zurückgibt, muss dieses zur seinem Rollenobjekt „geliftet“ werden, bevor es `m` letztlich zurückgeben kann.

Die im Beispiel verwendete Methodenabbildung `m -> concr_m` kann nur deshalb angewendet werden, da `m` und `concr_m` bezüglich ihrer Rückgabewerte und formalen Parameter „passen“. Um genauer zu fassen, was „passen“ bedeutet, kann man sich anhand eines kleinen Beispiels daran erinnern, wann in der standard-objektorientierten Welt Parameter zueinander passen.

Würde man eine dem vom Compiler generierten Code ähnliche Methode ohne Object Teams implementieren:

```
public Type1 m(Type2 t2) {
    return concr_m(t2);
}
```

so wäre klar, unter welchen Umständen `m` typ-sicher benutzt werden kann:

- der Parametertyp von `m` ist konform zum Parametertyp von `concr_m`, und
- der Ergebnistyp von `concr_m` ist konform zum Ergebnistyp von `m`,

wobei unter konform „typ-gleich oder spezieller“ zu verstehen ist.

Dies läßt sich auf Object Teams übertragen, wobei die Menge der zueinander konformen Typen durch zwei Regeln erweitert wird:

- die Menge der konformen Parameter-Typen wird durch den Typ erweitert, der Ergebnis des *lowering* ist, und
- die Menge der konformen Rückgabe-Typen wird durch den Typ erweitert, der Ergebnis des *lifting* ist.

Der erste Zusatzpunkt kommt im vom Compiler generierten Methodenrumpf durch den Ausdruck `r2.base`, der zweite Zusatzpunkt durch `liftToRole1(...)` zum Ausdruck. Die Anwendung von *lifting* und *lowering* und die Regeln, wann diese angewendet werden, wird *translation polymorphism* genannt.

Sind Parameter und Rückgabewerte nicht konform, so können sogenannte *parameter mappings* verwendet werden. Auch hier zur Illustration ein Beispiel. Bei der folgenden Rollenmethoden-Signatur:

```
abstract public int m(Integer integer);
```

und dieser Basismethode-Signatur:

```
public Integer concr_m(int i) { ... }
```

ist ein Binding wie im obigen Beispiel nicht möglich, da sowohl Rückgabewert als auch der formale Parameter nicht konform sind. Durch ein *parameter mapping* können die Methoden trotz nicht-konformer Signaturen aufeinander abgebildet werden:

```
int m(Integer integer) -> Integer concr_m(int i) with {
    integer.intValue() -> i,
    result <- result.intValue()
};
```

Im Block nach dem Schlüsselwort `with` sind die Parameterabbildungen enthalten. Dabei steht das Object Teams-Schlüsselwort `result` für den Rückgabewert, der umgedrehte Pfeil in der zweiten Parameterabbildung soll die in diesem Fall umgekehrte Datenflussrichtung andeuten. Innerhalb eines *parameter mapping block* können auch beliebige Ausdrücke stehen, was folgendes Beispiel andeuten soll:

```
    integer.intValue() + 10 -> i,
    result <- other_m(result)
```

In der zweiten Zeile wird zuerst eine weitere Rollenmethode mit dem Rückgabewert aufgerufen, bevor dieser an `m` zurückgegeben wird. Das letzte Beispiel resultiert in der folgenden vom Object Teams-Compiler erzeugten Implementierung von `m`:

```
return other_m(base.concr_m(integer.intValue() + 10));
```

Die letzte Generierungsaufgabe vor der Attribuierungsphase stellt das Kopieren aller Rollen eines Super-Teams in das aktuell zu compilierende Team dar, wenn diese noch nicht im Team existieren. Falls sie bereits existieren, muss ein *Merging* der Features der Rollen des Super-Teams mit den Features der bereits definierten Rollen erfolgen. Diese Object Teams-Phase realisiert den ersten Teil der Umsetzung der oben beschriebenen *implicit inheritance*, den zweiten Teil realisiert Phase (6).

Kopieren der Rollen / Merging der Features (5)

In diesem Schritt werden die Rollen aus dem Super-Team in das zu compilierende Team kopiert. Man kann sich diesen Vorgang als „textuelles“ Kopieren des Rollen-Codes vorstellen, ähnlich dem, was eine `include`-Anweisung in einem C-Programm bewirkt. Deshalb wird diese Umsetzung von *implicit inheritance* als *copy inheritance* bezeichnet.

Die Umsetzung selbst muss in zwei Schritte unterteilt werden: dem Kopieren der Symbole und – zu einem späteren Zeitpunkt – dem Kopieren des Codes für alle kopierten Rollenmethodensymbole. Das Kopieren der Rollen kann nicht – wie bei allen vorangegangenen Modifikationen – auf der Basis von AST geschehen, da dem Compiler diese nicht notwendigerweise zur Verfügung steht. Wurde das Team mit der zu kopierenden Rolle vorher bereits compiliert, so wurde für diese Rolle und deren Features keine AST während des Compilierungsvorgangs aufgebaut. Jedoch liegt diese Rolle als Symbolbaum vor, der vom Compiler, unabhängig davon, ob von einer bereits compilierten Klasse (*class file*) oder von Sourcecode (*java file*) ausgehend, auf jeden Fall erzeugt wurde. Ein Symbolbaum hat dabei ein Klassensymbol vom Typ `Symbol.ClassSymbol` als Wurzel, an diesem hängen Symbole vom Typ `Symbol.MethodSymbol` oder `Symbol.VarSymbol`.

Beim Kopieren der Symbole müssen bestimmte Informationen aufgrund der kovarianten Rollen-Redefinition angepasst werden. So kann es z.B. notwendig werden, die Parametertypen von Methodensymbolen anzupassen, wenn darin Rollentypen enthalten sind. Befindet sich die zu kopierende Methode z.B. in einer Rolle, die sich ihrerseits in einem Team mit dem Namen `Collaboration` befindet, und hat die zu kopierenden Methode die Signatur `void concr_m(Role1 r1)`, so ist dies eigentlich eine Abkürzung für `void concr_m(Collaboration.Role1 r1)`. Die Signatur des kopierten Methodensymbols, das sich in einer Rolle befindet, die Teil des Teams `Connector` ist, muss zu `void concr_m(Connector.Role1 r1)` modifiziert werden. Das gleiche gilt für den Typ von Rückgabewerten und für die Typen von Variablensymbolen.

Existiert die Rolle im zu compilierenden Team bereits, so müssen die Features *gemerged*, die beiden Feature-Mengen vereinigt werden. Hatte die Rolle für eine geerbte abstrakte Methode ein *callout binding deployment* definiert, so wurde im Schritt (5) dafür eine Implementierung generiert, die durch das *Merging* nicht überschrieben werden darf. Auch wenn die Rolle bestimmte Methoden der Rolle des Super-Teams überschrieben hat, darf durch das *Merging* keine erneute Überschreibung erfolgen. Damit eine überschriebene Methode aus der Rolle des Super-Teams durch `tsuper` trotzdem verfügbar bleibt, wird diese nach dem Kopieren umbenannt. Im Code vorkommende *tsuper calls* können dann durch einen Aufruf dieser umbenannten Methode ersetzt werden.

Zu einem Methodensymbol gehört das Code-Segment, also die Folge von Bytecode-Anweisungen. Bem Kopieren dieses Symbols kann von der Existenz dieses Feldes (`MethodSymbol.code != null`) zu diesem Zeitpunkt noch nicht ausgegangen

werden, da das Team, das die Rolle mit dieser Methode enthält, - im obigen Beispiel das Team `Collaboration` – nicht notwendigerweise compiliert vorliegen muss. Deshalb werden die Methodensymbole, die bereits kopiert wurden, für die jedoch zum Kopierzeitpunkt kein Code-Segment vorlag, in einer Liste der Klasse `TransBindingsInfo` gespeichert und das Kopieren – und das Anpassen – des Code-Segments, das den zweiten Schritt der Umsetzung von *copy inheritance* darstellt, auf einen späteren Zeitpunkt (siehe (6), „Kopieren / Anpassen des Codes geerbter Rollen- und Team-Methoden") verschoben.

Damit waren alle Aufgaben, die sich auf die Sourcecodegenerierung von AST und Symbolen beziehen, bearbeitet. Der erzeugte Object Teams-Code konnte damit genau wie konventionell geschriebener Java-Code den weiteren Compilierungsphasen unterzogen werden.

Erweiterung der Attribuierungsphase

In der Klasse `Attr` wurden für die in der Klasse `Tree` hinzugefügten AST-Konstrukte die jeweiligen `_case`, d.h. accept-Methoden implementiert. Dadurch werden z.B. für ein `Tree.BindingDeployment`-Element alle notwendigen Symbole und Typinformationen erzeugt und diese an das Element angefügt.

Erweiterung der Flusskontroll- und der Sourcecode-Transformationsphase

Für die Flusskontrollphase, für die Umwandlung von *generic* zu konventionellem Java und das „Flachklopfen" der inneren Klassen zu „normalen" Klassen mussten fast keine Anpassungen programmiert werden. Lediglich für die Klasse `TransTypes`, die einen AST-Visitor darstellt, musste eine `accept`-Methode implementiert werden, die den AST vom Type `Tree.BindingDeployment` unverändert zurück gibt.

Erweiterung der Bytecode-Generierungsphase

Die Klasse `Gen` wurde dahingehend modifiziert, dass ihre Methode `normalizeDefs` verändert wurde. Der *javac* verlegt alle Variableninitialisierungen, die im Sourcecode mit der Variablendeklaration zusammen notiert sind, in den jeweiligen Konstruktor der Klasse und verwirft anschliessend alle AST, die nicht vom Typ `Tree.MethodDef` sind. Dies würde ebenfalls `Tree.BindingDeployment`-Konstrukte betreffen, die allerdings im weiteren Compilierungsvorgang für Object Teams-Aufgaben noch benötigt werden. Die Methode `normalizeDefs` wurde deshalb so angepasst, das `Tree.BindingDeployment`-Konstrukte nicht verworfen werden.

Bevor der compilierte Bytecode durch die Klasse `ClassWriter` in eine Datei geschrieben werden kann, müssen noch zwei Object Teams-relevante Erweiterungen durchgeführt werden: das vorher aufgeschobene Kopieren und Anpassen des Codes geerbter Methoden und das Generieren von Attributen im Fall von *callin*-Bindings.

Kopieren / Anpassen des Codes geerbter Rollen-und Team-Methoden (6)

Wie bereits erwähnt ist die Voraussetzung für diesen Schritt, dass das zu kopierende Element, das Feld `code` des jeweiligen Methodensymbols, überhaupt vorliegt, dass also das Team inklusive der Rolle und all seiner Methoden, die kopiert werden sollen,

vorher compiliert wurde. Dies wird dadurch erreicht, dass in der „Hauptroutine" `compile` der Klasse `JavaCompiler` jedes Team, das eine Rolle enthält, für die Bytecodes für eine ihrer Methoden kopiert werden müssen, an das Ende der Liste der zu compilierende Klassen geschoben wird (siehe Feld `deferredEnvs` der Klasse `JavaCompiler`). Das hat wiederum zur Folge, dass Teams, die selbst oder deren Rollen Methoden definieren, deren Code von anderen Teams oder Rollen benötigt wird, zuerst compiliert werden und das Codesegment auf jeden Fall verfügbar ist, wenn es kopiert werden soll. Als Beispiel für eine solche Konstellation kann das erste Object Teams-Beispiel herangezogen werden. Wenn die Rolle `ConnectorA.Role1`, die von `GetSetCollaboration` durch *copy inheritance* nach `ConnectorA` kopiert wurde, compiliert werden soll, so muss auch die konkrete Methode `collaborate()` kopiert werden. Damit deren Bytecode-Anweisungen beim Kopieren verfügbar sind, muss die Klasse `GetSetCollabortion.Role1` auf jeden Fall vor `ConnectorA.Role1` compiliert werden.

Beim Kopieren des Bytecodes muss dieser angepasst werden, falls er Referenzen in den Constant Pool der umgebenden Klasse enthält, die im Kontext der neuen Rolle und des neuen Teams nicht mehr stimmen. Wie bereits erwähnt stellt der Constant Pool eines *class file* ein Register aller möglichen Konstanten dar, die in einer Klasse benutzt werden, z.B. fallen darunter Literale und Namen aller Art. Wenn z.B. im Code einer Methode die Methode `collaborate` aufgerufen wird, so wird nicht „collaborate", sondern eine Referenz auf einen Eintrag im Constant Pool im Code auftauchen, der den Namen „collaborate" enthält. Dies bewirkt ein kompakteres *class file*-Dateiformat, wenn dieser Aufruf an mehreren Stellen auftauchen sollte, da die Referenz auf einen Namen meist weniger Bytes beansprucht als der Name selbst.

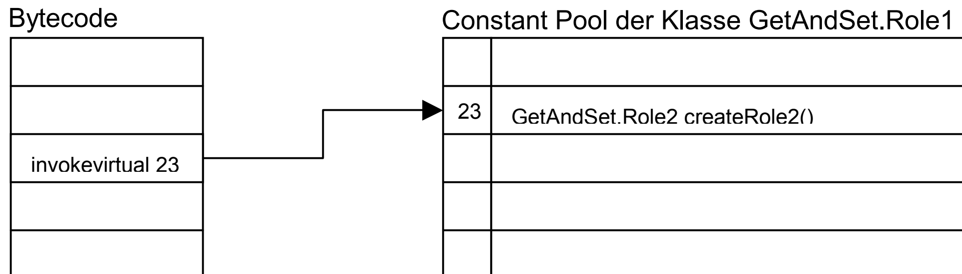
Wenn eine Klasse compiliert wird, erzeugt der *javac* eine Repräsentation des Constant Pool in Form einer Instanz der Klasse `Pool` des Pakets `com.sun.tools.javac.v8.code`. Dieser Pool nimmt während des Compilierungsvorgangs u.a. Symbole auf. Wenn nun z.B. der Code der Methode `collaborate()` der Rolle `GetAndSet.Role1` nach `ConnectorA.Role1.collaborate()` kopiert werden soll, so könnte im Bytecode folgendes auftauchen:

```
invokevirtual 23
```

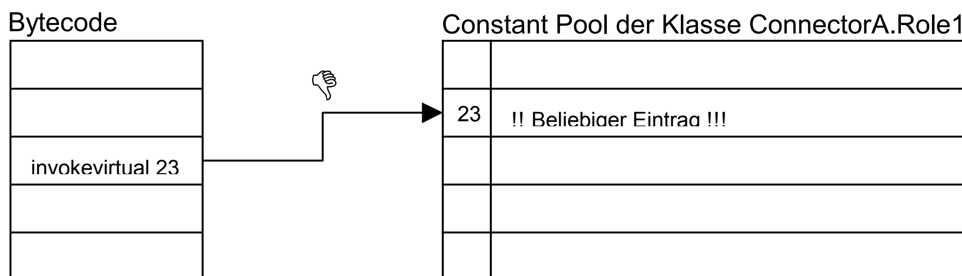
wobei `invokevirtual` das Mnemonic für den Aufruf einer nicht-statischen Methode ist. Die Zahl stellt `23` eine Referenz auf ein Element im Constant Pool der Rolle dar, in der Informationen über die aufzurufende Methode stehen, z.B. könnte dies die Signatur in der Form `GetAndSet.Role2 createRole2()` sein. Diese Referenz bezieht sich allerdings auf den Constant Pool der ursprünglichen Rolle, also auf `GetAndSet.Role1`. Dies macht im Team `ConnectorA` keinen Sinn, da erstens im Constant Pool der Klasse `ConnectorA.Role1` an der Stelle `23` etwas ganz anderes stehen kann und zweitens die Methode mit der Signatur `GetAndSet.Role2 createRole2()` im Constant Pool der Klasse `ConnectorA.Role1` überhaupt nicht verfügbar sein muss. Es muss deshalb im Constant Pool der Klasse `ConnectorA.Role1` nach dem Symbol mit der Signatur `ConnectorA.Role2 createRole2()` gesucht werden – es ist auf jeden Fall vorhanden, da das Symbol in

einer früheren Phase, dem ersten Teil der Umsetzung von *copy inheritance*, erzeugt wurde –, die Referenz auf dieses Symbol ermittelt und anstelle der alten Referenz gesetzt werden. Abbildung 6 soll den Anpassungsalgorithmus verdeutlichen:

Ausgangssituation: Code von `GetAndSet.Role2 GetAndSet.Role1.createRole2()`



Code wird kopiert nach `ConnectorA.Role2 ConnectorA.Role1.createRole2()`



Finden des entsprechenden Eintrags im Constant Pool der Klasse ConnectorA.Role1:
`ConnectorA.Role2.createRole2()`
und Anpassen der Referenz im Bytecode

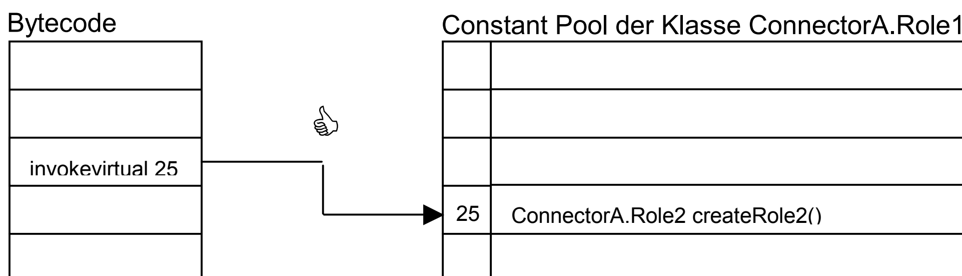


Abbildung 6: Algorithmus zur Anpassung von Constant Pool-Referenzen

Nach diesem Schema müssen innerhalb des zu kopierenden Codes alle Constant Pool-Referenzen überprüft und gegebenenfalls angepasst werden, sowohl im Code der geerbten Rollen- als auch der geerbten Team-Methoden.

Generieren von Attributen im Fall von *callin*-Bindings (7)

Während bei *callout*-Bindings der Object Teams-Compiler alle syntaktischen Konstrukte in ein lauffähiges Team umsetzen kann, so ist dies bei *callin*-Bindings nicht möglich, da *callin*-Bindings Aspekt-Funktionalität in sich tragen und zur Laufzeit aktivierbar und deaktivierbar sein sollen. Die Aufgabe des Compilers besteht bei *callin*-Bindings darin, die Binding-Informationen in Form von Bytecode-Attributen zu erzeugen, damit das Object Teams-RE diese Informationen auslesen kann. Dabei sind Bytecode-Attribute C-Struct ähnlich aufgebaut, es handelt sich also um strukturiert aneinander gereihte Byte-Arrays. Bytecode-Attribute sind in [21] spezifiziert.

Die von Object Teams verwendeten Attribute heissen `CallinRoleBaseBindings`, `CallinMethodMappings` und `CallinParamMappings` und sind im folgenden aufgelistet. *`u2`* bezeichnet dabei den Typ *2 Byte unsigned*. Das Suffix `_index` eines Elements zeigt an, dass es sich hierbei um eine Referenz auf einen Constant Pool-Eintrag handelt.

Das Attribut `CallinRoleBaseBindings` ist ein Attribut des Teams. Es beinhaltet alle Rollen-Basis-Bindings, die in diesem Team definiert sind und *callin*-Methoden-Bindings enthalten.

```
CallinRoleBaseBindings {
  u2 attribute_name_index;
  u2 callin_bindings_count;
  CallinBinding callin_bindings[callin_bindings_count];
}
CallinBinding {
  u2 role_name_index;
  u2 base_name_index;
}
```

`CallinMethodMappings` ist ein Attribut einer Rolle, das aussagt, welche Rollen-methode per *callin* mit welchem Modifier auf welche Basismethode abgebildet wird.

```
CallinMethodMappings {
  u2 attribute_name_index;
  u2 method_mappings_count;
  CallinMethodMapping method_mappings[method_mappings_count];
}
CallinMethodMapping {
  u2 role_method_name_index;
  u2 binding_modifier_index;
  u2 base_method_name_index;
}
```

`CallinParamMappings` ist ein Attribut einer Rollenmethode und gibt an, in welcher Weise Parameter der Rollenmethode auf Parameter der Basismethode abgebildet werden. Dabei besteht ein `CallinParamMapping`-Eintrag aus einem Constant Pool-Index-Paar, das jeweils dereferenziert die Positionen der formalen Parameter innerhalb der Signatur ihrer Rollen- bzw. Basismethode kennzeichnet.

```
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;
}
```

Die Informationen für diese Attribute sind als Felder der Klassen `Symbol.ClassSymbol` und `Symbol.MethodSymbol` deklariert und werden im Anschluss an das Kopieren des Methodencodes (Phase 6) durch Objekte der Klasse `TransBindingsCallin` erzeugt. Am Ende des Compilierungsvorgangs wandelt die Klasse `ClassWriter` den Inhalt dieser Felder in Bytecode um.

Damit sind alle Object Teams-Modifikationen durchgeführt. Als letztes muss der Bytecode durch die Klasse `ClassWriter` erzeugt werden.

Es folgen einige Klassen, die modifiziert bzw. neu angelegt wurden, deren Beitrag zur Object Teams-Erweiterung jedoch eher „Utility-Charakter“ hat:

- `com.sun.tools.javac.v8.util.Names`
  Alle während des Compilierungsvorgangs verwendeten Namen werden in dieser Klasse gehasht. Einige Object Teams-Namen wurden hier hinzugefügt.
- `com.sun.tools.javac.v8.comp.TransBindingsConstants`
  Enthält alle wichtigen Object Teams-Konstanten.
- `com.sun.tools.javac.v8.comp.TransBindingsInfo`
  Enthält Utility-Methoden, die Object Teams-relevante Informationen liefern.
- `com.sun.tools.javac.v8.comp.TransBindingsMappingAnalyser`
  Analysiert die Parameterabbildungen, die – bisher – ausschliesslich bei *callout*-Bindings vorkommen können.
- `com.sun.tools.javac.v8.comp.TransBindingsVisitor`
  Enthalt „kleine“ Visitor-Klassen als innere Klassen, die entweder Informationen über einen AST sammeln oder diesen modifizieren. Sie wurden in eine einzige Klasse gepakt, da die meisten ihrer `accept`-Methoden identisch sind.
- `com.sun.tools.javac.v8.comp.TransBindingsException`
  Diese Klasse stellt Ausnahmen dar, die Object Teams-spezifisch sind.

Zusammenfassung der Compilermodifikationen

Der Object Teams-Compiler (*otc*) erzeugt per *copy inheritance* in einem Team alle Rollen, die im jeweiligen Super-Team definiert sind. Sind diese Rollen bereits definiert, so findet ein *Merging* statt, wobei überschriebene Rollen-Features umbenannt werden und so für `tsuper` sichtbar und zugänglich bleiben. Werden in einer Rolle *callout bindings* definiert, so passt *otc* die Methodenrümpfe der Rollenmethoden entsprechend den Bindings an. Enthält ein Team dadurch keine abstrakten Methoden mehr, sind also alle abstrakten Methoden entweder

implementiert oder durch *callout* an konkrete Basismethoden gebunden, dann kann es unabhängig von einem Object Teams-RE genutzt werden. Im *callin*-Fall werden Attribute im Bytecode erzeugt, die das Object Teams-RE auslesen kann, um die entsprechenden Modifikationen zu erzeugen.

Die Auswertung der Attribute durch das Object Teams-RE

Grundlage für die Modifikation von Java-Bytecode zur Laufzeit ist die Möglichkeit, die Klassen, die die Java Virtual Machine dynamisch lädt, vorher zu inspizieren und gegebenenfalls Modifikationen durchzuführen. Dies wird durch den Austausch des *System Class Loader* ermöglicht, wovon das Framework JMangler [14] Gebrauch macht. Der Benutzer schreibt für JMangler sogenannte *Transformer*-Klassen, in denen definiert ist, wie eine Klasse, bevor sie von der JVM geladen wird, transformiert werden soll. Die Transformer-Klassen lesen also die vom Compiler generierten Attribute aus und modifizieren danach die Basisklassen, bevor diese an die JVM weiterleitet werden. Diese Technik ist Thema einer weiteren Diplomarbeit, die in Kürze erscheinen wird.

Ausstehende Arbeiten am Compiler

Die Implementierung des Object Teams-Compiler wurde mit dieser Arbeit begonnen, sie ist jedoch noch weit von ihrem Abschluss entfernt. Die ausstehenden Arbeiten werden im Folgenden in der Reihenfolge ihrer Dringlichkeit genannt, die dringlichsten zuerst.

<u>Vervollständigung der elementaren Object Teams-Funktionalität</u>

Darunter fallen einige Punkte, die unmittelbar aus dem beschriebenen Programmiermodell folgen, jedoch noch nicht implementiert sind:

- Kopieren und Umbennen der geerbten Rollenkonstruktoren
  Wenn beim Vorgang des *Merging* ein Konstruktor der beerbten Rolle kopiert werden soll und ein Konstruktor mit derselben Signatur von der erbenden Rolle bereits definiert ist, so muss ersterer nach dem Kopieren umbenannt werden. Dabei verliert der Konstruktor in Java jedoch seinen typischen Namen, der mit dem Rollennamen identisch sein muss. Diese Namensabweichung in den Compiliervorgang von Konstruktoren zu integrieren steht noch aus.
- Fehlende Umsetzung von `tsuper`
  Mit dem ersten Punkt zusammenhängend ist die noch fehlende Umsetzung von *tsuper calls*. Die Umsetzung des Aufrufs `tsuper()`, also der Aufruf des über die Teamgrenze hinweg geerbten Rollenkonstruktors schliesst sich direkt an die Realisierung des ersten Punkts an. Die Ersetzung eines Methodenaufrufs wie z.B. `tsuper.originalMethod()` müßte dagegen ohne weitere Voraussetzungen implementierbar sein, da die jeweilige Methode kopiert und anschliessend umbenannt wurde.

- Geerbte Team-Methoden und deren Signaturveränderung
  Ein nicht ohne weitere Vorüberlegungen lösbares Problem ergibt sich durch geerbte Team-Methoden, deren Rückgabewert einen Rollentyp darstellt. Deren Code-Anpassung ist Teil der Object Teams-Compiler-Phase 7 und wurde bereits erläutert. Durch die spezielle Vererbungssemantik ändert sich jedoch auch deren Signatur, z.B. von `TeamA.RoleA createRoleA()` zu `TeamB.RoleA createRoleA()`. Wenn diese Methode *public* oder *protected* ist, dann greift der konventionelle Vererbungsmechanismus von Java, d.h. eine derart angepasste Methode wird von Java verboten, da bei der Methodenverbung keine Signaturveränderungen hinsichtlich des Rückgabewerts erlaubt sind. Wie trotzdem eine modifizierte Team-Methode generiert und benutzt werden kann, muss noch herausgearbeitet werden.
- Fehlende Umsetzung des *smart lifting*-Verfahrens
  Wie im konzeptuellen Teil bereits beschrieben ermöglicht dieses Verfahren, für Basis-Objekte das jeweils speziellste Rollenobjekt zu finden, dessen Typ auf den Typ des Basisobjekts über ein Binding abgebildet wurde. Dies umzusetzen bedeutet, die generierten `liftTo`-Methoden dahingehend zu erweitern, dass sie anhand des aktuellen Basis-Parameters entscheiden können, welches Rollenobjekt sie zurückliefern. Eine intuitive Lösung wäre, per *instanceof*-Operator die Basis-Objekte auf ihren Typ zu testen und aufgrund der Testergebnisse die Rolle auszusuchen. Da diese Möglichkeit wenig performant ist, wurde eine zweite Variante in Betracht gezogen, die aufwendiger zu implementieren, dafür erheblich performanter zu sein verspricht. Danach generiert der Compiler in den `liftTo`-Methoden die Typ-Tests anhand von *tags* (Integer-Werten), die in den Basisklassen erwartet werden und deren Typ identifizieren. Diese *tag*-Felder sind in den Basisklassen ursprünglich natürlich nicht definiert. Deshalb muss das Object Teams-RE beim Laden der Basisklassen für diese *tags* Felder in den Basisklassen erzeugen und diese Felder anhand der vom Compiler verwendeten *tag*-Werte initialisieren, bevor die eigentliche Programmlogik ablaufen kann.
- Deklariertes Lifting
  Teil des Object Teams-Programmiermodell ist die in dieser Arbeit bisher nicht dokumentierte Möglichkeit, den Lifting-Prozess explizit auszulösen. Dies macht bei Team-Methoden Sinn, wenn diese als Parameter ein Rollen-Objekt erwarten, trotzdem aber ausserhalb der Teams benutzbar sein sollen, wo normalerweise keine Rollen-Objekte existieren. Object Teams ermöglicht deshalb die Deklaration von Team-Methoden nach dem Muster `public <ReturnType> team_method(BaseType as RoleType param)`. Die Methode `team_method` erwartet gegenüber dem Benutzer als Argument ein Objekt von Typ `BaseType`, intern verwendet sie jedoch das dem aktuellen Parameter zugeordnete Rollenobjekt vom Typ `RoleType`. Umzusetzen wäre diese Funktionalität durch eine Parser- und AST-Erweiterung und durch das Einfügen eines `liftTo`-Ausdrucks als erste Anweisung innerhalb des Rumpfs von `team_method`.

Damit der Object Teams-Compiler nach der Lösung dieser dringendsten Proleme als Werkzeug von Entwicklern einsetzbar ist, die die Implementations-Interna nicht kennen, sondern ausschliesslich das Programmiermodell und die Informationen, die u.a. auf der Object Teams-Homepage [30] abzurufen sind, müssen noch einige weitere Aufgaben bearbeitet werden. Dazu gehören:

Type-Checking

Bisher werden die Methodenabbildungen bei *callout*-Bindings hinsichtlich ihrer formalen Parameter und ihrer Rückgabewerte nur implizit dadurch typ-überprüft, dass nach der Generierung der neuen Methoden, die durch das Binding enstanden sind, diese Methoden standard-mässig compiliert und dabei Typfehler vom Compiler gemeldet werden. Der Compiliervorgang sollte jedoch bereits vorher bei einer fehlerhaften Methodenabbildung abgebrochen werden, da die Generierung der neuen Rollenmethoden für den Entwickler unsichtbar bleiben sollte. Denn tritt bei der Compilierung dieser generierten Methoden ein Fehler auf, so bezieht sich die Fehlermeldung auf die generierte Methode, die der Entwickler nicht explizit implementiert, sondern deren Generierung er durch das Binding veranlasst hat. Die Fehlermeldung verweist somit nicht auf den vom Entwickler verursachten Fehler, sondern auf den sich anschliessenden Generierungsschritt und kann der Fehlerbehebung weniger gut dienen als ein direkter Verweis auf das fehlerhafte Binding.

Generell ungeprüft ist z.Z. das Binding im *callin*-Fall, da kein Code sondern nur Attribute generiert werden und damit Fehler nur zur Laufzeit bemerkt werden können. Der Compiler könnte jedoch das *callin*-Binding dahingehend prüfen, inwieweit die Signaturen der Basisklassen nach Object Teams-Maßstäben zu den Signaturen der Rollenmethoden passen, auf die sie abgebildet wurden.

Ausser dem reinen Type-Checking muss der Object Teams-Compiler weitere Anforderungen überprüfen und bei deren Verletzung entsprechend reagieren. Dazu gehören u.a. die folgenden Punkte, die man als *well formedness rules* bezeichnen kann:

- Ein Team kann nicht instantiiert werden, wenn eine abstrakte (Rollen-) Methode nicht implementiert oder gebunden wurde.
- Bei einer abstrakten Rollenmethode kann nicht `=>` (*callout override*) verwendet werden, bei einer konkreten nicht `->` (*callout*).
- Die Deklaration einer Klasse als Team oder als Rolle muss mit dem Modifier *open* versehen sein, wenn sie als Rolle *callin*-Methoden oder Methoden-Bindings hat oder wenn sie als Team eine *open* Rolle besitzt.
- *callin*- und *callout*-Bindings dürfen keine (indirekten) Zyklen bilden.
- Das Thema Exceptions ist im Object Teams-Compiler bisher nicht ausreichend berücksichtigt. Es ist bei der Implementierung zu bedenken, dass *callin*-Methoden keine Exceptions werfen dürfen (bei der Implementierung der Basis ist davon nichts bekannt und somit auch nicht einplanbar). Für den *callout-Fall* gilt folgendes: wenn eine Basismethode eine Exception wirft und

die Rollenmethode nicht, dann muss die Exception im Binding aufgefangen werden, z.B. durch:

```
roleMeth -> baseMeth catch (Exception e) {...};
```

Wenn beide eine Exception werfen, dann sollten sie kompatibel sein. Sind sie es nicht, so muß eine Indirektion benutzt werden, bei der eine mögliche Basis-Exception explizit behandelt wird:

```
...
abstract role_method_with_ex() throws RoleException;
void role_method() {
    try {
        role_method_with_ex();
    } catch (Exception e) { ...}
}
 role_method_with_ex -> base_method_with_ex;
```

- Findet ein *callin*-Binding einer Basismethode auf eine Rollenmethode mit dem Modifier *replace* statt, so muss der Compiler überprüfen, ob die Rollenmethode einen *base call* enthält. Fehlt dieser, so sollte eine Warnung ausgegeben werden.
- Verwendet der Benutzer einen Namen, der von Object Teams intern verwendet wird, so muss eine entsprechende Warnung ausgegeben werden.

Diese Liste ist sicherlich nicht vollständig und dürfte während der Implementierung der noch fehlenden Funktionalitäten noch anwachsen.

Parameterabbildungen bei *callin*-Bindings

Parameter mappings sind bisher nur für den *callout*-Fall implementiert. Eine ähnliche Umsetzung kommt für den *callin*-Fall nicht in Frage, da in diesem Fall kein Sourcecode generiert werden kann, sondern die *parameter mappings* in einer geeigneten Weise in Attributen codiert werden müssen. Diese Funktionalität wird in Synchronisation mit einer Diplomarbeit implementiert werden, die sich gerade in Bearbeitung befindet.

Aussagekräftige Fehlermeldungen des Compilers

An vielen Stellen, an denen Object Teams-Anpassungen implementiert wurden, können Fehler auftreten, z.B. die oben beschriebenen Typfehler. Diese Fehler – die teilweise auch zu gefangenen Exceptions führen, die jedoch nicht Object Teams-spezifisch sein müssen – werden z.Z. nicht entsprechend behandelt, die resultierenden Fehlermeldungen sind somit nicht nur wenig aussagekräftig, sondern teilweise für den Benutzer eher verwirrend. Im Paket `com.sun.tools.javac.v8.resources` befinden sich vier *property*-Dateien, in denen mögliche Fehlermeldungen und ihre vom Compiler verwendeten Abkürzungen enthalten sind. Diese Dateien sind entsprechend zu erweitern.

Ein weitere Voraussetzung für aussagekräftige Fehlermeldungen ist die durch den Compiler ausgegebene Positionsangabe des Fehlers bei dessen Auftreten. Dies wird im *javac* dadurch erreicht, dass im Parser beim Erzeugen eines AST der

Erzeugungsmethode die aktuelle Quelltext-Position des syntaktischen Konstrukts im Quellcode mitgegeben wird. Die Position wird durch den Scanner bereitgestellt. Durch die Object Teams-Erweiterungen war die Erzeugung von AST unabhängig vom Parsen und damit einer Scanner-Position notwendig. Dadurch ist das Positionsfeld dieser nachträglich erzeugten AST nicht immer aussagekräftig. Für diese für Object Teams generierten AST muss eine Möglichkeit gefunden werden, deren Position nachträglich zu berechnen, um bei auftretenden Fehlern dem Benutzer die Stelle des Fehler anzeigen zu können.

Integration in eine Software-Entwicklungsumgebung

Software wird selten ausschließlich mit den unbedingt notwendigen Tools erstellt, worunter man einen Text-Editor, einem Compiler und eine Laufzeitumgebung verstehen kann. Hinzu kommen Tools zum Finden von Quelltextstellen (z.B. *grep* unter Unix), Debugger (z.B. *gdb)*, Versionskontrollsysteme (z.B. Visual Source Safe, CVS) oder auch Modellierungstools (z.B. Together, Rational Rose, ArgoUML). Neben der von jedem Tool erbrachten Funktionalität kann die Integration dieser Funktionalitäten in eine einzige Entwicklungsumgebung (*Integrated Development Environment*, IDE) einen großen Produktivitätsgewinn darstellen, da ein ständiges Wechseln von Tool zu Tool und damit auch die ständige Umstellung auf das jeweilige Tool-Interface wegfiele. Der Object Teams-Compiler und die Object Teams-RE sollten mittelfristig ebenfalls in eine Entwicklungsumgebung integriert werden. Dies setzt allerdings eine IDE voraus, die sowohl die Verwendung eines eigenen Compilers als auch die Integration eines eigenen bzw. modifizierten Laufzeitsystems erlaubt. Zur Zeit bieten sich dafür zwei IDEs an: NetBeans [22] und Eclipse [17]. Während NetBeans schon seit einigen Jahren existiert ist das Eclipse-Projekt erst vor etwas mehr als einem halben Jahr gestartet. Von beiden existieren einsetzbare und erweiterbare Versionen, beide stehen unter einer Open Source-Lizenz. Beide IDE sollten auf Object Teams-Erweiterbarkeit hin evaluiert werden.

Ausblick

Für die ausstehenden Aufgaben, deren Bearbeitung über die bisher geleistete Arbeit hinausgehen könnte, stellt sich die Frage, ob die daraus abgeleiteten Features überhaupt für den Standard-Java-Compiler implementiert werden sollten, denn der *javac* ist nicht unbedingt für mögliche Erweiterungen entworfen worden. Hauptursachen dafür sind die mangelnde Verständlichkeit des Codes – was die Analyse auch deshalb erschwert, da weitere Dokumentation fehlt –, und die verwendeten Datenstrukturen, die Erweiterungen nur unter einigem Aufwand zulassen. Der folgende Abschnitt beschreibt im einzelnen, wodurch die Erweiterbarkeit des *javac* erschwert wird.

Kritikpunkte am *javac*

Folgende Punkte sind, was das *javac*-Design und dessen Implementierung betrifft, negativ anzumerken:

Nebeneffekte von Methoden

Häufig auftretende Nebeneffekte von Methoden – z.B. die Anzahl von Elementen einer Liste, die sich vom Methodenanfang zum Methodenende ohne direkte Modifikation verändert – machen den Programmfluss schwer verständlich. Durch den Verzicht auf *getter*- und *setter*-Methoden für die meisten Klassenfelder und damit die Aufgabe des Geheimnisprinzips (siehe [1]) wird die Verständlichkeit des Codes weiter verringert, wenn dies auch den Verzicht auf zusätzliche Methodenaufrufe bedeutet (die Felder sind dabei *public* bzw. *protected*) und einen damit einhergenden Performanzgewinn darstellen mag.

Kombinierte Verwendung von Shift- und Bit-Operatoren

Dies soll nicht als generelle Kritik am verwendeten Programmierstil verstanden werden, denn dadurch wird Effizienz und Performanz erreicht, leider aber auch geringere Verständlichkeit.

Unpassende Feature-Namen

Oft sind Methoden und Felder wenig aussagkräftig benannt, es fehlen sprechende Bezeichner. Teilweise versteckt sich hinter „einfachen“ Namen wie `put`, die an Standard-Methoden aus dem JDK erinnern, eine sehr viel spezifischere Implementierung, die aus dem Namen nicht hervorgeht. Für die Methode `clone` eines Symbols dagegen gilt das Gegenteil, sie leistet weniger als erwartet. Man könnte sagen, dass die Invariante einer Methode, die normalerweise durch Standard-Namen vom Benutzer antizipiert und vorausgesetzt wird, durch die spezifischen Implementierungen oft verletzt wird.

Bad smells

Im Sinne von Fowler et al. [23] liegen viele *bad smells* vor. Im *javac* „riecht es“ deshalb: viel zu langer Code einzelner Methoden (*long method*), teilweise zu grosse Klassen (*large classes*) und Methoden mit zu langen Parameterlisten (*long parameter list*).

Fehlende Polymorphie

Zu selten wird Polymorphie verwendet. Stattdessen besitzen AST-Klassen häufig ein *tag*, einen Integer-Wert, der die Klasse identifiziert. Auch hier liegt die Motivation mutmaßlich im erhofften Performanzgewinn des Compilers. Zur Folge hat dieser Stil jedoch oft viele verschachtelte *if*- und *switch*-Anweisungen (was auch einen *smell* namens *switch statements* darstellt). Die *tag*-Abfragen bringen auch die Verletzung des single choice-Prinzips (siehe [1]) zum Ausdruck: die möglichen Ausprägungen eines Elements kommen nicht nur in einem die Ausprägungen definierenden Modul, sondern auch in allen Modulen vor, deren Verhalten von diesen Ausprägungen abhängt. Bezogen auf bestimmte Klassen im *javac* bedeutet dies, dass die möglichen *tags* zwar in einer Klasse definiert sind, jedoch auch in vielen anderen

Klassen in Fallunterscheidungen genutzt werden. Das *single choice*-Prinzip könnte dadurch angewandt werden, dass aufgrund der AST-Klassen-Instanz das jeweilige Verhalten per Polymorphie ausgewählt und auf *tags* verzichtet werden würde.

Weitere Optimierungen

Diese machen den Code ebenfalls undurchsichtig. Beispielsweise hat jede zu compilierende Klasse ein *Pool*-Objekt, das den zu generierenden Constant Pool darstellt. Dieses Feld wird nicht instantiiert, sondern alle Klassen teilen sich dieses Pool-Objekt durch die Verwendung des Pool-Feldes der Klasse `Gen`, es liegt also eine Zuweisung der Art `classSymbol.pool = gen.pool` vor. Auf den ersten Blick scheint dies unsinnig, da jede Klasse letztendlich einen eigenen Constant Pool benötigt. Es ist aber dennoch korrekt implementiert, da nach jeder Zuweisung der Code der Klasse generiert und in eine Datei geschrieben wird und ihr Feld für den Pool damit hinfällig wird (das Klassen-Objekt wird dann nicht mehr verwendet). Das Pool-Feld von `Gen` wird anschliessend neu initialisiert. Auch hier scheint die Performanz, die man sich von der Vermeidung von Konstruktor-Aufrufen und der verminderten Speicherallokation durch Vermeidung zusätzlicher Objekte erhofft Ursache für diesen schwer verständlichen Stil.

Problematisches *Visitor Pattern*

Oft wird das *Visitor Pattern* verwendet. Wie bereits erläutert hat dieses Muster den Vorteil, die Datenstruktur und die Bearbeitung dieser Struktur voneinander zu trennen. Wenn nun die Datenstruktur – und dies ist im Falle des *javac* der AST – zusätzlich bearbeitet werden soll, so ist ein neuer Visitor zu schreiben, die AST-Klassen müssen dabei nicht modifiziert werden. Für die Erweiterung der Datenstruktur – im Falle der Object Teams-Erweiterung bedeutete dies, alle Binding-relevanten AST zu integrieren – ist allerdings eine Anpassung aller Visitor-Klassen die notwendige und aufwendige Folge.

Mögliche Compiler-Weiterentwicklungen

Wenn aus den aufgezeigten Gründen auf *javac* verzichtet werden soll, so könnte man versuchen, den bisher entwickelten Object Teams-Code auf einen anderen Compiler zu portieren. Als eine mögliche Alternative böte sich Kopi [24] an, ein Java-Compiler, der unter einer *Open Source*-Lizenz steht und der nach seiner Beschreibung dafür entworfen wurde, Weiterentwicklungen zu unterstützen. Diese Möglichkeit muss jedoch erst ausreichend evaluiert werden.

Generell bietet sich natürlich das Vorgehen an, mit dem viele Compiler anderer Sprachen entwickelt wurden: nachdem ein Compiler für eine Sprache A in einer beliebigen Sprache entwickelt wurde – was auch als als *bootstrapping* bezeichnet wird –, kann ein neuer Compiler für die Sprache A in der Sprache A selbst geschrieben werden, der durch den Ausgangscompiler übersetzt werden kann. Dies wurde wie beschrieben auch für den *javac* durchgeführt, wobei der Ausgangscompiler in der Sprache C geschrieben wurde. Mit dem *javac* wiederum konnten seine eigenen Quellen, erweitert um Object Teams-Funktionalität zum Object Teams-Compiler compiliert werden.

Analog zu diesem Vorgehen könnten die restlichen Object Teams-Funktionalitäten ebenfalls als Erweiterung des *javac* implementiert werden, um so den *bootstrapping*-Vorgang abzuschliessen. Anschliessend ergäben sich viele Möglichkeiten, *javac* als Code-Grundlage zu verlassen und so den Compiler selbst verständlicher und modularer zu schreiben. So könnte versucht werden, mit der gewonnenen Ausdruckskraft von Object Teams, dem Object Teams-Ausgangscompiler und den Quellen eines weiteren Java-Compilers, z.B. Kopi oder wiederum *javac*, einen weiteren Object Teams-Compiler zu erzeugen. Oder man könnte versuchen, gänzlich ohne Vorarbeit mit dem Object Teams-Ausgangscompiler einen neuen Object Teams-Compiler zu entwickeln. Statt dem Object Teams-Ausgangscompiler könnte dabei auch AspectJ oder HyperJ zum Einsatz kommen. Mit letztgenannten Ansätzen und dem Code, mit dem der *javac* erweitert und angepasst wurde, könnte ebenfalls versucht werden, einen Object Teams-Compiler zu erzeugen; im Falle von z.B. AspectJ müßten die Object Teams-Erweiterungen als Aspekte definiert werden. Dies wäre insofern relativ einfach umsetzbar, da beide Ansätze als auch die *javac*-Erweiterungen in Java implementiert wurden. Die gewählte Technik müßte aber in beiden Fällen mit einem Object Teams-RE kombiniert werden, das wie bereits erläutert Aspektverhalten bei Object Teams zur Laufzeit realisiert wird und das *weaving* von Bytecode erfordert. Dies leisten die Ansätze AspectJ und HyperJ nicht.

In Anbetracht der Tatsache, dass für den *javac* bereits Expertise erworben wurde, bietet sich als aussichtsreichste Perspektive an, den *bootstrapping*-Vorgang mit *javac* abzuschliessen und einen funktional vollständigen Object Teams-Compiler zu entwickeln. Trotz der obigen Kritikpunkte am *javac* ist ohne Evaluation nicht gesichert, dass diese Kritikpunkte auf die aufgeführten Alternativen nicht zutreffen, teilweise sind sie – wie die Kritik am verwendeten Visitor Muster – mit rein objektorientierten Mitteln auch nicht aufzulösen. Desweiteren zeigt die Auflistung dieser Punkte auch, dass deren Problematik erkannt wurde und entsprechend darauf reagiert werden kann.

Es wäre nach Vervollständigung der *otc*-Implementierung zuerst wichtig herauszufinden, wie modular die Object Teams-spezifischen Erweiterungen am *javac* selbst sind, d.h. inwieweit die Erweiterungen in eigenen Paketen und Klassen kapselbar sind. Wäre dies möglich, so könnte man die dann vorliegenden Object Teams-Compiler-Module in Kombination mit Object Teams-Features einsetzen und damit wie oben angedeutet neue Object Teams-Compiler erzeugen. Der nächste Abschnitt analysiert, inwieweit der bisher erstellte Object Teams-Code *tangled* mit *javac*-Code ist und eine Trennung der beiden Code-Sorten erreichbar ist.

Analyse der Kapselbarkeit von Object Teams-Code

Tabelle 1 gibt einen Überblick darüber, an welchen und wie vielen Stellen Modifikationen und Erweiterungen des *javac* für Object Teams erstellt wurden.

WAS UND WO	ANZAHL
Zusätzlichen Code am Anfang einer Methode eingefügt	2
Zusätzlichen Code mitten in einer Methode eingefügt	26
Zusätzlicher Code am Ende einer Methode eingefügt	1
Methoden in bestehende Klassen eingeführt	32
Felder in bestehende Klassen eingeführt	51
Felder in bestehenden Klassen geändert	8
Neue innere Klassen in bestehende Klassen eingeführt	7
Sichbarkeit von bestehenden Features geändert	4
Import in bestehenden Klassen erweitert	2
Elemente in *property*-Dateien (bespielhaft) eingefügt	3
Neue Klassen in bestehenden Paketen angelegt	8
darin enthalten: Felder	50
Methoden	132
Innere Klassen	4

Tabelle 1: Stellen und Häufigkeit der Modifikation des *javac* für Object Teams

Die ersten zehn Zeilen der Tabelle stellen invasive Veränderungen des *javac*-Sourcecodes dar, die letzte Zeile listet die Klassen und deren Features auf, die durch die Object Teams-Anforderungen gänzlich neu enstanden sind. Anhand der Tabelle lassen sich vier Elementgruppen erkennen, die aus *javac* herausgelöst werden müssten:

- neu in *javac*-Paketen eingeführte Object Teams-Klassen,
- Object Teams-spezifische Methodenanpassungen,
- neue Object Teams-Features,
- und neu eingeführte innere Klassen.

Neu eingeführte Object Teams-Klassen

Diese sind insofern ebenfalls *javac*-invasiv, als dass sie in einem *javac*-Paket untergebracht wurden. Der Grund hierfür liegt in der häufigen Verwendung des

default Modifier für Klassen-Features im *javac*. Dieser Modifier gibt dem Feature die Sichtbarkeit für alle Sub-Klassen und alle Klassen innerhalb desselben Pakets. Wurde ein solches Feature für Object Teams-Erweiterungen benötigt, so wurde die neue Klasse im gleichen Paket erstellt, ein Erben von der das Feature enthaltenden Klasse kam nicht in Frage, da man oft von verschiedenen Klassen *default* Features benötigte und ein Erben darüberhinaus semantisch keinen Sinn gemacht hätte. Um die neu erstellten Object Teams-Klassen in ein eigenes Paket auszulagern, müssten entweder viele Sichtbarkeits-Modifier von *javac*-Elementen geändert oder neue Zugriffsmethoden (*getter*-Methoden) für diese Elemente in ihren Klassen eingeführt werden. Beides wurde bereits mehrmals durchgeführt, wenn der Zugriff auf Elemente aus anderen Paketen notwendig, deren Sichtbarkeit aber zu eingeschränkt definiert war. Beide Alternativen bedeuten jedoch wiederum einen invasiven Eingriff in bestehenden *javac* Sourcecode.

Object Teams-spezifische Methodenanpassungen

Dies wäre durch Object Teams und *callin*-Bindings zu realisieren. Eine *javac*-Methode könnte bei ihrem Aufruf um *otc*-Funktionalität ergänzt werden. Leider war zusätzlicher Object Teams-Code fast immer mitten im Methodenrumpf nötig, wehalb diese Fälle mit *replace*-Bindings zu behandeln wären, wobei neben den neuen Anweisungen der gesamte Code der anzupassenden *javac*-Methode auch in der *callin*-gebundenen Rollenmethode vorkommen müsste. Selten lagen die Modifikationen direkt am Anfang oder Ende der Methode, was mit *before*- bzw. *after*-Bindings und schlankeren Rollenmethoden lösbar wäre. Die Object Teams-Modifikationen, die mitten in Methodenrümpfen auftauchen, sollten deshalb daraufhin untersucht werden, inwieweit sie an den Anfang oder an das Ende der Methode verschoben werden können, oder inwieweit es möglich ist, deren Anweisungen an Anfangs- oder Endstellen anderer Methoden zu verschieben.

Neue Object Teams-Features

Prinzipiell können sämtliche Methodenänderungen in neu eingeführte Methoden ausgelagert werden, was teilweise bereits durchgeführt wurde. Diese neuen Methoden wären Methoden von Rollen und würden über definierte Bindings aufgerufen werden. Im *javac*-Code eingeführte Felder könnten zu Feldern von Rollen umgesetzt werden. Wenn Felder in bestehenden *javac*-Klassen geändert wurden, so ist die Ursache hierfür fast immer der Sichtbarkeits-Modifier, dessen Problematik bereits beschrieben wurde. Wenn der Typ eines Feldes geändert wurde, so geschah dies meist aus Bequemlichkeit und wäre auf Notwendigkeit zu überprüfen.

Neu eingeführte innere Klassen

Neben der Sichbarkeit von für Object Teams notwendigen *javac*-Features ist die Heraustrennung von eingeführten *inner classes* aus bestehenden *javac*-Klassen eine aufwendigere Aufgabe. Es handelt sich hierbei vor allem um die von Object Teams eingeführten AST-Klassen, die Subklassen von `Tree` und als statische innere Klassen in `Tree` realisiert sind, da der Compiler bei der Bearbeitungen der AST im Verlauf des Compilierungsvorgangs stets Subklassen von `Tree` erwartet. Zur Object

Teams-AST-Erweiterung gehören vier `Tree`-Subklassen, z.B. die Klasse `Tree.BindingDeployment`. Diese müßten aus der Klasse `Tree` herausgenommen und separat definiert werden, wobei sie weiterhin von `Tree` erben müßten. Die Bearbeitung aller AST erfolgt in den meisten Compiler-Phasen durch Visitor-Klassen, die im *javac* enthalten sind, z.B. die Klassen `Attr`, oder `Gen`. Die Methoden zur Bearbeitung von Object Teams-AST müsste ein extern definierter Visitor kapseln, um die Modifikationen an den *javac*-Visitors zu umgehen. Damit hätte man sowohl die Object Teams-Datenstruktur als auch die Bearbeitungsklasse für diese Datenstruktur modular und unabhängig vom *javac* erfasst. Das korrekte Zusammenspiel zur Laufzeit von Objekten dieser Klassen mit Objekten der *javac*-Klassen könnte der folgende Entwurf gewährleisten.

Es findet bei der Verwendung des Visitor Musters ein sogenannter *double dispatch* bei Methodenaufrufen statt: der Aufruf `tree.visit(visitor)` hat zur Folge, dass erstens die `visit`-Methode der Klasse der dynamischen `Tree`-Instanz aufgerufen wird (erster *dispatch*). Im Rumpf dieser Methode steht typischerweise die Anweisung `visitor.visit(this)`, die eine der überladenen `visit`-Methoden anhand des Typs des aktuellen Parameters (der `Tree`-Instanz) dynamisch auswählt und aufruft (zweiter *dispatch*). Mit Object Teams-Funktionalität können beide Auswahlverfahren „abgehört" werden.

Alle *javac*-Visitor haben eine *accept*-Methode mit der Signatur `void _case(Tree tree)`, die eine Exception wirft; diese Methode wird normalerweise nie aufgerufen, da das Argument immer spezieller als `Tree` ist, wofür wiederum Methoden definiert sind. Diese Methode müsste per *callin* an eine Art „Interceptor"-Rollenmethode gebunden werden:

```
public open class JavacListener playedBy JavacVisitor {
    void intercept(Tree tree) <- replace void _case(Tree tree);
    ...
}
```

Ohne dieses Binding würde die allgemeine `_case`-Methode des `JavacVisitor` (im konkreten Fall z.B. Objekte der Klassen `Attr` oder `Gen`) aufgerufen werden, da der `JavacVisitor` den Typ `BindingDeployment` nicht kennt und dafür nur diese allgemeine Methode zur Verfügung stellt. Die Bindings-Definition hat jedoch zur Folge, dass wenn – für *javac* fälschlicherweise – `_case` mit einem Argument aufgerufen wird, das von einem unbekanntem Subtyp von `Tree` ist, der Aufruf zuerst an eine Methode der Team-Rolle `JavacListener` geht. Damit wäre das erste *dispatch* nach Object Teams-Klassen „umgebogen". Die Methode `intercept` könnte wie folgt implementiert werden:

```
public callin void intercept(Tree tree) {
    if (tree instanceof BindingDeployment) {
        tree.visit(otVisitor);
    } else {
        base(tree);
    }
}
```

Der Parameter `otVisitor` hat die Object Teams-Klasse zum Typ, die den AST mit Object Teams-Informationen ausstatten kann, womit der zweite *dispatch* vollzogen wäre. Damit der Kontrollfluss bei der Bearbeitung des AST nicht im Object Teams-Visitor steckenbleibt und zu Fehlern führt, wenn *sub trees* der `BindingDeployment`-Instanz *javac*-`Tree`-Typen sind, muss es auch einen `OtListener` geben, der den `OTVisitor` „abhört“:

```
public open class OtListener playedBy OTVisitor {
    void intercept(Tree tree) <- replace void _case(Tree tree);
    ...
}
```

Diese `intercept`-Methode wäre entsprechend implementiert:

```
public callin void intercept(Tree tree) {
    if (!(tree instanceof BindingDeployment)) {
        tree.visit(javacVisitor);
    } else {
        base(tree);
    }
}
```

Um alle speziellen Object Teams-AST-Typen abzudecken, müßte das Beispiel analog erweitert werden.

Nach diesen bisherigen Überlegungen scheint es prinzipiell möglich, die Object Teams-spezifischen Erweiterungen von *javac*-Code zu lösen und mit Teams und Bindings zumindest aus *javac* erneut einen Object Teams-Compiler zu generieren, auch wenn bei der tatsächlichen Umsetzung der Überlegungen sicher weitere Schwierigkeiten auftauchen werden. Inwieweit mit einem *otc* und Teamdefinitionen ein anderer Compiler wie Kopi oder auch der in der Eclipse-IDE verwendete Eclipse-Java-Compiler um Object Teams-Funktionalität erweitert werden kann, könnte erst eine Analyse des Sourcodes dieser Open Source-Java-Compiler sicherstellen.

Das letzte Kapitel dieser Arbeit versucht, die Erfahrungen und gelernten Strategien beim Erweitern einer ursprünglich unbekannten Software zu beschreiben, dabei erkannte Fehler in der eigenen Arbeitsweise zu dokumentieren und Verbesserungsvorschläge herauszuarbeiten.

Dokumentation der eigenen Arbeitsmethodik

Die durch diese Arbeit gestellte Aufgabe lautete „Erweiterung des Java-Compilers für verbesserte Modularität durch aspekt-orientierte Techniken“, wobei die behandelte Haupttechnik das Programmiermodell Object Teams darstellte. Dies umfaste drei Teilaufgaben: zuallererst das Verständnis des Compilers, dann die Erarbeitung eines Konzepts, wie die Anforderungen des Object Teams-Modells umgesetzt werden können und schliesslich die Implementierung dieser Anforderungen. Diese Aufgaben wurden nicht sukzessive bearbeitet, sondern iterativ. Eine sukzessive Abarbeitung im Sinne eines Top-Down-Prozesses war nicht möglich. Viele Funktionsweisen des *javac* wurden erst durch die Implementierung von Object Teams-Anforderungen erkannt; z.B. wurde durch den Versuch und die dabei auftauchenden Probleme,

callin-Attribute zu generieren, vieles darüber in Erfahrung gebracht, wie der *javac* Bytecode erzeugt. Oder es wurde bei der Umsetzung einer Anforderung deutlich, dass deren Implementierung einen unverhältnismäßig hohen Aufwand nach sich ziehen würde, worauf die Anforderung angepasst wurde, was z.B. für die Anforderung der compiler-seitigen Umsetzung eines *base call* innerhalb einer *callin*-Methode geschah.

Analyse-Prozess

Als erstes galt es, sich die Grundarchitektur eines Compilers und dessen Phasen zu vergegenwärtigen. Auch wenn Compiler-Technik Thema in einigen Grund- und Hauptstudiumsveranstaltungen ist, so wird in diesen eher selten ein kompletter Compiler entwickelt oder ein vorliegender angepasst. Die z.B. in [28] beschriebenen Compiler-Konzepte konnten den Paketen und Klassen des *javac* teilweise zugeordnet werden, was einen groben Überblick über die Sourcen und die darin verteilten Funktionalitäten ermöglichte. Ausserdem zeigte sich, dass ein gewisses Mass an Softwarearchitektur-Kenntnis sehr zur dazu beiträgt, einige grundsätzliche Design-und Implementierungsdetails einer vorliegenden Software bereits dadurch zu erkennen, dass man dieser eine bekannte Architektur zuordnen kann. Im Falle des *javac* konnte gemutmaßt werden, dass es sich um eine Art *Pipes 'n' Filters*-Architektur handelte, wobei die Compilerphasen die Filter, die Pipes die wie auch immer gearteten Schnittstellen zwischen den Phasen darstellen mußten. Dadurch wurden bereits Annahmen über die Implementierung möglich (die sich größtenteils auch bestätigten):

- *Pipes 'n' Filters*-typisch: die Reihenfolge, in der die einzelnen Filter durchlaufen werden, ist stets dieselbe, und wird nicht zur Laufzeit entschieden. Annahme: diese feste Reihenfolge muss sich im Code irgendwo widerspiegeln.
- *Pipes 'n' Filters*-typisch: zwischen den Phasen werden Daten übergeben. Annahme: es existieren definierte Schnittstellen zwischen den Phasen, an denen die Daten übergeben werden. Desweiteren wird in allen Phasen dieselbe Grunddatenstruktur verwendet.
- *Pipes 'n' Filters*-typisch: die einzelnen Phasen kennen sich gegenseitig nicht, laufen jedoch nicht unabhängig voneinander ab, sondern bilden einen Dependenzgraphen.
  Annahme: es existieren keine Assoziationen zwischen Klassen, die einzelne Phasen implementieren. Ausserdem folgt daraus ebenfalls die erste Annahme.

Durch diese Annahmen konnte die „Hauptroutine" des Compilers, in der die einzelnen Phasen angestossen werden, realtiv einfach identifiziert werden: die Methode `compile` der Klasse `JavaCompiler`. Dieser ist zwar eine andere Methode vorgeschaltet (die Methode `compile` der Klasse `Main`), diese arbeitet jedoch lediglich die Kommandozeilen-Argumente ab und ruft dann die Hauptroutine auf.

Neben einer Architektur-Grundannahme kann es ebenfalls nützlich sein, Kenntnisse von allgemeinen Anforderungen an bestimmte Software-Typen zu haben. Dazu gehören auch nicht-funktionale Anforderungen wie z.B. Performanz und Ressourcenverbrauch. Letztere werden an einen Compiler im besonderen Maße gestellt. Er wird während der Softwareentwicklung oft gestartet und muss damit performant sein. Desweiteren wird er oft dahingehend entwickelt, auf sehr unterschiedlichen Hardwareumgebungen einsetzbar zu sein, auch auf stark ressourcenbegrenzter Hardware wie z.B. Mobil-Geräten, wodurch seine Benutzung ressourcenschonend sein sollte. In Kombination mit dem Wissen um die oben genannte *Pipes 'n' Filters*-Architektur kann von einem Compiler deshalb erwartet werden, dass er, was die Bearbeitung der verwendeten Datenstrukturen betrifft, nur soviel Informationen wie nötig dazu heranzieht. Diese Erwartung spiegelt sich im Aufbau und in der Verwendung eines AST direkt wieder, der z.B. einen Klassentyp mit allen weiteren Typen darstellen kann, die sich in dessen Scope befinden. Zur Compilierung müssen jedoch nicht alle diese Typen verfügbar sein, wenn sie nicht von einem AST-Element direkt verwendet werden. Zur Compilierung einer Klasse, die das klassische *hello world*-Beispiel implementiert, muss z.B. nicht der Typ `ThreadGroup` verfügbar sein, obwohl dieser im Scope der Klasse liegt. Die komplette Austattung des AST mit allen Typ-Elementen wäre somit performanz- und ressourcenverschwendend. Sinnvoller ist es, Informationen zu Typen nur dann zu suchen und zu erzeugen, wenn sie tatsächlich gebraucht werde, also eine Art *lazy type resolving* durchzuführen. Diese Technik zur Erfüllung der genannten Anforderungen spiegelt sich im *javac* insofern wider, als dass die Symbole von Features, die Klassensymbolen angehören, nur bei Bedarf mit zusätzlichen Informationen ausgestattet werden. Die *javac*-Implementierung sieht dafür die Methode `complete` der Klasse `Symbol` vor, die ein Objekt der Klasse `ClassReader` benutzt. Dieser *ClassReader* implementiert seinerseits ein `Completer`-Interface und lädt für die Klasse `Symbol` nur bei Bedarf die Typen, die zur Compilierung notwendig sind.

Ab diesem Zeitpunkt kam der Debugger zum Einsatz, mit dem in beliebigen Phasen der Compilerungsvorgang unterbrochen und der Zustand des Compilers über das Auslesen von Variablenwerten inspiziert werden konnte. Auch wenn dies erste detaillierte Einblicke in die Funktionsweise des Compilers ermöglichte, so konnte diese dadurch nicht insoweit verstanden werden, als dass die Analysephase damit endgültig abgeschlossen werden konnte. Die Abläufe während des Compilierungsvorgangs sind dafür einfach zu komplex, es würde unverhältnissmaßig viel Zeit in Anspruch nehmen, vor der Implementierung neuer Funktionalitäten alles verstehen zu wollen.

Auch wenn nach der Analyse somit einige Fragen offen blieben, so wurden auch einige erschöpfend beantwortet, was nicht unwesentlich durch die *javac*-seitige Verwendung von *Design Pattern* begünstigt wurde. Ähnlich wie die erkannte Software-Architektur Rückschlüsse auf Design- und Implementierungsdetails ermöglichte, so ermöglichten die verwendeten Design Patterns (vor allem *Visitor*,

Composite und *Abstract Factory*) das rasche Verständnis grundlegender Abläufe während des Compilierungsvorgangs, sowohl was die Erzeugung von Objekten als auch was die Interaktion zwischen Objekten zur Laufzeit im *javac* betrifft.

Konzepterstellungsprozess

Während und nach dem vorläufigen Abschluß der Analyse wurden die schriftlichen Arbeiten begonnen, die vor einer Erweiterung des *javac*-Sourcecodes notwendig waren. Dazu gehörte die Erweiterung der Java-Syntax, in die die Object Teams-Konstrukte integriert wurden, und die Erstellung eines Konzepts, wie die Object Teams-Konstrukte implementiert werden sollten, also deren genaue Umsetzung. Dazu wurde ein einfaches und effektives Medium verwendet. Für alle erfassten Informationen, die man einem klar umrissenen Teilthema der Object Teams-Realisierung zuordnen konnte, wurde eine ASCII-Datei erstellt. Diese wird im folgenden „Karte" genannt, da der Ausdruck dieser Datei auf DIN A6-Format erfolgen sollte und diese Größe „Kartencharakter" hat. Hier ein Beispiel für den Inhalt einer Karte:

```
Eine Klasse muß als open deklariert werden, wenn sie
(a) (mindestens) eine callin Methode, oder
(b) inner open classes, oder
(c) bindings
enthält.

Die open Deklaration signalisiert, daß die Klasse nicht
vollständig ist, und daß die fehlende Funktionalität nicht
per Klassen-Vererbung, sondern per Objekt-Vererbung ergänzt
werden muß.

Eine Klasse muß als abstract deklariert werden, wenn sie
(a) abstract im Sinne der Sprache Java ist, oder
(b) (mindestens) eine nicht gebundene callin Methode
enthält.

Die abstract Deklaration signalisiert, daß die Klasse nicht
vollständig ist, und daß die fehlende Funktionalität per
Klassen-Vererbung (a) bzw. per Objekt-Vererbung (b) ergänzt
werden muß.


Die Kombination von open und abstract ist möglich.

@@well-formedness
@author=stephan,christof
@date=020718
@state=fixed
```

Diese Karte erfaßt die Voraussetzungen für die Verwendung der Modifier *abstract* und *open* und stellt eine „Wohlgeformtheitsregel" dar, die im Compiler implementiert werden kann. Nach dem eigentlichen Informationsblock wird die Karte klassifiziert und mit Informationen über die Autoren, das Datum und den Zustand versehen. Von diesen Karten sind bis zum aktuellen Zeitpunkt dreissig erstellt worden, es enstanden die Kategorien *well-formedness*, *implementation*, *concept*, *syntax* und *dynamic-semantic*. Das Kartensystem erwies sich aus hauptsächlich zwei Gründen als besonders praktikabel. Erstens konnte es von den Beteiligten – dem

Autor dieser Arbeit, der Kommilitonin, die das Object Teams-RE enwickelte und dem gemeinsamen Betreuer – unkompliziert genutzt werden. Es enstand dadurch sehr wenig organisatorischer Overhead, trotzdem konnten wichtige Informationen erfasst und für alle über CVS (*Concurrent Versions System)* zugänglich gemacht werden. Zweitens unterstützte es den dynamischen Charakter des Konzepts: anfangs war noch nicht klar, wie bestimmte Teilkonzepte konkret umzusetzen sind, manches änderte sich auch während der Implementierung des *otc* und des Object Teams-RE. In diesem Fall konnten die Karten einfach angepasst werden, weil wenig Abhängigkeiten unter ihnen bestand und jede ein bestimmtes Teilkonzept kapselte. Ein formal strengerer Dokumentationsansatz hätte die Konzeptentwicklung eventuell weniger gut unterstützt.

Implementationsprozess

Nach der Erstellung der erweiterten Syntax wurde damit begonnen, die auf den Karten erfassten Konzepte umzusetzen, d.h. die *javac*-Sourcen anzupassen. Dies konnte teilweise durch direkten Eingriff geschehen, wenn die notwenige Code-Stelle und die Modifikation selbst offensichtlich waren, z.B. wenn die Menge der möglichen Modifier um *team* erweitert werden musste. Waren Code-Stelle und Modifikation allerdings nicht offensichtlich, so musste der *javac* durch zu compilierende Teams „provoziert“ werden, Fehler zu melden, um so die notwendige Stelle und Form der Modifikation herauszufinden. Der *javac* bekam also ein Team als Eingabe und meldete an der ersten Stelle, an der er mit den Teamdefinitionen „Schwierigkeiten“ hatte, einen Fehler. Durch das Setzen von *breakpoints* im Debugger konnte die Stelle, die den Fehler hervorrief, nach und nach lokalisiert werden. Nach der Lokalisation erfolgte die Anpassung des *javac*, wonach er einerseits Object Teams-Konstrukte akzeptieren und andererseits die notwenigen Informationen für diese Konstrukte generieren musste. Dieser Vorgang – Fehler provozieren, Code-Stelle finden, Code anpassen – wurde für alle Compilerphasen durchgeführt, wobei der Code des *javac* immer besser verstanden und mögliche Probleme teilweise schon im Voraus erkannt werden konnten.

Als wichtige Variable in diesem Prozess erwies sich der Input des Compilers, also die zu compilierenden Object Teams-Definitionen. Da vor allem am Anfang des Anpassungsvorgangs die *javac*-Funktionalität noch nicht gänzlich verstanden war, war es wichtig, das *javac*-Verhalten durch möglichst einfache Object Teams-Definitionen zu „beruhigen“, d.h. die Anzahl der Methodenaufrufe möglichst gering zu halten, um möglichst eindeutige Rückschlüsse darüber ziehen zu können, wo und vor allem wodurch ein Fehler verursacht wurde. Konnte man, was die Ursache des Fehlers betrifft, eine Hypothese aufstellen, so konnte man diese Hypothese anhand regulärer Java-Klassen überprüfen, indem man die regulären Java-Klassen als Input benutzte, den Compiler an der gewissen Stelle erneut anhielt und dessen Zustand überprüfte. Lag man mit der Hypothese richtig, so konnte die entsprechende Anpassung am Code erfolgen. Lag man falsch, so hatte man eine falsche Möglichkeit ausgeschaltet und musste neu überlegen. Als Beispiel sei eine durch Object Teams-Code aufgetretene `NullPointer`-Exception angeführt. Die Stelle, an der diese

Exception auftrat, wurde in einer Klasse gefunden, die Type-Checking durchführt. Sie kam dadurch zustande, dass ein Symbol-Feld dereferenziert wurde, das `null` war. Als Hypothese für die Ursache wurde angenommen, dass das Feld des Symbols, das zu einem Object Teams-Konstrukt gehörte, fälschlicherweise noch nicht initialisiert worden war. Eine Überprüfung dieser Hypothese mit einer regulären Java-Klasse ergab jedoch, dass die Stelle, an der die Dereferenzierung auftrat, überhaupt nicht durchlaufen wurde, wodurch eine neue Hypothese aufgestellt werden musste. Die Ursache des Fehlers wurde schliesslich darin gefunden, dass ein anderes Feld des Object Teams-Konstrukts mit einem falschen Wert ausgestattet war und aufgrund dieses Werts die Methode fälschlicherweise durchlaufen wurde, die den Fehler nach sich zog.

Die Object Teams-Definitionen, die als Input dienten, waren anfangs, wie bereits erwähnt, einfach gehalten und wurden nach erfolgreicher Compilierung mit immer mehr Features angereichert, deren Umsetzung dann erneut nach dem obigen Verfahren implementiert wurden. Zuerst wurden Teams mit einfachen *callout*-Bindings verwendet. Diese *callout*-Bindings wurden durch *parameter mappings* erweitert, diese wiederum anschliessend mit *expressions*. Danach fand eine Zäsur statt und Teams mit *callin*-Bindings wurden als Input verwendet. Nach einer erneuten Zäsur wurden schliesslich Teams compiliert, in denen Rollenvererbungen definiert waren. Die Zäsuren zwischen den zu compilierenden Teams hatten manchmal den bekannten Effekt, dass neue Features korrekt umgesetzt wurden, alte Features, deren Umsetzung bereits vorher implementiert worden war, nun nicht mehr funktionierten. Diesem Problem kann man normalerweise mit Regressionsstests begegnen, indem man für ausgewählte Methoden Tests schreibt (z.B. mit dem Testframework JUnit) und diese nach jeder grundlegenden Code-Veränderung durchführt. Das Erstellen von Tests, die eine gewisse Aussagekraft besitzen, ist innerhalb eines Compilers allerdings schwierig, da die Pre-und Post-Conditions von Compilermethoden – vor allem derer, die Hauptfunktionalitäten implementieren – schwierig zu erfassen sind. Grund hierfür sind wiederum die vielen Seiteneffekte dieser Methoden, die auch nicht unbedingt konstant sind, sondern z.B. vom Typ aktueller Parameter abhängen können.

Als praktikabler Test erwies sich die Re-Compilierung aller bereits mit Erfolg compilierter Object Teams nach jeder grundlegenden Änderung des Compiler-Codes. Damit dieser Vorgang so einfach wie möglich durchgeführt werden konnte, wurde das Build-Tool *ant* [29] eingesetzt, dessen *build file* (ähnlich einem Makefile) die *otc*-Aufrufe für alle bereits bearbeiteten Team-Dateien beinhaltete. Lief der *ant*-Aufruf mit diesem *build file* ohne Fehlermeldungen durch, so konnte man relativ sicher sein, dass die vorher implementierten Features weiterhin korrekt umgesetzt wurden. Um gänzlich sicher zu gehen, mussten die erzeugten *class files* decompiliert werden, um die gegebenenfalls erzeugten Object Teams-spezifischen Elemente zu überprüfen. Dieser Vorgang war jedoch nicht Teil des ant-Skripts, da die decompilierte Klasse mit einer ASCII-Datei zu vergleichen wäre, die für jeden Testfall erzeugt werden müsste. Wenn dieser Vergleich von Formatierungsunterschieden,

wie sie in der Vergleichsdatei und in der vom Decompiler erzeugten Datei auftauchen, abstrahieren soll, müßte für diesen Test zuerst ein Programm erstellt werden, das diesen Vergleich durchführt. Da der Aufwand dafür im Vergleich zum Nutzen als zu hoch eingeschätzt wurde, wurden Object Teams-*class files* nur im Bedarfsfalle selektiv decompiliert und „mit dem Auge“ auf ihre Korrektheit überprüft.

Was die Input-Dateien betrifft so sollte man abschliessend betonen, wie wichtig deren genaue Anfertigung war. Sie waren Ausgangspunkt der meisten Compiler-Modifikationen, ein vermeintlich unwesentlicher Fehler konnte weitreichende Auswirkungen auf die Modifikationen am *javac*-Code haben. Würde eine Input-Team z.B. eine Rolle enthalten, in der nach einem Binding-Deployment das Semikolon fehlen würde und der Parser daraufhin angepasst, so hätte dies u.U. mehrere falsch angepasste Parsermethoden zur Folge. Auch waren sprechende Namen der Elemente in diesen Teams hilfreich beim Debuggen des *javac*. In den Input-Teams war z.B. in allen Rollennamen das Präfix „Role“ enthalten, was es z.B. beim Inspizieren eines Classensymbols ermöglichte, dieses sofort als Rollensymbol zu erkennen.

Prozessübergreifende Erfahrungen

Während zu Beginn der Umsetzung der Object Teams-Features vor allem die dahinter liegenden Konzepte das Hauptgewicht der Arbeit darstellten, so kippte dies im Laufe dieser Arbeit immer mehr zugunsten reiner Implementierungsaufgaben. Wichtig hierbei war, trotz aller wichtiger Code-Details, das „Big Picture“, das Gesamtkonzept nicht aus den Augen zu verlieren. Sonst hätte die Gefahr bestanden, am Konzept vorbei zu programmieren. Eine gute Hilfe hierbei waren neben den häufigen Treffen der Beteiligten die bereits dargestellten Karten. Durch das gelegentliche Lesen, durch das Anlegen neuer und das Anpassen bestehender Karten konnte man sich die Anforderungen immer wieder vergegenwärtigen.

Der letzte Punkt, der rückblickend auf die Erstellung der Object Teams-Implementierung von Bedeutung ist, ist weniger technischer als menschlicher Art. Man könnte ihn vielleicht mit „Keine Angst vor schwarzen Löchern“ oder „...vor Sümpfen“ umschreiben. Einen Entwickler, der ein bestehendes Softwaresystem modifizieren will und dieses nicht kennt, kann man als eine Art Entdecker ansehen; ob er dabei im Weltall oder im Dschungel umherstreift, ist dabei vielleicht eher zweitrangig. Wichtig bei diesem Vergleich ist, dass er die Elemente in diesem System und die Regeln, nach denen die Elemente interagieren, nicht kennt. Diese Unwissenheit sollte jedoch keine Angst oder Scheu davor aufkommen lassen, spielerisch Veränderungen am System vorzunehmen, denn nur so können die Zusammenhänge in Erfahrung gebracht werden. Die Einstellung, nach der man erst bei absoluter Sicherheit über die Konsequenzen des Eingriffs mit diesem Eingriff startet, hätte die Entwicklung des Object Teams-Compiler nicht beschleunigt, wahrscheinlich eher verlangsamt. Eine weitere Scheu, die es abzulegen galt, war, Lösungen von Problemen zu verschieben und mit der Lösung anderer Probleme fortzufahren. Wurde dies nämlich getan, so

zeigten sich bei der Lösung des neuen Problems oft Hinweise darauf, wie das erste zu lösen war.

Abschliessen möchte ich diese Arbeit mit einem Zitat vom Donald E. Knuth, dem Verfasser des berühmt-berüchtigten fünf-bändigen Werks „The Art of Computer Programming“. Dieses Zitat ist nicht aus einem dieser Bände, sondern erschien in einem Interview mit Knuth in der Computerzeitschrift ´ct (5/2002, S.190) und beschreibt meine Arbeitsweise beim Erweitern des *javac* für Object Teams perfekt:

„Man versteht etwas nicht wirklich, wenn man nicht versucht, es zu implementieren.“

Literatur und Quellen

[1] Betrand Meyer. *Object-orientated Software Construction*. Prentice Hall PTR, 1997.

[2] *Object Management Group* (OMG) zu UML, WWW: http://www.uml.org.

[3] P. Tarr, H. Ossher, W. Harrison and S. Sutton Jr. *N Degrees of Separation: Multi-Dimensional Separation of Concerns*. In: Proceedings of ICSE'99.

[4] P. Tar and H.Ossher. *HyperJ User and Installation Manual*. IBM Corporation, 2000.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. *Getting Started with AspectJ*. Xerox Corporation, submitted for CACM 2001.

[6] Open Source Initiative (OSI). *Open Source Approved Licenses,* http://www.opensource.org/licenses/index.php

[7] M. Mezini und K. Lieberherr. *Adaptive plug-and-play components for evolutionary software development*. In C. Chambers, editor, *Object-Orientated Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, number 10, pages 97-116, Vancouver, October 1998. ACM.

[8] K. Lieberherr, D. Lorenz und M. Mezini. *Programming with Aspectual Components*. In: Technical Report, Northeastern Universitiy, April 1999.

[9] S. Herrmann and M. Mezini. *PIROL: A case study for multidimensional separation of concerns in software engineering environments*. In: *Proceedings of OOPSLA 2000*. ACM, 2000.

[10] S. Herrmann und M. Mezini. *Combining composition styles in the evolvable language LAC*. In: *Proceedings of AsoC workshop at the 23nd ICSE*, 2001.

[11] Die Programmiersprache LUA, WWW: http://www.lua.org

[12] S. Herrmann. *Object Teams: Improving Modularity for Crosscutting Collaborations*. In: Proceedings of Net Object Days 2002.

[13] Java 2 Platform, Standard Edition (J2SE), Sun Community Source Licensing. WWW: http://wwws.sun.com/software/java2/index.html

[14] G. Kniesel, P. Costanza, M. Austermann. *JMangler - A Framework for Load-Time Transformation of Java Class Files*. In: *Proceedings of IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, November 2001. WWW: http://javalab.cs.uni-bonn.de/research/jmangler

[15] M. Campione, K. Walrath, A. Huml, *The Java Tutorial: A Short Course on the Basics*, 3. Edition, Dezember 2000. Addison Wesley. WWW: http://java.sun.com/docs/books/tutorial/index.html

[16] A. V. Aho, R. Sethi, J. D. Ullmann. *Compilerbau*. 2. Auflage, Oldenbourg.

[17] Die Eclipse IDE, WWW: http://www.eclipse.org

[18] Der Decompiler *jad*, WWW: http://kpdus.tripod.com/jad.html

[19] Das graphische Front-End *dj* für jad WWW: http://members.fortunecity.com/neshkov/dj.html

[20] G. Bracha, J. Gosling, B. Joy and G. Steele: *The Java Language Specification*. 2. Auflage, Juni 2000, Addison Wesley. WWW: http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

[21] T. Lindholm, F. Yellin. *The Java Virtual Machine Specification*. 2. Auflage, April 1999, Addison Wesley. WWW: http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html

[22] Die NetBeans IDE, WWW: http://www.netbeans.org

[23] M. Fowler, K. Beck, D. Roberts, W. Opdyke: *Refactoring: Improving the Design of Existing Code*. 1999, Addison Wesley.

[24] Der Kopi Java-Compiler, WWW: http://www.dms.at/kopi

[25] Die Programmiersprache Ruby, WWW: http://www.ruby-lang.org

[26] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns – Elements of Reusable Object-Orientated Software*. 1995, Addison Wesley.

[27] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler. *Making the future safe for the past: Adding Genericity to the Java Programming Language*. In: *Proceedings OOPSLA 1998*, Vancouver, October 1998. ACM. WWW: http://www.cis.unisa.edu.au/~pizza/gj/Documents

[28] Andrew W. Appel. *Modern Compiler Implementation in Java*. 1998, Cambridge University Press.

[29] Stefan Edlich. *Ant – kurz & gut*. 2002, O'Reilly. WWW: http://jakarta.apache.org/ant/index.html

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