# Schemaerhaltende Bytecodetransformationen zum Aspektweben zur Programmlaufzeit

Diplomarbeit
bei Prof. S. Jähnichen

vorgelegt von
Michael Flüh
Matr. Nr. 190238
Fachgebiet Softwaretechnik
am Institut für Softwaretechnik und Theoretische Informatik
Technischen Universität Berlin

Gutachter: Prof. Dr-Ing. Stefan Jähnichen
2.Gutachter: Dr.-Ing. Stephan Herrmann
Betreuung: Dipl.-Inf. Christine Hundt

Berlin, 8. Dezember 2006

# Inhaltsverzeichnis

# Einleitung

Der von Dijkstra geprägte Begriff *Separation of Concerns*, bezeichnet die Methode, die Komplexität eines Problems in möglichst unabhängige, weniger komplexe Teilprobleme herunterzubrechen. Durch die Komposition der gelösten Teilprobleme kann dann das ursprüngliche Problem leichter gelöst werden. In der Software-Entwicklung entspricht diese Herangehensweise der Fokussierung auf ein Modul oder die Prozedur eines Programms. In der Evolution der Programmiersprachen war es stets ein Bestreben, ideale Konzepte der Modularisierung zu finden. *Aspektorientierte Programmierung* (AOP) bietet eine neue Modellierungsebene, um Anforderungen zu separieren. Eine Technik zur Realisierung dieser neuen Modularisierungsform wird durch *Code-Transformationen* erzielt. Dafür wird der Code der Aspekt-Module in eine Basisanwendung ”hineingewoben”. Im Gegensatz zum üblichen Vorgehen, in dem die Transformationen zu einem festgelegten Zeitpunkt erfolgen, erlauben dynamische Webeverfahren das Weben von Aspekten zur Laufzeit eines Programms.

Der überwiegende Teil der bestehenden AOP Implementierungen erweitert vorhandene Programmiersprachen und stellt zudem keine eigenen Laufzeitumgebungen zur Verfügung, welche auf die Bedürfnisse der AOP zugeschnitten sind. Dies ist insbesondere für ein *dynamisches Weben* ein Problem, da Laufzeit-Transformationen eine Schnittstelle zur einer Laufzeitumgebung benötigen.

Die Laufzeitumgebung der Programmiersprache Java bietet solche Schnittstellen, jedoch in einem beschränkten Rahmen. Der Bytecode von Java-Klassen darf zur Laufzeit zwar verändert werden, jedoch das *Schema* (Signatur) der Klasse nicht. Die Code-Transformationen müssen also *schemaerhaltend* sein.

Die vorliegende Arbeit beschreibt die Umsetzung eines Laufzeit-Webers für die aspektorientierte Sprache *ObjectTeams/Java*, die sich diesen Beschränkungen stellen muss.

Eine Einführung in die grundlegenden Konzepte der aspektorientierten Programmierung wird in Kapitel 1 vorgenommen.

Kapitel 2 beschäftigt sich im ersten Teil mit den Konzepten des Programmierungsmodells *Object Teams* und erläutert im zweiten Teil die Webestrategie, die zur Zeit Verwendung findet und nicht dynamisch ist.

Ein Überblick zum dynamischen Weben im Java-Umfeld wird in Kapitel 3 gegeben. Es stellt darüber hinaus die Java-API *Java Programming Language Instrumentation Service* vor, die Bytecode-Transformationen zur Laufzeit ermöglicht und Grundlage des zu realisierenden Webers sein wird.

Kapitel 4 stellt im ersten Teil die Probleme der Transformationseinschränkungen dar und schlägt anschließend eine Lösung vor, die einem Laufzeit-Weben gerecht wird.

Ein dynamischer Weber für Object Teams benötigt neben der Möglichkeit, überhaupt Laufzeittransformationen durchzuführen, zusätzliche Webeinformationen, die nicht ohne weiteres zur Laufzeit zur Verfügung stehen. Welche Laufzeitinformationen dies sind, und wie sie zur Verfügung gestellt werden können, wird unter anderem im Kapitel 5 beschrieben.

Kapitel 6 fasst die Ergebnisse der Diplomarbeit zusammen und gibt einen Ausblick auf mögliche zukünftige Entwicklungen.

# Kapitel 1

# Aspektorientierte Programmierung

## 1.1 Abstraktion und Modularisierung

Abstraktion und Modularisierung sind zwei fundamentale Konzepte, um Komplexität beherrschbar zu machen. Abstraktion erlaubt unterschiedliche Sichten auf ein Softwaresystem bezüglich der Detailtreue. Modularisierung strukturiert das System nach seinen Aufgaben bzw. Anforderungen. Je ausgeprägter diese Konzepte in Programmiersprachen wiederzufinden sind, desto größer ist die Möglichkeit, gut strukturierte Softwaresysteme zu entwerfen, die den Kriterien der Wiederverwendbarkeit und der Wartbarkeit Folge leisten. Nach deren Anforderungen klar voneinander abgegrenzte Moduleinheiten begünstigen während des gesamten Entwicklungszyklus eine effiziente Umsetzung. So sind z.B. Modultests durchführbar, bevor die Module in das Gesamtsystem integriert werden. Des Weiteren wird das Auffinden von Fehlern innerhalb eines Programms durch die Lokalität einer implementierten Anforderung erleichtert.

Die *objektorientierte* Programmierung (OOP) ist das vorherrschende Programmierparadigma für *imperative* Programmiersprachen. Der Weg dorthin führte über die *prozedurale* Programmierung, die das Programm in einzelne Prozeduren zerlegte, hin zur *modularen* Programmierung, die zusammengehörige Daten und Prozeduren in Module fasst. Die OOP führt den Begriff der Klasse als instanzierbares Modul ein. Objekte sind Instanzen einer Klasse, die interagieren. Durch die Vererbungsbeziehung zwischen Klassen wurde eine neue und mächtige Strukturierungsdimension geschaffen, denn sie erlaubt es, ein Modul auf verschiedenen Abstraktionsebenen zu betrachten und zu benutzen.

### 1.1.1 Cross-Cutting-Concerns

Die Konzepte der OOP haben sich in der Praxis als sehr geeignet erwiesen, jedoch existieren Anforderungen an Systeme, welche zwar die konzeptionellen Eigenschaften eines Moduls haben, sich jedoch nicht entlang der objektorientierten Modulstruktur implementieren lassen. Anforderungen werden als *Cross-Cutting-Concerns* bezeichnet, wenn bei deren Implementierung unweigerlich mehrere Klassen involviert werden, wodurch die Modulstruktur aufgeweicht wird.

Typische strukturelle Eigenschaften von Cross-Cutting-Concerns, die einer sauberen Modularisierung des Programmcodes entgegenwirken, sind:

- *Code-Scattering*: die Verteilung einer Anforderung auf mehrere Klassen
- *Code-Tangling*: die Vermischung verschiedener Anforderungen in einer Klasse

Ein Beispiel für ein Cross-Cutting-Concern zeigt sich in der Umsetzung des *Model-View-Controller* Architekturmusters. Ziel des Musters ist eine klare Trennung der Bestandteile Model, View und Controller. Eine Anforderung an diese Architektur ist, dass das Model bei der Änderung eines bestimmten Zustands die View in Form eines *Updates* benachrichtigt (*Observer-Pattern*). Betrachten wir die Implementierungsebene des Models so stellen wir fest, dass diese Anforderung ein Cross-Cutting-Concern ist, denn wir finden verstreut über dessen Implementierung eine entsprechende *Update*-Funktionalität.

Das Muster erlaubt es, Model und View als zwei abstrakte, austauschbare Dinge zu sehen. Existiert jedoch eine andere View, so ist nicht garantiert, dass die in einem konkreten Model implementierten Update-Funktionen dieser View entsprechen und somit auch keine echte Unabhängigkeit der Implementierung gegeben ist.

Weitere, oft zitierte typische Beispiele für Cross-Cutting-Concerns sind globale System-Anforderungen wie z.B. *Synchronisation*, *Sicherheit* oder Persis*tenz*, oder die als Paradebeispiele geltenden *Tracing*- und *Logging*-Funktionalitäten.

Die Abbildung 1.1[1] veranschaulicht die typische modulübergreifende Implementierung des Loggings. Die roten Markierungen zeigen die Stellen in den unterschiedlichen Modulen an, an denen diese Anforderung realisiert wird.

Aufgrund des Code-Scatterings und Code-Tanglings verhindern Cross-Cutting-Concerns eine saubere Modularisierung des Programmcodes. Dies führt dazu, dass der Programmcode schwieriger zu warten ist, da veränderte Anforderungen nicht mehr lokal korrigierbar sind. Außerdem wird die Wiederverwendbarkeit der Module in anderen Systemen erschwert, da nicht alle implementierten Anforderungen auf das neue System passen.

[1] entnommen aus http://www.parc.com/research/csl/projects/aspectj/downloads/OOPSLA2002-demo.ppt

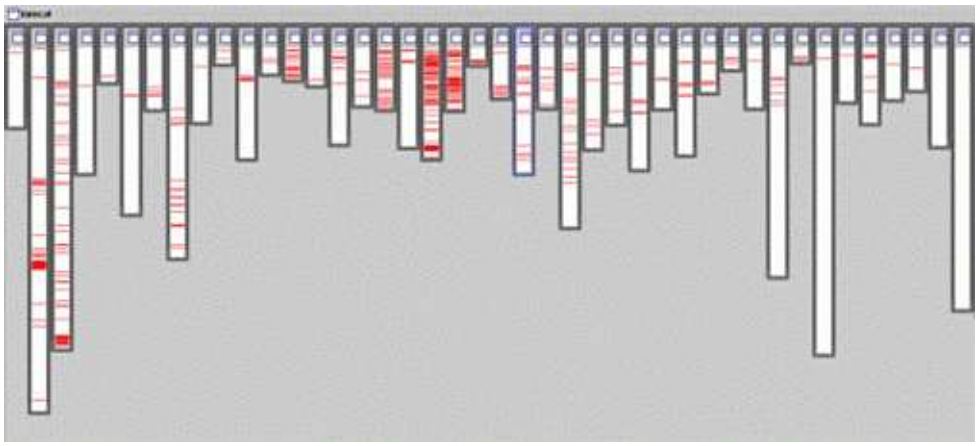

Abbildung 1.1: Zerstreute Implementierung des Loggings

### 1.1.2 Aspektorientierte Programmierung

Aspektorientierte Programmierung [11] (AOP) hat es sich zum Ziel gesetzt, eine bessere Modularisierung von Cross-Cutting-Concerns zu realisieren. Sie versteht klassenübergreifende Anforderungen als die *Aspekte* eines Programms. Kernidee ist es, den *Aspekt-Code* aus der Basisanwendung zu separieren und in *Aspekt-Modulen* zu isolieren. Die Koppelung zwischen Aspekt und Programm erfolgt an sogenannten *Joinpoints*. Joinpoints sind Punkte im Programmfluss an denen ein *Advice* (Aspekt-Code) ausgeführt wird. Es existieren verschiedene Formen eines Advices:

- *before-Advice*: der Advice wird ausgeführt, bevor der Joinpoint auftritt
- *after-Advice*: der Advice wird ausgeführt nach dem der Joinpoint auftritt
- *around-Advice*: der Joinpoint wird durch den Advice ersetzt

Der around-Advice hat zusätzlich die Möglichkeit, durch eine *proceed*-Anweisung den Joinpoint auszuführen.

In Abbildung 1.2 ist anhand eines einfachen Sequenzdiagramms dargestellt, wie eine Methode durch einen before- und einen after-Advice eines Aspekts adaptiert ist.

*Pointcuts* beschreiben eine Menge von Joinpoints und werden z.B. wie in der Sprache AspectJ[10] deklarativ formuliert. Ein Aspekt setzt sich also aus dem Aspekt-Code und einem Pointcut zusammen.

Typische Joinpoints für Aspekte sind die Methodenausführung (*method-execution-point*), der Methodenaufruf (*method-call-point*) oder die Referenzierung eines bestimmten Klassenattributs.

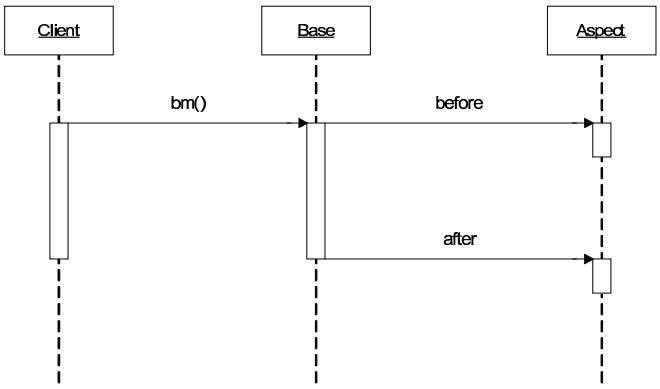


Abbildung 1.2: Adaption einer Methode durch einen Aspekt

## 1.2 Weben von Aspekten

Als Konsequenz der strikten Trennung von Aspekt- und Basiscode benötigt eine aspektorientierte Sprache eine Komponente, welche die Aspekte in die Basisanwendung integriert. Man nennt diesen Prozess *Weben* (*weaving*). Der Aspekt-Weber ist die Komponente die das Weben realisiert. Abbildung 1.3 zeigt die Trennung der Module der Basisanwendung von den Cross-Cutting-Concerns, die durch den Weber zusammengeführt werden.

Es wird unterschieden zwischen statischem Weben, welches die Aspekte zur Compilezeit einfügt und dynamischem Weben, bei dem die Anbindung zur Laufzeit geschieht. *Load-Time-Adaption* ist ein weiteres Verfahren und meint das Einweben der Aspekte zur Ladezeit.

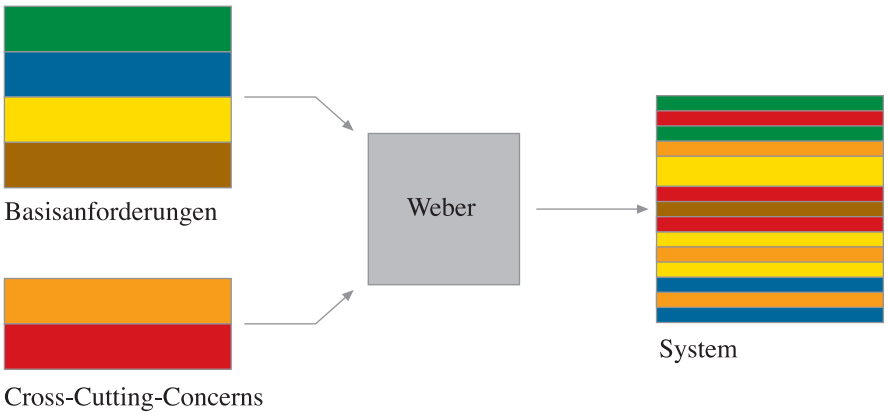


Abbildung 1.3: Weben von Aspekten

Die AOP versteht sich nicht als ein Ersatz, sondern als eine Erweiterung des objektorientierten Programmierparadigmas. Mittlerweile existieren für fast jede geläufige Programmiersprache, wie z.B. *Java*, *C++*, *PHP* oder *Ruby*, aspektorientierte Erweiterungen. AspectJ gilt als ein Hauptvertreter dieser Sprachen.

Im nächsten Kapitel wird die aspektorientierte Programmiersprache *Object Teams/Java* (OT/Java) vorgestellt, in deren Umfeld diese Diplomarbeit entstanden ist.

# Kapitel 2

# Object Teams/Java

## 2.1 Das Programmiermodell Object Teams

*Object Teams* [7] ist sicherlich in dem Sinne eine aspektorientierte Programmiersprache, da es grundlegende Konzepte einer solchen beinhaltet, allerdings sind diese Konzepte eingebettet in ein *kollaborations-* und *rollenbasiertes* Programmiermodell. Unter einer *Kollaboration* versteht man eine Menge von interagierenden Objekten innerhalb eines bestimmten Kontexts zur Erfüllung einer Aufgabe. Die Existenz eines solchen Zusammenschlusses kann auf eine bestimmte Zeitspanne begrenzt sein und Objekte können zur gleichen Zeit verschiedenen Kollaborationen zugehörig sein. Um auf den Begriff Cross-Cutting-Concerns zurückzukommen, lässt sich ein *Kollaborationsmodul* genau als die Menge aller Klassen auffassen, die eine bestimmte Anforderung implementiert.

### 2.1.1 Teams und Rollen

*Teamklassen* bilden in Object Teams die Kollaborationsmodule. Innerhalb von Teams kommt das Konzept der *Rolle* zur Geltung. Unter einer Rolle versteht man eine kontextbezogene Sicht auf ein Objekt. Abhängig von diesem Kontext kann die Sicht oder das Verhalten des Rollenobjektes gegenüber seinem Rolleninhaber erweitert bzw. eingeschränkt sein. Objekte können Rollen dynamisch annehmen und auch wieder ablegen und zur gleichen Zeit können sie mehrere Rollen besitzen, wobei die Objektidentität im Kontext eines Teams stets erhalten bleibt. Die Akteure innerhalb einer Object Teams Kollaboration, also einer Team-Instanz, sind Rollenobjekte und werden in Form von *InnerClasses* beschrieben. Rollenklassen sind in der Regel einer Basisklasse, dem Rolleninhaber, zugehörig.

Listing 2.1: Team- und Rollenklasse

```
1 team class MyTeam {
2         public class MyRole playedBy MyBase { ... }
3 }
```

Im Listing 2.1 wird durch das Schlüsselwort `team` eine Teamklasse deklariert. Teamklassen erben implizit von der Klasse `Team`. Diese Klasse implementiert die Infrastruktur eines Kollaborationsmoduls im Sinne von Object Teams. Ein Team stellt den Kontext der in ihr definierten Rollenklasse und kann daher auch als *Mediator* agieren. Die Rolle deklariert mit Hilfe des `playedBy`-Konstruktes die Zugehörigkeit zu einer Basisklasse.

### 2.1.2 Rolle und Basis

Die Rollen-Basis-Beziehung bewirkt intern bei Instanzierung der Rolle eine Komposition auf Objektebene nach dem Prinzip der objektbasierten Vererbung (*Delegation*). D.h. ein Rollenobjekt hat über einen begehbaren Link Zugriff auf sein Basisobjekt. Dieser Link wird jedoch durch die Team-Infrastruktur verwaltet und gekapselt. Damit wird sichergestellt, dass eine Rolle immer einem Basisobjekt zugehörig und diese Verbindung unveränderlich ist. Nur über explizit definierte Schnittstellen in der Rollenklasse können Aufrufe vom Rollenobjekt an das Basisobjekt erfolgen.

### 2.1.3 Rollenmethoden-Bindungen

Eine Rolle legt über *Methodenbindungen* die Schnittstelle zum Basisobjekt fest. Diese Bindungen sind gerichtet. Eine *Callout*-Bindung meint den Zugriff der Rolle auf das Basisobjekt und eine *Callin*-Bindung die entgegengesetzte Richtung.

#### Callout-Bindungen

*Callout-Bindungen* realisieren eine Schnittstelle zu den Basisklassen und ermöglichen so das Aufrufen von Basismethoden. Dafür wird in einer Rollenklasse eine abstrakte Methode deklariert. Ihre Implementierung findet sie durch ein deklaratives Mapping auf eine existierende Methode in der Basisklasse.

Listing 2.2: Callout-Bindung

```
1 public class MyRole playedBy MyBase {
2         abstract String baseToString();
3         String baseToString -> String toString()
4
5         public String toString() {
6                 return BaseToString() + "_playedBy_" + "
                          MyRole";
7         }
8 }
```

Im Listing 2.2 wird ein einfaches Callout-Beispiel gezeigt. Die `baseToString()`-Methode ist an die Basismethode `toString()` gebunden. Der Aufruf der Rollenmethode `toString()` liefert demnach eine String-Repräsentation des Rollenobjekts, die sich teilweise aus dem Resultat der `toString()`-Methode des Basisobjekts zusammensetzt.

Die Attribute der Basisklasse können über Callouts zusätzlich verfügbar gemacht werden. Dazu dienen die Modifier `get` und `set` in der Bindungs-Deklaration, wie im folgenden Listing zu sehen ist:

Listing 2.3: Zugriff auf Basisattribute

```
1 abstract int  getX();
2 abstract void setX(int x);
3 int  getX()        -> get int x;
4 void setX(int x)   -> set int x;
```

**Callin-Bindungen**

Die *Callin-Bindungen* sind der Teil von Object Teams, in dem die Konzepte der aspektorientierten Programmierung wiederzufinden sind. Eine Callin-Bindung besteht, wie eine Callout-Bindung, aus einem deklarativen Mapping zwischen einer Rollenmethode (Advice) und einer Basismethode (method-execution-point), wobei die Richtung des Aufrufs jetzt entgegengesetzt ist. Über einen zusätzlichen Modifier wird ausgedrückt, ob ein *before-*, *after-* oder *replace*[1]-Callin-Bindung gemeint ist. Eine replace-Bindung ersetzt (überschreibt) eine Basismethode. Die überschreibende Rollenmethode kann jedoch optional mit einem *base-call*[2] die Basismethode aufrufen.

[1] entspricht dem im Kapitel 1 erwähnten *around*-Advice

[2] entspricht der im Kapitel 1 erwähnten proceed-Operation

Listing 2.4: Callin-Bindung

```
1 public class MyRole playedBy MyBase {

3         callin String roleToString() {
4                 return "MyBase" + "_playedBy_" + base.
                                  roleToString();
5          }
6         String roleToSting() <- replace String toString();
7 }
```

Im Listing 2.4 wird ein Beispiel für eine Callin-Bindung gegeben. Die Rollenmethode `roleToString()` ist an die Basismethode `toString()` mit einem replace-Callin gebunden. Dadurch wird die Basismethode überdeckt, so dass beim Aufrufen dieser die `roleToString()`-Methode ausgeführt wird. Durch den base-call (Zeile 4), wird nun nachträglich die Basismethode ausgeführt.

Die Abbildung 2.1 zeigt die Bindungen aus den Listings 2.2 und 2.4 anhand eines UFA-Diagrams. *UFA* (UML for Aspects) ist eine Erweiterung der UML, zur Modellierung von Object Teams Anwendungen.

### 2.1.4 Parameter-Mapping

Um eine flexible Integration von Teamklassen in bestehende Anwendungen zu ermöglichen, ist eine zusätzliche Angabe eines Parameter-Mappings bei den Bindungs-Deklarationen möglich. D.h. man kann genau angeben, welcher Parameter der Basismethode auf welchen Parameter der Rollenmethode abgebildet werden soll.

### 2.1.5 Vererbung von Bindungen

Basisklassen vererben ihre Bindungen an ihre Subklassen (*Aspektvererbung*). Wird also eine Basismethode, die durch einen Callin gebunden ist, in einer Subklasse überschrieben, so bewirkt der Aufruf dieser Methode auch die Ausführung der geerbten Callins.

### 2.1.6 Teamvererbung

Bei der Vererbung von Teams werden die Rollen des Super-Teams implizit weiter vererbt (implizite Vererbung). Rollen können durch Namensgleichheit mit ihrer Super-Rolle verfeinert werden. Mit Hilfe des *tsuper-calls* können, analog zum *super-call*, Methoden in der impliziten Oberklasse aufgerufen werden.

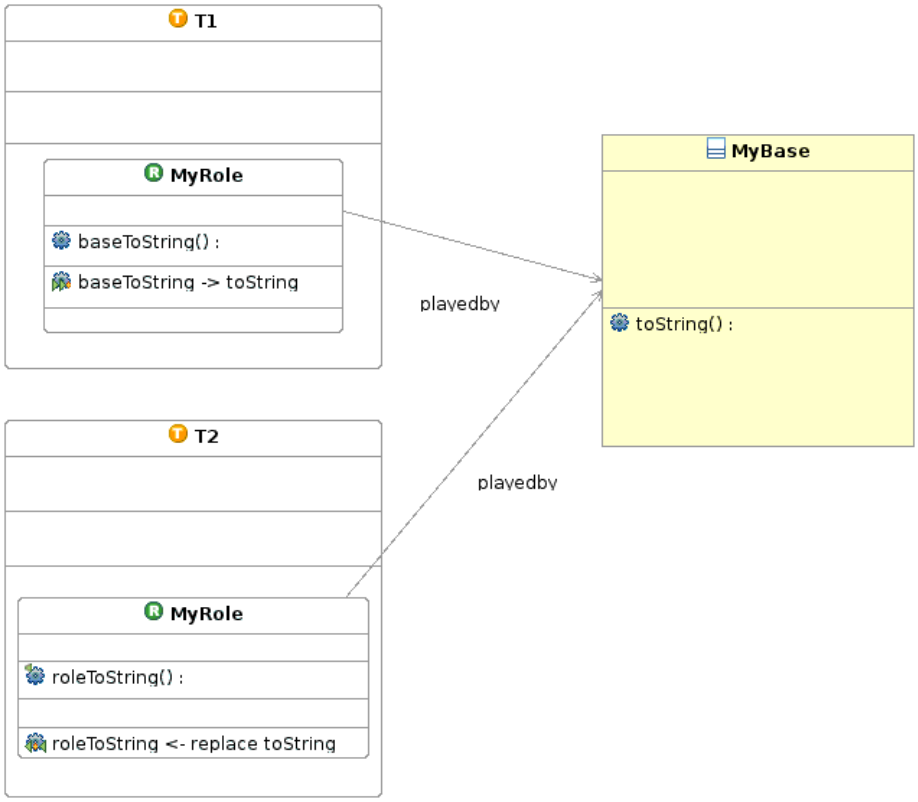


Abbildung 2.1: UFA-Diagramm für die Syntax-Beispiele

### 2.1.7 Teamaktivierung

Team-Instanzen können zur Programmlaufzeit aktiviert und deaktiviert werden. Durch eine Aktivierung werden die Callin-Bindungen der Rollen des Teams wirksam. Eine explizite Aktivierung wird über die Team-Methode `activate()` erzielt. Eine implizite (automatische) Aktivierung ist nötig, wenn eine Team-Methode aufgerufen wird, deren Verhalten von einer Callin-Bindung abhängt, die von einer ihrer Rollen deklariert wird.

### 2.1.8 Lifting und Lowering

Ein Team verwaltet, wie schon erwähnt, die Rollen-Basis-Beziehung. Bei einem Callin, also dem Übergang von einem Basisobjekt zu einem Rollenobjekt, wird zunächst überprüft, ob schon ein korrespondierendes Rollenobjekt existiert. Ist dies nicht der Fall, so wird ein Rollenobjekt erzeugt und zusammen mit der Basisreferenz in einem internen Cache gespeichert. Das Mapping zwischen Basis- und Rol-

lenobjekt wird *Lifting* genannt. Demgegenüber steht das so genannte *Lowering*, der Übergang von einer Rolle zur Basis, das bei einem Callout benötigt wird.

## 2.2 Bytecode-Transformationen zur Ladezeit

Ein Antrieb von Object Teams ist die einfache und flexible *a-posteriori-Integration* der Aspekte in bestehende Programmsysteme. Im Gegensatz zu statischen Webeverfahren, die auf Sourcecode-Ebene operieren, bindet der Weber von Object Teams die Aspekte auf Bytecode-Ebene. Daher ist der Quellcode der Basisapplikation für das Weben nicht notwendig.

Der Aspektweber von Object Teams basiert auf Load-Time-Adaption. Load-Time-Adaption meint das Transformieren eines Programms zur Ladezeit. Dieses Verfahren ist aufgrund der technischen Voraussetzung hauptsächlich in javabasierten AOP-Sprachen zu finden. Grundlage der Transformationen ist das *Java-Classfile-Format*.

### Java-Classfile-Format

Klassen werden von einem Java-Compiler in das maschinenunabhängige Classfile-Format übersetzt. *Classfiles* dienen der *Java Virtual Machine* (JVM) [5] als Klassendefinition und enthalten unter anderem den *Bytecode* der von der JVM interpretiert und ausgeführt wird. Die Struktur des Classfile-Formats ist im Folgenden dargestellt:

Listing 2.5: Classfile-Struktur

```
1  ClassFile {
2        u4 magic;
3        u2 minor_version;
4        u2 major_version;
5        u2 constant_pool_count;
6        cp_info constant_pool[constant_pool_count-1];
7        u2 access_flags;
8        u2 this_class;
9        u2 super_class;
10       u2 interfaces_count;
11       u2 interfaces[interfaces_count];
12       u2 fields_count;
13       field_info fields[fields_count];
14       u2 methods_count;
15       method_info methods[methods_count];
16       u2 attributes_count;
17       attribute_info attributes[attributes_count];
18    }
```

Der umfangreichste Teil des Classfiles ist der *Constant-Pool*. Er dient als Symboltabelle, in der alle konstanten Werte eines Classfiles abgelegt und referenziert werden. Dies sind unter anderem die Namen von Klassen, Methoden oder Feldnamen, die in irgendeiner Weise im Java-Bytecode benutzt werden. Wird z.B. ein neues Objekt erzeugt, wird der Eintrag im Constant-Pool referenziert, an dem der Name des Klassentyps verzeichnet ist. Der Bytecode selbst ist innerhalb einer `method_info`-Struktur in Form eines Code-Attributs gespeichert. Das Classfile-Format kennt eine Anzahl festgelegter Attribute, die für die jeweiligen Klassenelemente benutzt werden. Die Spezifikation der JVM schreibt vor, dass unbekannte Attributarten ignoriert werden sollen. Dies nutzt der Object Teams-Compiler, um die Bindungs-Informationen der Teamklassen mittels zusätzlicher Attribute zu transportieren, die von dem Aspekt-Weber zur Ladezeit ausgelesen werden.

### Der Classloader

Das Laden von Klassen ist ein wesentlicher Bestandteil der Java-Architektur. Für den Ladevorgang ist die Klasse `Classloader` aus dem Paket `java.lang` zuständig. Ihre Aufgabe ist es, zu einem gegebenen Klassennamen den repräsentierenden Bytecode zu lokalisieren und in ein `byte`-Array einzulesen, um daraus eine `Class`-Instanz zu erzeugen.

Die JVM erzeugt standardmäßig ein Exemplar der Klasse `Classloader`, den so genannten *System-Classloader*. Er lokalisiert und liest *.class*-Dateien im Datei-System, wobei die Umgebungsvariable *Classpath* dazu dient, den Suchraum zu beschränken. Anwendungsspezifische Classloader können andere Mechanismen implementieren, um den Bytecode zu finden und auszuwerten (z.B. einen verschlüsselten Bytecode aus einer Netzwerkadresse lesen).

Bei der Erzeugung eines eigenen Classloaders kann man optional die Instanz eines übergeordneten Classloaders angeben, andernfalls ist dies immer der System-`Classloader`. Dadurch wird ein Delegationsmodell realisiert.

Clienten (wie beispielsweise die JVM selbst) rufen die Methode `ClassLoader.loadClass()` auf, um ein Klasse zu laden. Diese Methode ruft als erstes die Methode `ClassLoader.loadClass()` des übergeordneten Classloaders auf. Wird die Klasse dort nicht gefunden, übernimmt der aufgerufene Classloader die Aufgabe. Man kann dieses Delegationsmodell vermeiden, indem man die `loadClass()`-Methode überschreibt und stattdessen direkt die überschriebene Methode `ClassLoader.findClass()` aufruft, um die Klasse zu lokalisieren und den Bytecode in ein `byte`-Array einzulesen. Die `Class`-Instanz wird über die vererbte Methode `ClassLoader.defineClass()` erzeugt.

### 2.2.1 Der Aspektweber von Object Teams

Aufgabe des Aspektwebers von Object Teams ist die Realisierung der Callin-Bindungen. Ausgangspunkt des Webens ist zum einen der vom OT/Java-Compiler generierte Bytecode der Teamklassen und zum anderen der Bytecode der Basisapplikation, bzw. der Basisklassen. Der OT/Java-Compiler ist eine Erweiterung eines Java-Compilers mit den Sprachkonzepten von Object Teams [2]. Der Bytecode der Teamklassen enthält unter anderem die Bindungs-Informationen in Form von Attributen, die für die Transformationen der Basisklassen nötig sind.

#### Load-Time-Adaption mit JMangler

*JMangler* [12] ist ein Framework, um Bytecode der Klassen zur Ladezeit abzufangen und zu transformieren. Die Bytecode-Transformationen mit JMangler basieren auf der *BCEL* (Byte Code Engineering Library) API [4]. Die BCEL API abstrahiert von dem binären Format der Java-Classfiles und erlaubt dadurch eine einfache Manipulation, wie z.B. das Hinzufügen neuer Methoden und Felder.

Bytecode-Transformationen werden innerhalb von *Transformerklassen* definiert. JMangler unterscheidet zwischen *Interface-Transformern* und *Code-Transformern*. Interface-Transformationen beinhalten Operationen, die das Klassenschema verändern, z.B. das Hinzufügen, Entfernen, Umbenennen und die Veränderung der Sichtbarkeit von Methoden und Attributen. Code-Transformationen erlauben nur das Verändern des Bytecodes innerhalb einer Methode.

Beim Laden der vom OT/Java-Compiler generierten Classfiles werden die Bindungs-Informationen mit Hilfe von BCEL aus den Classfile-Attributen ausgelesen und in eine Datenstruktur, den so genannten `CallinBindingManager`, abgelegt. Die dort gespeicherten Informationen dienen als Grundlage der Transformationen an den Basisklassen.

Die Abbildung 2.2 [3] zeigt das Zusammenspiel des OT/Java-Compilers mit dem OT/Weber, der auf die JMangler-Architektur aufbaut. Eine genaue Beschreibung des Transformations-Algorithmus wird in [9] vorgenommen.

Die Webestrategie des Load-Time-Weavers von Object Teams basiert sowohl auf Interface-Transformationen, als auch auf Code-Transformationen, die im nächsten Abschnitt näher beschrieben werden.

[3] Die Abbildung wurde aus der Diplomarbeit [9] entnommen.

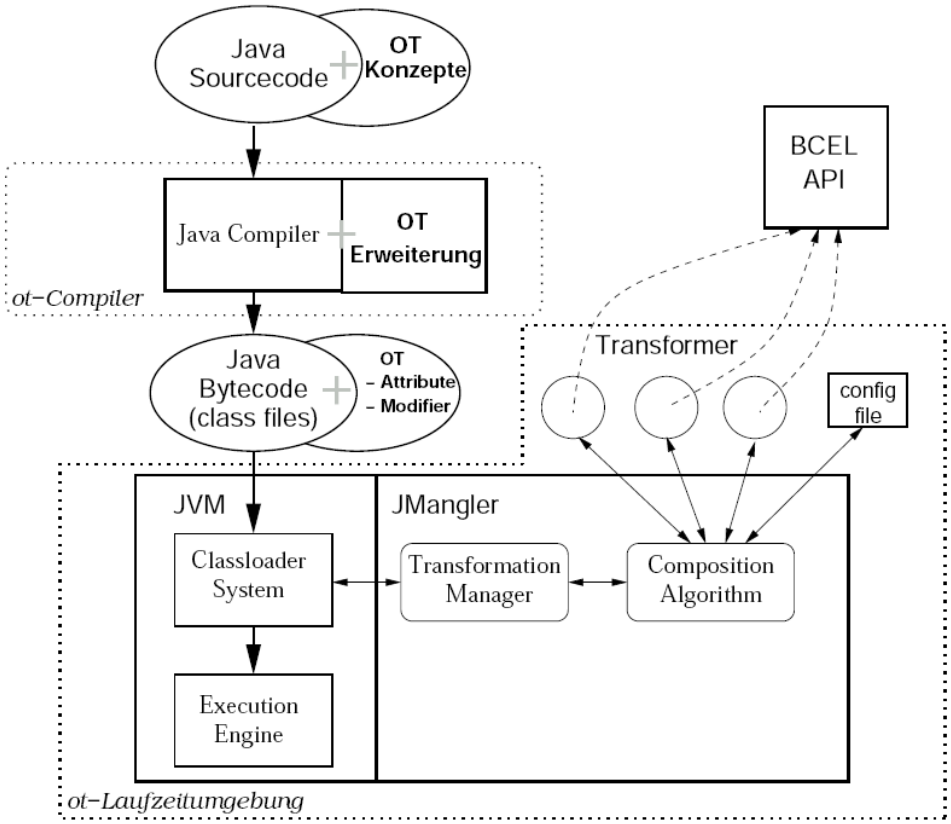


Abbildung 2.2: JMangler-Architektur mit OT/Java Erweiterung

### 2.2.2 Bisherige Webestrategie

Hauptaufgabe des Aspektwebers ist die Transformation von Basisklassen, deren Methoden von mindestens einem Callin gebunden sind. Die Transformation generiert eine Infrastruktur (*Dispatch-Logik*), die im Wesentlichen dem *Observer-Pattern* entspricht. Dabei agiert die Basisklasse als das *Subject* und die Teamklasse als Observer. Dem Muster entsprechend werden Klassenelemente für ein An- und Abmelden (Aktivieren und Deaktivieren siehe 2.1.7) von Teaminstanzen in die Basisklasse hinzugefügt:

Listing 2.6: An- und Abmeldeelemente der Basisklassen

```
1 protected static int _OT$activeTeamIDs[];
2 protected static Team _OT$activeTeams[];
3 public static void _OT$addTeam(Team team, int teamID)
4 public static void _OT$removeTeam(Team team)
```

Beim Anmelden wird die Teaminstanz in eine Registrierungsliste (`_OT$activeTeams[]`) abgelegt. Zusätzlich wird in einer anderen Liste (`_OT$activeTeamIDs[]`) eine eindeutige *Team ID* gespeichert, welche die Teamklasse der Instanz kennzeichnet. Die Klassenelemente sind statisch, da sich Teams bei Basis-Klassen und nicht bei Basis-Objekten anmelden.

Bei der Anmeldung einer neuen Team-Instanz, wird ein neues Array initialisiert, in dem die neue Team-Instanz sich im ersten Index des Arrays befindet. Für die Ausführungsreihenfolge der Callin-Bindungen der einzelnen Team-Instanzen bedeutet dies, dass die before-Callins und replace-Callins eines zuletzt angemeldeten Teams zuerst und die after-Callins zuletzt ausgeführt werden.

Um die Callin-Bindungen der registrierten Team-Instanzen zu realisieren, übernimmt die Dispatch-Logik bei einem Aufruf einer gebundenen Basismethode den Kontrollfluss (*method interception*). Um das zu realisieren, wird die Basismethode umbenannt und als Ersatz (*Stub*) ein *initialer Wrapper* generiert[4]:

```
1 public RType _OT$m$orig(AType1 a1, ..., ATypen an) {
2         /* Umbenannte Original-Methode */
3 }
4 public RType m(AType1 a1, ..., ATypen an) {
5         /* Stub - mit neuer Implementierung */
6 }
```

Aufgabe des *Initial-Wrappers* ist es, eine Arbeitskopie der Registrierungslisten der Teams zu erstellen und damit den so genannten *Chaining-Wrapper* aufzurufen.

### Der Chaining-Wrapper

Ein Chaining-Wrapper wird für jede gebundene Basismethode genau einmal generiert. Er implementiert das Dispatching zu den aktivierten Team-Instanzen. Jede callin-gebundene Rollenmethode besitzt eine öffentliche Callin-Wrappermethode in der umschließenden Teamklasse. Die Callin-Wrapper sind die Team-Schnittstellen für die Chaining-Wrapper in den Basisklassen. Sie übernehmen unter anderem das Lifting des Basisobjekts zum Rollenobjekt. Das Aufrufen eines Callin-Wrappers ruft die Rollenmethode des zum Basisobjekt korrespondieren Rollenobjekts auf. Im Folgenden wird das Aussehen des Chaining-Wrappers formal beschrieben, wobei wir zunächst die Notation einer Callin-Bindung festlegen.

[4] *RType* soll von einem konkreten Rückgabewert abstrahieren und entsprechend *AType* von einem konkreten Parametertypen.

Eine Callin-Bindung können wir als ein 6-Tupel darstellen:

$$(T, R, B, rm, bm, M)$$

$T$ : Teamklasse
$R$ : Rollenklasse
$B$ : Basisklasse
$rm$ : Rollenmethode
$bm$ : Basismethode
$M \in \{before, replace, after\}$ : Bindungsart

Die Komponenten einer konkreten Bindung $b$ sollen wie folgt notiert werden:

$$(T_b, R_b, B_b, rm_b, bm_b, M_b)$$

Durch die Deklarierung einer Callin-Bindung werden direkte und indirekte (vererbte) Bindungen definiert (Aspektvererbung).

Sei $CB_{all}$ die Menge aller existierenden Bindungen und eine Methode $A.m$ mit der Signatur $RType\ m(AType_1, ..., AType_n)$.

Die Menge der Callin-Bindungen $CB_{A,m}$ ist also:

$$CB_{A,m} = \{\ b \in CB_{all} \mid B_b \sqsubseteq A \wedge bm_b = m\ \}\ ^5$$

Wir zerlegen die Bindungen $CB_{A,m}$ in folgende Teilmengen:

$$BEFORE_{A,m} = \{\ b \in CB_{A,m} \mid M_b = before\ \}$$
$$REPLACE_{A,m} = \{\ b \in CB_{A,m} \mid M_b = replace\ \}$$
$$AFTER_{A,m} = \{\ b \in CB_{A,m} \mid M_b = after\ \}$$

Für eine Callin-Bindung b = $\{T_b, B_b, R_b, rm_b, bm_b, M_b\}$ hat der entsprechende Callin-Wrapper des Teams $T_b$ folgende Signatur:

$void\ \_OT\$R_b\$rm_b\$bm_b(B_b,a_1,...,a_n)$ für $M_b \in \{\ before, after\ \}$
$RType\ \_OT\$R_b\$rm_b\$bm_b(B_b,Team[],int[],int,a_1,...,a_n)$ für $M_b \in \{\ replace\ \}$

Der entsprechende Chaining-Wrapper für $A.m$ ist im Listing 2.8 dargestellt und wird im nächsten Abschnitt genauer erklärt.

[5] $A \sqsubseteq B \Leftrightarrow A$ ist Subklasse von $B$

Listing 2.7: Chaining-Wrapper für A.m()

```
1  public RType _OT$m$chain(Team _OT$teams[], int _OT$teamIDs[],
                            int _OT$idx, AType1 a_1, ..., ATypeN a_n) {

3      RType _OT$result = null;
4      if(_OT$idx >= _OT$teams.length) {
5          _OT$result = _OT$m$orig();
6          return _OT$result;
7      }

9       Team _OT$team = _OT$teams[_OT$idx];

11     // before-Callins
12     switch(_OT$teamIDs[_OT$idx]) {
13         // ∀ b ∈ BEFORE_{A,m}
14         case : T_b.idx:
15             ((T_b)_OT$Team).R_b$rm_b$m(this, a_1, ..., a_n)
16         break;

18     }
19     // replace-Callins
20     switch(_OT$teamIDs[_OT$idx]) {
21         // ∀ b ∈ REPLACE_{A,m}
22         case T_b.idx:
23             _OT$result = ((T_b)_OT$Team).R_b$rm_b$m(this, _OT$teams,
                                _OT$teamIDs, _OT$idx+1, a_1, ..., a_n);
24             break;

26         default:
27             _OT$result = _OT$m$chain(_OT$teams, _OT$teamIDs,
                                _OT$idx+1, a_1, ..., a_n);
28             break;
29     }
30     // after-Callins
31     switch(_OT$teamIDs[_OT$idx])
32     {
33         // ∀ b ∈ AFTER_{A,m}
34         case T_b.idx:
35             ((T_b)_OT$Team).R_b$rm_b$m(this, a_1, ..., a_n);
36             break;

38     }

40     return _OT$result;
41 }
```

Da Callin-Bindungen vererbt werden (Aspektvererbung) gilt:

$$CB_{B,m} \subset CB_{A,m} \text{ für alle } A \sqsubset B$$

Damit für eine Submethode von *m* in einer Klasse *B* die nötigen Callins ausgeführt werden, müsste also auch ein Chaining-Wrapper für *B.m* generiert werden. Hat *B.m* jedoch keine weiteren Bindungen, als die von *A.m* geerbten, reicht es aus, den vererbten Chaining-Wrapper von *A.m* aufzurufen. Dafür wird *B.m* wie folgt transformiert:

Listing 2.8: Vererbung der Dispatch-Logik

```
1  public class B extends A {
2
3      ...
4      public RType _OT$m$orig(AType1 a1, ..., ATypen an) {
5          /* Umbenannte Original-Methode von B.m */
6      }
7
8      public RType bm(AType1 a1, ..., ATypen an) {
9          super.m(AType1 a1, ..., ATypen an)
10     }
11     ...
12 }
```

Durch den Super-Aufruf in Zeile 9 in der Methode *B.m* wird der Initial-Wrapper von *A.m* aufgerufen, der nun wiederum den entsprechenden Chaining-Wrapper ausführt. Durch das dynamische Binden wird am Rekursionsende nun jedoch die Methode `B._OT$m$orig()` ausgeführt.

Nur wenn für *B.m* Bindungen existieren, die *A.m* nicht besitzt, wird ein neuer Chaining-Wrapper in *B* generiert.

### Abarbeitung der Teams im Chaining-Wrapper

Der Chaining-Wrapper arbeitet das ihm übergebene Team-Array rekursiv ab (verschränkte Rekursion). Für jedes Team werden die entsprechenden Callin-Wrapper aufgerufen. Dies wird durch die `switch`-Anweisung realisiert, die anhand der Team ID des aktuellen Teams diskriminiert wird (Zeilen 12, 20, 31).

Der Ablauf eines Rekursionsschritts lässt sich wie folgt erklären: Als erstes wird in der Zeile 4 überprüft ob alle Teams abgearbeitet worden sind (Rekursionsende). Wenn dies der Fall ist, liefert der Chaining-Wrapper das Ergebnis der umbenannten Original-Methode zurück. Ist dies nicht der Fall, werden für das aktuelle Team zunächst alle before-Callins ausgeführt und dann die replace-Callins. Existieren keine replace-Bindungen für das Team wird die Rekursion fortgeführt

(Zeile 27). Existieren replace-Bindungen, hängt es von der Callin-Methode der Rollen ab, ob die Rekursion weiter fortgeführt wird, und damit früher angemeldete Teams abgearbeitet werden, oder nicht. Die Fortführung der Rekursion findet nur dann statt, wenn ein replace-Callin einen base-call ausführt. Ist dies der Fall, wird der Chaining-Wrapper für das nächste Team aufgerufen, ansonsten kehrt der Aufruf des replace-Callins wieder unmittelbar zurück und die after-Callins der bisher abgearbeiteten Teams werden ausgeführt.

### Realisierung des base-calls

Der base-call ist, wie schon erwähnt, die Fortführung der rekursiven Abarbeitung der Teams innerhalb eines replace-Callins, was bedeutet, dass der Chaining-Wrapper für die nächste Team-Instanz in der Teamliste aufgerufen wird. Daher besitzen die Callin-Wrapper für replace-Bindungen eine erweiterte Signatur, um die zusätzlichen Argumente des Chaining-Wrappers durchzureichen.

### Schichtweise Abarbeitung der Teams

Das durch den Chaining-Wrapper realisierte Dispatching zu den Team-Schnittstellen, setzt das Konzept von Object Teams um, in dem sich Team-Instanzen *schichtweise* auf die gebundenen Basismethoden legen. Abbildung 2.3 veranschaulicht dieses Prinzip anhand von drei Teams. Es zeigt die Reihenfolge, in der die jeweiligen Callin-Bindungen der Rollen ausgeführt werden. Dabei ist auch immer die Aktivierungsreihenfolge der Team-Instanzen entscheidend. In diesem Fall ist $t1$ die zuletzt und $t3$ die zuerst angemeldete Instanz.

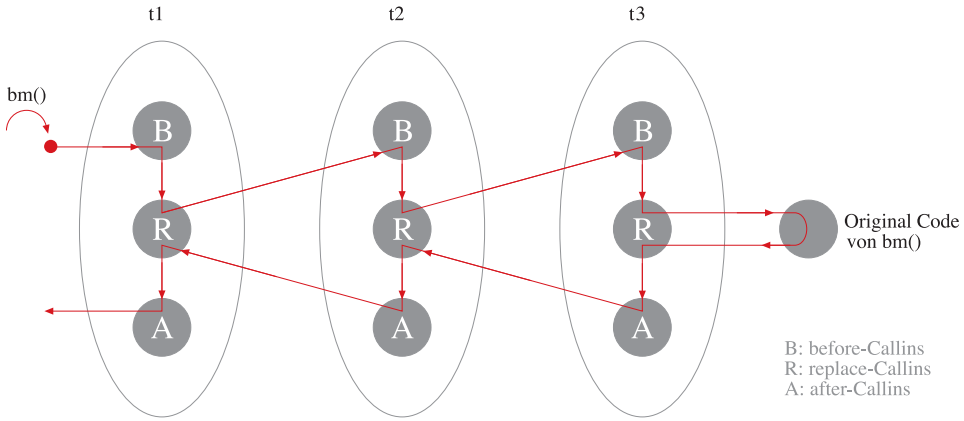


Abbildung 2.3: Schichten von Team-Instanzen

Abbildung 2.4 zeigt einen Dispatch anhand eines Sequenzdiagramms. Die Methode *bm* ist durch die Rollenmethoden $R1.rmb()$ und $R1.rma()$ durch einen before-

Callin bzw. einen after-Callin und durch die Rollenmethode *R2.rmr*() durch einen replace-Callin gebunden. Die Reihenfolge, in der der Chaining-Wrapper die jeweiligen Callin-Wrapper der Teams aufruft, entspricht der schichtweisen Abarbeitung der Teams.

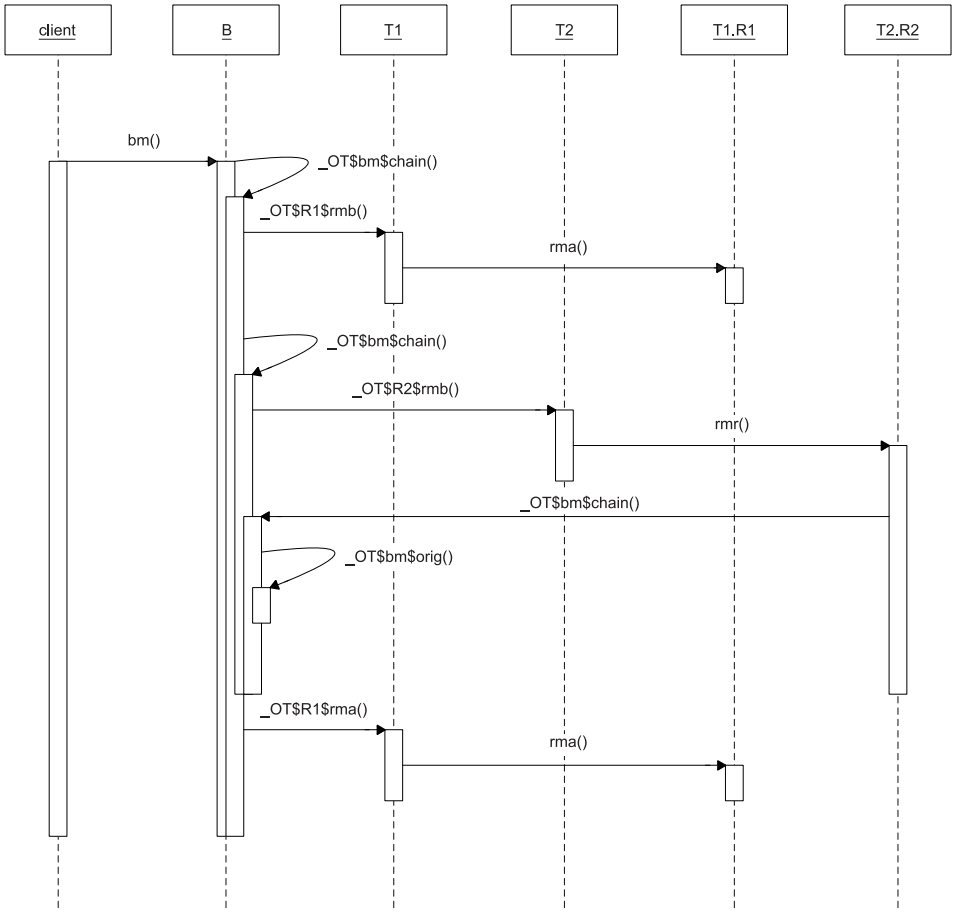


Abbildung 2.4: Ablauf eines Callin-Dispatch

### 2.2.3 Probleme und Nachteile der Webestrategie

Das Load-Time-Weaving des Webers ist in einem gewissen Sinne statisch, da man den Bytecode der Klassen nur während der Ladezeit transformieren kann. Ist der Ladevorgang einer Klasse abgeschlossen, ist keine weitere Veränderung mehr möglich.

Diese Einschränkung führt zu folgendem Problem: Die Bindungs-Informationen werden während der Ladezeit aus den Classfile-Attributen der Teamklassen gelesen. Eine Basisklasse kann also erst dann gewoben werden, wenn die für sie relevanten Bindungs-Informationen bekannt sind. Daher müssen die Teamklassen vor den Basis-Klassen geladen werden.

Der System-Classloader implementiert das sogenannte *lazy-loading*. Das bedeutet, dass Klassen zum spätest möglichen Zeitpunkt geladen werden. Diese Ladestrategie des Classloaders kann nicht garantieren, dass die Teamklassen zuerst geladen werden. Daher wird eine Ladereihenfolge wie folgt erzwungen: Der OT/Compiler generiert in jede OT/Java-Klasse, die eine oder mehrere Teamklassen referenziert, ein *ReferencedTeams*-Attribut. Dieses Attribut enthält die Namen dieser Teamklassen. Werden diese Attribute zur Ladezeit vom Weber ausgelesen, wird unmittelbar das Laden dieser Klasse über die Methode `Classloader.loadClass()` erzwungen.

Diese Vorgehensweise stößt jedoch an ihre Grenzen, wenn die Abhängigkeiten zwischen Basisklassen und Teamklassen transitiv sind. Nehmen wir an, eine Klasse *A* referenziert eine Klasse *B*, die wiederum ein Team *T* referenziert, dass *A* bindet. Da die Klasse *T* erst geladen wird, wenn das ReferencedTeams-Attribut aus *B* ausgelesen wurde, kann die schon geladene Klasse *A* nicht mehr gebunden werden.

Dieses Problem würde nicht existieren, wenn wir Klassen unabhängig davon weben könnten, ob sie schon geladen worden sind oder nicht. Dies führt uns zu dem Begriff des *dynamischen Webens*, welcher im nächsten Kapitel behandelt wird.

# Kapitel 3

# Weben von Aspekten zur Programmlaufzeit

Bei statischen Webeverfahren werden die Aspekt- und Basismodule zu einem bestimmten Zeitpunkt in ein ausführbares Artefakt verbunden. Es wird also vorher festgelegt, welche Aspektanforderungen in einer Basisanwendung wirksam sein sollen und welche nicht.

*Dynamisches AOP* (DAOP) erlaubt das Weben von Aspekten zu beliebigen Zeitpunkten. Aus der zeitlichen Unabhängigkeit des Webevorganges ergeben sich, wie wir sehen werden, Vorteile gegenüber dem statischen Webeverfahren und sie erschließt darüber hinaus neue Anwendungsfelder der aspektorientierten Programmierung.

## 3.1 Vorteile des dynamischen Webens

Wenn Anforderungen an ein laufendes System gestellt werden, die vorher noch nicht definiert wurden oder sich während des Betriebs kurzfristig oder längerfristig ändern, sind wir in der Regel dazu gezwungen, das System mit einer neuen Version auszutauschen. Sind diese wechselnden Anforderungen von einer bestimmten Häufigkeit und unterliegt das System zugleich einer hohen Verfügbarkeit, so sind statische Webeverfahren für solche Anwendungen unzureichend. Im Folgenden werden einige Beispiele aufgelistet, in denen DAOP eine Anwendung finden könnte.

- **Logging**: In der Implementierungsphase ist es typisch, dass man wiederholt an den verschiedensten Stellen des Programmes Logging-Ausgaben einfügt, um bestimmte Programmabschnitte zu testen. Bei sehr großen Systemen kann der Deployment-Vorgang, wie das Recompilieren und Reinstallieren der Komponenten zu einer zeitkonsumierenden Angelegenheit werden. Mit DAOP könnte man ein Werkzeug realisieren, das während des Programmablaufs Logging-Ausgaben an den gewünschten Stellen einfügt und später wieder entfernt.

- **Debugging**: Bestimmte Programmfehler treten erst nach längerer Laufzeit ein und lassen sich zu dem typischer Weise nicht ohne weiteres rekonstruieren. Will man diese Fehler finden, ist man beim herkömmlichen Debugging gezwungen, die Applikation zu beenden, um diese in einem *Debug-Modus* zu untersuchen. Dies hat den offensichtlichen Nachteil, dass durch den Neustart der fehlerhafte Zustand verloren geht. Mit einem dynamischen Debugging dagegen, könnte man genau dann die Anwendung untersuchen, wenn sie das Fehlverhalten aufweist.

- **Rapid-Prototyping**: *Rapid-Prototyping* ist eine Methode, mit möglichst wenig Aufwand einen Prototyp zu entwickeln. Gezielte Änderungen mit DAOP an einem laufenden System können helfen, einen Prototypen zu realisieren, der neue Anforderungen implementiert.

- **Profiling**: *Profiling* ist das Messen der Performance eines Systems. Beim Profiling spricht man von *Instrumentierungen*, wenn ein Analysecode in eine Anwendung eingefügt wird, um Laufzeitverhalten oder Speichernutzung auszuwerten. Ein typisches Beispiel ist die Messung der Ausführungszeit einer Methode. Dabei wird am Anfang und am Ende der Methode ein entsprechender Messcode generiert. Ein dynamischer Weber könnte diesen Code zur Laufzeit der Applikation hinzufügen und wieder entfernen.

- **Security**: Sicherheitsanforderungen an ein System führen im Allgemeinen dazu, dass dieser Aspekt über das ganze System verteilt implementiert ist. Ein adaptiver Aspekt könnte verschiedene Sicherheitsstufen definieren und so eine flexible Anwendung realisierbar machen, die ihre *Security-Policies* dynamisch verändern könnte.

Ein Problem, welches durch eine dynamische Webestrategie bezüglich der bisherigen Strategie von Object Teams gelöst werden könnte, ist die in Kapitel 2 beschriebene Abhängigkeit der Reihenfolge, in der die Klassen geladen werden müssen. Die Realisierung eines dynamischen Webers bedeutet, dass diese Abhängigkeit nicht mehr bestünde, da Basisklassen zu beliebigen Zeitpunkten transformierbar wären.

Eine sehr interessante Anwendung, die mit Hilfe von DAOP entwickelt werden könnte, ist ein *Aspekt-Manager* für Object Teams. Diese Anwendung, in Form eines Java-Agenten[1], könnte zusammen mit der Basisanwendung gestartet werden, um zur Laufzeit einem Benutzer zu erlauben interaktiv, Aspekte hinzuzufügen oder zu entfernen.

## 3.2 AOP im Java-Umfeld

Dynamische AOP Implementierungen sind größtenteils im Java-Umfeld zu finden. Dies liegt nicht nur daran, dass Java in den letzten Jahren eine der wichtigen Mainstream-Programmiersprachen geworden ist, sondern begründet sich im Wesentlichen auch darauf, dass die Laufzeitumgebung der Sprache, also die Java Virtuell Machine, Schnittstellen anbietet, um zur Laufzeit in eine Anwendung einzugreifen.

### 3.2.1 Debugger-basiertes Weben

Die ersten Ansätze, dynamisches AOP im Java-Umfeld zu realisieren, benutzen das JVMDI (*Java Virtuell Machine Debug Interface*). Das JVMDI ist eine Programmierschnittstelle zur virtuellen Maschine für die Entwicklung von *Debuggern* oder ähnlichen Programmierwerkzeugen. Client-Anwendungen des JVMDI können während der Laufzeit einer Java-Anwendung *Breakpoints* innerhalb von Methoden setzen und sich für spezielle Events, wie z.B. den Zugriff auf ein bestimmtes Feld einer Klasse registrieren lassen. AOP Implementierungen, wie beispielsweise *Prose* [13] und *Wool* [3] nutzen diese Technik, um Joinpoints zu generieren. Beim Erreichen eines Breakpoints während der Programmausführung wird die Basis-Anwendung von der JVMDI angehalten und die registrierten *Event-Handler* benachrichtigt, die dann den für den Joinpoint entsprechenden Advice ausführen. Charakteristisch für dieses Vorgehen ist, dass man den Code der Basisanwendung durch das Generieren von Joinpoints nicht verändern muss, was zunächst vorteilhaft erscheint. Jedoch ist die eventbasierte Benachrichtigung sehr inperformant. Da die JVMDI-Schnittstelle dafür gedacht ist, externe Debugger-Werkzeuge zu entwickeln, die in einem eigenen Thread laufen, bedingt jedes Breakpoint-Event einen kostspieligen Kontextwechsel. Hinzu kommt, dass zur Benutzung der JVMDI die JVM in einem *Debug-Modus* gestartet werden muss, der die Performance der Basisanwendung erheblich beeinträchtigt.

[1] Ein Java-Agent ist eine Klasse, die eine *premain*-Methode implementiert, die vor der *main*-Methode ausgeführt wird. Diese wird bei Programmstart als Parameter in Form eines *jar-Files* übergeben.

### 3.2.2 Runtime Weaving mit Hotswap

Mit der Java 1.4 SDK wurde die JVMDI um ein sogenanntes *Hotswapping* erweitert. Hotswap meint den Austausch von Klassen während der Laufzeit, ohne diese neu laden zu müssen. Über die Hotswap-Schnittstelle kann eine Klasse redefiniert werden, in dem man für sie eine neue Bytecode-Repräsentation zur Verfügung stellt.

Die oben genannten debugger-basierten AOP Implementierungen nutzen die Hotswap-Technik, um den Overhead der eventbasierten Benachrichtigung zu vermeiden. Anstelle der Breakpoints werden Methoden-Aufrufe für den Advice-Code eingewoben. Das Setzen eines Breakpoints ist nur einmalig notwendig, um eine Bytecode-Transformation an dem betreffenden Joinpoint durchzuführen. Ist dies getan, kann der Breakpoint gelöscht werden.

### 3.2.3 Java Programming Language Instrumentation Service

Mit der Java 1.5 SDK wird die Hotswap-Funktionalität in die Java-API integriert. *Java Programming Language Instrumentation Service* (JPLIS) wird als ein Framework eingeführt, um Bytecode-Instrumentalisierungen für Profiling und Monitoring-Tools zur Verfügung zu stellen. Die Benutzung von JPLIS erfordert, im Gegensatz zu dem in der JVMDI vorhandenen Hotswapping, keinen Debug-Modus mehr.

Das Interface `Instrumentation` zum Redefinieren von Klassen befindet sich im Paket `java.lang.instrument`. Um dies zu benutzen, wird ein sogenannter *Java-Agent* in Form eines jar-Files beim Starten der JVM angegeben. Ein Java-Agent ist eine Klasse, die eine `premain()`-Methode implementiert. Diese Methode wird vor der `main()`-Methode ausgeführt und wird vom System mit einer Instanz des `java.lang.Instrumentation`-Interfaces aufgerufen (siehe Listing 3.1 Zeile 105).

Listing 3.1: Beispiel eines Java-Agenten

```
101 public class MyAgent {

103     private static Instrumentation instInstance;

105     public static void premain(String options,
                    Instrumentation inst) {

107         instInstance = inst;
108         MyTransformer transformer = new MyTransformer();
109         instInstance.addTransformer(transformer);

111     }

113     public static Instrumentation getInstrumentation() {
114         return instInstance;
115     }

117 }
```

Die Methode des `Instrumentation`-Objekts zum Redefinieren ist:

```
void redefineClasses(ClassDefinition[] definitions)
```

Das Argument dieser Methde (`ClassDefinition[] definitions`) ist ein Array von Klassen, die zu redefinieren sind. Dies zeigt sich, wenn wir uns den Konstruktor von `ClassDefintionen` betrachten:

```
ClassDefinition(Class<?> theClass, byte[] theClassFile)
```

Ein Objekt dieser Klasse wird mit der aktuellen `Class`-Instanz der zu redefinierenden Klasse und dem neuen Byte-Code initialisiert. Die erforderlichen Schritte, um eine Klasse zur Laufzeit zu redefinieren, sind also wie folgt:

1. Den neuen Bytecode einer Klasse erstellen
2. Das aktuelle `Class`-Objekt der zu redefinieren Klasse finden
3. Eine Instanz der Klasse `ClassDefinition` mit der `Class`-Instanz und dem Byte-Code erzeugen
4. Ein Array von `ClassDefinition`-Instanzen erzeugen, auch wenn es nur Objekt ist
5. Das Instrumentation-Objekt besorgen und darauf die `redefineClasses()`-Methode mit dem `ClassDefiniton`-Array als Parameter Aufrufen

Wir werden im Kapitel 5 sehen, dass die benötigten Daten für Schritt 1 und 2 nicht ohne weiteres verfügbar sind, um einen Laufzeit-Weber zu realisieren.

Neben der Redefinition von Klassen zur Laufzeit ist es zusätzlich möglich, den initialen Bytecode zur Ladezeit zu transformieren. Dafür wird eine Instanz des Interfaces `ClassFileTransformer`, das ebenfalls im Package `java.lang.Instrumentation` zu finden ist, beim Intsrumentation-Objekt registriert. Dies geschieht über die Methode `Instrumentation.addTransformer()` (siehe Listing 3.1 Zeile 108).

In der `ClassFileTransformer`-Klasse muss die Methode `transform()` implementiert werden:

Listing 3.2: Signatur der transform-Methode

```
1 transform(ClassLoader loader,
2                   String className,
3                   Class<?> classBeingRedefined,
4                   ProtectionDomain protectionDomain,
5                   byte[] classfileBuffer)
```

Ist eine Instanz des `ClassFileTransformer` bei dem Instrumentation-Objekt registriert, wird beim Aufrufen der Methode `ClassLoader.defineClass()` automatisch die `transform()`-Methode aufgerufen, in der der Zugriff und damit die Veränderung des Bytecodes möglich ist. Damit bietet JPLIS also auch die Möglichkeit, Load-Time-Adaption zu realisieren.

Zusätzlich wird die `transform()`-Methode beim Aufruf von `redefineClasses()` ausgeführt, so dass Redefinitionen auch innerhalb eines `ClassFileTransformer` möglich sind.

### Einschränkungen des Instrumentation Interfaces

Das Instrumentation Interface beschränkt die Redefinitionen zur Laufzeit[2], gibt jedoch gleichzeitig an, dass diese Beschränkungen in zukünftigen Versionen aufgehoben werden könnten.

Betrachten wir die Tabelle 3.1, so stellen wir fest, dass die Redefinitionen das *Schema* der Klasse nicht verändern dürfen. Die Redefinition einer Klasse darf weder Methoden bzw. Felder hinzufügen, entfernen oder ändern, noch die Methodensignaturen, sowie die Vererbungsbeziehungen ändern.

Durch die Redefinitionsbeschränkungen von JPLIS sind also alle die Webestrategien für ein Laufzeitweben mit JPLIS nicht mehr geeignet, die neue Methoden und Felder in die Basisklassen generieren, um die Dispatch-Logik zu realisieren.

[2] Die Beschränkungen gelten nicht für Transformationen zur Ladezeit.

Tabelle 3.1: Redefintionsbeschränkungen von JPLIS

| *Erlaubte Operationen* |
|---|
| Die **Redefinition** des **Bytecodes** einer beliebigen **Methode** |
| Die **Redefinition** eines beliebigen **Attributs** |
| Die **Redefinition** des **Constant-Pools** |
| *Nicht erlaubte Operationen* |
| Das **Hinzufügen**, **Entfernen** oder **Ändern** von **Feldern** |
| Das **Hinzufügen**, **Entfernen** oder **Ändern** von **Methoden** |
| Die **Veränderung** der **Methodensignaturen** |
| Die **Veränderung** der **Vererbungsbeziehungen** |

Dies betrifft auch, wie wir in Kapitel 2 gesehen haben, die Strategie des OT/Java-Webers.

Dynamische AOP Frameworks, die JPLIS basierte Webestrategien implementieren, wie beispielsweise *AspektWerkz* [14], umgehen dieses Problem mit einem sogenannten *2-Phasen-Weben*.

### 3.2.4 2-Phasen-Weben

Um die Redefinitionsbeschränkungen zu umgehen, wird der Webevorgang in zwei Phasen unterteilt. Die erste Phase findet zur Ladezeit statt und dient der *Präparierung* aller oder einer festgelegten Menge von Klassen. In dieser Phase werden potenziell alle schemaverändernden Transformationen getätigt, die eine bestimmte Webestrategie hervorrufen würden. Es werden also Klassenelemente und insbesondere leere Methoden generiert, die dann zur Laufzeit entsprechend schemaerhaltend redefiniert werden können.

Für die Webestrategie von Object Teams-Weber müsste also für jede Methode einer Klasse, die geladen wird, eine leere Wrapper-Hülse des Chaining-Wrappers, sowie eine leere *orig()*-Methode generiert werden (siehe Listing 3.3)

Listing 3.3: Präparieren von Wrapper-Methoden

```
1  public class B {

3      RType _OT$bm$orig(AType_1 a_1, ..., AType_n a_n) {
4          // leere Methode
5      }

7      RType _OT$bm$chain(Team _OT$teams[], int _OT$teamIDs[],
                          int _OT$idx, AType_1 a_1, ..., AType_n a_n) {
8          // leere Methode
9      }
10     ...

12 }
```

Diese Vorgehensweise hat offensichtlich zur Folge, dass man im Allgemeinen für die meisten Klassen *Hooks* präpariert, die während der Laufzeit nicht benötigt werden und man somit einen entsprechenden Overhead zur Ladezeit produziert.

Auf der anderen Seite müssen wir uns jedoch Fragen, in wie weit eine ”gute” Webestrategie realisiert werden kann, die auf eine Präparierungs-Phase verzichtet. Damit beschäftigt sich unter anderem das nächste Kapitel und es beschreibt die grundlegenden Probleme eines schemaerhaltenden Webens. Zudem werden wir sehen, dass nicht nur die Redefinitionsbeschränkungen ein Problem des Laufzeit-Webens darstellen, sondern auch die Anforderungen bewältigt werden müssen, wenn wir nicht mehr von einer festgelegten Menge von Aspekten ausgehen können.

# Kapitel 4

# Schemaerhaltende Webestrategie zur Laufzeit

Im vorangegangenen Kapitel wurde beschrieben, welche Vorteile dynamisches Aspektweben gegenüber dem statischen Weben hat. Insbesondere für Object Teams würde das Problem der Ladereihenfolge zwischen Basis- und Teamklasse gelöst werden. Ziel ist es daher, eine schemaerhaltende und dynamische Webestrategie basierend auf JPLIS für *OT/Java* zu definieren, die das gleiche leisten soll, wie die im Kapitel 2 beschriebene Webestrategie. Die entsprechenden Anforderungen werden im nächsten Abschnitt aufgelistet.

Gegenüber der bisherigen Strategie sind zwei wesentliche Herausforderungen, die bewältigt werden müssen:

- Redefinitionsbeschränkungen an den Basisklassen zur Laufzeit
- Dynamisches Weben von Callin-Bindungen zu beliebigen Zeitpunkten

Die Redefinitionsbeschränkungen werden uns durch JPLIS auferlegt und bilden die Rahmenbedingungen für neue Transformationen zur Laufzeit.

Eine dynamische Webestrategie muss im Gegensatz zur alten Strategie eine sich verändernde Menge von Callin-Bindungen annehmen. Dies hat insbesondere Auswirkungen auf die Transformationen an einer bestimmten Basisklasse, da für sie jederzeit neue Bindungen hinzukommen können. Dies setzt ein inkrementelles (wiederholtes) Weben der Basisklassen voraus. Um dies zu realisieren, müssen zusätzliche Laufzeitinformationen verwaltet werden, auf die der Weber effizient zugreifen soll.

## 4.1 Anforderungen an die dynamische Webestrategie

Zunächst sollen die grundlegenden Anforderungen an die neue Webestrategie definiert werden. Dabei wird kurz erläutert, in wie fern sie für eine schemaerhaltende und dynamische Webestrategie Probleme aufwerfen.

- **Anforderung 1: Ausführung eines Callin-Dispatches**
  Diese Anforderung unterteilen wir in zwei zu lösende Aufgaben:
  - **1a:** Zur Ausführung eines Callins muss auf der einen Seite ein Dispatch beim Aufrufen der Basismethode initiiert werden und auf der anderen Seite die Originalmethode aufrufbar sein. D.h. wir müssen den Konflikt zwischen der Dispatch-Anforderung der Basismethode und die Ausführung der selbigen mit unverändertem Verhalten auflösen (im Folgenden: *Basis-Dispatch-Konflikt*).
  - **1b:** Ein Sonderfall der Aspektbindung tritt auf, wenn eine geerbte und nicht überschriebene Methode *B.bm* gebunden wird. Hier stellt sich nämlich zusätzlich die Frage, wie eine Dispatchanforderung realisiert werden kann, wenn die Methode nicht Bestandteil der eigentlichen Basisklasse ist. Ein Überschreiben von *bm* zum Abfangen des Methodenaufrufes wäre trotz Vererbung der Methode *bm* im Sinne von JPLIS eine schemaverändernde Operation.

- **Anforderung 2: Abarbeitungsreihenfolge der Teams in Schichten**
  Die Abarbeitung der Teams in Schichten (vgl. Kapitel 2) wird durch die verschränkte Rekursion des Chaining-Wrappers realisiert. Da wir zur Laufzeit diese Wrapper nicht generieren können, müssen wir einen alternativen Mechanismus finden, der ohne eine Schemaveränderung auskommt.

- **Anforderung 3: Teamregistrierung**
  Das Generieren von Anmeldeschnittstellen in die Basisklassen ist eine schemaverändernde Transformation, die wir ebenfalls zur Laufzeit nicht umsetzen können. Des weiteren werden wir sehen, dass die bisherige Realisierung der Teamregistrierung im Konflikt mit der Forderung nach einem dynamischen Weben steht (darauf wird in Abschnitt 4.4 eingegangen).

- **Anforderung 4: Decapsulation**
  OT/Java erlaubt Bindungen an private Methoden und Attribute der Basisklassen. *Decapsulation* meint das dafür nötige Durchbrechen der Kapselung. Um dies zu realisieren, erweitert die alte Webestrategie innerhalb einer Transformation die Sichtbarkeit der betroffenen Elemente. Eine Veränderung der Sichtbarkeit zur Laufzeit ist im Sinne von JPLIS eine schemaverändernde Redefinition und daher nicht möglich.

Anforderungen die übergreifend sind, also für 1 bis 4 gelten:

- **Performance des Webens**
  Ein inkrementelles Weben zur Laufzeit hat eine Verzögerung der Basisanwendung aufgrund der erforderlichen Bytecode-Transformationen zur Folge. Daher sollte dieser Prozess möglichst performant realisiert sein, um auch zeitkritischen Anwendungen gerecht zu werden.

- **Performance der Dispatchausführung**
  Des weiteren ist die Performance des Dispatchings zu berücksichtigen. Gemeint ist die Zeit, die benötigt wird, um den Aufruf einer gebundenen Methode an eine verknüpfte Rollenmethode zu leiten.

Es soll nun eine Webestrategie gefunden werden, die den gestellten Anforderungen gerecht wird. Dafür werden zunächst zwei Ansätze untersucht, deren Ziel es ist, die alte schemaverändernde Webestrategie auf eine schemaerhaltende abzubilden, was eine Wiederverwendung der alten Transformer ermöglichen würde.

## 4.2 Erster Ansatz zum schemaerhaltenden Weben

Betrachten wir die bisherige Webestrategie, so erkennen wir ein grundlegendes Prinzip der Dispatch-Realisierung: das Wrappen von Methoden, in Form der Umbenennung einer gebundenen Basismethode und die Generierung einer Wrapper-Methode als Stub für die Initiierung des Dispatchings. Da wir die schemaverändernden Transformationen der bisherigen Webestrategie zur Laufzeit nicht ausführen dürfen, wollen wir im ersten Ansatz untersuchen, ob dieses Prinzip des Wrappings auf Klassen- bzw. auf Objektebene übertragbar ist.

Der naive Ansatz soll also darin bestehen, eine Klasse anstatt einer Methode zu wrappen. Ist dies möglich, könnten wir auf einer Kopie[1] des Bytecodes einer zu webenden Basisklasse die alten Webeoperationen ausführen und damit eine neue Klasse definieren (*on-the-fly*). Die Methoden der Basisklasse dagegen sollen nur noch als Schnittstelle zu der neuen Klasse fungieren. Die dafür nötigen Transformationen wären schemaerhaltend. Auf diese Weise würden wir die Redefinitionsbeschränkungen von JPLIS umgehen, da das Generieren von Klassen zur Laufzeit keine Redefinition darstellt.

[1] Woher wir den Bytecode einer bestimmten Klasse bekommen können, wird in Kapitel 5 beschrieben.

Diese Vorgehensweise ähnelt dem Design-Muster *Adapter mit Delegation*, wobei die Basisklasse zu einem Adapter redefiniert wird und die neue Klasse die zu adaptierende ist. Aufrufe eines Klienten werden von dem Adapter an die zu adaptierende Klasse delegiert. Das Ziel eines *Classwrappings* muss es sein, dass der Aufruf auf dem Adapter-Objekt für einen Klienten ein transparentes Verhalten hervorruft.

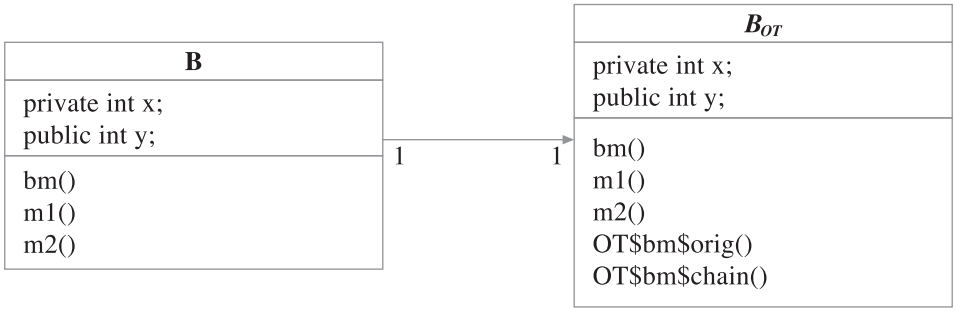


Abbildung 4.1: Classwrapping

### 4.2.1 Classwrapping

Sei dazu nun *B* eine Klasse, die wie folgt zu einer Wrapper-Klasse transformiert wird: Wir generieren eine neue Klasse $B_{OT}$, die im Wesentlichen eine Kopie von *B* ist, jedoch mit geringfügigen, noch zu bestimmenden Modifikationen (siehe Abbildung 4.1). Zusätzlich steht es uns frei, weitere Methoden und Attribute zu $B_{OT}$ hinzuzufügen. Alle Methoden von *B* werden schemaerhaltend redefiniert. Die neue Implementierung dieser Methoden erzeugt zunächst nur eine Instanz $b_{OT}$ (*Surrogate-Objekt*) von $B_{OT}$ und delegiert den Aufruf an ihre Kopie in $B_{OT}$, deren Resultat sie zurückliefert (Listing 4.3 und 4.2).

Listing 4.1: Klasse B mit redefinierter Methode B.bm

```
1  public class B {
2
3      ...
4      private int x = 1;
5      public  int y = 2;
6
7      public int bm(int i) {
8          B_OT b_OT = new B_OT();
9          return b_OT.bm(i)
10     }
11 }
```

Listing 4.2: Generierte Klasse $B_{OT}$ mit Methode $B_{OT}$.bm

```
1  public class  B_OT {
2
3      private int x;
4      public  int y;
5
6      public int bm(int i) {  // kopierte Methode von B.bm()
7          return i + this.y + this.x;
8      }
9      ...
10 }
```

**Objektzustand und Objektidentität**

Da wir davon ausgehen müssen, dass schon ein Objekt *b* der Klasse *B* existiert, bevor wir diese redefinieren, wird $b_{OT}$, da neu initialisiert, einen anderen Zustand als *b* besitzen und somit $b_{OT}$.*bm* ein anderes Ergebnis oder allgemeiner ein anderes Verhalten zeigen, als es Objekt *b* getan hätte. Um dieses Problem vorerst zu lösen, wird $B_{OT}$ mit einem speziellen Konstruktor ausgestattet (Listing 4.4 ab Zeile 7), der es erlaubt, den Zustand des *Sichtbarkeitsraumes* (alle öffentlichen und privaten Felder[2]) einer Instanz von $B_{OT}$ für ein Objekt *b* zu initialisieren (*shallow copy*). Der Aufruf des Konstruktors befindet sich im Listing 4.3 in Zeile 7.

Des Weiteren ist zu berücksichtigen, dass auch die Identität eines Objekts das Gesamtverhalten interagierender Objekte beeinflusst. Daher ist bei der Implementierung von $B_{OT}$ zu beachten, dass jede ausgehende Nachricht eines $B_{OT}$ Objekts, die ihre Identität (`this`-Operator) enthält, so modifiziert wird, dass stattdessen die Identität des entsprechenden *b*-Objekts geliefert wird.

Listing 4.3: Redefinierte Methode B.bm mit Initialisierung des Surrogate-Objekts

```
1  public class B {
2
3      private int x = 1;
4      public  int y = 2;
5
6      public int bm(int i) {
7          B_OT b_OT = new B_OT(this,x,y);
8          return b_OT.bm(i)
9      }
10     ...
11 }
```

[2] Wir gehen zunächst davon aus, dass *B* keine Vererbungsbeziehung besitzt, so dass wir an dieser Stelle *protected*-deklarierte Felder vernachlässigen können.

Listing 4.4: Generierte Klasse $B_{OT}$ mit neuem Konstruktor

```
1  public class B_OT {

3      private int x;
4      public  int y;
5      private B base;

7      public B_OT (B base,int x,int y) {
8          this.base = base;
9          this.x = x;
10         this.y = y;
11     }

13     public int bm(int i) {
14         return i + this.y + this.x
15     }
16     ...
17 }
```

**Persistenz des Surrogate-Objekts**

Bisher erzeugen wir jedes mal beim Aufrufen einer Methode von *B* ein neues Surrogate-Objekt. Wird innerhalb einer Aufrufsequenz das gleiche Objekt von Typ *B* zweimal aufgerufen, werden zwei verschiedene Surrogate-Objekte benutzt, die nun nicht mehr zwingend den gleichen Zustand besitzen müssen. Um die Persistenz des Surrogate-Objekts zu wahren, konstruieren wir uns ein Mapping, das zu einem gegebenen Objekt der Klasse *B* das zugehörige Objekt der Klasse $B_{OT}$ liefert. Eine dafür benötigte Datenstruktur (z.B. eine `HashMap`) stellen wir für *B* in $B_{OT}$ in Form eines Klassenattributs zur Verfügung. Nun wird vor der Delegation geprüft, ob schon ein korrespondierendes $B_{OT}$-Objekt für *b* existiert. Ist dies nicht der Fall, wird zuvor eine neue Instanz erzeugt und gespeichert (siehe Listing 4.5 ab Zeile 8).

Ein weiteres Problem ergibt sich, wenn der Wert eines öffentlichen Feldes von *B* von einem beliebigen Objekt verändert wird. Da bisher ein $B_{OT}$ Objekt auf den initialen Kopien aller Werte von $b_{OT}$ operiert, wird diese Veränderung nicht berücksichtigt.

Die Lösung dafür ist, die Werte der öffentlichen Felder nicht bei der Initialisierung zu übergeben, sondern $B_{OT}$ so zu implementieren, dass es auf *B* zugreift, wenn öffentliche Felder in einer Methode referenziert werden[3].

[3] Die privaten Objektzustände von *b* können nicht mehr geändert werden, da alle Methoden (auch öffentliche) von *B* so redefiniert wurden, dass die entsprechenden Methoden auf ihr Surrogate-Objekt zugreifen, welches auf den initialen Kopien operiert.

Listing 4.5: Vollständige Redefinition der Methode *B.bm*

```
1  public class B {

3      private int x = 1;
4      public  int y = 2;

6      public int bm(int i) {

8          B_OT b_OT = B_OT.getInstance(this);
9          if (b_OT == null) {
10           b_OT = new B_OT(this,x);
11           B_OT.addInstance(this,b_OT);
12         }
13           return b_OT.bm(i)
14     }
15     ...
16 }
```

Listing 4.6: Vollständig generierte Klasse $B_{OT}$

```
1  public class B_OT {

3      private int x;
4      private B base;

6      private static HashMap <B,B_OT> instance
7      public  static void addInstance(B base) { ... }
8      public  static B_OT getInstance(B base) { ... }

10     public B_OT(B base,int x) {
11         this.base = base;
12         this.x = x;
13    }

15     public int bm(int i) {
16         return i + base.y + this.x;
17     }
18     ...
19 }
```

**Vererbung**

Ziehen wir als nächstes eine Vererbungsbeziehung von *B* in Betracht, so entsteht folgendes Problem: Erbt *B* von einer Klasse *A*, dann müssen wir auch eine Vererbungsbeziehung für die Klasse $B_{OT}$ festlegen, da die kopierten Methoden von *B* auf vererbte Methoden von *A* zugreifen könnten. Wir haben die Wahl, $B_{OT}$ von *A* oder von einer Klasse $A_{OT}$ erben zu lassen, die analog zu $B_{OT}$ eine Kopie von *A* ist.

Egal wie wir uns jedoch entscheiden, werden wir ein $b_{OT}$-Objekt im Allgemeinen nicht vollständig initialisieren können. Wenn wir davon ausgehen, dass die Klasse *A* auch private Felder deklariert, so sind diese nicht im Sichtbarkeitsraum von *B*. Bei der Initialisierung des Surrogate-Objekts $b_{OT}$ in *B* gehen uns also die aktuellen Zustände verloren. Stattdessen werden die Zustände erzeugt, die für die Initialisierung eines *b*-Objekts herangezogen werden. Dies führt dazu, dass das Verhalten von $b_{OT}$ nun nicht mehr zwingend dem eines *b*-Objekts entspricht.

Die Kapselung der Klassen und damit der Zugriff auf die privaten Objektzustände stellt also das wesentliche Problem dar, das ein Wrappen von Klassen zur Laufzeit nur bedingt realisierbar macht. Daher wird ein weiterführender Ansatz beschrieben, der diese Problematik umgeht.

### 4.2.2 Statische Dispatch-Klassen

Die Idee dieses Ansatzes basiert, ähnlich wie im vorangegangenen, auf der Generierung neuer Klassen zur Laufzeit. Jedoch wollen wir nun die Klassen nicht wrappen, sondern nur die schemaverändernden Elemente der bisherigen Transformationen in die neuen Klassen auslagern und eine Delegation zwischen einer gebundenen Basismethode und der Dispatch-Logik in der neuen Klasse konstruieren.

Für eine zu webende Basisklasse *B* erzeugen wir uns also wieder eine Klasse $B_{OT}$ (im Folgenden: *Dispatch-Klasse*), die jetzt nur noch den Initial-Wrapper und den Chaining-Wrapper der gebundenen Methoden enthält. Gebundene Basismethoden werden so redefiniert, dass sie einen Aufruf an ihren entsprechenden Initial-Wrapper in der Dispatch-Klasse delegieren. Alle ungebundenen Methoden bleiben unverändert in der Basisklasse bestehen (siehe Abbildung 4.2).

Da nun die Dispatch-Logik in $B_{OT}$ ausgelagert wird, aber der Kontext des Basisobjekts *b* relevant bleibt, prüfen wir, in wie weit dies mit der Kapselung der Basisklasse *B* in Konflikt steht. Tatsächlich stellen wir fest, dass der Initial-Wrapper keinen Kontext des Basisobjekts benötigt und der Chaining-Wrapper in seiner Implementierung nur an zwei Stellen das Basisobjekt referenziert:

- als Parameter beim Aufruf der Callin-Wrapper (Lifting)
- beim Aufruf der Original-Methode beim Rekursionsende

Die Referenz des Basisobjekts könnten wir den Wrapper-Methoden jedoch explizit als ein zusätzliches Argument übergeben. Somit wären wir in der Lage, die Dispatch-Logik als statische Klassenelemente zu implementieren und müssen folglich auch keine Objekte der Dispatch-Klassen erzeugen[4].

[4]Dies setzt voraus, dass wir auch keine Vererbung zwischen den Dispatch-Klassen bräuchten. Die Initial-Wrapper benutzen zwar Vererbung (Vererbung der Anmeldeschnittstellen), jedoch ist diese Vorgehensweise für ein Laufzeitweben, wie schon in den Anforderungen angedeutet, nicht mehr realisierbar und muss ohne Vererbung umgesetzt werden.

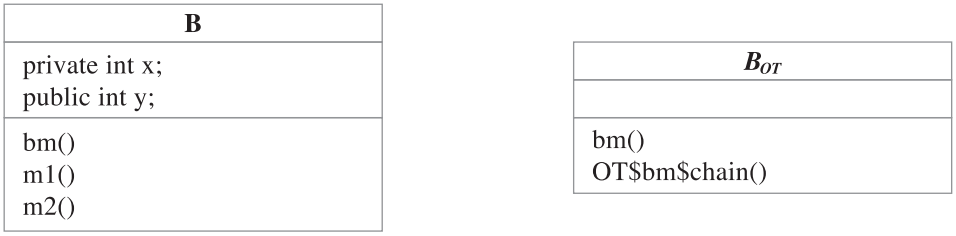


Abbildung 4.2: Dispatch-Klasse

Durch den ausgelagerten Chaining-Wrapper könnten wir den Dispatch-Mechanismus der alten Strategie und damit die schichtweise Abarbeitung der Team-Instanzen auf ein Laufzeit-Weben übertragen, jedoch bleiben Probleme bezüglich der gestellten Anforderungen offen, die nun dargelegt werden.

**Basis-Dispatch-Konflikt**

Da bei diesem Ansatz alle Methoden in der Original-Klasse bleiben, müssen wir für die gebundenen Methoden den Basis-Dispatch-Konflikt auflösen. In statischen Webeverfahren, wie z.B. vom AspectJ-Compiler [8] verwendet, wird dieses Problem mit Hilfe von *Java-Closures* gelöst.

Eine Closure (ein Abschluss) ist ein Konstrukt, das aus der *funktionalen Programmierung* stammt und bezeichnet eine *anonyme Funktion*, die an einen bestimmten *lexikalischen Kontext* gebunden ist.

In Java wird eine Closure mit Hilfe von *lokalen* oder *inneren* Klassen[5] konstruiert, da diese im Kontext der umschließenden Klasse bzw. Methode instanziert wird. Innerhalb einer Closure-Klasse ist es nun möglich, den sogenannten *Joinpoint-Shadow* (gemeint sind die Instruktionen, die zu einem Joinpoint gehören) zu verschieben und stattdessen den Dispatch-Code zu generieren. Der Dispatch-Code erzeugt eine Instanz der Closure-Klasse und übergibt diese dem Advice-Code. Dieser kann bei einer proceed-Operation den verschobenen Joinpoint-Shadow in der Closure-Instanz aufrufen.

Das Hinzufügen von inneren Klassen, auch wenn sie nur in einer Methode deklariert werden, führt jedoch auf Bytecode-Ebene im Allgemeinen zu einer Veränderung des Schemas der umschließenden Klasse. Dies liegt daran, dass eine innere Klasse vom Java-Compiler in ein separates Classfile übersetzt wird und diese genauso wie andere Klassen, nicht ohne weiteres auf die privaten Felder ihrer

[5] Da man zwar die Wirkungsweise einer Closure mit einer inneren Klasse simulieren kann, ist eine Closure aber immer noch eine anonyme Funktion und keine Klasse, so dass man mit dem Begriff Java-Closure etwas vorschichtig sein sollte. Für zukünftige Versionen der Programmiersprache Java sind Closures angekündigt.

äußeren Klasse zugreifen darf. Um dies trotzdem möglich zu machen, wird vom Compiler für jedes private Klassenelement der äußeren Klasse, auf das die innere Klasse zugreift, eine öffentliche *access*-Methode generiert, um die Kapselung aufzuheben. Diese Vorgehensweise kommt für ein Laufzeit-Weben mit JPLIS jedoch nicht in Frage, da die Generierung der access-Methoden nicht erlaubte Redefinitionen sind.

Wenn der Basis-Dispatch-Konflikt durch das Verschieben des Basiscodes nicht lösbar ist, könnte eine andere Lösung in Frage kommen: Die Benutzung einer *globalen Variable*.

Ein *Dispatch-Flag* in Form eines zusätzlichen Klassenattributs von $B_{OT}$ könnte einer modifizierten Basismethode *B.bm* signalisieren, wie sie sich verhalten soll. Listing 4.7 zeigt wie so eine Implementierung aussehen könnte. Eine `switch`-Konstruktion ruft aufgrund des Zustands des Dispatch-Flags entweder den Initial-Wrapper in der Dispatch-Klasse auf, oder führt den Original-Code der Basismethode aus. Der Chaining-Wrapper dagegen setzt vor dem Aufrufen der Basismethode das Flag so, dass der Original-Code ausgeführt wird (Listing 4.8 Zeile 9).

Listing 4.7: Klasse B mit modifizierter Methode B.bm

```
1  public class B {
2
3      private int x;
4      public  int y;
5
6      public int bm(int i) {
7
8          switch (B_OT.dispatchFlag)
9          {
10             case(DISPATCH) :
11                 return B_OT.bm(this,i);
12                 break;
13             default :
14                 B_OT.dispatchFlag = DISPATCH;
15                 // Original Code
16                 ...
17         }
18     }
19 }
```

Listing 4.8: Aufruf der Basismethode von $B_{OT}$ mit Dispatch-Flag

```
1  public class B_OT {

3      public static int dispatchFlag = DISPATCH;

5      public _OT$bm$chain(B base,Team _OT$teams[], int
                   _OT$teamIDs[],int _OT$idx,int i) {

7          Object res = null;
8          if(_OT$idx >= _OT$teams.length) {
9              dispatchFlag = NO_DISPATCH;
10             res = base._OT$bm$orig(i);
11             return res;
12         }
13         ...
14     }
15 }
```

Bei der Verwendung von globalen Variablen muss jedoch die Existenz von Nebenläufigkeit berücksichtigt werden. Die Anweisungsfolge, beginnend beim Setzen des Dispatch-Flags im Chaining-Wrapper (Listing 4.8 Zeile 9) bis zum Zurücksetzen in der Basismethode (Listing 4.7 Zeile 14), ist ein kritischer Abschnitt, der entsprechend synchronisiert werden muss.

Um dies zu realisieren, könnten wir das Dispatch-Flag in einem `ThreadLocal`-Objekt verwalten. Damit wäre sichergestellt, dass jeder Thread auf ein ihm zugehöriges Flag zugreift und damit ein nebenläufiger Zugriff ausgeschlossen wird. Die Verwaltung einer `ThreadLocal`-Variablen wird jedoch intern über eine `Map` implementiert. Dies bedeutet, dass ein zusätzlicher Aufwand während des Callin-Dispatches zu berücksichtigen ist.

### Aspektbindung bei vererbter Methode

Wird eine Methode *B.bm* von einer Klasse *A* vererbt und diese Methode in *B* gebunden, stehen wir vor dem Problem, keinen Dispatch-Code in *B* generieren zu können, wenn die Methode in dieser Klasse nicht redefiniert wurde. Ein Aufruf von *B.bm* wird unmittelbar zur Superklasse *A.bm* delegiert. Ist *A.bm* auch gebunden, muss der Dispatch-Code der Methode feststellen, ob der Aufruf von *B* oder von *A* kam, um den richtigen Initial-Wrapper aufzurufen. Eine Lösung dafür wäre es, eine Typ-Prüfung der `this`-Referenz mit Hilfe des `instanceof`-Operators in *A* durchzuführen. Der `instanceof`-Operator ist jedoch eine relativ teure Operation, da er einen Lookup im Vererbungsbaum benötigt, was einem performanten Callin-Dispatch entgegensteht.

### Decaspulation

Ein weiteres Problem bildet das Durchbrechen der Kapselung, ohne das Klassenschema zu verändern. Wir dürfen zur Laufzeit weder *Getter*- bzw. *Setter*-Methoden in die Basisklassen generieren, noch die Sichtbarkeit der betroffenen Klassenelemente heraufsetzen, um direkt auf diese zuzugreifen.

Eine Möglichkeit, die sich jedoch anbietet, ist die Benutzung der *Java-Reflection-API*. Über die `Class`-Instanz eines Objekts erhält man die Meta-Informationen der Felder und der Methoden der Klasse des Objekts:

- `Field getDeclaredFields(String name)`
  Liefert ein `Field`-Objekt eines deklarierten Felds
- `Method getDeclaredMethods(String name)`
  Liefert ein `Method`-Objekt einer deklarierten Methode

Für die Klasse `Field` existieren Methoden, um für ein konkretes Objekt den Wert des deklarierten Felds zu setzen bzw. zu lesen. Analog dazu besitzt die Klasse `Method` die Methode `invoke()`, um auf einem Objekt die entsprechende Methode aufzurufen.

Sowohl die `Field`- als auch die `Methode`-Klasse erweitern die Klasse `java.lang.reflect.AccessibleObject`. Diese Klasse definiert die Methode `setAccessible()` mit der man den Zugriff auf private Elemente ermöglicht.

Ein wesentlicher Nachteil dieser Vorgehensweise ist die schlechte Performance [1] des Reflection-Mechanismus, da der Zugriff nicht direkt erfolgt, sondern erst von der JVM gesondert ausgewertet werden muss. Ein anderer Nachteil besteht darin, dass das Setzen der Zugriffsrechte über die Methode `setAccessible` nur erlaubt ist, wenn eine entsprechende *Java-Permission* vorhanden ist. Dies muss jedoch nicht in allen Umgebungen der Fall sein, was eine Einschränkung der Funktionalität des Webers bedeuten würde.

### Bewertung der Strategie

Die Strategie basiert auf dem Auslagern der bisherigen Dispatch-Logik in andere Klassen. Da die Elemente der Dispatch-Logik statisch implementiert werden können, entfällt ein Erzeugen von zusätzlichen Objekten dieser Klasse und ein entsprechendes Mapping, wie wir es im ersten Ansatz beschrieben haben. Zudem basiert die Dispatch-Klassen-Strategie im Wesentlichen auf der alten Webestrategie (Generierung von Initial- und Chaining-Wrappern), was eine Wiederverwendbarkeit der bestehenden Transformer möglich macht.

Jedoch bleiben folgende Probleme offen:

- Anforderung 1a: Basis-Dispatch-Konflikt mit Dispatch-Flag
  **Problem:** Synchronisation einer globalen Variablen

- Anforderung 1b: Bindung bei vererbter Methode mit `instanceof`-Operator
  **Problem:** Performance der Operation

- Anforderung 4: Decapsulation mit Java-Reflection
  **Problem:** Performance der Reflection-API und Verfügbarkeit der Security-Policy

Die Anforderungen könnten bezüglich der Verfügbarkeit der Security-Policy bedingt erfüllt werden, allerdings liegt die Performance der einzelnen Lösungen weit hinter der der alten Webestrategie.

Des Weiteren sind es nicht nur die Redefinitionsbeschränkungen, die einer Übertragbarkeit der alten Webestrategie auf ein Laufzeit-Weben entgegenstehen. Wir müssen auch die Eignung dieser Strategie für ein dynamisches Weben in Frage stellen. Ein entscheidendes Kriterium, und daher eine Hauptanforderung an einen dynamischen Weber, ist ein effizientes Weben zur Laufzeit, um die Aspektbindung für Benutzer und Anwendung möglichst transparent, d.h. mit einer minimalen Verzögerung des Programmablaufs zu realisieren. Die Implementierung der Chaining-Wrapper ist abhängig von den definierten Callin-Bindungen der zugehörigen Basismethoden und daher komplex. Beim Laufzeitweben ist die Menge der Bindungen nicht festgelegt, weil Aspekte hinzukommen oder eventuell entfernt werden können. Daher müsste der Dispatch-Code immer wieder aufwändig durch Bytecode-Transformationen angepasst werden. Diese invasive Vorgehensweise sollte zur Laufzeit möglichst vermieden werden. Ziel einer dynamischen Webestrategie sollte daher sein, den Umfang der Bytecode-Transformationen zu minimieren.

Im Folgenden wird eine neue Webestrategie herausgearbeitet, die den gestellten Anforderungen gerecht werden soll. Die Anforderungen, für die im ersten Ansatz keine performanten Lösungen gefunden wurden, sollen mit Hilfe des 2-Phasen-Webens gelöst werden (siehe Kapitel 3, Abschnitt 3.2.4). Dabei sollen die Klassen jedoch nur in einem minimalen und konstanten Umfang präpariert werden. Wir werden im Kapitel 5 sehen, dass die dafür notwendigen Modifikationen zur Ladezeit sehr effizient durchgeführt werden können und daher ein probates Mittel darstellen, um zur Laufzeit einen Performancegewinn zu sichern.

## 4.3 Generische Teamschnittstellen

Um eine Minimierung der Bytecode-Transformationen zur Laufzeit zu erreichen, wäre es optimal, wenn wir eine Basismethode nur genau einmal weben müssten. Der Dispatch-Code müsste infolgedessen unabhängig von dem Typ und der Anzahl der unterschiedlichen Bindungen sein. Dies ist, wie schon erwähnt, bei den individuellen Implementierungen der Chaining-Wrapper nicht der Fall, da die Signaturen der Team-Schnittstellen nicht einheitlich sind. Für jedes Team muss ein spezieller `case`-Fall in den Wrappern erzeugt werden, der die entsprechenden Callin-Wrapper aufruft.

Dies führt zu dem Ansatz, den Dispatch-Code in den Basisklassen generisch zu definieren. Dementsprechend muss auch eine generische Schnittstelle zum Aufrufen der Callin-Wrapper für die Team-Klassen realisiert werden.

Bevor diese Lösung beschrieben wird, soll kurz auf die Präparations-Phase der Klassen zur Ladezeit eingegangen werden.

### 4.3.1 Object Teams-Interface für Basisklassen

Zur Ladezeit werden alle Klassen mit dem im Listing 4.9 dargestellten Interface `IBASE` präpariert (die Realisierung wird in Kapitel 5 beschrieben). Da die jeweiligen Implementierungen des Interfaces zur Ladezeit noch unbestimmt sind, enthalten die Methoden einen ”Dummycode”[6]. Auf ihre Bedeutung wird im Laufe des Kapitels eingegangen.

Listing 4.9: Interface IBASE

```
1  public interface IBASE {

3      public Object _OT$callOrig(int boundMethodId,
4                                 Object[] args);


7      public Object _OT$access(int accessId,int op_kind,
8                               Object[] args);

10     protected Object _OT$callAllBindings(int boundMethodId,
11                                          Object[] args);

13 }
```

[6] Sie liefern eine *null*-Referenz zurück.

### 4.3.2 Eliminierung der Chaining-Wrapper

Die Lösung der nun vorgestellten Strategie basiert auf folgender Idee: Anstatt die individuellen Callin-Wrapper der Teams aufzurufen, soll dem Dispatch-Code eine generische (einheitliche) Schnittstelle in den Basismethoden zur Verfügung gestellt werden. Ist so eine Schnittstelle vorhanden, wird ein Chaining-Wrapper überflüssig, da sich aufeinanderfolgende Teams in der abzuarbeitenden Registrierungsliste durch genau diese Schnittstelle direkt, also ohne den Umweg über den Chaining-Wrapper, aufrufen können.

Listing 4.10: Generische Teamschnittstelle

```
1  public final Object callAllBindings(IBase ibase,
2                    int boundMethodId,
3                    Team[] teams,
4                    int idx,
5                    Object[] args) {

7     Object res = null;

9     callBefore(ibase,boundMethodId,args);
10    res = callReplace(ibase,boundMethodId,teams,idx,args);
11    callAfter(ibase,boundMethodId,args);

13    return res;
14
15 }
```

Die Implementierung der neuen Team-Schnittstelle (`Team.callAllBindings()`) ist im Listing 4.10 dargestellt[7]. Sie ist eine *Template-Methode* (Design-Pattern: *Template and Hook*) die hintereinander folgende *Hook-Methoden* ausführt:

- `Team.callBefore()`: Ausführung der before-Callins eines Teams
- `Team.callReplace()`: Ausführung der replace-Callins eines Teams
- `Team.callAfter()`: Ausführung der after-Callins eines Teams

Die Hook-Methoden enthalten, wie wir später sehen werden, einen teamklassen-spezifischen Dispatch-Code zum Ausführen der entsprechenden Callin-Wrapper eines Teams.

Der Aufruf der `callAllBindings()`-Methode bewirkt die Aufrufsequenz der Callinwrapper für eine Team-Instanz. Für die schichtweise Ausführung mehrerer Instanzen dient die neue Team-Methode `Team.callNext()`.

[7] Die Signatur im Listing ist noch nicht vollständig und wird im Folgenden erweitert.

Listing 4.11: Neue callNext()-Methode

```
2    public final Object callNext(IBase ibase,
3                        int boundMethodId,
4                        Team[] teams,
5                        int idx,
6                        Object[] args) {

8       if (idx+1 < teams.length)
9          return teams[idx+1].callAllBindings(ibase,
                        boundMethodId, teams, idx+1, args);
10      else
11         return ibase._OT$callOrig(boundMethodId,args);

13   }
```

Die `callNext`-Methode wird je nach Ausgangslage an zwei Stellen aufgerufen:

- Besitzt ein Team keinen replace-Callin, dann implementiert es auch nicht die Hook-Methode `callReplace()`. Stattdessen wird die vererbte Methode der Klasse `Team` aufgerufen (siehe Listing 4.12), welche unmittelbar `callNext()` ausführt.
- Besitzt ein Team einen replace-Callin, wird die `callNext()`-Methode durch einen base-call aufgerufen.

Listing 4.12: Default Implementierung von callReplace()

```
1 public Object callReplace(IBase ibase,
2              int boundMethodId,
3              Team[] teams,
4              int idx,
5              Object[] args) {

7       return callNext(ibase, boundMethodId, teams, idx,
                     args);
8  }
```

Der Rekursionsschritt des Chaining-Wrappers wird also dadurch ersetzt, dass die aktuelle Team-Instanz nun jeweils direkt die nachfolgende Instanz in der Liste über die neue Team-Schnittstelle aufruft. Dafür wird der `callAllBindings()`-Methode sowohl die Liste der aktiven Teams, als auch ein *Iterator*, der den aktuellen Team-Index transportiert, übergeben (analog zum Chaining-Wrapper). Sind alle Teams abgearbeitet, wird der Original-Code der Basismethode aufgerufen, der in die neue `IBASE._OT$callOrig()`-Methode verschoben wurde.

### Verschieben des Basiscodes

Um den Basis-Dispatch-Konflikt aufzulösen, soll die Methode `IBASE._OT$callOrig()`, im Gegensatz zur alten Webestrategie, als Platzhalter für potenziell alle Methoden einer Klasse fungieren. Die Parameter der Methode bestehen aus einem `Object`-Array und einer *Bound Method ID*, die eine gebundene Methode innerhalb einer Basisklasse eindeutig identifiziert. Das Array soll die zuvor *geboxten* Argumente eines Aufrufs einer gebundenen Basismethode empfangen. Im Listing 4.13 wird gezeigt, wie über eine switch-Anweisung anhand der Bound Method ID die jeweiligen Code-Fragmente zur Ausführung gebracht werden.

Listing 4.13: Generische callOrig-Methode

```
1  public Object _OT$callOrig(boundMethodId,Object[] args) {

3      Object res = null;

5      switch(boundMethodId) {
6      case 1:
7          // unbox args
8          // Code für boundMethodId=1
9          break;
10     case 2:
11         // unbox args
12         // Code für boundMethodId=2
13         break;
14     case 3:
15             ...
16         break;
17     }
18     return res;

20 }
```

### Initialer Dispatch

In der Basis-Methode wird, anstelle des verschobenen Basis-Codes, der neue Dispatch-Code generiert. Dieser Dispatch-Code (Listing 4.14) ruft nur das erste Team in der Registrierungsliste[8] über die neue Team-Schnittstelle auf.

[8] Der neue Verwaltungsmechanismus der Team-Registrierung wird im Abschnitt 4.4 behandelt.

Listing 4.14: Generischer Dispatch in der Basismethode

```
1  RType bm(AType_1 a_1, ..., AType_n a_n) {

3      // Box Arguments a_1 ... a_n
4      Object[] args = new Object[n];

6      Team[] teams = ... // Initialisierung des Team-Arrays

8      Object res = teams[0].callAllBindings(this,
9                          1,      /*Bound Method ID*/
10                         teams,  /*Team-Array*/
11                         0,      /*Iterator*/
12                         args,   /*Boxed Args*/);

14     return ... //unbox res;
15 }
```

Damit die Argumente der Basismethode in die einheitliche Schnittstelle integrierbar sind, werden sie einmalig bei dem initialen Dispatch in ein *Object*-Array "verpackt"[9]. Neben der Liste der aktiven Teams wird die Bound Method ID der entsprechenden Basismethode übergeben, damit der entsprechende Basis-Code in der `_OT$callOrig()`-Methode ausgeführt werden kann.

Die Abbildung 4.3 zeigt den Kontrollfluss der schichtweisen Abarbeitung mit Hilfe der neuen Team-Schnittstellen und der `_OT\$callOrig`-Methode[10]. Für jedes Team werden zunächst die `callBefore()`-Methoden ausgeführt. Im nächsten Schritt dann die `callReplace()`-Methoden. Sie rufen immer dann die `callNext()`-Methode auf, wenn die entsprechenden Rollenmethoden einen base-call ausführen (ansonsten würde die Fortführung der Abarbeitung, wie im Chaining-Wrapper, vorzeitig beendet werden, ohne die verbleibenden Teams zu berücksichtigen). Sind weitere Teams in der Liste vorhanden, wird die `callAllBindings()`-Methode für das nächste Team aufgerufen und die Abarbeitung fortgesetzt (vgl. Rekursionsschritt im Chaining-Wrapper). Sind alle Teams abgearbeitet, wird schließlich die `_OT$callOrig()`-Methode ausgeführt und das Ergebnis zurückgeliefert (vgl. Rekursionsende im Chaining-Wrapper). Nun wird in umgekehrter Reihenfolge die `callAfter()`-Methode der Teams aufgerufen.

Damit haben wir die geforderte schichtweise Abarbeitung der Teams ohne einen Chaining-Wrapper aufgezeigt.

[9] Diese Technik wird auch in der alten Webestrategie verwendet, um unbenutzte Parameter im Rahmen eines Parameter-Mappings zu tunneln.

[10] Die Aufrufe der Callin-Wrapper wurden bewusst weggelassen, da die Abbildung nur den Kontrollfluss entlang der neuen Team-Methoden veranschaulichen soll.

### 4.3.3 Teamdispatch zu den Callin-Wrappern

Nachdem der neue Abarbeitungsmechanismus der registrierten Team-Instanzen festgelegt worden ist, müssen wir uns jetzt überlegen, wie ein Team selbst entscheidet, welche ihrer Callin-Wrapper es für einen Dispatch ausführen muss und welche nicht. Ziel ist es, einen effizienten `switch`-Dispatch für die Teamklassen zu implementieren, um damit unserer Anforderung nachzukommen, einen performanten Callin auszuführen.

Die Aufrufe der Callin-Wrapper befinden sich in der alten Strategie in den Chaining-Wrappern einer zugehörigen Basismethode *B.bm*. Innerhalb des Wrappers wurde anhand des Typs $T$ der jeweiligen Team-Instanz bestimmt, welche der Callin-Wrapper aufzurufen sind. Die Menge der auszuführenden Callin-Wrapper für eine Team-Instanz wird also durch zwei Parameter festgelegt, die Basismethode *B.bm* und den Typ $T$ der Team-Instanz.

Die Vermutung liegt daher nahe, dass die Bound Method ID der Basismethode für diesen Zweck nicht ausreichen kann. Dies zeigt folgendes Beispiel:

Dafür seien $B1$, $B2$, $B3$, $B4$ Basisklassen mit folgenden Subklassenbeziehungen:

$$B1 \sqsubset B2 \sqsubset B3 \sqsubset B4$$

Des Weiteren deklarieren zwei Teams $T1$ und $T2$ Callin-Bindungen $b1$ und $b2$ bzw. $b3$ und $b4$ wie folgt:

$$\begin{aligned} b1 &= \{\ T1, R1, B1, rm, bm, before\ \} \\ b2 &= \{\ T1, R2, B2, rm, bm, before\ \} \\ b3 &= \{\ T2, R3, B3, rm, bm, before\ \} \\ b4 &= \{\ T2, R4, B4, rm, bm, before\ \} \end{aligned}$$

Also existieren für die Klassen $B1$ und $B2$ folgende Bindungen:

$$\begin{aligned} BEFORE_{B1,bm} &= \{\ b1\ \} \\ BEFORE_{B2,bm} &= \{\ b1, b2\ \} \end{aligned}$$

Und für die Klassen $B3$ und $B4$:

$$\begin{aligned} BEFORE_{B3,bm} &: \{\ b1, b2, b3\ \} \\ BEFORE_{B4,bm} &: \{\ b1, b2, b3, b4\ \} \end{aligned}$$

Wollen wir die Bound Method ID für eine `switch`-Anweisung für $T1$ und $T2$ benutzen, müssen die Teams feststellen können, ob ein Aufruf von $B1.bm$ oder $B2.bm$ bzw. $B3.bm$ oder $B4.bm$ kommt, um die entsprechenden Callin-Wrapper für ihre Bindungen auszuführen.

Listing 4.15: callBefore-switch in Team T1

```
1  void callBefore(IBase ibase, int boundMethodId,...) {

3      switch(boundMethodId) {
4          case ID_B1.bm : // Aufruf von B1.bm()
5              // Callin-Wrapper für R1.rm();
6              this._OT$R1$rm$bm$(ibase, ...);
7              break;
8              // Aufruf der Callin-Wrapper R1.rm() und R2.rm()
9          case ID_B2.bm : // Aufruf von B2.bm()
10             this._OT$R1$rm$bm$(ibase, ...);
11             this._OT$R2$rm$bm$(ibase, ...);
12             break;
13     }
14 }
```

Listing 4.16: callBefore-switch in Team T2

```
1  void callBefore(IBase ibase, int boundMethodId,...) {

3      switch(boundMethodId) {
4          case ID_B3.bm :
5              // Aufruf von Callin-Wrapper R3.bm()
6              this._OT$R3$rm$bm$(ibase, ...);
7              break;
8          case ID_B4.bm :
9               // Aufruf der Callin-Wrappers R3.rm() und R4.rm()
10             this._OT$R3$rm$bm$(ibase, ...);
11             this._OT$R4$rm$bm$(ibase, ...);
12             break;
13     }
14 }
```

Betrachten wir die Listings 4.15 und 4.16, so finden wir keine Belegung der Bound Method ID für die einzelnen `case`-Fälle, um einen korrekten Dispatch für $T1$ und $T2$ zu realisieren. Denn zum einen muss $ID_{B3}.bm$ ungleich $ID_{B4}.bm$ sein, damit $T2$ einen Aufruf von $B3.bm$ und $B4.bm$ unterscheiden kann. Zum anderen muss $ID_{B4}.bm$ gleich $ID_{B2}.bm$ sein, damit ein Aufruf von $B4.bm$ alle Callins für $T1$ und $T2$ ausführt. Dann gilt aber $ID_{B3}.bm$ ungleich $ID_{B2}.bm$ und ein Aufruf von $C.bm$ ruft nicht die richtigen Callins von $T1$ auf.

Das Problem lässt sich wie folgt umschreiben: Die Bound Method ID muss zum einen weiter vererbt werden, damit ein Aufruf von *B*3.*bm* oder *B*4.*bm* die Callins des Teams *T*1 ausführt, und zum anderen unterschiedlich sein, damit *T*1 und *T*2 innerhalb einer Vererbungskette unterscheiden können, von welchem Instanz-Typ der Aufruf kam.

Wir legen daher für das folgende Vorgehen fest, dass die Bound Method ID an ihre Submethoden weitervererbt wird und versuchen eine Lösung zu finden, den fehlenden Parameter für ein Dispatching in den Teamklassen zu realisieren.

### Lösung mit instanceof-Operator

Eine Lösung für das Problem ist die Typprüfung der Basisklasse unter Verwendung des `instanceof`-Operators. Wenden wir den Operator auf die übergebene Basisreferenz an, können wir herausfinden, ob der Aufruf von einem *B*2 oder einem *B*1 Typ kam (Listing 4.17 und 4.18). Dabei müssen wir jedoch auf die Reihenfolge achten, in der wir die Typprüfungen durchführen. Genauer gesagt, müssen wir immer mit dem speziellsten Typ anfangen (also in unserem Beispiel mit *B*2 bzw. *B*4).

Listing 4.17: callBefore-switch in Team T1 mit instanceof-Operator

```
1  void callBefore(IBase ibase,int boundMethodId,...) {

3      switch(boundMethodId) {
4          case ID_bm :
5              if (ibase instanceof B2) {
6                  // Aufruf von B2-Instanz
7                  this._OT$R1$rm$bm$(ibase, ...);
8                  this._OT$R2$rm$bm$(ibase, ...);
9              } else if (ibase instanceof B1) {
10                 // Aufruf von einer B1-Instanz
11                 this._OT$R1$rm$bm$(ibase, ...);
12             }
13             break;
14     }
15 }
```

Listing 4.18: callBefore-switch in Team T2 mit instanceof-Operator

```
1  void callBefore(IBase ibase, int boundMethodId,...) {

3      switch(boundMethodId) {
4          case ID_bm :
5              if (ibase instanceof B4) {
6                  // Aufruf von einer B4-Instanz
7                  this._OT$R3$rm$bm$(ibase, ...);
8                  this._OT$R4$rm$bm$(ibase, ...);
9              } else if (ibase instanceof B3) {
10                 // Aufruf von einer B3-Instanz
11                 this._OT$R3$rm$bm$(ibase, ...);
12             }
13             break;
14     }
15 }
```

Der `instanceof`-Operator ist, wie schon erwähnt, eine relativ teure Operation. Daher steht diese Lösung im Widerspruch mit unserer Forderung nach einem performanten Dispatch. Optimal wäre es, wenn ein Team neben der Bound Method ID, eine weitere Id für eine `switch`-Konstruktion zur Verfügung gestellt bekäme, anhand derer es entscheiden könnte, welche ihrer Callin-Wrapper es für eine bestimmte Methode aufrufen muss.

### Herleitung einer Callin ID für ein Team-Dispatch

Im Folgenden soll eine *Callin ID* für ein Team $T$ und eine Basisklasse $B$ gefunden werden, mit der eine Instanz von $T$ entscheiden kann, welche ihrer Callin-Wrapper sie für einen Dispatch einer konkreten Methode von $B$ aufrufen muss.
Sei dafür $T$ ein Team mit folgenden before-, replace- und after-Bindungen:

$$BEFORE_T = \{ b_1, ..., b_i \}$$
$$REPLACE_T = \{ r_1, ..., r_j \}$$
$$AFTER_T = \{ a_1, ..., a_k \}$$

Und $BOUNDCLASSES_T = \{ B_1, ..., B_n \}$ die Menge der Basisklassen, die von $T$ gebunden werden.

Für jedes Element von $B_i \in BOUNDCLASSES_T$ definieren wir folgende Callin-Mengen:

$$BEFORE_{B_i} = \{ \text{b} \in BEFORE_T \mid B_b \sqsubseteq B_i \}$$
$$REPLACE_{B_i} = \{ \text{b} \in REPLACE_T \mid B_b \sqsubseteq B_i \}$$
$$AFTER_{B_i} = \{ \text{b} \in AFTER_T \mid B_b \sqsubseteq B_i \}$$

Dies sind also die Mengen von Callin-Bindungen von $T$, die für die gebundenen Klassen $B_i \in BOUNDCLASSES_T$ gelten. Die Menge der Bindungen von $T$, die für eine beliebige Basisklasse $B$ gelten, sind:

$$BEFORE_B = \{ \text{b} \in BEFORE_T \mid B_b \sqsubseteq \text{B} \}$$
$$REPLACE_B = \{ \text{b} \in REPLACE_T \mid B_b \sqsubseteq \text{B} \}$$
$$AFTER_B = \{ \text{b} \in AFTER_T \mid B_b \sqsubseteq \text{B} \}$$

Für alle $B_i \in BOUNDCLASSES_T$ und ein beliebiges $B$ gilt:

**P1**: $B_i \sqsubseteq \text{B} \wedge \neg \exists\, B_j \in BOUNDCLASSES_T : B_i \sqsubset B_j \sqsubset B \Longrightarrow BEFORE_B = BEFORE_{B_i} \wedge REPLACE_B = REPLACE_{B_i} \wedge AFTER_B = AFTER_{B_i}$

Wenn es also kein $B_j$ gibt, das weder echte Subklasse von $B_i$ noch eine echte Superklasse von $B$ ist, dann kann $B$ auch keine weiteren Callins von $T$ vererbt bekommen haben, die $B_i$ nicht schon hat.

Wir vergeben nun eine eindeutige Callin ID $ID_{T_{B_i}}$ für jedes $B_i \in BOUNDCLASSES_T$ und konstruieren uns für das Team $T$ eine Abbildung $f_T$, die einer Basisklasse $B$ mit $B \sqsubseteq B_i$ eine entsprechende $ID_{T_{B_i}}$ zuweist:

$$f_T : (\, B \,) \rightarrow ID_{T_{B_i}}$$

Für $f_{T1}$ und alle $B_i \in BOUNDCLASSES_T$ und eine Basisklasse $B$ soll gelten:

**P2**: $B_i \sqsubseteq \text{B} \wedge \neg \exists\, B_j \in BOUNDCLASSES_T : B_i \sqsubset B_j \sqsubset \text{B} \Longleftrightarrow f_T(B_i) = f_1(B)$

Ermitteln wir also zu einer Basisklasse $B$ die Callin ID $f_T(B)$ und es existiert eine $f_T(B_i)$ mit $f_T(B_i) = f_T(B)$, dann gilt wegen **P2** und **P1**, dass die Callin-Mengen von $B$ und $B_i$ gleich sind. Existiert keine Callin ID für $B$, dann ist $B$ auch keine Subklasse eines $B_i \in BOUNDCLASSES_T$ und kann auch keine Callin-Bindungen von T1 besitzen.

Für die Implementierung der Hook-Methoden des Teams $T$ bedeutet das, dass die Aufrufe der Callin-Wrapper für die Bindungen $BEFORE_T$, $REPLACE_T$, $AFTER_T$ mittels der IDs $ID_{T_{B_i}}$ mit $B_i \in BOUNDCLASSES_{T1}$ gruppiert werden können. Beispiel 4.2 zeigt anhand der Bindungen zweier Teams $T1$ und $T2$ wie die Callin IDs vergeben werden.

**Beispiel 4.2:**
Basisklassen mit : $B1 \sqsubseteq B2 \sqsubseteq B3 \sqsubseteq B4$

$T1$ mit Bindungen:

$$b1 = \{\ T1, R1, B1, rm1, bm, before\ \}$$
$$b2 = \{\ T1, R1, B3, rm2, bm, before\ \}$$

| Methode | Bindungen | Callin ID | Bound Method ID |
|---|---|---|---|
| $B1.bm$ | b1 | $f_{T1}(B1) = 1$ | 1 |
| $B2.bm$ | b1 | $f_{T1}(B2) = 1$ | 1 |
| $B3.bm$ | b1,b2 | $f_{T1}(B3) = 2$ | 1 |
| $B4.bm$ | b1,b2 | $f_{T1}(B4) = 2$ | 1 |

$T2$ mit Bindungen:

$$b3 = \{\ T2, R1, B2, rm1, bm, before\ \}$$
$$b4 = \{\ T2, R1, B4, rm2, bm, before\ \}$$

| Methode | Bindungen | Callin ID | Bound Method ID |
|---|---|---|---|
| $B1.bm$ | - | - | 1 |
| $B2.bm$ | b3 | $f_{T2}(B2) = 1$ | 1 |
| $B3.bm$ | b3 | $f_{T2}(B3) = 1$ | 1 |
| $B4.bm$ | b3,b4 | $f_{T2}(B4) = 2$ | 1 |

**Implementierung der Hook-Methoden**

Mit der Bound Method ID und der zusätzlichen Callin ID können wir uns jetzt zwei geschachtelte `switch`-Anweisungen konstruieren. Der erste `switch` diskriminiert anhand der Bound Method ID und der zweite anhand der Callin ID. Die Listings 4.19 und 4.20 zeigen die `callBefore()`-Methoden für T1 und T2 aus Beispiel 4.2.

Listing 4.19: callBefore-switch in Team T1 mit Bound Method ID und Callin ID

```
1  void callBefore(IBase ibase, int boundMethodId, int callinId,
                   ...) {

3      switch(boundMethodId) {
4      case 1:
5          switch (callinId) {
6          case 1:
7              // execute BEFORE(B1)
8              break;
9          case 2:
10             // execute BEFORE(B3)
11             break;
12         }
13     }
14 }
```

Listing 4.20: callBefore-switch in Team T2 mit Bound Method ID und Callin ID

```
1  void callBefore(IBase ibase, int boundMethodId, int callinId,
                   ...) {

3      switch(boundMethodId) {
4      case 1:
5        switch (callinId) {
6        case 1:
7            // execute BEFORE(B2)
8            break;
9        case 2:
10           // execute BEFORE(B4)
11           break;
12       }
13     }
14 }
```

Die formale Darstellung der Hook-Methoden für ein Team $T$ mit folgenden Bindungen, sind in den Listings 4.21 und 4.22 dargestellt:

$$BEFORE_T = \{\, b \in CB_T \mid M_b = before \,\}$$
$$REPLACE_T = \{\, b \in CB_T \mid M_b = replace \,\}$$
$$AFTER_T = \{\, b \in CB_T \mid M_b = after \,\}$$

Listing 4.21: callBefore-Methode

```
1  void callBefore(IBase ibase,
2       int boundMethodId,
3       int callinId,
4       Object[] args) {

6      // unbox args ...
7      switch(boundMethodId) {
8      // ∀ b ∈ BEFORE_T
9      case (ID_bbm):
10         switch (callinId) {
11         // ∀ b ∈ BEFORE_T | boundMethodId = ID_bbm
12         case f_T(B_b):
13             this.R_b$rm_b$bm_b(this, a_1, ..., a_n);
14             break;
15         }
16     }
17 }
```

Listing 4.22: callReplace-Methode

```
1  Object callReplace(IBase ibase,
2        int boundMethodId,
3        int callinId,
4        Team[] teams,
5        int idx,
6        Object[] args) {

8      // unbox args ...
9      switch(boundMethodId) {
10     // ∀ b ∈ REPLACE_T
11     case (ID_bbm):
12         switch (callinId) {
13         // ∀ b ∈ REPLACE_T | boundMethodId = ID_bbm
14         case f_T(B_b):
15             return this.R_b$rm_b$bm_b(this, boundMethodId, teams, idx
                                  , a_1, ..., a_n);
16             break;
17         }
18     }
19 }
```

Die `callAfter()`-Methode wird analog zur `callBefore()`-Methode konstruiert.

Die Abbildung $f_T$ kann über eine `HashMap` realisiert werden. Damit eine Team-Instanz die entsprechende Callin ID bestimmen kann, muss sie jedoch wissen aus welcher Basisklasse der Dispatch erfolgte. Dafür könnte der initiale Dispatch eine eindeutige *Class ID* als zusätzlichen Parameter der `callAllBindings()`-Methode übergeben. Der Lookup der Callin ID für jede Team-Instanz stellt jedoch einen zusätzlichen Aufwand dar, den wir berücksichtigen müssen. An einer späteren Stelle wird im Rahmen der neuen Team-Registrierung eine Lösung vorgestellt, die den Lookup zur Dispatch-Zeit vermeidet.

### 4.3.4 Generierung der Hook-Methoden

Da der wesentliche Teil des neuen Dispatchings in den Hook-Methoden der Team-Klassen realisiert wird, bietet es sich an, dass der OT/Java-Compiler diese Methoden generiert. Zum Dispatching wird zum einen die Bound Method ID verwendet und zum anderen die Callin ID. Letztere ist nur für eine Team-Klasse eindeutig und könnte daher vom Compiler problemlos festgelegt werden. Die Bound Method ID dagegen muss global eindeutig sein, denn sie wird sowohl von allen Teamklassen, als auch von dem Dispatch-Code der Basisklassen verwendet. Der inkrementelle Übersetzungsvorgang des OT/Java-Compilers kann die Generierung einer global eindeutigen ID nicht leisten. Um dieses Problem zu lösen, könnten zur Übersetzungszeit der Teamklassen nur lokal eindeutige Bound Method IDs erzeugt und diese zur Ladezeit gegen global eindeutige IDs ausgetauscht werden, die der Weber erzeugt.

### 4.3.5 Aspektbindung bei nicht redefinierter Methode

Ein Spezialfall der Aspektbindung muss gesondert behandelt werden. Wird eine Methode *bm* einer Klasse *B* von einer Klasse *A* geerbt und diese nicht überschrieben, können wir bei einer Bindung dieser Methode keinen Dispatch-Code in *B* generieren, da diese in *A* implementiert ist. Dies führt dazu, dass ein Aufruf von *B.bm* unmittelbar nach *A.bm* delegiert wird. Da auch für *A.bm* ein Dispatch-Code existieren könnte, müsste in *A.bm* unterschieden werden, ob der Aufruf von *A.bm* oder *B.bm* ausging. Um dieses Problem zu lösen, soll die generische Methode `IBASE._OT$callAllBindings()` verwendet werden. Analog zur `_OT$callOrig()`-Methode soll sie unterschiedliche Code-Fragmente ausführen, welche über die Bound Method ID diskriminiert werden. Die erforderlichen Redefinitionen zur Laufzeit für eine gegebene Methode *bm* der Klasse *B*, die von einer Klasse *A* erbt, bestehen aus zwei Schritten. Im ersten Schritt wird die `_OT$callAllBindings()`-Methode von *B* wie folgt redefiniert:

Listing 4.23: Generische callAllBindings-Methode

```
1  public Object _OT$callAllBindings(boundMethodId, Object[]
                 args) {

3     Obeject res = null;

5     switch(boundMethodId) {
6        case 1:
7          Team[] teams = ... // Initialisierung des Team-Arrays
8          teams[0].callAllBindings(this,1,teams,0,args)
9     }
10    return null;
11 }
```

Wir erzeugen also den Dispatch-Code für *B.bm* in dem für *bm* zugehörigen `case`-Fall. Der zweite Schritt unterscheidet zwei Fälle:

- 1. Fall: *A.bm* ist schon gebunden
- 2. Fall: *A.bm* ist noch nicht gebunden

Listing 4.24: Redefinition von A.callAllBindings Fall 2

```
1  public Object _OT$callAllBindings(boundMethodId, Object[]
                 args)
2  {
3        Object res = null;
4        switch(boundMethodId) {
5          case 1:
6          res = _OT$callOrig(boundMethodId, args);
7          break;
8        }
9        return res;
10 }
```

Wenn *A.bm* noch nicht gebunden ist, verschieben wir ihren Code in die `_OT$callOrig()`-Methode von *A* und redefinieren die `_OT\$callAllBindings()` wie in Listing **??** gezeigt. Andernfalls befindet sich der Original-Code von *A.bm* schon dort. Stattdessen verschieben wir den Dispatch-Code von *A.bm* in die `_OT$callAllBindings()`-Methode von *A*, so wie wir es vorher mit *B.bm* getan haben. Für beide Fälle wird nun *A.bm* wie folgt redefiniert:

Listing 4.25: Redefiniton von A.bm() für Fall 1

```
1 public RType bm(AType_1 a_1, ..., AType_n a_n)
2 {
3       return this._OT$callAllBindings(1 /*boundMethodId*/,
                    args);
4 }
```

Mit diesen Transformationen haben wir nun Folgendes erreicht: Bei einem Aufruf von *B.bm* wird die vererbte Methode *bm* in *A* ausgeführt. Durch das dynamische Binden erfolgt jetzt aber der Aufruf der `_OT$callAllBindings()`-Methode von *B* (mit der selben Bound Method ID), der nun den Dispatch-Code für *B.bm* ausführt.

Wird dagegen *A.bm* aufgerufen, wird die `_OT$callAllBindings()`-Methode in *A* ausgeführt, die den Dispatch-Code für *A.bm* enthäht oder im Fall 2 direkt die `_OT$orig()`-Methode von *A* aufruft.

Wir müssen jedoch noch etwas nachbessern: Beim Aufrufen der Original-Methode (in der `callNext()`-Methode), wird die `_OT$callOrig()`-Methode von *B* aufgerufen (dynamisches Binden). Ausgeführt werden muss aber, wegen der Vererbung, *A.bm*, deren Code wiederum nach `A._OT$callOrig()` verschoben worden ist. Um den Aufruf der Original-Methode an die richtige Stelle zu leiten, generieren wir einen `super`-Aufruf als einen `default`-Fall in den `_OT$callOrig`-switch von *B* (Listing 4.26), womit nun der Code von *A.bm* ausgeführt wird.

Listing 4.26: Generische callOrig-Methode mit super-Call

```
1  public Object _OT$callOrig(boundMethodId, Object[] args) {
2
3      Object res = null;
4      switch(boundMethodId) {
5          case 1:
6              res = // Code für boundMethodId = 1
7              break;
8          case 2:
9              res = // Code für boundMethodId = 2
10             break;
11         default:
12             res = super._OT$callOrig(boundMethodId, args);
13             break;
14     }
15     return res
16
17 }
```

## 4.4 Teamregistrierung

Wie schon in den Anforderungen erwähnt, steht die Teamregistrierung, wie sie in der alten Strategie realisiert ist, in Konflikt zu einem dynamischen Weben. Dies soll im Folgenden begründet und eine alternative Vorgehensweise vorgeschlagen werden.

Eine Teamklasse $T$ deklariert eine Menge von Callin-Bindungen. Durch jede Bindung wird eine Menge von Joinpoints definiert, nämlich die der gebundenen Basismethoden und deren Submethoden.

Aufgabe der Teamregistrierung ist die dynamische Aktivierung einer Teaminstanz. In dem bisherigen Anmeldeverfahren erfolgt die Aktivierung dadurch, dass sich eine Instanz $t1$ eines Teams $T$ bei einer Basisklasse $B$ anmeldet. Dafür besitzt $B$ eine Anmeldeschnittstelle, inklusive einer Registrierungsliste $L$, welche sie entweder direkt implementiert oder von einer Superklasse erbt. Für die Klasse $A \sqsupseteq B$, in der $L$ liegt, gelten folgende Bedingungen:

1: Die durch $T$ definierten Joinpoints innerhalb einer Basishierarchie[11] von $B$ liegen in $A$ oder in einer Subklasse von $A$.

2: Es existieren keine weiteren Joinpoints von anderen Teams in einer Superklasse von $A$.

Die Anmeldeliste $L$ von $B$ liegt also in der obersten gebundenen Klasse innerhalb einer Basishierarchie von $B$. Der Grund dafür ist, dass sich ein Team nur bei einer einzigen Anmeldeliste pro Basishierarchie registrieren muss und nicht bei jedem einzelnen Joinpoint (Vererbung der Teamliste).

Team-Registrierung und Dispatch-Code greifen also innerhalb einer Basishierarchie von $B$ auf eine gemeinsame Liste zu. Anzumerken ist, dass sich Teaminstanzen in einer Liste $L$ eintragen können, deren Joinpoint-Mengen disjunkt sind. Für einen Joinpoint $p$ existieren also Instanzen in $L$, deren Callin-Bindungen $p$ nicht als Joinpoint definiert sind. Somit ist $L$ im Allgemeinen für einen bestimmten Joinpoint nicht optimal besetzt. Der Chaining-Wrapper ist so implementiert, dass er die für ihn nicht relevanten Team-Instanzen ignoriert[12] und die Rekursion fortführt.

Die schemaverändernde Generierung von Anmeldeschnittstellen in den Basisklassen ist zur Laufzeit nicht möglich. Dies können wir jedoch dadurch vermeiden, in dem wir die Anmeldeschnittstellen in einen neuen *Team-Manager* auslagern. Der Zugriff auf die jeweiligen Registrierungslisten geschieht über einen Indirektionsschritt mit einem Registrierungsschlüssel.

[11] Eine Klasse $C$ liegt in der Basishierarchie von $B$, wenn gilt: $B \sqsubseteq C$ oder $C \sqsubseteq B$

[12] Es existiert kein entsprechender *case*-Fall.

Das eigentliche Problem, das sich zur Laufzeit ergibt, ist Bedingung 2: Nehmen wir an, es existieren zwei Klassen *B* und *C*, die von einer Klasse *A* erben. Des Weiteren besitzen *B* und *C* jeweils eine Liste, die Bedingung 1 und 2 erfüllen. Wird nun *A* gebunden, müssen die Listen von *B* und *C* in *A* wegen Bedingung 2 zusammengelegt werden. Das Zusammenlegen zweier Listen ist aber nicht möglich, da sonst die Reihenfolge der Aktivierung verloren geht. Dies zwingt uns dazu, die Listen so aufzuteilen, dass dies nicht mehr vorkommen kann. Da wir aber nachträglich jederzeit eine neue Basisklasse binden können, die Superklasse von zwei disjunkten Basishierarchien ist, können wir keine andere Aufteilung finden, als eine globale Liste für alle Joinpoints. Eine globale Liste hätte jedoch Nachteile:

- Für einen Joinpoint ist eine globale Liste im Allgemeinen sehr dünn besetzt. Der Dispatch-Code muss die für ihn relevanten Instanzen herausfiltern. Dadurch verzögert sich der Callin-Dispatch.
- Die Performance des An- und Abmeldens wird durch eine große Liste verschlechtert.

### 4.4.1 Joinpoint-Registrierung

Die oben genannten Nachteile einer globalen Liste können durch eine veränderte Anmeldung der Teams vermieden werden, indem sich die Teaminstanzen direkt bei den von ihnen definierten Joinpoints anmelden, also bei allen Methoden und deren Submethoden, die durch ihre Rollen gebunden sind. Jeder Joinpoint bekommt in dieser Lösung eine exakte Liste von Teams zugewiesen, die von dem Team-Manager verwaltet wird. Über eine *Joinpoint ID* ruft der Dispatch-Code seine individuelle Liste vom Team-Manager ab.

Gegenüber einer globalen Liste haben wir also eine optimal besetzte Joinpoint-Liste. Jedoch hat diese Vorgehensweise Einfluss auf den Aufwand der Team-Registrierung. Nehmen wir an, der Aufwand für die Registrierung bzw. Deregistrierung in einer Liste ist linear, dann beträgt dieser bei einer globalen Liste mit n-Instanzen immer $\Theta(n)$.

Bei einer Joinpoint-Registrierung hängt der Aufwand davon ab, wie viele Joinpoints ein Team definiert und wie viele unterschiedliche Joinpoints insgesamt existieren. Der Aufwand liegt zwischen dem best-case: $1 * \Theta(\frac{n}{j})$, wenn $j$ die Anzahl der Joinpoints ist und jedes Team genau einen, von jedem anderen Team unterschiedlichen Joinpoint besitzt und dem worst-case: $j_T * \Theta(n)$, wenn $j_T$ die Anzahl der Joinpoints für ein Team $T$ ist und alle Teams die gleichen Joinpoints besitzen.

Bei einer zeitkritischen OT/Java-Anwendung, in der sich eine Team-Instanz sehr oft innerhalb einer kurzen Zeit aktiviert und deaktiviert, ist eine performante Umsetzung dieser Prozesse wichtig. Da die Listenoperationen im worst-case sehr umfangreich sein können, ist dies nicht mehr zu garantieren.

Um solchen Anwendungen gerecht zu werden, wäre ein erweitertes Konzept der Team-Aktivierung bzw. Deaktivierung innerhalb von OT/Java denkbar. Diese Erweiterung würde zwischen einer Registrierung und einer Aktivierung bzw. einer Deregistrierung und einer Deaktivierung unterscheiden:

Eine Registrierung bzw. Deregistrierung meint dabei nur das Hinzufügen bzw. Entfernen von Team-Instanzen in die entsprechenden Listen.

Eine Aktivierung bzw. Deaktivierung dagegen meint die Aktivierung bzw. Deaktivierung der Callin-Bindungen für eine registrierte Team-Instanz.

So könnte eine Aktivierung implizit, falls nötig, eine Registrierung mit sich ziehen, aber eine Deaktivierung keine Deregistrierung.

Während des Dispatchvorgangs können sich demnach bei diesem Konzept deaktivierte Team-Instanzen in der abzuarbeitenden Team-Liste befinden. Für eine deaktivierte Instanz müsste die `callAllBindings()`-Methode unmittelbar die `callNext()`-Methode aufrufen:

Listing 4.27: Überspringen von deaktivierten Instanzen

```
1 public final Object callAllBindings(IBase ibase,
2                     int boundMethodId,
3                     Team[] teams,
4                     int idx,
5                     Object[] args) {
6
7     if (!this.isActive) callNext(ibase,boundMethodId,teams,
                    idx,args);
8     ...
9 }
```

Durch diese 2-stufige Vorgehensweise ist es möglich, das aufwändige Registrieren/Deregistrieren mit einer billigen Operation des Aktivierens und Deaktivierens abzufedern.

### 4.4.2 Joinpoint-Registrierung mit einer Callin ID

Um für einen Dispatch die richtigen Callin-Wrapper aufzurufen, muss ein Team bisher jedes mal die entsprechende Callin ID aus der übergebenen Class ID über einen Lookup (z.B. aus einer `HashMap`) ermitteln. Dies verzögert die Ausführungszeit eines Callin-Dispatches. Mit der Realisierung einer Joinpoint-Registrierung ist es möglich, den Lookup einmalig bei der Registrierung zu berechnen und zusammen mit der Team-Instanz abzuspeichern.

Dafür weisen wir einem Joinpoint eine weitere Liste zu, in der bei Registrierung einer Instanz die korrespondierende Callin ID vermerkt wird.

Nehmen wir das Beispiel 4.2 mit den Team-Instanzen $t1$ der Klasse $T1$ und $t2$ der Klasse $T2$, so ergeben sich folgende Joinpoint-Listen:

| Methode | Joinpoint ID | Team-instanzen | Callin IDs |
|---|---|---|---|
| $B1.bm$ | 1 | $t1$ | $f_{T1}(B1) = 1$ |
| $B2.bm$ | 2 | $t1$ | $f_{T1}(B2) = 1$ |
| $B3.bm$ | 3 | $t1$ | $f_{T1}(B3) = 2$ |
| $B4.bm$ | 4 | $t2, t1$ | $f_{T2}(B4) = 1,\ f_{T1}(B4) = 2$ |
| $B5.bm$ | 5 | $t2, t1$ | $f_{T2}(B5) = 2,\ f_{T1}(B5) = 2$ |

Die Liste der Callin IDs wird der `callAllBindings()`-Methode mit übergeben. Die jeweiligen Teams können aus dieser Liste die Callin ID ermitteln.

Listing 4.28: Generischer Dispatch in der Basismethode

```
1  RType bm(AType_1 a_1, ..., AType_n a_n) {
2
3      Object res;
4
5      int jpID = 1; /*Joinpoint ID*/
6
7      Team[] teams = TeamManager.getTeams(jpID);
8      int[] callinIds = TeamManager.getCallinIds(jpID);
9
10     res = teams[0].callAllBindings(this,
11                         1,          /*Bound Method ID*/
12                         team,       /*Team-Array*/
13                         callinIds,  /*Callin-Array*/
14                         0,          /*Iterator*/
15                         args        /*Boxed Args*/);
16
17     return ... /unbox res;
18 }
```

## 4.5 Zugriff auf private Klassenelemente zur Laufzeit

Object Teams erlaubt sowohl für Callout-Bindungen das Zugreifen auf private Klassenelemente von Basisklassen, als auch für den base-call das Aufrufen von privaten Methoden, die durch einen Callin gebunden sind. Die bisherige Webestrategie durchbricht die Kapselung, in dem sie die Sichtbarkeit der betroffenen Felder zur Ladezeit erweitert. Beim dynamischen Weben wissen wir jedoch nicht im Voraus, welche Felder und Methoden dies sind und wir haben auch nicht die Möglichkeit, nachträglich Änderungen an der Sichtbarkeit vorzunehmen.

Da wir den Zugriff auf private Klassenelemente über die Java-Reflection-API aus schon erwähnten Gründen nicht realisieren wollen, soll die Methode `IBASE._OT$access()` diese Zugriffe direkt ermöglichen.

Die `_OT$access()`-Methode (Listing 4.29) bekommt eine *Access ID* als Parameter, die ein bestimmtes Element (Attribut oder Methode) der Klasse kennzeichnet. Soll ein Zugriff auf ein Attribut erfolgen, gibt der Parameter `op_kind` an, ob das Feld gelesen oder gesetzt werden soll. Im letzteren Fall befindet sich das geboxte Argument im ersten Index des `Object`-Arrays. Beim Zugriff auf eine Methode befinden sich im Array die geboxten Argumente des Aufrufs.

Listing 4.29: access-Methode für Decapsulation

```
1  public Object _OT$access(int accessId,int op_kind,Object[]
                 args) {

3    Object res = null;

5    switch(accessId) {
6    case 1:
7       switch(operation) {
8       case(GET)
9         res = new Integer(this.x);
10        break;
11      case(SET)
12        this.x = ((Integer)args[0]).toInt();
13        break;
14      default:
15        break;
16      }
17   case 2:
18      // unbox Arguments
19      int r = this.bm(...);
20      res = // box Result
21      break;
22   case 3:
23   ...
24   }

26  return res
```

## 4.6 Überprüfung der Anforderungen

Die am Anfang des Kapitels gestellten Anforderungen an einen dynamischen Weber sollen nun anhand der neuen Webestrategie überprüft werden.

Ein Teil der Anforderungen wurde durch eine Präparations-Phase gelöst, in der das Interface `IBASE` den Klassen hinzugefügt wurde. Dies löste folgende Probleme:

- **Anforderung 1a: Basis-Dispatch-Konflikt**
  **Lösung**: Verschieben des Original-Codes in die `_OT$callOrig()`-Methode
- **Anforderung 1b: Aspektbindung bei vererbter Methode**
  **Lösung**: Dynamisches Binden der `_OT$callAllBindings()`-Methode
- **Anforderung 4: Decapsulation**
  **Lösung:** Zugriff auf private Klassenelemente durch die `_OT$access()`-Methode

**Anforderung 3: Teamregistrierung**
Wir haben gesehen, dass eine dynamische Webestrategie nicht von festgelegten Basishierarchien ausgehen kann. Dies macht die Aufteilung der Listen anhand dieser Hierarchien nicht möglich. Um die alte Team-Registrierung beizubehalten, wäre die Konsequenz, eine globale List zu benutzen. Mit der vorgeschlagenen 2-stufigen Registrierung (Anmelden und Aktivieren) wäre eine Joinpoint Registrierung eine gute Alternative, da zum einen exakte Listen existieren und zum anderen die Callin ID eines Joinpoints für ein Team nur einmalig berechnet werden müsste.

Der neue Dispatch-Mechanismus, bestehend aus der einheitlichen Team-Schnittstelle und dem generischen Dispatch-Code ermöglicht eine Abarbeitung der Teams, die ohne einen Chaining-Wrapper auskommt. Zudem vermeidet der generische Dispatch-Code, dass eine gebundene Methode mehrmals gewoben wird. Dies ist hinsichtlich eines dynamischen Webens eine wichtige Eigenschaft.

Ein Kritikpunkt, dem wir uns jedoch stellen müssen, ist die Notwendigkeit des Verpackens der Aufrufargumente in ein `Objekt`-Array. Dies sind teure Operationen, die zwar in der alten Strategie auch angewandt wurden, jedoch im Allgemeinen in einem kleineren Umfang.

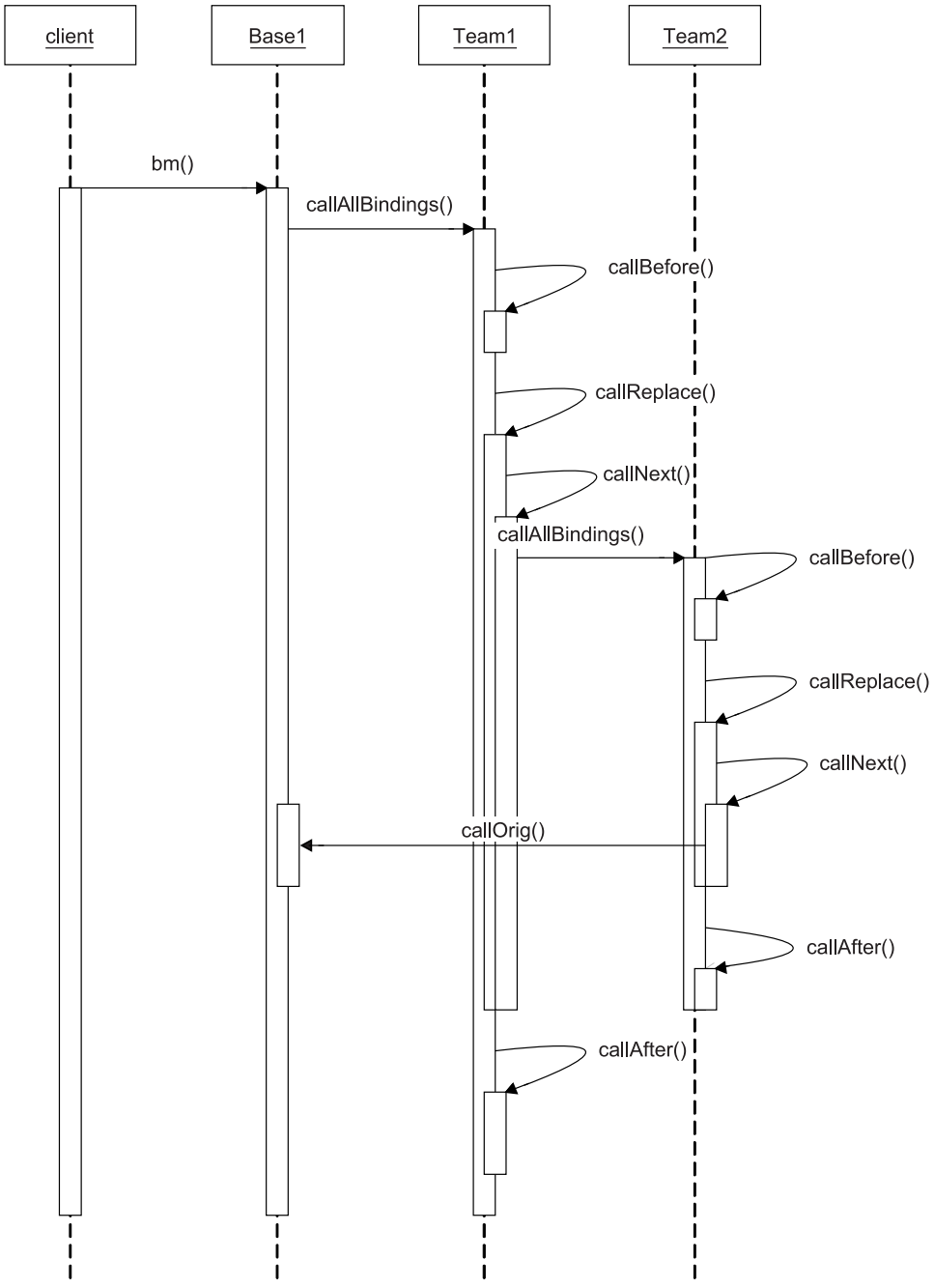


Abbildung 4.3: Ablauf eines generischen Callin-Dispatchings

# Kapitel 5

# Realisierung des Webers

Nachdem im Kapitel 4 die neue Webestrategie definiert wurde, wird nun der Blick auf die Anforderungen gerichtet, die ein dynamischer Weber für ObjectTeams/Java stellt. Wir unterscheiden zunächst zwei notwendige Zeitpunkte, an denen Basisklassen gewebt werden müssen, um dann die jeweils notwendigen Anforderungen zu bestimmen.

## 5.1 Webezeitpunkte

Der Weber muss sowohl zur Ladezeit, als auch zur Laufzeit die Basisklassen transformieren:

- **Weben zur Ladezeit:** Wird eine Basisklasse *B* geladen, muss geprüft werden, ob eine Bindung für *B* existiert oder ob *B* eine gebundene Methode aus einer ihrer Superklassen überschreibt (Aspektvererbung). In beiden Fällen muss *B* entsprechend transformiert werden.
- **Weben zur Laufzeit:** Wie im Kapitel 4 beschrieben, wird die Team-Registrierung nicht mehr direkt an den Basisklassen vorgenommen, sondern erfolgt über einen Team-Manager. Dieser muss bei der Registrierung einer Team-Instanz prüfen, ob die entsprechenden Bindungen für die Basisklassen schon gewoben sind. Ist dies nicht der Fall, muss er den Weber damit beauftragen.

Die Anforderungen, die bestehen, um zur Ladezeit Basisklassen zu transformieren sind:

- Abfangen des Bytecodes einer Klasse zur Ladezeit
- Die Transformierbarkeit des Bytecodes

Diese Voraussetzungen sind, wie in Kapitel 3 im Abschnitt 3.2.3 beschrieben, durch einen registrierten `ClassFileTransformer` gegeben. Die Methode `transform` wird mit dem `byte`-Array des Classfiles einer Klasse aufgerufen, die durch einen Classloader neu definiert wird. Innerhalb dieser Methode ist die Transformierbarkeit der Klassendefinition gegeben.

Die Notwendigkeit, eine bestimmte Klasse zu transformieren, hängt davon ab, ob für sie eine direkte Callin-Bindung oder eine indirekte (vererbte) Bindung besteht. Die Callin-Bindungsinformationen, die aus den Classfile-Attributen der Teamklassen gelesen und in dem `CallinBindingManager` verwaltet werden, enthalten nur die Informationen der direkten Bindungen von Basismethoden. Um die Existenz einer indirekten Bindung für eine Klasse festzustellen, muss überprüft werden, ob für ihre Superklassen direkte Bindungen bestehen. Daher brauchen wir zur Ladezeit einer Klasse die Namen ihrer Superklassen (*Superclass-Lookup*).

Die Anforderungen, die zur Laufzeit bestehen, unterscheiden sich darin, dass neben einem Superclass-Lookup, ein *Subclass-Lookup* benötigt wird. Immer dann, wenn eine Basismethode gewoben werden soll, muss der Weber wissen, welche Subklassen diese Methode überschreiben, um auch dort einen entsprechenden Dispatch-Code zu generieren. Zusätzlich stellt sich hier die Frage, woher wir den Bytecode der zu redefinierenden Klassen bekommen. Die Java Virtuell Machine und insbesondere das Instrumentation Interface bietet keine Schnittstelle, um den Bytecode von geladenen Klassen zu erhalten.

Im Folgenden wird eine Realisierung eines *Vererbungs-Lookups* für Klassen geschildert und eine Möglichkeit aufgezeigt, wie dem dynamischen Weber zur Laufzeit der benötigte Bytecode zur Verfügung gestellt werden kann.

## 5.2 Vererbungs-Lookup

Ziel ist es, für den Weber eine Datenstruktur (*Inheritance Manager*) bereitzustellen, die zu einem gegebenen Klassennamen die Namen der Superklassen bzw. die der Subklassen liefert. Zusätzlich müssen auch die implementierten Methoden dieser Klassen abfragbar sein, um bestimmen zu können, ob eine Aspektvererbung zwischen zwei Klassen vorliegt oder nicht.

Da für eine Anwendung sehr viele Klassen geladen sein können, sollte die dafür vorgesehene Datenstruktur speichereffizient sein und zudem einen performanten Lookup ermöglichen. Wir können uns des weiteren die Frage stellen, ob es möglich ist, eine minimierte Klassenhierarchie zu notieren, die nur die Klassen enthält, die Methoden überschreiben. Natürlich muss auch eine Vollständigkeit der Datenbasis bei jedem einzelnen Webevorgang gegeben sein, damit alle benötigten Klassen redefiniert werden.

### Inhalt der Datenstruktur

Die Klasse `java.lang.Class` ist eine Repräsentation einer von der JVM geladenen Klasse. Sie enthält Meta-Informationen, wie z.B. eine Referenz zu ihrer Superklasse und die Namen und Signaturen ihrer Methoden. Es bietet sich daher an, diese schon vorhandenen Informationen der JVM zu referenzieren, um der Anforderung einer speichereffizienten Datenstruktur nachzukommen.

### Aufbau der Datenstruktur

Da die Anzahl der geladenen Klassen nicht konstant ist, müssen wir uns überlegen, wie unsere Datenstruktur während der Laufzeit aktuell bleibt. Dabei existieren prinzipiell zwei Möglichkeiten: Entweder erfolgt eine Aktualisierung der Datenbasis unmittelbar beim Laden einer neuen Klasse, oder wir aktualisieren die Datenbasis vor jedem einzelnen Webevorgang. Wir betrachten zunächst eine Realisierung für den letzteren Fall.

Eine direkte Möglichkeit, `Class`-Instanzen zur Laufzeit zu erhalten, bietet das JPLIS Instrumentation-Interface über die Methode `Instrumentation.getAllLoadedClasses()`. Sie liefert ein Array der gegenwärtig geladenen Klassen der JVM. Aus diesem Array ließe sich vor dem ersten Webeprozess eine Vererbungshierarchie ableiten, da jede `Class`-Instanz eine Referenz auf ihre Superklasse besitzt. Der Nachteil dieser Vorgehensweise ist, dass bei jedem weiteren Webevorgang ein Abgleich erfolgen muss, welche Klassen seit dem letzten Webevorgang neu geladen wurden und welche schon in der Datenstruktur notiert sind. Existieren sehr viele geladene Klassen, ist dies ein sehr aufwändiger Prozess, der bei jedem Webevorgang wiederholt werden muss. Daher wird nun eine Überlegung angestellt, wie die Datenbasis zur Ladezeit aktualisiert werden kann, um dieses Problem zu vermeiden.

Eine alternative Vorgehensweise ist das Abfangen der vom Classloader erzeugten `Class`-Instanzen zur Ladezeit und das unmittelbare Speichern der benötigten Informationen in unserer Datenstruktur. Um dies zu realisieren, müssen wir uns einen geeigneten Punkt suchen, an dem alle `Class`-Instanzen abfangbar sind, unabhängig von welchem Classloader sie erzeugt werden[1].

Die `Classloader.defineClass()`-Methode erzeugt aus einem übergebenen `byte`-Array eine `Class`-Instanz. Da die Methode `final`-deklariert ist, kann sie von keinem anderen Classloader überschrieben werden und käme daher als geeigneter Punkt in Frage[2]. Durch eine einfache Redefinition dieser Methode können wir die erzeugten `Class`-Instanzen abfangen. Die Redefintion besteht darin, einen Aufruf zu generieren (`InheritanceManager.addClass()`), der dem Inheritance Manager die erzeugte `Class`-Instanz übergibt, bevor sie als Resultat der `define()`-Methode zurückgeliefert wird. (siehe Listing 5.1). Der dadurch realisierte Kontrollfluss ist im Sequenzdiagramm 5.1 dargestellt.

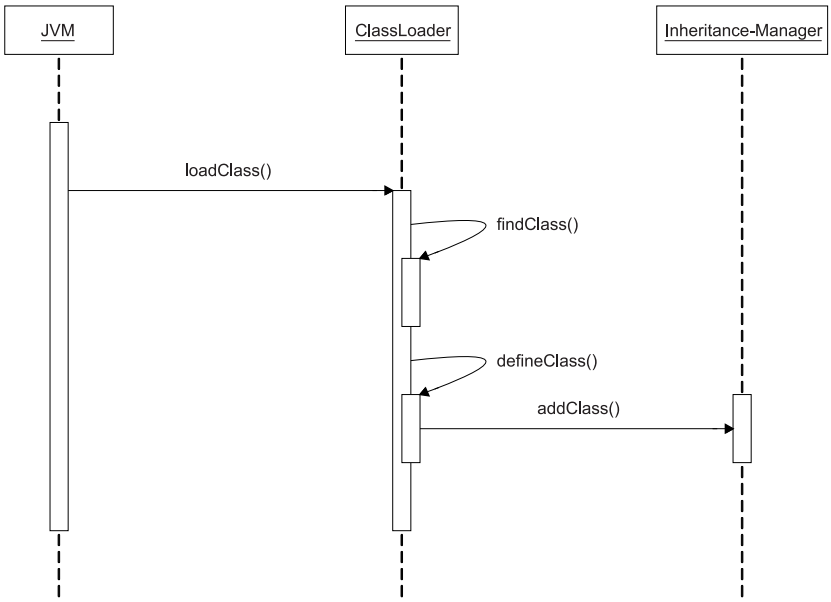


Abbildung 5.1: Abfangen der Class-Instanzen

[1] Innerhalb der *transform*-Methode des *ClassFileTransformers* haben wir zur Ladezeit zwar den Zugriff auf den Bytecode des Classfiles, jedoch existiert noch keine *Class*-Instanz der Bytecoderepräsentation.

[2] Ausgenommen von dieser Vorgehensweise sind die Klassen, die vom *Bootstrap-Classloader* geladen werden. Dies sind die Kern-Klassen der Java-API (Klassen aus dem Package *java.lang*).

Listing 5.1: Modifizierte define-Methode

```
1 protected final Class<?> defineClass(String name,byte[] b,
                int off,int len)
2 {
3     Class clazz = null;
4     ...
5     InheritanceManager.addClass(clazz);
6     return clazz;
7 }
```

Der `InheritanceManager` verwaltet eine `Hash`-Datenstruktur, die für einen gegebenen Schlüssel eine Liste von Subklassen liefert. Wird die `addMethode()` aufgerufen, kann die Superklasse aus der übergebenen Class-Instanz über die Methode `Class.getSuperClass()` ermittelt werden. Die Superklasse dient als Schlüssel für die Liste, in der nun die Class-Instanz als die Subklasse gespeichert wird. Das Listing 5.2 zeigt eine beispielhafte Implementierung dieser Vorgehensweise.

Listing 5.2: Speichern der Class-Instanz

```
1  public void Class addClass(Class clazz) {

3    Class superClazz = clazz.getSuperClass();
4    if (inheritanceTree.Contains(superClazz.getName()) {
5      (Vector)(inheritanceTree.get(superclazz)).add(clazz);
6    } else {
7      Vector subclazzes = new Vector();
8      subclazzes.add(clazz);
9      inheritanceTree.put(clazz.getName(),subclazzes);
10   }
11 }
```

Für die Realisierung eines minimierten Vererbungsbaums könnte vor dem Speichern der Klasse überprüft werden, ob die vom Classloader abgefangene Klasse eine Methode in einer Superklasse überschreibt. Dafür müssen wir für jede Methode, die von der Klasse implementiert wird (`Class.getMethods()`), im Vererbungsbaum schrittweise über die Methode `Class.getSuperClass()` aufsteigen. Finden wir keine Methode in der Superklasse, die unsere abgefangene Klasse implementiert, brauchen wir sie auch nicht zu speichern. Wenn man davon ausgeht, dass ein Überschreiben von Methoden relativ selten ist, können wir unsere Datenbasis erheblich einschränken. Dies würde einem performanten Subklass-Lookup entgegen kommen. Jedoch müssen wir durch den Abgleich im Vererbungsbaum eine gewisse Verzögerung der Ladezeit einer Klasse in Kauf nehmen.

## 5.3 Bytecode-Verwaltung

Die Verfügbarkeit des Bytecodes zur Ladezeit wurde in der alten Webestrategie durch das JMangler-Framework realisiert. Das Instrumentation Interface bietet uns dagegen, wie schon erwähnt, keine Schnittstelle, um zur Laufzeit auf den Bytecode einer geladenen Klassen zuzugreifen. Wir müssen dem Weber also eine eigene Infrastruktur anbieten, die genau dies umsetzt.

Für Klassen, die mit dem System-Classloader geladen werden exisitiert die Möglichkeit über die Methode `ClassLoader.getSystemResourceAsStream()` ein im Classpath befindliches Classfile einzulesen, so dass bei Anwendungen, die keinen speziellen Classloader benutzen, die Verfügbarkeit des Bytecodes sichergestellt ist.

Für Klassen die von einem anwendungsspezifischen Classloader geladen werden, ist diese Funktionalität jedoch nicht garantiert. Hier ist die Frage zu beantworten, wie man deren Bytecode erhält.

Eine Lösung dafür wäre es, den Bytecode innerhalb der `ClassFileTransFormer.transform()`-Methode für spezielle Classloader beim Laden der Klasse in ein dafür vorgesehenes temporäres Verzeichnis zu speichern bzw. zu verwalten, um somit einen Zugriff der Classfiles zur Laufzeit zu ermöglichen.

Da der Zugriff auf den sekundären Speicher relativ aufwändige Lese-Operationen bedingt, ist dies im Hinblick auf ein performantes Laufzeit-Weben eine unbefriedigende Lösung. Daher bietet es sich zumindest an, einen Cache für den transformierten Bytecode bereit zu halten, um ein inkrementelles Weben performant zu gestalten.

## 5.4 Präparieren der Klassen

Die Bedingung, die wir uns für ein 2-Phasen Weben gestellt haben, ist ein performantes Präparieren der Klassen zur Ladezeit. In der alten Webestrategie wird jede Klasse, die geladen wird, in eine sogenannte `ClassGen`-Repräsentation überführt. Die `ClassGen`-Klasse ist eine zentrale Datenstruktur innerhalb der BCEL-API, um das Transformieren und das Generieren neuer Klassen zu ermöglichen. Das Erzeugen von `ClassGen`-Objekten, ausgehend von dem `byte`-Array des Classfiles ist jedoch sehr aufwändig, da der Initialisierungsprozess die einzelnen Strukturbestandteile des Classfiles auswertet und auf umfangreiche BCEL-Datenstrukturen abbildet. Die Benutzung von BCEL ist daher sehr schwergewichtig für eine für alle Klassen einheitliche Präparationsphase. Stattdessen soll eine Transformation der Klassen vorgezogen werden, die direkt auf dem `byte`-Array eines Classfiles operiert, um die Verzögerung zur Ladezeit zu minimieren.

Wie in Kapitel 4 beschrieben, werden die Klassen mit dem Interface `IBASE` präpariert. Dafür müssen dem Classfile folgende Elemente hinzugefügt werden:

- Deklarierung des Interfaces `IBase`
- leere Implementierung der `IBASE._OT$callOrig()`-Methode
- leere Implementierung der `IBASE._OT$callAllBindings()`-Methode
- leere Implementierung der `IBASE._OT$access()`-Methode

Da diese Elemente, insbesondere die leeren Methoden, für alle Klassen die gleiche Bytecode-Repräsentation besitzen, besteht die Aufgabe des Präparierens nur darin, die jeweils festgelegten `byte`-Arrays in das `byte`-Array des Classfiles zu platzieren. Somit beschränkt sich der Aufwand nur auf ein Kopieren des ursprünglichen Classfiles in ein neues (größeres) `byte`-Array, in dem zusätzlich die neuen Classfile-Elemente hinzugefügt werden. Diese Operationen stellen bezüglich des Gesamtaufwandes eine Klasse durch den Classloader zu Laden, nur einen relativ kleinen Mehraufwand dar.

## 5.5 Modell eines Webers

Im Folgenden wird ein Modell des Laufzeitwebers skizziert, das die Interaktion der beteiligten Komponenten veranschaulichen soll. Die Abbildung 5.2 zeigt dafür die Abläufe während des Webens zur Ladezeit (blaue Pfeile) und die zur Laufzeit (rote Pfeile). Die einzelnen Schritte werden nun erklärt:

Ablauf zur Ladezeit:

1. Der Classloader wird beauftragt, eine neue Klasse zu definieren (`ClassLoader.defineClass()`).
2. Die `ClassFileTransformer.transform()`-Methode wird mit dem Namen der Klasse, dem `byte`-Array des Classfiles und der Instanz des Classloaders aufgerufen, der für den Ladeprozess verantwortlich ist.
3. Für jede Klasse wird die Präparierungs-Transformation durchgeführt (`BaseClassTransformer`).
4. Wird die Klasse von einem anwendungsspezifischen Classloader geladen, wird das Classfile in ein temporäres Verzeichnis gespeichert, um die Verfügbarkeit des Classfiles zur Laufzeit zu sichern.
5. Der Bytecode und der Name der Klasse wird an den Weber weitergereicht.

6. Der Weber überprüft anhand des Klassennamens ob die Klasse direkte Bindungen besitzt. Zum Feststellen der indirekten Bindungen wird ein Superclass-Lookup durchgeführt. Sind Bindungen vorhanden wird der Bytecode transformiert.

7. Wurde eine Klasse gewoben, wird der Bytecode im Cache vom Bytecode-Manager gespeichert.

8. Die neu erzeugte `Class`-Instanz wird in dem Inheritance-Manager gespeichert.

Ablauf zur Laufzeit:

1. Eine Team-Instanz registriert sich beim Team-Manager mit dem Namen ihrer Teamklasse.

2. Der Team-Manager ruft den Weber mit dem Namen der Teamklasse auf.

3. Der Weber überprüft, ob der Dispatch-Code für die deklarierten Callin-Bindungen der Teamklasse schon gewoben sind. Ist dies nicht der Fall, werden die Klassen bestimmt, die für diese Bindungen gewebt werden müssen. Zusätzlich hält der Vererbungs-Manager die zur Redefinition benötigten `Class`-Instanzen bereit.

4. Der Weber fordert den Bytecode der zu transformierenden Basisklassen am Bytecode-Manager an.

5. Falls der Bytecode sich nicht im Cache des Bytecode-Managers befindet, werden die Classfiles im Classpath oder in dem temporären Verzeichnis gesucht und gelesen.

6. Der Weber ruft mit dem Bytecode und den `Class`-Instanzen die `redefineClasses()`-Methode auf.

7. Die `ClassFileTransformer.transform()` wird mit dem Bytecode und den zu redefinierenden Klassen aufgerufen.

8. Die Klassen werden durch den Laufzeit-Weber gewoben.

9. Der Bytecode der redefinierten Klassen wird im Cache des Bytecode-Managers gespeichert.

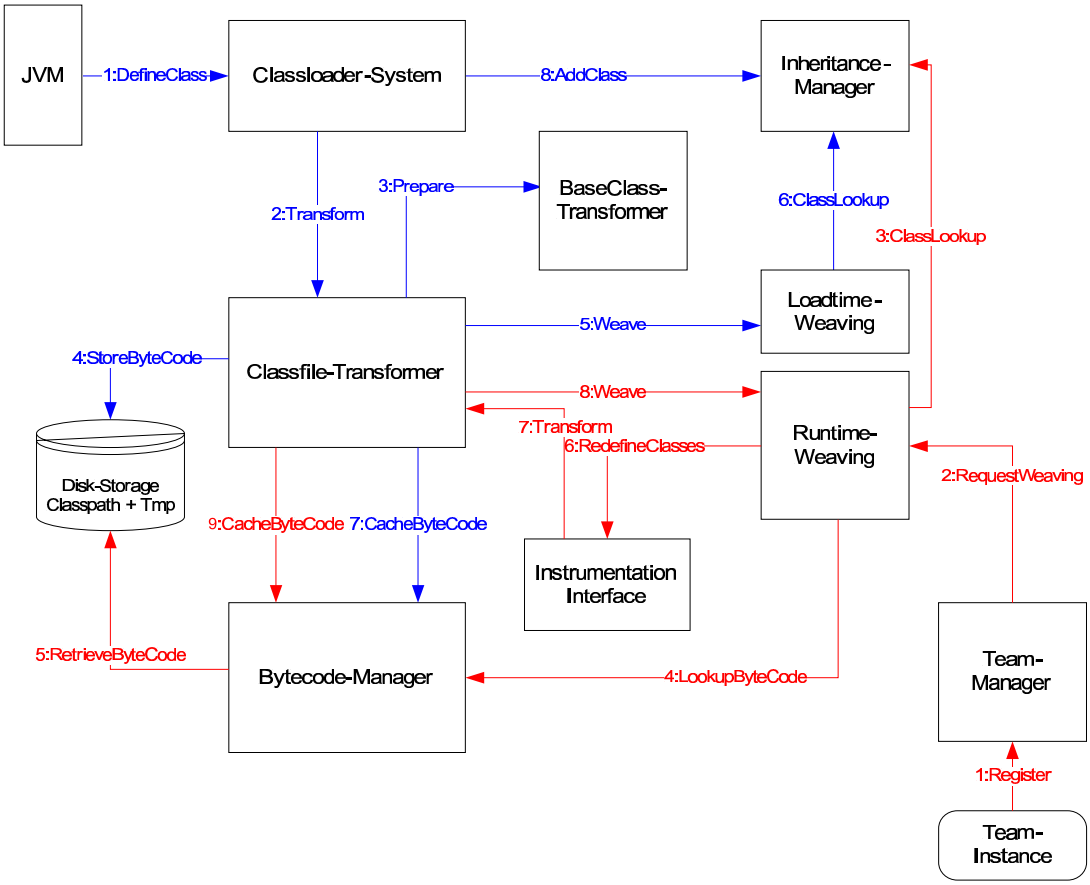


Abbildung 5.2: Interaktionen während des Webens

# Kapitel 6

# Zusammenfassung und Ausblick

## 6.1 Zusammenfassung

Die Motivation am Anfang dieser Arbeit war es, ein ”striktes” schemaerhaltendes Weben zu realisieren, das ohne eine Präparierungs-Phase auskommt. Dabei stellte sich jedoch heraus, dass die Konsequenzen der Redefinitionsbeschränkungen so schwerwiegend sind, dass eine Umsetzung einer schemaerhaltenden Webestrategie nicht mit den Konzepten von Object Teams und der Anforderung nach einer performanten Aspektbindung in Einklang zu bringen ist. Insbesondere die Aspektbindung an vererbten Methoden und das Decapsulation Problem lassen sich nur bedingt und in einem nicht zufriedenstellenden Maße lösen.

Daher wurden die Beschränkungen dadurch aufgeweicht, dass die Klassen zur Ladezeit mit einer festgelegten Schnittstelle präpariert werden. Dies kann zur Ladezeit schnell durchgeführt werden, so dass die daraus resultierende Verzögerung in den meisten Fällen vernachlässigbar ist.

Eine weitere Einschränkung des Instrumentation Interfaces bezüglich eines Laufzeitwebens, ist eine fehlende Schnittstelle um den Bytecode der geladenen Klassen abzurufen. Der Weber muss daher annehmen, dass der Bytecode in Form von Classfiles im Dateisystem zur Verfügung steht. Mit der Anforderung nach einem Lookup in dem Vererbungsbaum der geladenen Klassen, stehen wir einem ähnlichen Problem gegenüber. Auch hier müssen wir eine zusätzliche Datenbasis verwalten, die dem Weber die erforderlichen Informationen bereitstellt.

Weitestgehend unabhängig von den Redefinitionsbeschränkungen war es, eine neue Webestrategie zu finden, die mit möglichst wenigen Bytecode-Transformationen zur Laufzeit auskommt. Dafür wurde die Komplexität des Dispatchings der alten Webestrategie durch einen generischen Mechanismus ersetzt.

Eine Implementierung der neuen Transformationen liegt vor. Diese setzen sich aus der Präparierungs-Phase der Klassen und der Generierung des neuen Dispatch-

Codes für die Basisklassen zusammen. Die neuen Transformer konnten jedoch nur anhand von Simulationen getestet werden, da die notwendige Unterstützung des neuen Dispatchings, seitens des OT/Java-Compilers noch nicht existiert (siehe Abschnitt Offene Punkte).

Der Vererbungs-Lookup mit Hilfe eines modifizierten Classloader, wie in Kapitel 5 beschrieben, konnte anhand eines Prototypen in der Simulation mit einbezogen werden.

### 6.1.1 Offene Punkte

Um eine vollständige Integration der neuen Webestrategie zu realisieren müssen noch folgende Punkte erfüllt werden.

- Erweiterung der Klasse `Team` mit der neuen Schnittstelle `callAllBindings()`, sowie der `callNext()` und der Default-Implementierung der `callReplace()`-Methode (siehe Kapitel 4, Abschnitt 4.3.2)
- Generierung des Dispatch-Codes in den Hook-Methoden `callBefore()`, `callReplace()` und `callAfter()` durch den Compiler (Kapitel 4, Abschnitt 4.3.3)
- Die durch den Compiler erzeugten Hook-Methoden müssen von der Laufzeitumgebung zur Ladezeit angepasst werden. Die Anpassung besteht darin, die vom Compiler erzeugten lokal eindeutigen Bound Method IDs, gegen global eindeutige auszutauschen, wie in Kapitel 4, Abschnitt 4.3.3 erwähnt.
- Realisierung der neuen Teamregistrierung über einen Team-Manager

Der Zugriff auf den Bytecode zur Laufzeit wurde in der Simulation über die BCEL-Klasse `Repository` umgesetzt. Die Klasse realisiert weitestgehend den im Kapitel 5 beschriebenen Bytecode-Manager, jedoch beschränkt sie sich nur auf die Klassen, die sich im Classpath befinden. Das Hinzufügen eines zusätzlichen Verzeichnisses zum Classpath könnte jedoch vermutlich ausreichen, um eine Lösung mit temporären Verzeichnissen für die Verfügbarkeit des Bytecodes von speziellen Classloadern zu realisieren.

OT/Java erlaubt auch Callin-Bindungen an statischen Basismethoden. Dies wurde in Kapitel 4 noch nicht berücksichtigt. Eine Lösung dafür könnte es sein, die Klassen mit einer weiteren statischen Methode zu präparieren, um den Code von statischen Methoden zu verschieben und aufrufbar zu machen. Die `_OT$access()`-Methode, könnte stattdessen statisch implementiert werden. Dafür müsste der Methode bei einem Zugriff auf nicht statische Klassenelemente zusätzlich noch die Referenz des entsprechenden Basisobjekts mitgeliefert werden.

Im Rahmen dieser Diplomarbeit wurden die einzelnen Lösungsvorschläge nicht anhand von Performance-Messungen bewertet. Insbesondere fehlt ein Vergleich zwischen der alten und der neue Webestrategie bezüglich der Aspektbindungs-Performance, also der Ausführungszeit eines Dispatches.

### 6.1.2 Ausblick

#### Joinpoint-Weaving

Object Teams unterstützt bisher nur den execution-point als Joinpoint in Form der Methodenbindungen durch Callins. Die neue Webestrategie ist jedoch so ausgelegt, dass Aspektbindungen für weitere Joinpoints realisierbar sind. Prinzipiell ist es möglich beliebige Bytecode-Instruktionen eines Joinpoints einer Methode zu verschieben und stattdessen einen Dispatch-Code zu erzeugen. Durch die einheitliche Schnittstelle können die zusätzlich benötigten Kontextparameter des Joinpoints transportiert werden. Auch innerhalb eines schon verschobenen Code-Fragments in die `_OT$callOrig()`-Methode könnte ein Dispatch-Code für weitere Joinpoints generiert werden.

#### AOP Unterstützung in der JVM

Aspektorientierte Programmierung ist dabei sich in der Praxis betrieblicher Software-Entwicklung zu etablieren. Wird dieser Trend fortgeführt, ist zu erwarten, dass auch der Bedarf an einem dynamischen AOP in Zukunft weiter zunehmen wird. Ist dies der Fall, wird es wohl auch nur eine Frage der Zeit sein, bis die *Sun JVM* als die maßgebende Implementierung und meist verwendete Plattform für die Programmiersprache Java, sich dieser Entwicklung öffnet. Dies könnte bedeuten, dass die Redefinitionsbeschränkungen des Instrumentation Interfaces aufgehoben werden, um ein Präparieren der Klassen zur Ladezeit zu vermeiden. Des Weiteren könnte die Verfügbarkeit des Bytecodes geladener Klassen direkt von der JVM gewährleistet werden. Andere Implementierungen der Virtuellen Machine sind dem Trend der aspekt-orientierten Programmierung schon gefolgt. Steamloom [6] ist eine Erweiterung der Virtuellen Maschine, die das AOP-Paradigma explizit unterstützt und dabei auch das dynamische Weben von Aspekten berücksichtigt.

# Literaturverzeichnis


[1] developerWorks - IBM's resource for developersobject - java programming dynamics : http://www-128.ibm.com/developerworks/java/library/j-dyn0603/.

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

[3] S. Chiba, Y. Sato und M. Tatsubori. Using hotswap for implementing dynamic aop systems, 2003.

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

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

[6] Michael Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. Dissertation, Technische Universität Darmstadt, 2005.

[7] S. Herrmann. Object teams: Improving modularity for crosscutting collaborations, 2002.

[8] Erik Hilsdale und Jim Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, Seiten 26–35. ACM Press, New York, NY, USA, 2004. ISBN 1-58113-842-3.

[9] Christine Hundt. *Bytecode-Transformationen zur Laufzeitunterstützung von aspekt-orientierter Modularisierung mit Object-Teams/Java*. Diplomarbeit, Technische Universität Berlin, 2003.

[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm und William G. Griswold. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, Seiten 327–353. Springer-Verlag, London, UK, 2001. ISBN 3-540-42206-4.

[11] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier und John Irwin. Aspect-oriented programming. In Mehmet Akşit und Satoshi Matsuoka (Herausgeber), *Proceedings European Conference on Object-Oriented Programming*, Band 1241, Seiten 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

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

[13] A. Popovici, T. Gross und G. Alonso. Dynamic weaving for aspect oriented programming, 2002.

[14] A. Vasseur. Dynamic AOP and Runtime-Weaving for Java - How does Aspectwerkz Address it? In *In AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*. 2004.