Praxistaugliches dynamisches Aspektweben für ObjectTeams/Java im Kontext des OSGi Komponentenframeworks

Diplomarbeit bei
Prof. Dr.-Ing. Stefan Jähnichen

vorgelegt von
Oliver Frank
Matr. Nr. 211856
Fachgebiet Softwaretechnik
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin

Betreuung durch
Dr.-Ing. Stephan Herrmann

Berlin, 19.06.2009

Eidesstattliche Versicherung

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, den 19.06.2009

Inhaltsverzeichnis

Einleitung

Seit es Software gibt[1], wurden nach immer neuen Techniken gesucht, mit denen Software bestmöglich entwickelt werden kann. Vielleicht das wichtigste Kriterium für die Bewertung dieser Techniken war dabei immer die Frage, wie gut sich Software in *Module* aufteilen, also *modularisieren*, lässt. Durch jede neue Technik hat sich auch der Begriff *Modul* bzw. *Modularisierung* verändert. Allgemein gesagt, beschreibt er eine Einheit, in der Software gleicher oder ähnlicher Funktionalität gekapselt ist bzw. die Aufteilung von Software in solche Einheiten.

Die momentan wohl verbreitetste Art der *Modularisierung* stellt die *Objektorientierung* dar. In der *objektorientierten Programmierung* (*OOP*) stellen Klassen bzw. Objekte die elementaren[2] Module dar. Trotz ihrer hohen Verbreitung sollte auch der *OOP* nicht blind vertraut werden. Es lohnt sich also zu hinterfragen, ob die *objektorientierte Programmierung* wirklich die bestmögliche Art der *Modularisierung* ermöglicht. Dazu ist es zunächst nötig, den Begriff *Modularisierung* genauer zu definieren.

Bertrand Meyer beschrieb die Modularisierung 1988 durch fünf Kriterien:

- *Decomposability*
- *Composability*
- *Understandability*
- *Continuity*
- *Protection*

Hier soll eines dieser Kriterien besonders hervorgehoben werden, die *Kontinuität* (*Continuity*). Bertrand Meyer beschrieb dieses Kriterium folgendermaßen:

[1] Der Begriff *Software* wurde 1958 von John Wilder Tukey geprägt.
[2] In vielen objektorientierten Sprachen, lassen sich Klassen und Objekte zu größeren Modulen verbinden.

> *A method satisfies Modular Continuity if, in the software architectures that it yields, a small change in a problem specification will trigger a change of just one module, or a small number of modules.*
>
> *Quelle: [Mey97]*

Um das Kriterium *Kontinuität* zu erfüllen, muss eine Softwarearchitektur also so beschaffen sein, dass eine kleine Änderung der Spezifikation eines Softwaresystems in einem oder zumindest wenigen Modulen umgesetzt werden kann.

Unter Einsatz der *OOP* kann dieses Kriterium in den meisten Fällen eingehalten werden, allerdings nicht in allen.

Mit der *aspektorientierten Programmierung* (*AOP*), die eine Erweiterung der *OOP* darstellt lässt sich auch in diesen Fällen ein noch höherer Grad der *Modularisierung* erreichen.

Eine Spezialform der *AOP* stellt die Sprache *ObjectTeams/Java* (*OT/J*) dar. Hier kann das Verhalten von Klassen bzw. Objekten in sog. *Kollaborationen* beeinflusst werden, ohne den Code der Klassen zu verändern.
Eine Möglichkeit dieses Verhalten zu erreichen, ist das *Aspektweben*. Hierbei werden Klassen für den Benutzer transparent von einer *Laufzeitumgebung* verändert. Dies kann statisch beim Start eines Programms[1] oder dynamisch während seiner Laufzeit durchgeführt werden.
Momentan werden *Aspekte* von der *OT/J-Laufzeitumgebung* statisch gewoben.

Ziel dieser Arbeit ist es, einen Ansatz, wie die *OT/J-Laufzeitumgebung* auch dynamisch arbeiten kann, aufzuzeigen und umzusetzen.

Dazu wird im ersten Kapitel zunächst dargestellt, in welchen Fällen die Möglichkeiten der *OOP* zur *Modularisierung* von Programmen nicht ausreichen und wie durch die *aspektorientierte Programmierung* eine besser *Modularisierung* erreicht werden kann.

Das zweite Kapitel beschäftigt sich dann mit der Frage, mit welchen Methoden die *AOP* technisch umgesetzt werden kann. Hier wird unter anderem auch das *Aspektweben* genauer erläutert.

Im dritten Kapitel wird die Sprache *ObjectTeams/Java* als Spezialform der *AOP* genauer vorgestellt.

Anschließend wird im vierten Kapitel die aktuelle *Laufzeitumgebung* von *OT/J* beschrieben, die die Features von *OT/J* durch statisches *Aspektweben* umsetzt. Hier wird auch darauf eingegangen, welche Nachteile durch die Verwendung von

[1] Der Zeitpunkt "beim Start des Programms" ist etwas ungenau. Er wird im Laufe dieser Arbeit konkretisiert werden.

statischem *Aspektweben* entstehen und es wird überprüft, welche anderen Techniken in Frage kommen, um die Sprachelemente von *ObjectTeams/Java* umzusetzen.

Im fünften Kapitel wird dargelegt wie die Funktionalität von *OT/J* mit Hilfe von dynamischem *Aspektweben* umgesetzt werden kann.

Kapitel sechs beschreibt die technische Implementierung des neuen dynamischen *Aspektwebers*.

In Kapitel sieben wird dann das *OSGi Komponentenframework* vorgestellt. Weiterhin wird die Umsetzung von *OT/J* im Kontext des *OSGi Komponentenframeworks* (*OT/Equinox*) beschrieben und es wird dargelegt, wie diese Implementierung an das dynamische *Aspektweben* angepasst werden kann.

Kapitel acht fasst die Arbeit zusammen, bewertet die Ergebnisse dieser Arbeit und beschreibt offene Punkte, die in weiteren Arbeiten behandelt werden sollten.

1 Grundlagen der aspektorientierten Programmierung (AOP)

Aspektorientierte Programmierung ist eine Erweiterung der *objektorientierten Programmierung* (*OOP*). Im Allgemeinen gilt für Erweiterungen einer bestimmten Technik, dass es einen Grund für die Entwicklung dieser neuen Technik, also ein Problem bei der Alten, gibt.
Im Folgenden soll dieses Problem dargestellt und davon ausgehend erklärt werden, wie die *AOP* dieses Problem löst bzw. zu lösen versucht.

Eines der wesentlichen Prinzipien von Techniken zur Softwareentwicklung ist es, Programme so zu strukturieren, dass sich thematisch gleiche Funktionalitäten eines Programms auch an gleicher Stelle im Code wiederfinden. Dieses Prinzip bezeichnet man als *Modularisierung*.
In der *OOP* wird dieses Prinzip im Wesentlichen durch Klassen realisiert. Eine Klasse ist eine Sammlung von zusammengehörenden Datenstrukturen und Funktionen zu diesen Datenstrukturen. Das heißt, ähnliche Funktionalitäten sollten in einer Klasse oder einer kleinen Menge von Klassen gekapselt sein.

1.1 Crosscutting concerns

Allerdings existieren Inhalte, die sich nicht kapseln lassen. Diese Inhalte werden *Crosscutting concerns (CCC)* genannt. Um einen *CCC* zu implementieren ist es also nötig, eine große Menge von Klassen anzupassen. Ein gutes Beispiel für einen *CCC* ist Logging. Um beispielsweise jeden Anfang und jedes Ende einer Methode zu loggen, sind in jeder Methode (mindestens) zwei Zeilen Code zu implementieren. Das heißt die Funktionalität Logging ist nicht in einer Klasse gekapselt, sondern verteilt sich auf alle Klassen eines Programms.

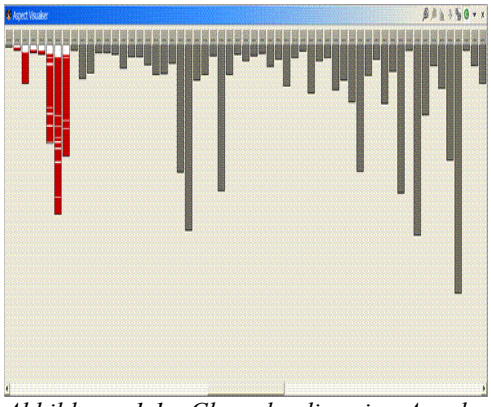

Abbildung 1.1: Class loading im Apache Tomcat Quelle: [Wik07]

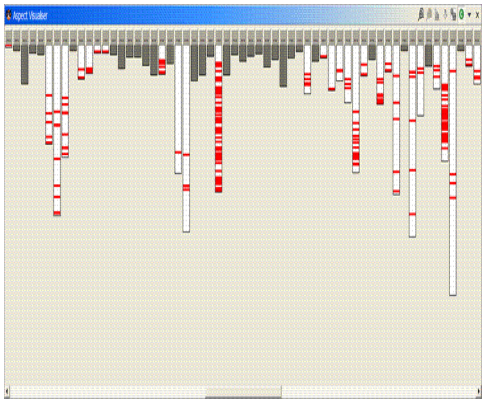

Abbildung 1.2: Logging im Apache Tomcat Quelle: [Wik07]

In den Abbildungen 1.1 und 1.2 ist der Unterschied zwischen einem gut modularisieren *Concern* und einem *CCC* am Beispiel des Apache Tomcat Webservers dargestellt.
In Abbildung 1.1 ist die Verteilung des Codes, der benötigt wird, um Klassen zu laden, dargestellt. Horizontal sind die verschiedenen Pakete dargestellt. Auf der Y-Achse ist die Größe dieser Pakete zu sehen. In rot ist der Code dargestellt der diese Funktionalität betrifft. Wie man hier sehen kann, ist das Laden von Klassen in einigen wenigen Paketen implementiert. Das Logging (Abbildung 1.2) verteilt sich dagegen auf fast alle Pakete.

Ziel der *AOP* ist es, auch die *CCC* gut modularisieren zu können.

1.2 Aspekte

Aspekte sind mit Klassen zu vergleichen. Ein *Aspekt* fasst Code zusammen, der eine Funktionalität implementiert. Der Unterschied zu Klassen in der *OOP* ist, dass *Aspekte* das Verhalten von anderen Klassen oder *Aspekten* verändern können.
Dazu können innerhalb eines *Aspekts* sogenannte *Advices* definiert werden. Ein *Advice* ist zu vergleichen mit einer Methode.
Im Gegensatz zur *OOP* wird ein *Advice* nicht direkt aufgerufen, sondern es wird definiert, an welchen Stellen von bestehenden Code er aufgerufen werden soll. Diese Stellen werden als *Joinpoint* bezeichnet. Grundsätzlich können alle Stellen einer bestehenden Implementierung *Joinpoints* sein. Bei den meisten *aspektorientierten* Sprachen ist diese Menge jedoch auf besonders prägnante Stellen (z.B. Ausführung einer Methode , Aufruf einer Methode, Zugriff auf ein Feld eines Objekts u.s.w) begrenzt.
Wenn man das Beispiel des Loggings betrachtet stellt man fest, dass *Joinpoints* alleine nicht ausreichend sind, um zu beschrieben, an welchen Stellen ein *Advice* ausgeführt werden soll. Da das Logging beim Start und Ende jeder Methode

durchgeführt werden soll, müsste man für den entsprechenden *Advice* alle Methoden der bestehenden Implementierung auflisten, was zu einem hohen Aufwand führen würde. Eleganter ist es nicht jeden einzelnen *Joinpoint* aufzulisten sondern nur die Menge der *Joinpoints* zu definieren, an denen dieser *Advice* greifen soll. Eine solche Mengendefinition wird *Pointcut* genannt.
In der *aspektorientierten* Sprache *AspectJ* würde der *Pointcut* für das Logging folgendermaßen definiert:

```
pointcut log() : execution(* *(..) )
```

Hierdurch wird die Menge von allen *Joinpoints* definiert, an denen eine Methode mit beliebigem Namen (zweiter *), beliebigem Rückgabetyp (erster *) und beliebiger Parameterliste ((..)) ausgeführt (`execution`) wird.
Weiterhin kann sich ein *Pointcut* auch aus anderen *Pointcuts* zusammensetzen.

Einem *Advice* kann dann ein *Pointcut* zugewiesen werden , an dem er ausgeführt werden soll. Außerdem kann definiert werden, ob dieser *Advice* direkt vor dem *Pointcut* oder nach ihm ausgeführt werden soll oder ob der *Advice* den *Pointcut* ersetzen soll.

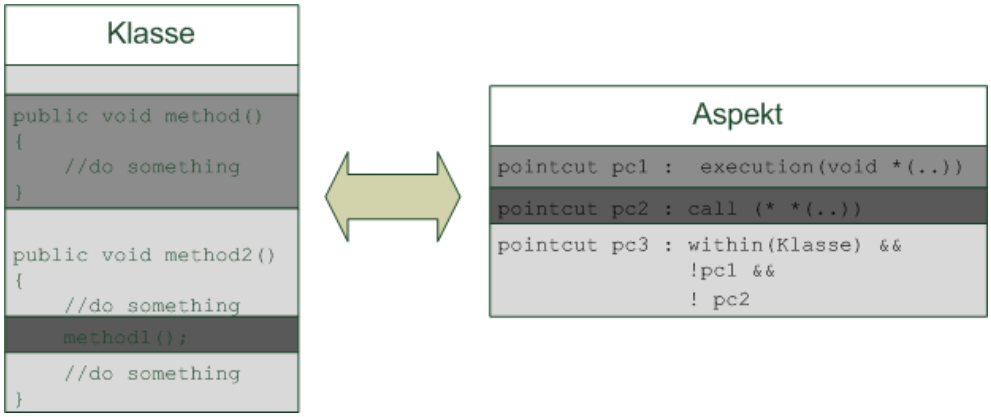

Abbildung 1.3: Beziehung zwischen Joinpoints und Pointcuts

In Abbildung 1.3 ist dargestellt, welche *Joinpoints* ein *Pointcut* betrifft. Der *Pointcut* `pc1` gilt während der Ausführung der Methode `method1`, `pc2` beim Aufruf der Methode `method1` und `pc3` in der gesamten Klasse, aber nur wenn `pc1` und `pc2` nicht gelten.

Im nächsten Kapitel soll nun dargestellt werden, wie die *aspektorientierte Programmierung* technisch umgesetzt werden kann.

2 Technische Umsetzung der aspektorientierten Programmierung

Um das gewünschte Verhalten von *aspektorientierten Sprachen* tatsächlich zu gewährleisten, sind Techniken nötig, die über die Standard Java API hinausgehen. Hierbei kann man im Wesentlichen drei Ansätze unterscheiden:

1. *Aspektweben*
2. *Delegation bzw. Proxies*
3. *Metaobject Protocol*

Beim ersten Ansatz wird der Code von Klassen so verändert, dass *Aspekte* ausgeführt werden können (s. 2.2). Die zweite Technik dagegen verändert keine Klassen bzw. Objekte, sondern benutzt Instanzen (*Proxies*), die stellvertretend für die eigentlichen Klassen/Objekte Anfragen an sie entgegennehmen (s. 2.3).
Bei der dritten Variante wird auf einer Metaebene auf Anfragen an eine Klasse reagiert (s. 2.4).

Diese Techniken und ihre verschiedenen Unterarten sollen in diesem Kapitel dargestellt werden.

Da beim *Aspektweben* Klassen verändert werden, ist es zunächst nötig, zumindest grob, zu verstehen, in welchem Format Informationen über Klassen, inklusive dem Code ihrer Methoden, abgelegt werden und wie sie beim Ablauf eines Programms geladen werden.
Im Folgenden sollen diese Fragen geklärt werden.

2.1 Interne Funktionsweise von Java

In Java laufen Programme nicht selbstständig in einem Prozess, sondern werden von einer Instanz verarbeitet, die eine Schnittstelle zwischen Programm und Betriebssystem darstellt, der *Java virtual machine* (*JVM*). Von ihr werden alle

Klassen geladen, die für ein Programm benötigt werden, und das Programm ausgeführt. Die Klassen liegen in einem binären Format, dem *Class file format*, vor. Die Spezifikation von Java legt dabei lediglich das Format fest. Wie diese binären Daten entstehen und mit welchem Medium sie gespeichert werden, ist dagegen grundsätzlich beliebig. In diesem binären Format sind drei Arten von Informationen gespeichert.

1. Allgemeine Informationen über die Klasse z.B. ihr Name, ihre Methoden, ihre Felder (s. 2.1.1)
2. Attribute der Klasse, Felder und Methoden(s. 2.1.2)
3. Der Code der Methoden, der *Bytecode* (s. 2.1.3)

Diese binären Daten werden dann von sog. *Class loadern* (s. 2.1.4) geladen und für die Benutzung in der *JVM* aufbereitet. Anschließend können Instanzen einer Klasse erzeugt werden und der *Bytecode* ihrer Methoden von einem Interpreter ausgeführt werden (s. Abbildung 2.1).

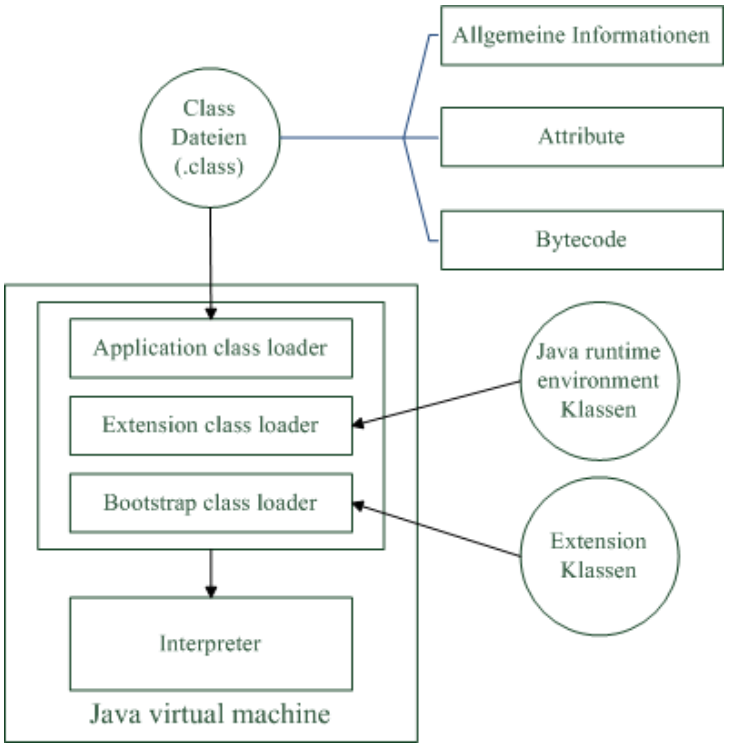

Abbildung 2.1: Funktionsweise von Java

Im Folgenden sollen nun diese einzelnen Teile genauer beleuchtet werden.

2.1.1 Class Dateien

Zunächst soll hier beschrieben werden, welche Informationen in der binären Repräsentation einer Klasse gespeichert sind. Die im Folgenden verwendete Notation entspricht dabei der Definition von Strukturen in C. Die Typen `u2` und `u4` entsprechen zwei bzw. vier Bytes, also den Datentypen `short` und `int` in Java.

```
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. }
```

Die einzelnen Bestandteile einer Class Datei haben folgende Bedeutung:

1. `magic`**:**
   Durch die *Magic number* wird eine Datei als Class Datei identifiziert. Sie muss immer den Wert `0xCAFEBABE` haben.
2. `minor_version` und `major_version`:
   Diese Nummern identifizieren die Version des Formats der Class Datei eindeutig
3. `constant_pool` und `constant_pool_count`:
   Im *Constant pool* sind sämtliche Konstanten hinterlegt, die in einer Class Datei benutzt werden. Er enthält alle konstanten Werte (`String`, `Integer`, `Float`, `Double`, `Long`) und alle Namen von Feldern, Methoden und Klassen, die in dieser Class Datei verwendet werden. Der *Constant pool count* gibt die Anzahl der Einträge im *Constant pool* plus eins an. Dabei ist zu beachten, dass die Einträge je nach Typ eine unterschiedliche Länge haben. Es ist also nicht möglich mittels des *Constant pool counts* zu bestimmen, an welcher Stelle in der Class Datei die nachfolgenden Einträge zu finden sind.
4. `access_flags`:
   Hier sind Modifier der Klasse (`public`, `private`, `protected`, `static`, `abstract`) eingetragen.
5. `this_class` und `super_class`:
   Diese Einträge speichern als Verweis auf den *Constant pool* den Namen

der Klasse und ihrer Superklasse

6. `interfaces`, `interfaces_count`: Hier werden, auch als Referenz auf den *Constant pool,* die Namen und Typen von Interfaces hinterlegt.
7. `methods`, `methods_count`, `fields` und `fields_count`: `methods` und `fields` sind Arrays der Strukturen `method_info` bzw. `field_info`. In ihnen werden der Name, die Signatur und die Attribute (s. nächster Abschnitt) von Methoden und Feldern gespeichert.
8. `attributes` und `attributes_count`:
   In diesem Eintrag sind alle Attribute der Klasse enthalten. Wie Attribute aufgebaut sind und welche Bedeutung sie haben wird im nächsten Abschnitt erläutert.

2.1.2 Attribute von Klassen Feldern und Methoden

Ein Attribut besteht aus einem Namen, der Länge seiner Informationen und den Informationen selbst. Diese Form ist so abstrakt gehalten, dass es möglich ist neue Attribute zu definieren und dort beliebige Informationen zu verpacken. Dazu schreibt die Spezifikation der *JVM* vor, dass jedes Attribut, das einer konkreten *VM* nicht bekannt ist, ignoriert werden muss und nicht die Semantik dieser Klasse verändern darf.
Beispiele für vordefinierte Attribute sind der Name der Quellcode Datei, Exceptions, die diese Klasse verwendet, Innere Klassen u.s.w. Auch der *Bytecode* der Methoden ist in einem Attribut mit dem Namen `Code` enthalten. Wie dieser Code aufgebaut ist und ausgeführt wird, beschreibt der nächste Abschnitt.

2.1.3 Aufbau und Ausführung Bytecode

Java *Bytecode* wird von einem Interpreter ausgeführt. Dieser setzt Anweisungen im *Bytecode* in Maschinencode für ein konkretes Betriebssystem um. Dieser Interpreter arbeitet *stack basiert*. Das heißt, für das Verwalten von Werten werden nicht wie im Prozessor Register benutzt, sondern ein Stack, auf dem diese Werte abgelegt werden können.
An dieser Stelle soll kurz eine Übersicht gegeben werden, welche Arten von Instruktionen im Java *Bytecode* verwendet werden können.

- Stack Manipulationen (Duplizieren eines Wertes auf dem Stack, Vertauschen zweier Werte auf dem Stack, ein Element vom Stack löschen u.s.w)
- Arithmetische Operationen (Addition, Subtraktion u.s.w)
- Sprungbefehle (Bedingte Sprünge und unbedingte Sprünge)
- Methodenaufrufe (z.B. Aufruf einer dynamisch gebundenen, einer privaten oder einer statischen Methode)
- Feldzugriffe (Lesende/schreibende auf Felder)
- Konstanten (Ablegen von konstanten Werten auf dem Stack)

Diese Auflistung ist natürlich nicht vollständig, da dies den Umfang dieser Arbeit sprengen würde. Sie soll nur einen kurzen Überblick über Java *Bytecode* liefern.

Nachdem nun geklärt wurde, wie Klassen und der Code ihrer Methoden in Java gespeichert wird, stellt sich nun die Frage, wie die binäre Repräsentation von Klassen in die *JVM* gelangt. Diese Aufgabe erledigen die *Class loader*, die im Folgenden erklärt werden.

2.1.4 Class loader

Jede Java Klasse, die in einem Programm verwendet wird, wird von einem *Class loader* geladen. Von der *JVM* wird automatisch ein solcher aufgerufen, wenn eine Klasse benötigt wird. Es ist es aber auch möglich, eine Klasse explizit zu laden.

Die *JVM* stellt standardmäßig drei *Class loader* zur Verfügung:

1. *Boostrap class loader*: Dieser lädt alle Klassen aus der J*ava Runtime Environment* (*JRE*).
2. *Extension class loader*: Von ihm werden alle Klassen geladen, die sich im Extension Verzeichnis der *JRE* befinden
3. *Application class loader*: Dieser *Class loader* ist dafür zuständig, sämtliche applikationsspezifische Klassen aus dem *Classpath* zu laden.

Zusätzlich zu diesen drei *Class loadern* ist es auch möglich eigene *Class loader* zu implementieren und einzusetzen. Jeder *Class loader* hat dabei einen *Parent class loader*. So entsteht eine baumartige Hierarchie von *Class loadern*. Bei den *Class loadern* der *JVM* ist der *Boostrap class loader* ganz oben in der Hierarchie. Darauf folgen der *Extension class loader* und der *Application class loader*.

Das Laden einer Klasse unterteilt sich dabei in folgende Schritte:

1. In der Methode `findLoadedClass` wird überprüft, ob diese Klasse schon geladen wurde. Ist das der Fall, wird sie dem Klienten zurückgeliefert.
2. Wenn die Klasse noch nicht geladen wurde, delegiert dieser *Class loader* die Anfrage zunächst an den *Class loader* weiter, der in der Hierarchie eine Stufe höher steht. Die ersten beiden Schritte werden dann so oft wiederholt, bis die Klasse gefunden ist oder der *Bootstrap class loader* erreicht ist.
3. Wurde die Klasse auch vom *Bootstrap class loader* nicht gefunden, muss der ursprüngliche *Class loader* versuchen diese Klasse zu laden. Wird sie auch von ihm nicht gefunden, wirft er eine `ClassNotFoundException` und der Vorgang bricht erfolglos ab.

Dieses Verhalten ist in Abbildung 2.2 beispielhaft dargestellt.

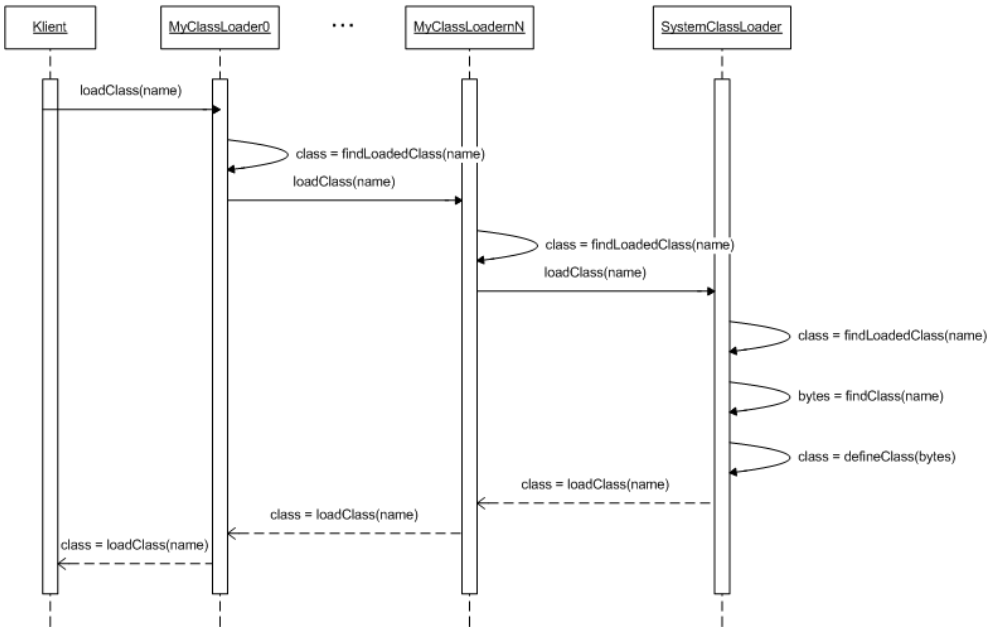

Abbildung 2.2: Zusammenarbeit mehrerer Class loader

2.2 Aspektweben

Beim *Aspektweben* wird der Code, auf den die Aspekte wirken sollen verändert.
Die Instanz, die diese Veränderung vornimmt wird *Aspektweber* genannt. Das *Aspektweben* kann mit verschiedenen Techniken umgesetzt werden. Diese Techniken lassen sich nach zwei Kriterien unterteilen:

1. Welcher Code wird gewoben?
2. Wann wird gewoben?

Nach dem ersten Kriterium lassen sich zwei Kategorien von *Aspektwebern* unterscheiden. *Aspektweber,* die auf Quellcode oder solche, die auf *Bytecode* arbeiten.
Aspektweber, die auf Quellcode arbeiten, haben dabei zwei wesentliche Nachteile. Zum einen müssen alle Programmteile, die gewoben werden sollen, als Quellcode vorliegen. Zum anderen ist auch ein Compiler nötig, der den gewobenen Quellcode in das binäre *Class file format* überführt. Gerade bei eingebetteten Systemen mit wenig Speicherplatz ist ein Compiler aber nicht vorhanden.
Aus diesen Gründen arbeiten die meisten Weber auf *Bytecode* Ebene, weswegen Weber, die auf Quellcode arbeiten, hier nicht weiter beachtet werden sollen.

Für das zweite Kriterium, nämlich den Webezeitpunkt, lassen sich ebenfalls zwei Kategorien von Webern unterscheiden. Diese werden in den beiden folgenden Abschnitten dargestellt.

2.2.1 Aspektweben zur Ladezeit

Der *Bytecode* von Klassen kann zu dem Zeitpunkt, an dem sie geladen werden, nahezu beliebig verändert werden. An dieser Stelle existiert seitens der *JVM* keine Überprüfung der Semantik der Klassen. Lediglich die Syntax des *Class file formats* wird geprüft. Dieser Zeitpunkt bietet sich also an, um den *Bytecode* von Klassen zu manipulieren.

Es existieren zwei Möglichkeiten, wie sich der *Aspektweber* in den Prozess des Ladens von Klassen einhängen kann, so dass er seine manipulierte Version des Bytecodes an die *JVM* übergeben kann. Diese werden im Folgenden vorgestellt.

Aspektweben im Class loader

Wie in 2.1.4 dargestellt, wird jede Klasse von einem *Class loader* geladen.
Ein selbstgeschriebener *Class loader* kann nun innerhalb dieses Prozesses *Bytecode* verändern. Dazu implementiert er die Methode

```
protected Class<?> loadClass(String name, boolean resolve)
```

Diese Methode bekommt den Namen der Klasse übergeben (auf den Parameter `resolve` soll hier nicht weiter eingegangen werden) und liefert eine Class Instanz zurück. Diese Methode kann nun nach dem folgendem (vereinfachten) Schema überschrieben werden:

```
1.  protected Class<?> loadClass(String name,
2.            boolean resolve){
3.       ...
4.       byte[] bytecode = getBytecode(name);
5.       b = manipulateBytes(b);
6.       return super.defineClass(name, b, off, len);
7.  }
```

Nachdem der `Bytecode` in der Methode `getBytecode` geholt wurde, kann er in der Methode `manipulateBytes` nun beliebig transformiert werden.
Es reicht allerdings nicht aus, einen eigenen *Class loader* zu implementieren. Man muss auch dafür sorgen, dass er verwendet wird.
Wie in 2.1.4 erwähnt, werden die Klassen eines Programms durch den *Application class loader* geladen. Dieses Verhalten kann auch nicht beeinflusst werden. Die einzige Möglichkeit einen eigenen *Class loader* zu verwenden, besteht darin, dass man in der `main` Methode einen *Class loader* erzeugt und mit ihm dann alle weiteren Klassen lädt. So wird nur die Klasse, die die `main` Methode enthält, vom *Application class loader* geladen (s. Beispiel).

```
1. public static void main(String args[]) {
2.     ClassLoader cl = new MyClassLoader(
3.             this.getClass().getClassLoader());
4.     Class<MyClass> clazz =
5.             cl.loadClass("foo.bar.MyClass");
6.     MyClass mc = clazz.newInstance();
7.     ...
8. }
```

Den Weg eine eigenen *Class loader* zu benutzen verfolgt u.a. das Framework *JMangler* (s. 4.3.1 und [Aus00]). Hier können *Transformer* implementiert werden, die den *Bytecode* verändern. Diese werden dann vom Framework aufgerufen.

Java Programming Language Instrumentation Services (JPLIS)

Eine weitere Möglichkeit, während des Ladens der Klassen *Bytecode* zu verändern, bietet Java mit dem Framework *JPLIS* an. Um dieses Framework zu nutzen, muss ein sogenannter *Java agent* implementiert werden. Ein *Java agent* ist grundsätzlich eine normale Jar-Datei, die allerdings eine Klasse besitzen muss, die die Methode

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

implementiert. Diese Klasse muss in der Manifest Datei des *Agents* durch das Attribut `Premain-Class` bekanntgegeben werden.
Außerdem muss beim Start einer Anwendung angegeben werden, welche *Agents* verwendet werden sollen. Dies geschieht mit dem Kommandozeilenparameter `-javaagent`. Wie der Name schon sagt, wird die Methode `premain` dann vor der `main` Methode aufgerufen. Sie bekommt eine Instanz des Interfaces `java.lang.instrument.Instrumentation` übergeben. Auf dieser Instanz können mit der Methode `addTransformer` Objekte von Klassen, die das Interface `java.lang.instrument.ClassFileTransformer` implementieren, als *Transformer* registriert werden.
Das Interface `ClassFileTransformer` enthält die Methode

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

Diese Methode wird beim Laden jeder Klasse aufgerufen. In ihr kann dann der *Bytecode* beliebig verändert werden.
Mit Hilfe des *JPLIS* Frameworks lässt sich der *Bytecode* von Klassen nicht nur zur Ladezeit, sondern auch zur Laufzeit verändern. Wie dies funktioniert, wird im nächsten Kapitel gezeigt.

2.2.2 Aspektweben zur Laufzeit mit JPLIS

Über das Framework *JPLIS* ist es ebenfalls möglich Code zur Laufzeit zu verändern. Dazu kann die Instanz des Interfaces `Instrumentation` verwendet werden, die ein Agent in seiner `premain` Methode übergeben bekommt. Neben der schon beschriebenen Möglichkeit `Transformer` zu registrieren, bietet dieses Interface die Methode

```
redefineClasses(ClassDefinition... definitions)
```

an.
An diese Methode werden beliebig viele Instanzen der Klasse `ClassDefinition` übergeben. Diese Klasse ist ein Paar aus den Bestandteilen `Class<?>` und `byte[]`. Es muss also für jede zu redefinierende Klasse die bestehende Class Instanz und der neue *Bytecode* übergeben werden.
Der gesamte Prozess der Veränderung von Code zur Laufzeit mittels *JPLIS* unterteilt sich also in folgende Schritte:
Beim Start des Programms:

1. Aufruf der Methode `premain` des Agenten.
2. Speichern der Instanz des Interfaces `Instrumentation` zur weiteren Verwendung

Zu dem Zeitpunkt, an dem die Implementierung einer oder mehrerer Klassen geändert werden soll:

3. Auffinden des Bytecodes der Klasse(n)
4. Verändern des Bytecodes
5. Auffinden des Class Objektes der Klasse(n)
6. Aufruf der Methode `redefineClasses`

Einschränkungen von Codetransformationen zur Laufzeit

Zur Ladezeit kann der *Bytecode* beliebig verändert werden. Das ist zur Laufzeit nicht mehr der Fall.
Hier sind nur folgende Änderungen erlaubt:

- Verändern von Methodenrümpfen
- Redefinition des *Constant pools*

Nicht erlaubt sind alle anderen Veränderungen, also:

- Verändern der Vererbungsbeziehung der Klasse
- Hinzufügen, Entfernen oder Umbenennen von Methoden
- Hinzufügen, Entfernen oder Umbenennen von Feldern
- Ändern der Signatur von Methoden oder Feldern

Es ist geplant in zukünftigen Versionen von Java diese Einschränkungen (zumindest teilweise) aufzuheben. Ob und wenn ja, wann dies allerdings der Fall sein wird, ist allerdings unklar (s. 8.3.1).

2.3 Delegation und Proxies

Bei der *delegationsbasierten* bzw. *proxybasierten* Umsetzung a*spektorientierter Programmierung* werden im Gegensatz zum *Aspektweben* (s. 2.2) die Klassen nicht verändert. Es werden dagegen Aufrufe an eine Klasse zu einer anderen Klasse (Proxy) umgeleitet (delegiert). Wie das technisch umgesetzt werden kann soll in diesem Kapitel beschrieben werden (eine ausführlichere Beschreibung findet sich in [HS07]). Dazu wird zunächst der Begriff Proxy genauer erläutert.

2.3.1 Proxies

Allgemein bezeichnet man als *Proxy* eine Klasse oder Instanz einer Klasse, die stellvertretend für die ursprüngliche Instanz Methodenaufrufe entgegennimmt. Der *Proxy* entscheidet dann, wie weiter verfahren werden soll, ob also z.B. die Originalmethode noch aufgerufen oder zusätzlicher Code ausgeführt werden soll.

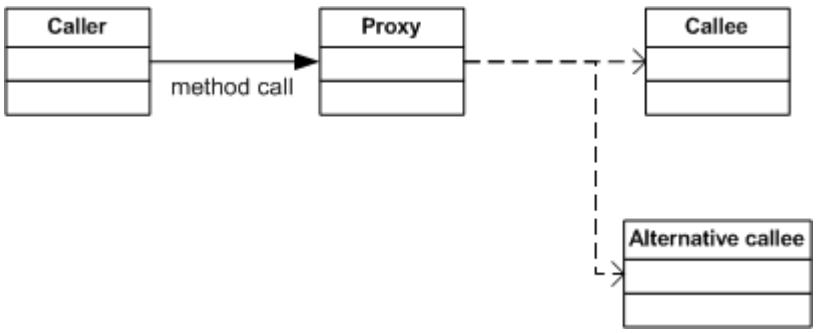

Abbildung 2.3: Funktionsweise eines Proxies

In Abbildung 2.3 wird schematisch die Funktionsweise eines *Proxies* dargestellt.

Es gibt zwei Ansätze wie aspektorientierte Programmierung mit *Proxies* umgesetzt werden kann:

1. *delegationsbasiert*
2. *proxybasiert*

Beide Ansätze verwenden das in Abbildung 2.3 dargestellte Schema. Der Unterschied zwischen ihnen besteht darin, dass beim *proxybasierten* Ansatz die *Proxies* auf Applikationsebene eingeführt werden. Diesen Ansatz verfolgt z.B. die *Dynamic proxy API* von Java. Hier wird also entweder eine Modifikation des Programms, in dem die *Proxies* laufen sollen, nötig oder das Programm muss von vornherein für den Einsatz von *Proxies* designed worden sein. Das widerspricht allerdings dem Grundsatz der AOP, dass die Funktionalität eines Programms ohne Modifikationen an seinem Code geändert werden soll.

Der *delegationsbasierte* Ansatz setzt dagegen auf der Ebene der Laufzeitumgebung von Sprachen an (bei Java also in der *JVM*). Diese muss dann so verändert werden, dass, für das Originalprogramm transparent, *Proxies* eingeführt werden können.

2.4 Metaobject protocol

Die dritte Möglichkeit AOP umzusetzen, stellt die Verwendung des *Metaobject protocols* einer Sprache dar. Über das *Metaobject Protocol* kann das Verhalten einer Sprache über *Metaobjects* verändert werden. So können z.B. Methodenaufrufe abgefangen werden.
Java bietet kein *Metaobject protocol* an.
Allerdings bietet Java mit der *Java Platform Debugger Architecture* (*JPDA*) die Möglichkeit die Funktionalität von Programmen zu verändern, wenn diese im *Debug Modus* laufen. Gleichzeitig bietet die *JPDA* genau wie *JPLIS* auch die Möglichkeit, den *Bytecode* von Klassen zur Lade- und Laufzeit zu verändern.
Nachfolgend soll erläutert werden, welche Möglichkeiten die *JPDA* zur Umsetzung der *AOP* bietet.

2.4.1 Java Platform Debugger Architecture

Die *Java Platform Debugger Architecture* bietet verschiedene Schnittstellen, um ein Programm zu unterbrechen, es zu untersuchen und es auch zu verändern.

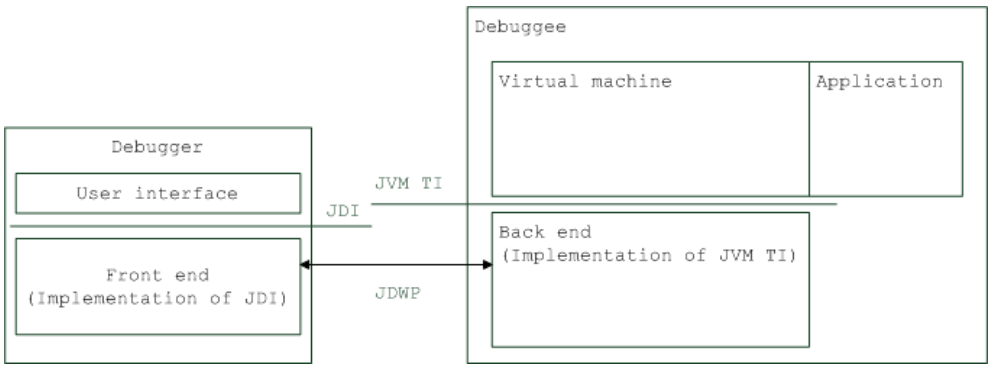

Abbildung 2.4: Funktionsweise der JPDA

In Abbildung 2.4 sind die Schnittstellen und ihre Zusammenarbeit dargestellt. Zunächst existiert eine Instanz, die überwacht bzw. beeinflusst werden soll (`Debuggee`). Das eigentliche Programm läuft auf einer Instanz der *Java Virtual Machine*, die die Schnittstelle *Java Virtual Machine Tool Interface* (JVM TI) anbietet. Über eine Implementierung dieser Schnittstelle (`Back end`) kann auf Events in der *JVM* reagiert werden. Eine Implementierung des *Java Debug Interfaces* (*JDI*) kann mit dem `Back end` über das *Java Debug Wire Protocol* (*JDWP*) kommunizieren. Der Debugger Prozess, dessen `User interface` diese

Implementierung benutzt, kann dabei auch in einem eigenen Prozess und einer eigenen *JVM* laufen. *JDI* und *JVM TI* bieten dabei grundsätzlich ähnliche Funktionalität, da das *JDI* auf dem *JVM TI* aufbaut. Der wesentliche Unterschied zwischen ihnen besteht darin, dass Implementierungen, die das *JVM TI* benutzen, in nativem Code (also C/C++, Assembler o.ä.) vorliegen müssen. Das *JDI* dagegen ist eine Java Schnittstelle. Deshalb verwenden Tools, die eines dieser beiden Interfaces benutzen, auch meist das *JDI*. Ein weitere Unterschied besteht darin, dass das JDI Debug Informationen in den Class Dateien benötigt.

Beide Interfaces bietet im Wesentlichen drei Möglichkeiten, um mit der *JVM* zu interagieren:

1. Abfragen bzgl. des Zustands der *JVM*: Hierzu existiert in beiden Interfaces eine Struktur bzw. Klasse (in *JVM TI* die Struktur `jvmtiEnv` und in *JDI* die Klasse `VirtualMachine`), über die sämtliche Zustandsinformationen (z.B. welche Klassen wurden geladen, welche Objekte existieren, welche Größe haben diese Objekte u.s.w.) abgefragt werden können.
2. Regieren auf Events, die von der *JVM* erzeugt werden:
   Die *JVM* erzeugt für jede Aktion, die geschieht (Laden einer Klasse, Methodenaufruf u.s.w.), ein Event und ruft eine Methode/Funktion auf, in der das Event behandelt werden kann. Für die Umsetzung der *AOP* könnte man sich z.B. vorstellen, dass, wenn ein Event behandelt wird, das den Aufruf einer Methode anzeigt, überprüft wird, ob vor dem Ausführen der Methode *Advice* Code ausgeführt werden soll.
3. Redefinition von Klassen: In beiden Interfaces ist es möglich, den *Bytecode* von Klassen zur Lade- oder zur Laufzeit zu verändern. Für die Redefinition von Klassen zur Laufzeit, gelten aber dieselben Restriktionen wie bei *JPLIS* (s. 2.2.2).

In diesem Kapitel wurde nun eine Übersicht über die verschiedenen Techniken gegeben, mit denen *aspektorientierte Programmierung* technisch umgesetzt werden kann.
An dieser Stelle soll noch keine Bewertung dieser Techniken stattfinden. Für die Bewertung ist es nötig, den Kontext zu kennen, in dem diese Techniken verwendet werden sollen. Dieser Kontext, nämlich die *aspektorientierte Sprache ObjectTeams/Java*, soll im nächsten Kapitel vorgestellt werden. Eine Bewertung dieser Techniken für *ObjectTeams/Java* findet dann in 4.4 statt.

3 Object Teams/Java (OT/J)

Die Sprache *ObjectTeams/Java* stellt eine Erweiterung von Java dar.
Die Idee hinter *OT/J* ist, wie bei der *OOP*, der realen Welt nachempfunden. Dort kommt es oft vor, dass sich Menschen (in der *OOP* also Objekte) in verschiedenen Kontexten unterschiedlich verhalten, also andere *Rollen* annehmen. So gibt sich ein Mensch im Berufsleben z.B. anders als im Privaten. *OT/J* setzt dieses Konzept in der Softwareentwicklung um.
Diesen Kontext nennt man *Kollaboration*, weswegen diese Art der Programmierung auch *kollaborationsbasierte Programmierung* genannt wird.

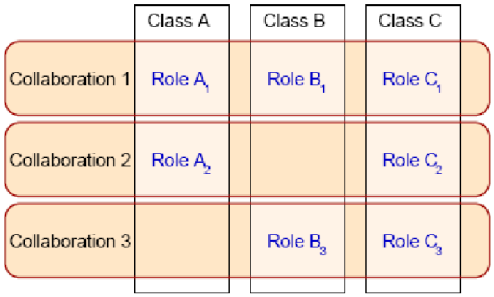

Abbildung 3.1: Kollaborationen und Rollen Quelle: [Her02]

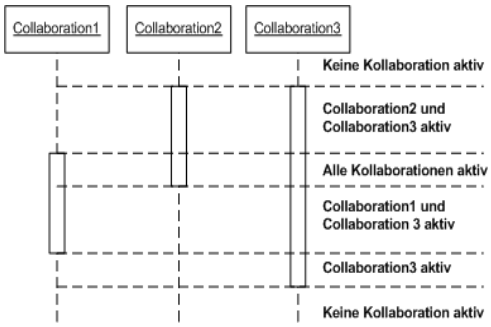

Abbildung 3.2: Kollaborationen im zeitlichen Verlauf

Eine *Kollaboration* kann beliebig viele Klassen umfassen und jede Klasse kann in jeder der *Kollaborationen*, an denen sie beteiligt ist, ein anderes Verhalten zeigen. Die Beteiligung einer Klasse an einer *Kollaboration* bezeichnet man als *Rolle* der Klasse.
In einer *Rolle* wird das veränderte Verhalten der Klasse für eine *Kollaboration* gekapselt.
In Abbildung 3.1 ist das Zusammenspiel von Klassen, *Rollen* und *Kollaborationen* dargestellt.

Eine *Kollaboration* kann dabei abhängig vom Zeitpunkt aktiv oder inaktiv sein. Es können auch mehrere *Kollaborationen*, an denen eine Klasse beteiligt ist zu einem Zeitpunkt aktiv sein (s. Abbildung 3.2).

Kollaborationsbasierte Programmierung kann als eine Unterart der *AOP* angesehen werden. *Rollen* entsprechen dabei den *Aspekten*. Genau wie *Aspekte* können sie *Advices* enthalten (bei *OT/J Callin Bindungen* genannt s. 3.2), die das Verhalten von bestehendem Code verändern. Der Unterschied besteht darin, dass ein *Advice* in der *AOP* beliebigen *Joinpoints* bzw. *Pointcuts* zugeordnet werden kann. *Rollen* können sich dagegen nur auf eine begrenzte, fest definierte Menge von *Joinpoints*, nämlich auf eine Klasse in einer Kollaboration, beziehen.
Bevor genauer darauf eingegangen wird, wie in OT/J *Advices* mittels *Callin Bindungen* definiert werden können, soll zunächst erläutert werden, wie Kollaborationen und Rollen in OT/J auf Klassen abgebildet werden.

3.1 Teams und Rollen

Eine *Kollaboration* wird in *OT/J* durch ein *Team* abgebildet. Ein *Team* ist eine Java Klasse, die mit dem *OT/J* Schlüsselwort `team` gekennzeichnet ist. *Teams* können auch geschachtelt werden, d.h. ein *Team* kann weitere *Teams* enthalten. Außerdem können einem *Team Rollen* zugeordnet werden. Jede innere Klasse eines *Teams* ist eine Rolle.

```
1. public team MyTeam {
2.     public class MyRole {} // MyRole ist eine Rolle
3. }
```

Jedes *Team* erbt dabei von der Klasse `org.objectteams.team`. Diese Vererbungsbeziehung ist dabei implizit, sie kann allerdings auch explizit deklariert werden. Die Klasse `Team` ist also vergleichbar mit der Klasse `Object`.

3.2 Callin Bindungen als Advices

Wie schon erwähnt, können die aus der *AOP* bekannten *Advices* in *OT/J* über *Callin Bindungen* abgebildet werden.

Dazu muss zunächst eine *Rolle* durch das Schlüsselwort `playedBy` an eine Klasse gebunden werden. Diese Klasse wird auch *Basisklasse* genannt. Auch *Rollen* und *Teams* können *Basisklassen* einer *Rolle* sein.

```
1. public team MyTeam {
2.     // MyRole ist eine Rolle mit der Basisklasse MyBase
3.     public class MyRole playedBy MyBase {}
4. }
```

Methoden dieser *Basisklasse* sind dann mögliche *Joinpoints* für *Callin Bindungen.* Zum Definieren einer *Callin Bindung* sind zwei Informationen nötig: Eine Rollenmethode, die den *Advice* darstellt und eine Methode der Basisklasse (Basismethode), die als *Joinpoint* fungiert.

Es existieren drei Arten von *Callin Bindungen*:

1. *Before callin Bindungen*
2. *After callin Bindungen*
3. *Replace callin Bindungen*

Je nach Art der *Callin Bindung* wird eine Rollenmethode direkt vor (1.) oder direkt nach (2.) der Basismethode ausgeführt oder sie kann die Basismethode ersetzen (3.). *Callin Bindungen* werden mit folgender Syntax definiert:

```
roleMethod <- [before | after |replace] baseMethod;
```

```
1.  public team MyTeam {
2.      public class MyRole playedBy MyBase {
3.          callin void roleMethod1() {
4.              //do something role specific
5.              base.roleMethod1();
6.              //do something role specific
7.          }
8.          public void roleMethod2() {
9.              //do something role specific
10.         }
11.         roleMethod1 <- replace baseMethod1;
12.         roleMethod2 <- after baseMethod1;
13.     }
14. }
```

In diesem Beispiel wird jeder Aufruf der Methode `baseMethod1` durch einen Aufruf der Methode `roleMethod1` ersetzt. Der Aufruf base.roleMethod1() in Zeile 5 führt das Verhalten aus, dass die Basismethode vor der Einführung dieser *Callin Bindung* gehabt hat[1]. Er wird auch *Base call* genannt. Dabei wird als Methodenname nicht der Name der Basismethode, sondern der Name der Rollenmethode benutzt. Da eine Rollenmethode auch in mehreren *Callin Bindungen* an verschiedene Basismethoden gebunden sein kann, ist in der Rollenmethode gar nicht bekannt, welche Basismethode sie ersetzt. Deshalb muss der Name der Rollenmethode benutzt werden.
Außerdem wird nach jedem Aufruf der Methode `baseMethod2` die Rollenmethode `roleMethod2` aufgerufen. Rollenmethoden, die durch eine *Replace callin Bindung* an eine Basismethode gebunden sind, dürfen nicht direkt aufgerufen werden. Das wird durch das Schlüsselwort `callin` ausgedrückt. Aus diesem Grund darf für sie auch keine Sichtbarkeit definiert werden.

[1] Wie wir in 3.4 sehen werden, muss das nicht das Verhalten der ursprünglichen Basismethode sein.

Callin Bindungen werden nur dann ausgeführt, wenn ein *Team* aktiv ist. Im Folgenden wird gezeigt, wie *Teams* aktiviert bzw. deaktiviert werden können.

3.3 Teamaktivierung/-deaktivierung

In OT/J existieren verschiedene Möglichkeiten, wie ein Team aktiviert/deaktiviert werden kann:

1. Explizite Aktivierung/Deaktivierung für alle Threads
2. Explizite Aktivierung/Deaktivierung für einen Thread
3. Explizite Aktivierung/Deaktivierung innerhalb eines `within` Blocks
4. Implizite Aktivierung/Deaktivierung

Für die Fälle 1. und 2. existieren die Methoden `activate()`, `deactivate()`, `activate(Thread thread)` und `deactivate(Thread thread)`. Diese Methoden können von jeder beliebigen Stelle im Programm aufgerufen werden.
Ebenso kann die explizite Aktivierung/Deaktivierung innerhalb eines `within` Blocks geschehen (3.) :

```
within(myTeam) { stmts }
```

Für alle Statements `stmts` innerhalb dieses Blocks ist die Instanz `myTeam` aktiv. Nach diesem Block wird der Zustand wiederhergestellt, in dem sich das *Team* vor dem Block befand.
Implizit aktiviert wird ein *Team*, wenn

1. eine Methode des *Teams* ausgeführt wird
2. eine Methode einer *Rolle* des *Teams* ausgeführt wird
3. es in ein anderes *Team* eingebettet ist und dieses *Team* aktiviert wird

3.4 Reihenfolge von Callin Bindungen

Es können mehrere *Rollen* existieren, die die gleiche Basisklasse binden. Auch können für eine Basismethode mehrere *Callin Bindungen* existieren. Ist das der Fall, müssen diese Bindungen in eine fest definierte Reihenfolge gebracht werden.

Bei der Bestimmung dieser Reihenfolge müssen folgende Fälle unterschieden werden:

1. Eine Basismethode ist von *Rollen* in unterschiedlichen *Teams* gebunden.
2. Eine Basismethode ist von verschiedenen *Rollen* im selben *Team* gebunden.
3. Eine Basismethode ist durch mehrere *Callin Bindungen* in derselben *Rolle* gebunden.

Natürlich sind auch sämtliche Kombinationen dieser drei Fälle möglich.

Das erste Kriterium für die Reihenfolge der Bindungen ist der Aktivierungszeitpunkt der *Teams*. Das *Team*, das zuletzt aktiviert wurde, hat dabei die höchste Priorität. Das heißt, seine *Before* und *Replace Callin Bindungen* werden zuerst und seine *After Callin Bindungen* zuletzt ausgeführt (s. Abbildung 3.3).
Wenn mehrere *Callin Bindungen* gleichen Typs (*Before, Replace, After*) zu einer Basismethode innerhalb eines Teams existieren (Fälle 2 und 3), reicht die Reihenfolge der Teamaktivierung natürlich nicht als Kriterium für die Ausführungsreihenfolge der Bindungen aus.
Hier muss explizit deklariert werden, in welcher Reihenfolge die Bindungen ausgeführt werden sollen. Dies geschieht durch das Schlüsselwort `precedence`. Für den zweiten Fall wird die Reihenfolge an einer beliebigen Stelle im *Team* folgendermaßen definiert:

```
precedence Role1, Role2, ... , RoleN; //wobei Role1, .. ,
                                      //RoleN Rollen sind.
```

Die Rolle, die zuerst in der Liste angegeben wird, hat dabei die höchste Priorität.
Wenn eine Basismethode durch mehrere *Callin Bindungen* in einer Rolle gebunden ist (Fall 3), müssen nicht die Rollen, sondern die *Callin Bindungen* selbst in eine Reihenfolge gebracht werden. Dazu ist es nötig, die *Callin Bindungen* zu benennen. Dies geschieht folgendermaßen:

```
NameDerCallinBindung: foo <- replace bar;
```

Nun können innerhalb einer *Rolle* hinter dem Schlüsselwort `precedence` die Namen der *Callin Bindungen* aufgelistet werden. Auch hierbei gilt, dass die *Callin Bindung*, die zuerst in der Liste definiert ist, die höchste Priorität hat.

Dabei ist nicht sichergestellt, dass alle Bindungen zu einer Methode auch ausgeführt werden. In jeder der *Replace callin Bindungen* kann entscheiden werden, ob weitere Bindungen bzw. die Originalmethode noch ausgeführt werden sollen. Dies ist nur dann der Fall, wenn die Rollenmethode, die durch eine *Replace callin Bindung* gebunden ist, einen *Base call* (s. 3.2) enthält. Ist das nicht der Fall, werden lediglich noch die *After callin bindungen* des aktuellen *Teams* ausgeführt. Dementsprechend ruft der *Base call* nicht zwangsweise die Originalmethode auf, sondern kann zur Ausführung weiterer *Callin Bindungen* führen.

Ein einfaches Beispiel soll die Ausführungsreihenfolge der *Callin Bindungen* noch einmal verdeutlichen :
Ex existieren zwei *Teams* T0 und T1, die wie folgt definiert sind. Dabei wird davon ausgegangen, dass jede Rollenmethode, die durch eine *Replace Callin Bindung* gebunden ist, einen *Base call* enthält.

```
1.  public team class T0 {
2.      protected class R0 playedBy B0{
3.          ...
4.          rm0 <- before bm0;
5.          callin1: rm1 <- replace bm0;
6.          callin2: rm2 <- replace bm0;
7.          rm3 <-after bm0;
8.          precedence callin1, callin2;
9.      }
10.  }
11.
12. public team class T1 {
13.     precedence R0, R1;
14.     protected class R0 {
15.         ...
16.         rm0 <- before bm0;
17.         rm1 <- replace bm0;
18.         rm2 <- after bm0;
19.     }
20.     protected class R1 {
21.         ...
22.         rm0 <- after bm0;
23.     }
24. }
25.
26. ...
27. public static void main(String args[]) {
28.     ...
29.     new T1().activate();
30.     new T0().activate();
31.     new B0().bm0();
32.     ...
33. }
34. ...
```

Diese Definition würde zu der in Abbildung 3.3 gezeigten Ausführungsreihenfolge führen. Dieses Diagramm geht dabei nicht auf technische Feinheiten ein, sondern soll die Ausführungsreihenfolge nur schematisch darstellen.
Da das *Team* `T0` zuletzt aktiviert wurde (Zeile 30), werden die *Before* und *Replace Bindungen* dieses *Teams* zuerst ausgeführt. Bei den *Replace Bindungen* hat die Bindung `callin1` durch die `precedence` Definition in Zeile 8 Priorität. Nachdem die *Before* und *Replace Bindungen* von `T0` abgearbeitet sind, werden jetzt die Rollenmethoden `rm0` und `rm1` von `T0.R0` ausgeführt . Diese Rolle hat durch Zeile 13 Priorität vor `T1.R1`. Anschließend folgt der Aufruf der Basismethode `bm0`. Nun werden die *After Callin Bindungen* `T1.R1.bm0`, `T1.R0.rm2` und `T0.R0.rm3` in dieser Reihenfolge ausgeführt.

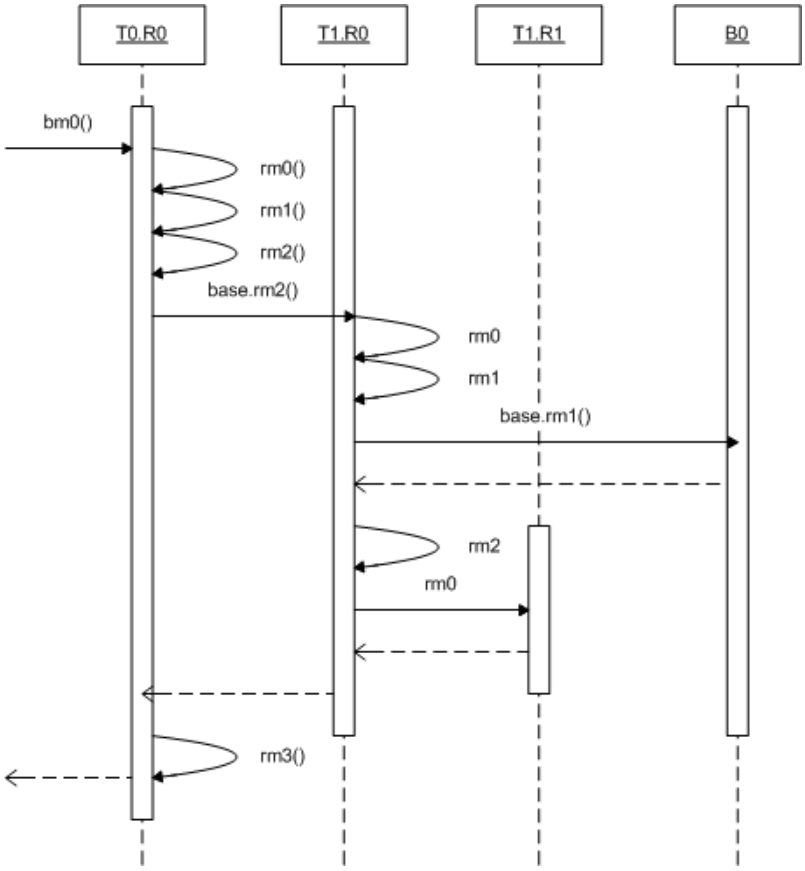

Abbildung 3.3: Aufrufreihenfolge von Callin Bindungen

3.5 Lifting und Lowering

Sollen für eine Basismethode *Callin Bindungen* ausgeführt werden, wird zunächst eine Instanz der Rolle benötigt, in der diese Bindungen definiert sind.
Den Vorgang, aus einem gegebenen Basisobjekt ein Rollenobjekt zu erzeugen, bezeichnet man als *Lifting*. Basisobjekte werden implizit geliftet, wenn Rollenobjekte benötigt werden, es ist aber auch möglich sie explizit zu erzeugen (s. [HHM09]).
Das Basisobjekt wird dabei in der Rolle gespeichert.
Den entgegengesetzten Weg, also aus einer Instanz der Rolle die Instanz der Basisklasse zu erhalten, wird als *Lowering* bezeichnet.

Das *Lifting* ist solange trivial, bis Bindungen auf Team- oder Basisseite vererbt werden. Wenn dies der Fall ist, muss eine intelligente Auswahl getroffen werden, für welche Klasse in der Basishierarchie welche Rolle der Rollenhierarchie verwendet werden soll. Diesen Vorgang bezeichnet man als *Smart lifting*. Bevor

erläutert wird, wie *Smart lifting* funktioniert, soll im nächsten Abschnitt zunächst dargestellt werden, wie Bindungen auf Rollenseite bzw. Basisseite vererbt werden können.

3.6 Vererbung von Bindungen

Bindungen können sowohl zwischen Superrolle und Subrolle, als auch zwischen Superbasisklasse und Subbasisklasse vererbt werden.
Das heißt, jede *Rolle* mit einer bestimmten *Basisklasse* kann auch von deren Subklasse eingenommen werden und hat dann die gleichen Bindungen wie zur Oberklasse. Das gilt allerdings nicht für *Callin Bindungen* zu privaten oder statischen Basismethoden. Da private oder statische Methoden in Java nicht vererbt werden, werden auch Bindungen zu ihnen nicht vererbt.

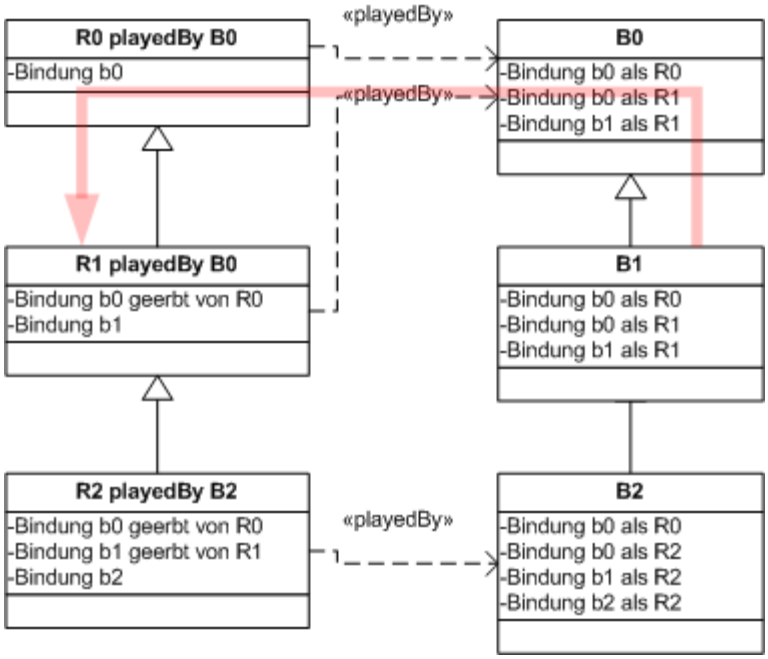

Abbildung 3.4: Bindungsvererbung und Smart Lifting

Dieses Verhalten wird in Abbildung 3.4 dargestellt. Hierbei wird davon ausgegangen, dass sich alle *Rollen* innerhalb eines *Teams* befinden.
Es existieren drei Basisklassen `B0`, `B1` und `B2`. Diese stehen in einer Generalisierungsbeziehung, d.h. es gilt `B2` ist Subklasse von `B1` ist Subklasse von `B0`.
Gleiches gilt für die drei *Rollen* `R0`, `R1` und `R2`. `R0` vererbt nun seine Bindung `b0` an `R1` und `R2`. Auch `R1` vererbt seine Bindung an `R2`. `B1` hat keine direkte Bindung zu einer *Rolle*, erhält diese aber durch die Generalisierungsbeziehung zu `B0`.

`B2` erhält zusätzlich zu `b0` alle geerbten und selbst definierten Bindungen von `R2`, also `b1` und `b2`.

3.6.1 Smart lifting

Wie in 3.5 beschrieben, muss bei Vererbung von Bindungen eine intelligente Auswahl getroffen werden, zu welcher Rolle ein Basisobjekt geliftet werden soll.
Diese Auswahl funktioniert nach folgendem Prinzip:
Es wird die speziellste Rolle verwendet, die für ein Basisobjekt möglich ist. Dazu wird zunächst überprüft, welche Rollen eine `playedBy` Relation zu dieser Basisklasse haben. Dabei wird nicht der statische, sondern der dynamische Typ des Basisobjekts verwendet. Werden keine gefunden, so wird dieser Vorgang mit der Superklasse der Basisklasse wiederholt. Ist nun innerhalb der Basishierarchie die speziellste Basisklasse gefunden, zu der eine direkte (nicht vererbte) `playedBy` Relation besteht, wird in der Rollenhierarchie in umgekehrter Richtung nach der speziellsten Rolle gesucht, die für diese Basisklasse möglich ist.
Dieses Verhalten ist durch den roten Pfeil in Abbildung 3.4 angedeutet. Hierbei soll ein Basisobjekt des dynamischen Typs `B1` geliftet werden. Da keine Rolle existiert, die eine direkte `playedBy` Beziehung zu B1 hat, muss in der Vererbungshierarchie von `B1` nun zunächst eine Klasse gesucht werden, zu der eine direkte Relation besteht. Diese Klasse ist `B0`. Nun wird innerhalb der Rollenhierarchie die speziellste *Rolle* gesucht, die `B0` als Basisklasse hat. Diese *Rolle* ist `R1`. Das bedeutet, dass die Rolle `R0` implizit abstrakt ist, da von ihr nie Objekte erzeugt, sondern Basisobjekte des Typs `B0` oder `B1` immer zu `R1` geliftet werden.

3.7 Weitere Bindungen

Neben den erwähnten *Callin Bindungen* können auch noch weitere Bindungen zur Basisklasse existieren. Diese Bindungsarten sollen im Folgenden dargestellt werden.

3.7.1 Callout Bindung

Über *Callout Bindungen* können abstrakte Rollenmethoden an Methoden der *Basisklasse* gebunden werden. Ein Aufruf der Rollenmethode wird dann direkt an die Methode der Basisklasse weitergeleitet.
Auch ist es über *Callout Bindungen* möglich Getter und Setter für Felder der Basisklasse bereitzustellen. Dies geschieht über die Schlüsselwörter `get` und `set`.
Callout Bindungen werden über den *OT/J* spezifischen Operator `->` abgebildet.

```
1.  public team MyTeam {
2.      public class MyRole playedBy MyBase {
3.          void roleMethod() -> void baseMethod();
4.          void setI(int i) -> set i;
5.          int getI() -> get i;
6.      }
7.  }
8.
9.  public class MyBase {
10.     public int i;
11.     public void baseMethod() {
12.         //do something
13.     }
14. }
```

In diesem Beispiel wird die Methode `roleMethod` der *Rolle* `MyRole` an die Methode `baseMethod` der *Basisklasse* gebunden. Jeder Aufruf von `roleMethod` wird direkt an `baseMethod` weitergeleitet. Außerdem kann durch die Methoden `getI` und `setI` auf das Feld `i` der Klasse `MyBase` zugegriffen werden.

3.7.2 Decapsulation

In einer *Rolle* ist es möglich, die Sichtbarkeitsregeln der Basismethoden und -felder aufzuheben, also auf Methoden und Felder zuzugreifen, die für die Rolle nicht sichtbar sind. Dieses Prinzip wird als *Decapsulation* bezeichnet. Zum einen können *Callout Bindungen* auf nicht sichtbare Felder und Methoden definiert werden, zum anderen kann innerhalb einer *Callin Bindung* ein *Base call* zu einer nicht sichtbaren Methode der Basisklasse erfolgen.

3.7.3 Parameter Mapping

Obwohl bei *Callout Bindungen* eine Methode auf eine andere weitergeleitet wird, muss die Rollenmethode nicht dieselbe Signatur haben wie die Methode der Basisklasse. Das wird durch *Parameter mapping* ermöglicht. Das *Parameter mapping* wird in einem Block definiert, der durch das Schlüsselwort `with` eingeleitet wird.
Für das *Mapping* können sowohl alle Identifier, die in diesem Scope sichtbar sind, als auch Konstanten benutzt werden. Weiterhin kann mit dem Schlüsselwort `result` auch der Rückgabewert gemappt werden:

```
1.  public team MyTeam {
2.     public class MyRole playedBy MyBase {
3.         int roleMethod(int i, int j) ->
4.                 String baseMethod(int k) with {
5.             i + j -> k,
6.             result <- new Integer(result).intValue()
7.         }
8.     }
9. }
```

Auch für *Callin Bindungen* kann *Parameter mapping* verwendet werden. Die Syntax ist dabei nahezu identisch zu den *Callout Bindungen*. Der einzige Unterschied besteht darin, dass die Richtung der Pfeile umgekehrt wird, d.h. für das *Mapping* der Parameter wird `<-` und für den Rückgabewert `->` benutzt.

Nachdem nun beschrieben wurde, welche neuen Features *OT/J* bietet, soll im nächsten Kapitel dargelegt werden, wie diese Sprachfeatures momentan technisch umgesetzt werden.

4 Umsetzung der Aspekte bei OT/J

In diesem Kapitel soll beschrieben werden, wie die Features von *OT/J* momentan technisch umgesetzt werden. Hierbei sollen nur die Sprachelemente betrachtet werden, die sich nicht ausschließlich im *Team* umsetzen lassen. Dies ist zum einen bei *Callin Bindungen* (s. 3.2) und zum anderen bei *Decapsulation* (s. 3.7.2) der Fall. Alle anderen Features werden ausschließlich in den *Teams* umgesetzt.
Um diese Features umzusetzen verwendet *OT/J Aspektweben zur Ladezeit* (s. 2.2.1). Wie genau die Basisklassen dazu redefiniert werden müssen, wird im Folgenden gezeigt. Zunächst wird die Umsetzung der *Callin Bindungen* beschrieben. Anschließend (4.2) wird dargelegt, wie die *Decapsulation* realisiert wird (4.2).

4.1 Callin Bindungen

Zunächst soll hier beschrieben werden, welche Schritte bei der Abarbeitung der *Callin Bindungen* zu einer Methode durchgeführt werden müssen. Dazu stellt Abbildung 4.1 diesen Vorgang schematisch dar. Anschließend werden aus diesem Schema die in OT/J verwendeten Methoden abgeleitet.

Die grundsätzliche Vorgehensweise für die Abarbeitung ist uns schon aus 3.4 bekannt. Diese wird hier noch verfeinert und mit technischen Aspekten angereichert.

1. Zunächst müssen beim Aufruf einer, durch beliebig viele *Callin Bindungen* gebundenen, Basismethode die aktiven Teams, die Bindungen zu dieser Basismethode haben, bestimmt werden.
2. Falls solche Teams existieren, muss nun das Team mit der höchsten Priorität bestimmt werden, sonst wird direkt der Originalcode der Basismethode aufgerufen.
3. Als nächstes werden alle *Before Bindungen* des Teams zu dieser Methode ausgeführt.

4. Jetzt muss die *Replace callin Bindung* mit der höchsten Priorität bestimmt werden.
5. Hat das Team keine *Replace Callin Bindungen*, muss das Team mit der nächst niedrigeren Priorität bestimmt werden, es wird also wieder Schritt 2 ausgeführt, sonst wird die *Replace callin Bindung* mit der höchsten Priorität ausgeführt.
6. Falls die zu der Bindung gehörende Rollenmethode einen *Base call* enthält, werden die weiteren *Replace Callin Bindungen* dieses Teams ausgeführt. Die Ausführung wird mit Schritt 4 fortgesetzt.
   Falls sie keinen *Base call* enthält werden nun noch alle *After callin Bindungen* dieses Teams und aller Teams mit höherer Priorität ausgeführt.

Dieses Schema verdeutlicht zwei wesentliche Punkte.
Zum einen wird Code benötigt, der als Einstiegspunkt in die *OT/J* Logik dient. Für jede gebundene Methode müssen hier die aktiven Teams bestimmt werden und die weitere OT/J Logik aufgerufen werden. Dieser Code wird als *initialer Dispatch* bezeichnet.

Zum anderen läuft ein großer Teil der Logik in einer Schleife ab (Schritt 2 bis Schritt 6). Hier werden für alle Teams alle Bindungen ausgeführt. Diese Schleife bricht ab, wenn entweder alle Bindungen ausgeführt wurden oder eine Rollenmethode, die durch eine *Replace Callin Bindung* gebunden ist, keinen *Base call* hat. Diese Schleife wird als *Chaining* bezeichnet.
Der Code, der diese beiden Punkte implementiert, wird *Dispatch Logik* genannt.

Wie diese *Dispatch Logik* technisch umgesetzt werden, soll im Folgenden gezeigt werden.

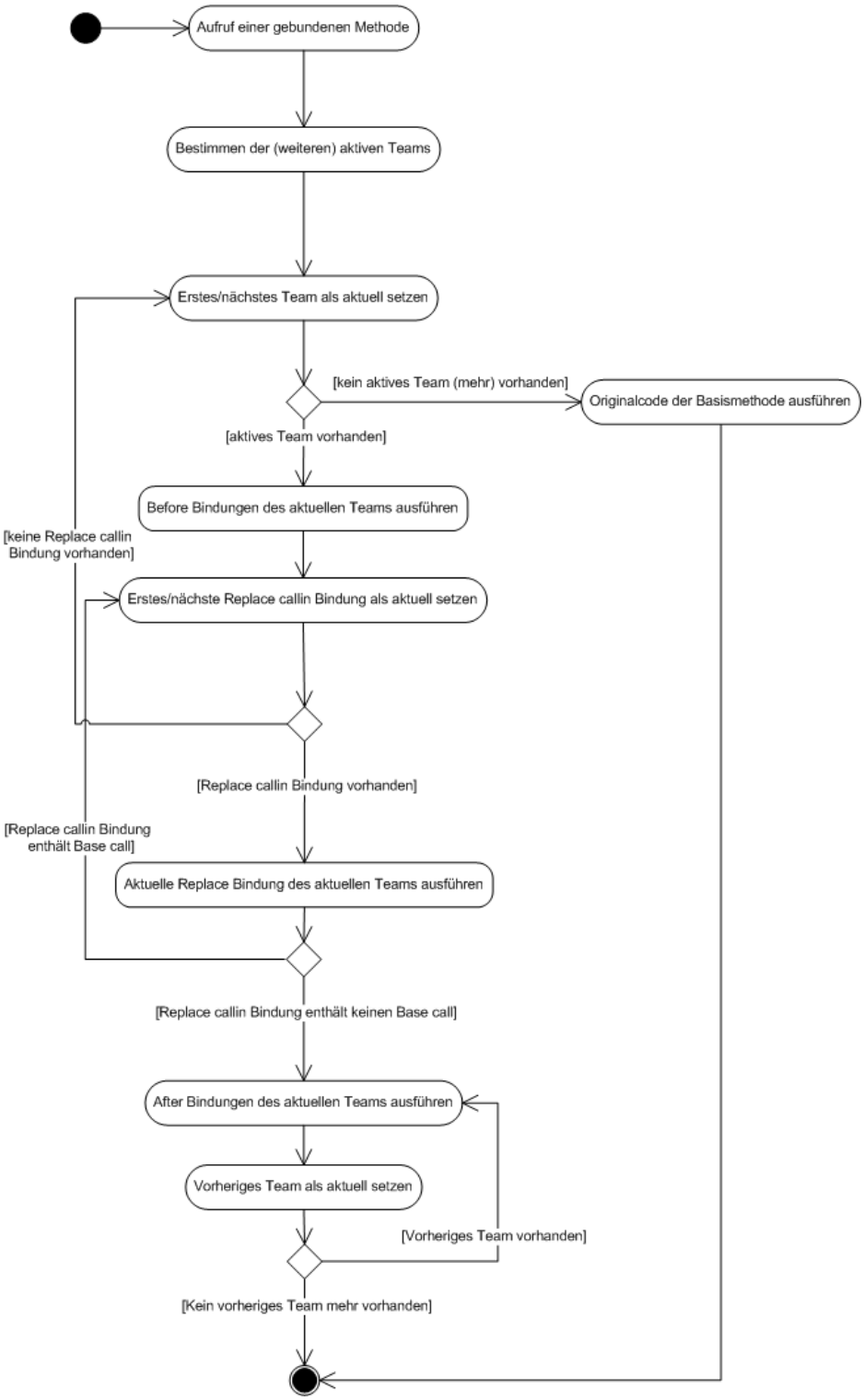

Abbildung 4.1: Abarbeitung von Callin Bindungen für eine Methode

4.1.1 Initialer Dispatch

Der *initiale Dispatch* geschieht in der Basismethode. Diese wird dazu zum sog. *Initial wrapper* redefiniert. Der Originalcode der Basismethode wird in eine neue Methode verschoben. Dies ist nötig, da es in der *Dispatch Logik* möglich sein muss, auf den Originalcode zuzugreifen (*Base call*). Hierbei darf dann natürlich kein weiterer Aufruf von Bindungen geschehen.
Sei

RType `bm(`*PType*$_1$ `a`$_1$`, ..., `*PType*$_n$ `a`$_n$`)`

eine Basismethode, so ist diese neue Methode folgendermaßen deklariert:

`public` *RType* `_OT$bm$orig(`*PType*$_1$ `a`$_1$ `, ... , `*PType*$_n$ `a`$_n$`)`

Das Präfix `_OT$` hat dabei eine besondere Bedeutung. Der *OT/J* Compiler stellt sicher, das Identifier, die mit diesem Präfix beginnen, in manuellem Code nicht verwendet werden können. Er sorgt also dafür, dass Methoden, deren Namen mit diesem Präfix beginnt, nicht überschrieben und nicht aufgerufen werden können.
Die Methode `bm` wird nun, wie folgt, zum *Initial wrapper* redefiniert.

```
1.  RType bm(AType1 arg1, ... , ATypen argn) {
2.      Team _OT$teams[] =
3.              new Team[_OT$activeTeams.length];
4.      int _OT$teamIDs[] =
5.              new int[_OT$activeTeamIDs.length];
6.      for (int i = 0; i < _OT$activeTeams.length; i++) {
7.          _OT$teams[i] = _OT$activeTeams[i];
8.          _OT$teamIDs[i] = _OT$activeTeamIDs[i];
9.      }
10.     Object args[] = {arg1, ... , argn};
11.     return _OT$bm$chain(_OT$teams, _OT$teamIDs, 0, 0,
12.             args);
13. }
```

Hier wird also der Schritt 1 aus Abbildung 4.1 durchgeführt.
Der Code des *Initial wrapper* ist dabei vereinfacht dargestellt.
Zusätzlich zu dieser Methode werden in der Basisklasse die Felder

```
protected static int _OT$activeTeamIDs[];
protected static Team _OT$activeTeams[];
```

definiert. Bei seiner Aktivierung bzw. Deaktivierung trägt sich ein Team in das Feld `_OT$activeTeams` ein bzw. wieder aus. Dazu werden in der Basisklasse die Methoden

```
public static void _OT$addTeam (Team team , int teamID);
public static void _OT$removeTeam (Team team);
```

definiert.
In dem Feld `_OT$activeTeamIDs` wird für jedes Team eine, von der OT/J-Laufzeitumgebung generierte, eindeutige Id abgelegt. Wie diese verwendet wird, wird im nächsten Abschnitt gezeigt.

Im *Initial wrapper* werden jetzt Arbeitskopien von `_OT$activeTeams` und `_OT$activeTeamIDs` angelegt und mit ihnen in Zeile 11 der *Chaining wrapper* aufgerufen.

4.1.2 Chaining wrapper

Auch das *Chaining* wird in der Basisklasse durchgeführt, im sog. *Chaining Wrapper*.
Um zu zeigen, wie dieser *Chaining wrapper* aufgebaut ist, soll hier zunächst der Begriff *Callin Bindung* genauer definiert werden:

Eine Bindung *b* ist ein Tupel *(Team, Role, RoleMethod, BaseClass, BaseMethod, Kind)* aus folgenden Bestandteilen:

1. *Team* Das Team, zu der die Rolle gehört, die eine Bindung zu ihrer Basisklasse hat.
2. *Role* Die Rolle, in der die Bindung definiert ist.
3. *RoleMethod* Die gebundene Methode der Rolle
4. *BaseClass* Die Basisklasse der Rolle
5. *BaseMethod* Die gebundene Methode der Basisklasse
6. *Kind* Der Typ der Bindung (*before*, *after*, *replace*)

Außerdem werden für eine Basisklasse *B'* mit der gebundenen Methode *bm'* die Mengen $BEFORE_{B,bm'}$, $AFTER_{B,bm'}$ und $REPLACE_{B,bm'}$, wie folgt, definiert:

$$BEFORE_{B'bm'}/AFTER_{B'bm'}/REPLACE_{B'bm'} =$$
$$\{ b = (T, R, rm, B, bm, before/after/replace \mid B = B' \wedge bm = bm'\}.$$

Die Menge $ACTIVE\_TEAM\_IDS_{B'}$ soll die Ids der aktiven Teams für die Basisklasse *B'* enthalten. ´

Zusätzlich definieren wir die Funktionen

teamId: Team -> Int,
bindingId: Binding -> Int

die jedem Team bzw. innerhalb eines Teams einer Bindung eine eindeutige Id zuordnet.

Dann wird der *Chaining wrapper* wie folgt implementiert:

```
1.  public Object _OT$bm'$chain(Team _OT$teams[],
2.          int _OT$teamIDs[], int _OT$idx, int _OT$bindIdx,
3.          Object _OT$args[]) {
4.      Object _OT$result = null;
5.      if(_OT$idx >= _OT$teams.length) {
6.          _OT$bm'$orig();
7.          return null;
8.      }
9.      Team _OT$team = _OT$teams[_OT$idx];
10.     if (_OT$bindIdx == 0) {
11.         switch (_OT$teamIDs[_OT$idx]) {
12.         ∀ id ∈ ACTIVE_TEAM_IDS_B'
13.         case id:
14.             ∀ b=(T,R,rm,B',bm',before) ∈ BEFORE_B',bm',
                    wobei teamId(T)=id
15.             ((T)_OT$team)._OT$R$rm$bm'(this, _OT$args);
16.             break;
17.         }
18.     }
19.     switch (_OT$teamIDs[_OT$idx]) {
20.     ∀ id ∈ ACTIVE_TEAM_IDS_B'
21.     case id:
22.         switch (_OT$bindIdx) {
23.         ∀ b=(T,R,rm,B',bm',replace) ∈ REPLACE_B',bm',
                    wobei teamId(T)=id
24.         case bindingId(b):
25.             _OT$result = ((T)_OT$team)._OT$R$rm$bm'(this,
26.                     _OT$teams, _OT$teamIDs, _OT$idx,
27.                     _OT$bindIdx + 1, 0, _OT$args);
28.             break;
29.         default:
30.             _OT$result = _OT$bm'$chain(_OT$teams,
31.                     _OT$teamIDs, _OT$idx + 1, 0,
32.                     _OT$baseMethTag, _OT$args);
33.             break;
34.         }
35.         break;
36.     default:
37.         _OT$result = _OT$bm'$chain(_OT$teams,
38.                     _OT$teamIDs, _OT$idx + 1, 0,
39.                     _OT$baseMethTag, _OT$args);
40.         break;
41.     }
```

```
42.     if (_OT$bindIdx == 0) {
43.         switch (_OT$teamIDs[_OT$idx]) {
44.         ∀ id ∈ ACTIVE_TEAM_IDS_B'
45.         case id:
46.             ∀ b=(T, R, rm, B', bm', after) ∈ AFTER_B',bm',
                      wobei teamId(T)=id
47.             ((T)_OT$team)._OT$R$rm$bm'(this, _OT$args);
48.             break;
49.         }
50.     }
51.     return _OT$result;
52. }
```

In diesem *Chaining wrapper* werden die Schritte 2 - 7 (bis auf den *Base call*) aus Abbildung 4.1 durchgeführt.
Zunächst wird in Zeile 5 überprüft, ob noch aktive *Teams* vorhanden sind (Schritt 3). Ist das nicht der Fall wird der Originalcode der Basismethode aufgerufen. Wenn noch weitere *Teams* existieren, wird über den Teamindex zunächst das aktuelle Team bestimmt. Beim initialen Aufruf des *Chaining Wrappers* ist dieser Index immer 0. Als nächstes werden in den Zeilen 10 - 18 alle *Before callin Bindungen* des aktuellen *Teams* in der richtigen Reihenfolge (s. 3.4) abgearbeitet. Um zu bestimmen, welche Bindungen das sind, wird die Team Id verwendet. Diese Id stellt lediglich eine günstigere Alternative zum `instanceof` Operator dar, mit dem diese Bestimmung genauso möglich wäre.
Auch bei den *Replace* und *After Callin Bindungen* wird diese Team Id verwendet.

Falls ein Team mehrere *Replace callin Bindungen* enthält, muss sichergestellt sein, dass die *Before* und *After callin Bindungen* nicht für jeden *Replace callin* wieder ausgeführt werden. Das geschieht mit der Überprüfung des Bindingindex in Zeile 10. Auch dieser ist beim initialen Aufruf 0.

Danach werden in den Zeilen 19 - 41 die *Replace callin Bindungen* des Teams aufgerufen. Zunächst wird natürlich nur die *Callin Bindung* mit der höchsten Priorität (s. 3.4) ausgeführt. Nur wenn diese einen *Base call* enthält, wird die Ausführung der *Replace callin Bindungen* fortgeführt.
Enthält die aufgerufene Rollenmethode einen *Base call*, so ruft sie wieder den *Chaining wrapper* mit dem um eins inkrementierten Bindingindex auf. So entsteht eine Rekursion, die die in 4.1 beschriebene Schleife abbildet.
Die Abbruchbedingung dieser Rekursion stellt genau die Überprüfung in Zeile 5 dar.
Falls es keine weitere *Replace Callin Bindung* in diesem Team gibt, also kein `case` Fall für den Bindingindex gefunden wird, ruft sich der *Chaining wrapper* selbst mit dem um 1 inkrementierten Teamindex auf. Dann werden entsprechend die Bindungen des nächsten Teams ausgeführt.

Nach der Abarbeitung der *Replace callin Bindungen* werden schließlich alle *After callin bindungen*, geordnet nach Priorität, ausgeführt.

Im nächsten Abschnitt soll nun gezeigt werden, wie die *Decapsulation* technisch umgesetzt wird.

4.2 Decapsulation

Um Zugriff auf ein Feld `f` vom Typ *TYPE* einer Basisklasse B zu nehmen, das für das *Team* bzw. die *Rolle* nicht sichtbar ist, werden in der Basisklasse die Methoden

```
public static TYPE _OT$get$f(B base_obj)
```

für lesenden Zugriff und

```
public static void _OT$set$f(B base_obj, TYPE new_value)
```

für schreibenden Zugriff erzeugt. Diese bekommen eine Instanz der Basisklasse übergeben, die in der Rolle gespeichert ist. Über diese Instanz kann dann innerhalb der Basisklasse das Feld gesetzt bzw. ausgelesen werden.

Der Zugriff auf nicht sichtbare Methode erfolgt dagegen dadurch, dass die Sichtbarkeit der Methode auf `public` geändert wird.

Nachdem nun gezeigt wurde, welche Veränderungen in der Basisklasse nötig sind, um *Callin Bindungen* und *Decapsulation* umzusetzen, soll im nächsten Abschnitt nun dargelegt werden, mit welcher Technik die Aspekte gewoben werden.

4.3 Weben der Aspekte zur Ladezeit

ObjectTeams/Java webt seine Aspekte zur Ladezeit der Klassen. Es existieren zwei verschiedene Varianten, mit welcher Technik die Aspekte gewoben werden. Dabei kann vom Benutzer eingestellt werden, welche der beiden Techniken verwendet werden soll.

4.3.1 JMangler

JMangler ist ein Framework zur Veränderung von Klassen zur Ladezeit. Dazu definiert *JMangler* einen eigenen *Class loader*. In diesem *Class loader*, wird das Framework aufgerufen. Um nun eigene Transformationen mit *JMangler* ausführen zu können, müssen sog. *Transformer* implementiert werden. Dabei existieren zwei verschiedene Arten von *Transformern*:

1. *Interface Transformer*: Durch diese *Transformer* werden Signaturen von Klassen oder Methoden verändert.
2. *Code Transformer*: *Transformer* dieses Typs verändern lediglich die Implementierung von Methoden.

Um solche *Transformer* zu implementieren, existieren die Interfaces `InterfaceTransformerComponent` und `CodeTransformerComponent`, die von den *Transformern* implementiert werden müssen. Das Framework sorgt dann soweit wie möglich dafür, dass diese *Transformer* in der richtigen Reihenfolge beim Laden einer Klasse aufgerufen werden. Dieser Prozess ist in Abbildung 4.2 schematisch dargestellt.

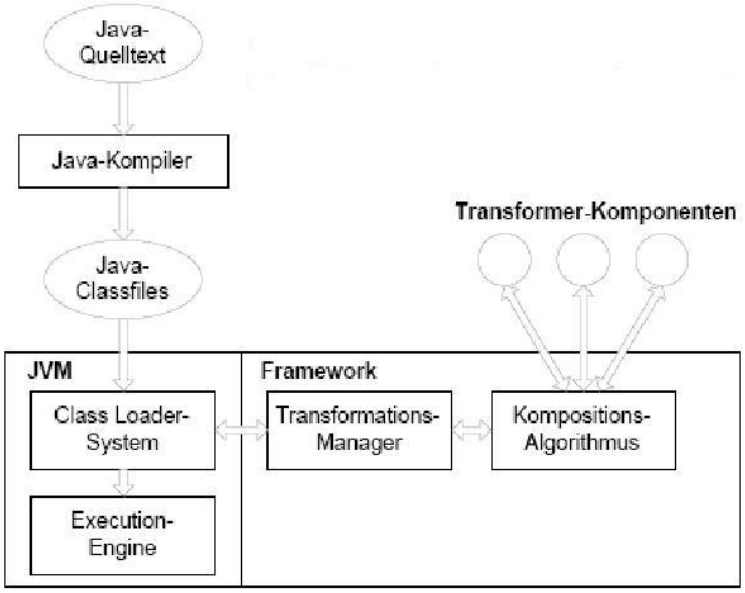

Abbildung 4.2: Transformationsprozess von JMangler Quelle: [Aus00]

In *ObjectTeams/Java* existieren diverse *Transformer* (z.B. für das Hinzufügen der Arrays, die die aktiven Teams speichern oder das Redefinieren einer gebunden Methode), über die die nötigen Redefinitionen vollzogen werden.

Die Benutzung von *JMangler* hat allerdings einen gravierenden Nachteil:
Wie gesagt, definiert *JMangler* einen eigenen *Class loader.* Da aber, wie in 2.2.1, zumindest die erste Klasse einer Applikation vom *Application class loader* geladen wird, ist es mit *JMangler* nicht möglich, Klassen, die vom *Extension* oder *Bootstrap class loader* geladen werden, zu redefinieren. Um diesen Nachteil zu beheben, wurde für *OT/J* eine zweite Variante geschaffen, wie die nötigen Redefinitionen vorgenommen werden können.

4.3.2 JPLIS

Die zweite Variante benutzt *JPLIS* (siehe 2.2.1), um die Transformationen zu vollziehen.

Dazu existiert die Klasse `otreAgent`, die als *Agent* fungiert und die nötige `premain` Methode implementiert. In dieser Klasse wird eine Instanz der Klasse `ObjectTeamsTransformer` als aufzurufender `ClassFileTransformer` registriert. Im `ObjectTeamsTransformer` werden dann über Delegation an weitere Klassen alle nötigen Transformationen erledigt.
Da die Benutzung von *JPLIS* keinen eigenen *Class loader* mehr erfordert, können hier auch Klassen redefiniert werden, die vom *Bootstrap* oder *Extension class loader* geladen werden.

4.3.3 Nachteile

Da beide Varianten die Aspekte zur Ladezeit weben, haben sie beide den Nachteil, dass die Ladereihenfolge von Basis- und Teamklassen beachtet werden muss.
Grundsätzlich gilt, dass Teamklassen vor den Basisklassen geladen werden müssen, damit beim Laden der Basisklassen schon feststeht, ob, und wenn ja, durch welche *Teams* bzw. *Rollen* sie gebunden werden.
Das stellt allerdings ein grundsätzliches Problem dar, da Klassen durch die *JVM* nach dem Prinzip des *Lazy loadings*, also genau dann, wenn sie benötigt werden, geladen werden. Das heißt es kann nicht sichergestellt werden, dass *Teams* wirklich vor den Basisklassen geladen werden.
Auch wenn bei *OT/J* inzwischen einige Mittel eingesetzt werden (z.B. Referenzen in Klassen, die Teams benutzen, auf die Teams), die dieses Problem abmildern, existieren Beispiele, in denen das Weben der Aspekte mit einer Fehlermeldung aufgrund der falschen Ladereihenfolge abbricht.

Außerdem wird bei diesem Ansatz davon ausgegangen, dass zur Ladezeit von Klassen überhaupt schon alle Teams bekannt sind, von denen sie adaptiert werden könnten. Besonders in hoch dynamischen Kontexten (wie z.B. 7.1) muss dies aber keineswegs der Fall sein.

Das Ziel muss es also sein, eine Strategie zu finden, wie Aspekte in jeder Situation umgesetzt werden können. Das nächste Kapitel zeigt die verschiedenen möglichen Ansätze dazu auf und bewertet ihre Einsetzbarkeit für *OT/J*.

4.4 Bewertung der Alternativen zur bisherigen Webestrategie

An dieser Stelle soll eine Bewertung der in Kapitel 2 vorgestellten Ansätze zur technischen Umsetzung von aspektorientierten Sprachen im Kontext von OT/J stattfinden.

Dafür ist es zunächst notwendig Kriterien aufzustellen, anhand derer diese Strategien bewertet werden können. Anschließend soll die eigentliche Bewertung erfolgen.

4.4.1 Kriterien zur Bewertung

Bei der Bewertung der verschiedenen Techniken zur technischen Umsetzung von *ObjectTeams/Java* sind verschiedene Kriterien zu beachten.

Am wichtigsten ist natürlich, dass alle Features von *OT/J* mit einer Technik auch umgesetzt werden können. Dies wird mit dem Kriterium *Korrektheit* ausgedrückt. Da dies das wichtigste Kriterium ist, wird es mit 40% gewichtet.
Um für den produktiven Einsatz tauglich zu sein, muss eine Technik es ermöglichen, effizient mit ihr zu arbeiten. Natürlich hängt die *Effizienz* nicht nur von der Technik selbst, sondern auch von der Implementierung ab, die auf dieser Technik aufbaut. Hier soll allerdings nur die *Effizienz* der Technik selbst bewertet werden. Dieses Kriterium wird mit 20% gewichtet.
Mit dem Kriterium *Wartbarkeit* soll bewertet werden, wie gut sich die Technik eignet, um an neue Anforderungen angepasst zu werden. Genau wie bei der *Effizienz*, hängt dies natürlich auch von der konkreten Implementierung ab. Dieses Kriterium ist vor allem für die (Weiter-)Entwicklung der *OT/J-Laufzeitumgebung* wichtig, für den Nutzer spielt es eine untergeordnete Rolle. Es soll deshalb nur mit 10% bewertet werden.
Ein großer Vorteil von Java ist die Plattformunabhängigkeit dieser Sprache. Durch den Einsatz der *JVM* (s. 2.1) wird gewährleistet, dass Java Programme ohne (oder nur mit wenig) Codeänderungen auch auf andere Plattformen portiert werden können. Da *OT/J* eine Erweiterung von Java darstellt, soll dies auch für *OT/J* Programme gelten. Mit dem Kriterium *Portabilität* soll diese Eigenschaft für die Techniken überprüft werden. Die *Portabilität* wird ebenfalls mit 10% gewichtet.
Der letzte zu bewertende Punkt ist die *Benutzerakzeptanz*. Hiermit soll bewertet werden, vor welche Hürden eine Technik den Benutzer stellt. Die *Benutzerakzeptanz* ist ein entscheidendes Kriterium für die produktive Verwendung von OT/J und soll deshalb mit 20% gewichtet werden.

4.4.2 Bewertung

Vor der eigentlichen Bewertung sollen an dieser Stelle noch einmal alle Techniken, die bewertet werden sollen, und alle Kriterien, mit denen bewertet wird, aufgelistet werden. Die Nummerierung wird dabei auch bei der Bewertung in Tabelle 4.1 verwendet. Die Bewertung der einzelnen Techniken nach den einzelnen Kriterien erfolgt dabei anhand von Schulnoten:

Techniken

1. *Aspektweben mit JMangler* (s. 2.1.4 u. 4.3.1)
2. *Aspektweben mit JPLIS* (s. 2.2.2)
3. *Aspektweben mit JDI* (s. 2.4.1)
4. *Aspektweben mit JVM TI* (s. 2.4.1)
5. *Delegation mit Proxies (delegationsbasiert)* (s. 2.3)
6. *Metaobject protocol mit JDI* (s. 2.4.1)
7. *Metaobject protocol mit JVM TI* (s. 2.4.1)

Kriterien

1. *Korrektheit*
2. *Effizienz*
3. *Wartbarkeit*
4. *Portabilität*
5. *Benutzerakzeptanz*

Techniken / Kriterien	Gewichtung	1.	2.	3.	4.	5.	6.	7.
1.	0,4	3	1	1	1	1	4	4
2.	0,2	2	3	4	3	1	4	3
3.	0,1	3	2	2	2	5	1	1
4.	0,1	2	2	2	5	6	2	5
5.	0,2	2	2	2	2	4	2	2
Gesamt	1	2,5	1,8	2	2,1	2,5	3,1	3,2

Tabelle 4.1: Vergleich der Techniken zur Umsetzung von OT/J

Wie in Tabelle 4.1 zu sehen ist, liegen alle Techniken relativ dicht beieinander, die Spanne reicht von 1,8 bis 3,2. Dementsprechend kann auch keine Technik als völlig ungeeignet bewertet, alle Techniken würden grundsätzlich in Frage kommen.

Im Folgenden soll dargelegt werden, wie die einzelnen Bewertung zustandekommen.

Aspektweben mit JMangler

Der größte Schwachpunkt des *Aspektwebens mit JMangler* liegt darin, dass hierbei Klassen nur zur Ladezeit gewoben werden können. Dadurch lassen sich, wie in 4.3.3 beschrieben, die Aspekte nicht in allen Konstellationen weben, da Teams immer vor den Basisklassen geladen werden müssen. Dies lässt sich aber nicht immer garantieren, weswegen diese Technik für das Kriterium *Korrektheit* nur die Bewertung befriedigend erhält. Diese Schwachstelle ist verantwortlich dafür, dass diese Technik in Zukunft nicht mehr eingesetzt werden soll.
Der zweite Punkt, bei dem diese Technik eine leichten Abzug erhält, ist die Wartbarkeit. Da bei dieser Technik eine externe, nicht standardisierte Bibliothek, nämlich *JMangler*, eingesetzt wird, kann die Wartung aufwendiger werden, wenn Fehler in der Bibliothek gefunden werden bzw. neuen Anforderungen Änderungen in der Bibliothek erfordern. Auch für dieses Kriterium erhält diese Technik ein befriedigend. Bei allen anderen Kriterien dagegen wurde diese Technik gut bewertet.

Aspektweben mit JPLIS

Das *Aspektweben mit JPLIS* hat im Gegensatz zu *JMangler* den Vorteil, dass hierbei Aspekte nicht nur zur Ladezeit, sondern auch zur Laufzeit gewoben werden können. Mit dieser Technik lassen sich Aspekte dann auch unabhängig von der Ladereihenfolge von *Teams* und Basisklassen weben und damit alle Features von *OT/J* umsetzen. Aus diesem Grund erhält diese Technik für das Kriterium Korrektheit ein sehr gut.
Allerdings kostet das Redefinieren von Klassen zur Laufzeit Performance, weswegen Implementierungen, die mit *JPLIS* arbeiten, grundsätzlich langsamer sein werden , als solche, die mit *JMangler* arbeiten. Dagegen wird mit *JPLIS* keine externe Bibliothek, sondern eine standardisierte Schnittstelle von Java verwendet, was die Wartbarkeit erhöht.

Aspektweben mit JDI

Das *Aspektweben mit JDI* bietet im Wesentlichen die gleichen Vor- und Nachteile wie die Verwendung von *JPLIS*. Der einzige Unterschied besteht darin, dass ein Programm für die Benutzung von *JDI* im *Debug Modus* laufen muss. Dieser Umstand verschlechtert allerdings die Performance[1]. Dadurch erhält *JDI* beim Kriterium *Effizienz* nur die Bewertung ausreichend.

Aspektweben mit JVM TI

Der große Nachteil von *JVM TI*, ist die nötige Verwendung von nativen Bibliotheken. Dadurch ist der Code, der *JVM TI* verwendet, nicht mehr leicht auf andere Plattformen zu portieren, weswegen diese Technik für das Kriterium *Portabilität* ein mangelhaft erhält. Auch die Performance ist hier nur leicht besser als bei *JDI.* Wird *JVM TI* genutzt müssen die Programme zwar nicht im Debug Modus laufen, allerdings kostet das Redefinieren von Klassen zur Laufzeit Zeit.

Delegation mit Proxies (delegationsbasiert)

Eine sehr interessante Alternative stellt die *delegationsbasierte* Umsetzung der Aspekte von *OT/J* dar. Dieser Ansatz ist wohl der effizienteste, da die *JVM* selbst so implementiert ist, dass dort direkt die Aspekte umgesetzt werden können. Teures *Aspektweben* wäre hier nicht mehr nötig. Auch ließen sich mit dieser Technik alle Features von OT/J umsetzen.
Leider existiert bis jetzt noch keine standardisierte *JVM*, in der Aspekte direkt umgesetzt werden können. Auch proprietäre Lösungen kommen momentan nicht über einen Prototyp Status hinaus. Es wäre also notwendig hier eine eigene *JVM* zu entwickeln oder eine bestehende Implementierung anzupassen. Dies würde die Wartbarkeit deutlich senken, da bei Fehlerbehebungen bzw. neuen Anforderungen unter Umständen auch die Implementierung der *JVM* angepasst werden müsste, was zu einem deutlichen Mehraufwand führt.

---

[1] Der Faktor, um den sich die Performance verschlechtert, hängt dabei stark vom Kontext ab.

Außerdem wären die *OT/J-Laufzeitumgebung* nicht mehr auf andere Plattformen zu portieren, ohne auch für diese Plattformen eine *JVM* zu entwickeln.
Schließlich würde es auch die Benutzerakzeptanz senken, da die Einführung einer proprietären *JVM* eine große Hürde für die Benutzung von *OT/J* darstellt.

Metaobject protocol mit JDI bzw. JVM TI

Die Umsetzung der Aspekte mittels *JDI* bzw. *JVM TI* hat zwei gravierende Nachteile. Zum einen lassen sich nicht alle Features von OT/J umsetzen. Es ist zwar möglich, vor oder nach dem Aufruf einer Methode zusätzlichen Code auszuführen (*Before* bzw. *After callin Bindungen*), allerdings lassen sich Methodenaufrufe nicht unterbinden. Jede Rollenmethode, die durch eine *Replace callin Bindung* gebunden ist, müsste also implizit einen *Base call* enthalten.
Außerdem ist das Abfangen von Methodenaufrufen über diese beiden Interfaces eine teure Operation. Selbst die *JVM TI* Spezifikation empfiehlt für performancekritische Anwendungen andere Ansätze zu wählen (s. [Sun04]). Zusätzlich müsste ein Programm für den Einsatz von *JDI* im *Debug Modus* laufen, was die Performance zusätzlich verschlechtert.

Beim Einsatz von *JVM TI* käme noch der oben erwähnte Nachteil der fehlenden *Portabilität* hinzu.

4.4.3 Fazit

Grundsätzlich bietet die *Delegation mit Proxies* den interessantesten Ansatz, da hier die Aspekte sehr effizient und elegant umgesetzt werden könnten. Allerdings müsste hierfür eine neue *JVM* geschaffen werden. Dies muss standardisiert erfolgen, da sonst die Chancen auf einen produktiven Einsatz von *OT/J* geringer werden würden.
Somit scheidet dieser Ansatz momentan leider aus.

JVM TI steht im Gegensatz zur Grundidee von Java, nämlich der Plattformunabhängigkeit. Der Einsatz dieses Interfaces würde viel Aufwand für die Anpassung der *OT/J-Laufzeitumgebung* an verschiedenste Plattformen bedeuten.
Bei der Verwendung von *JDI* wäre die Performance einer neuen Implementierung der *OT/J-Laufzeitumgebung* nicht ausreichend, da die Programme, für die Aspekte definiert werden sollen, im *Debug Modus* laufen müssen.

Somit bleibt nur noch das *Aspektweben mit JPLIS* übrig. Auch in der bisherigen Implementierung der *OT/J-Laufzeitumgebung* wird bereits *JPLIS* eingesetzt. Allerdings werden hier Klassen nur zur Ladezeit gewoben. Die Laufzeitumgebung müsste also so angepasst werden, dass sie auch Laufzeitweben einsetzt.

Im nächsten Kapitel soll nun gezeigt werden, wie diese neue Laufzeitumgebung umgesetzt werden kann.

5 Weben von Aspekten zur Laufzeit für OT/J

Diese Kapitel zeigt auf, wie das Weben von Aspekten zur Laufzeit für *OT/J* umgesetzt werden kann.

Wie in 2.2.2 beschrieben, ist es beim Weben der Klassen zur Laufzeit nicht möglich Methoden hinzuzufügen. Würde man die aktuelle *OT/J-Laufzeitumgebung* 1:1 auf das Laufzeitweben umstellen, wäre aber genau das nötig. Beim Bekanntwerden einer Bindung zur Laufzeit, müssten in einer Basisklasse der *Chaining wrapper* (s. 4.1.2) und eine Methode, die den Code der Originalmethode enthält, generiert werden.
Es ist also nötig, die bestehende Umsetzung der Aspekte, so zu verändern, dass zur Laufzeit Bindungen hinzugefügt werden können, ohne neue Methoden zu definieren. Wie dies möglich ist soll im Folgenden gezeigt werden.

5.1 Generische Schnittstelle zwischen Teams und Basisklassen

Eine Möglichkeit, wie die Umsetzung von *Callin Bindungen* zur Laufzeit erfolgen kann, wird in [Flü06] aufgezeigt.

Dieses Konzept basiert auf einer generischen Schnittstelle zwischen *Teams*/*Rollen* und Basisklassen. Das heißt, es soll zum einen nicht mehr nötig sein, dass die Basisklassen Kenntnis von den *Teams* bzw. *Rollen* benötigen, durch die sie gebunden werden und zum anderen soll es nicht nötig sein, für jede gebundene Basismethode in der Basisklasse neue Methoden zu definieren. Es werden zwar weiterhin zusätzliche Methoden in der Basisklasse benötigt, diese sollen aber unabhängig von der Anzahl der Bindungen und den *Teams*, zu denen Bindungen bestehen, sein.

Auch dieses Konzept basiert, wie die bisherige Implementierung, auf dem grundsätzlichen Schema aus Abbildung 4.1. Im Detail weist es allerdings deutliche Unterschiede auf.
Weiterhin soll die Basismethode zum *Initial wrapper* (s. 4.1.1) redefiniert werden. Für den Originalcode der Methode kann dagegen keine neue Methode mehr angelegt werden. Hier wird also eine generische Methode benötigt, die den Code aller gebundenen Methoden einer Klasse enthält.
Auch darf es nicht mehr nötig sein, für jede gebundene Methode einen *Chaining wrapper* zu definieren. Für die Änderung des *Chaining wrappers* könnte man sich grundsätzlich zwei Varianten vorstellen:

1. Der *Chaining wrapper* wird als generische Methode in der Basisklasse definiert.
2. Der *Chaining wrapper* wird als generische Methode im *Team* definiert.

Grundsätzlich wären beide Ansätze möglich. Wird der *Chaining wrapper* aber im *Team* definiert, kann die Schnittstelle von der Basisklasse zu den *Teams* deutlich schlanker gehalten werden. Sie muss, wie wir in 5.3 sehen werden, nur eine Methode, nämlich die Methode `callAllBindings`, umfassen.
Da die Basisklassen in der neuen Umsetzung von *OT/J* keine Kenntnis mehr von den konkreten *Teams*, von denen sie gebunden werden, haben sollen, ist es nötig, den *Chaining wrapper* an einer allgemeinen Stelle zu definieren. Dafür bietet sich die Klasse `Team` (s. 3.1) an.
Der neue *Chaining wrapper* arbeitet dabei, genauso wie der bisherige, rekursiv. Ihm wird ein Array aller aktiven *Teams* für eine Basismethode übergeben, die dann nacheinander abgearbeitet werden. Für den initialen Aufruf des *Chaining wrappers* wird dabei das erste *Team*, also das *Team* mit der höchsten Priorität (s. 3.4), verwendet.

Da sich im neuen Konzept der *Chaining wrapper* in der Klasse `Team` befindet, muss also auch der *Base call* direkt aus dieser Klasse heraus erfolgen. Bisher wurde in diesem Fall der *Chaining wrapper* der Basismethode aufgerufen.
Die Klasse `Team` kann aber nicht wissen, zu welcher Basismethode welcher Klasse der *Base call* erfolgen soll. Also müssen auch die Basisklassen eine generische Schnittstelle anbieten, die von der Klasse `Team` genutzt werden kann. Diese Schnittstelle stellt die Methode `callOrig` des Interfaces `IBoundBase` dar, die in 5.4.4 genauer erläutert wird.

5.1.1 Callin id und Bound method id

Da in der neuen Umsetzung von *OT/J* alle Methoden generisch sind, muss hier an zwei Stellen eine Entscheidung stattfinden, von welcher Methode aus welcher Basisklasse der Aufruf kommt und welcher Code für diese Methode ausgeführt werden soll.
Zum einen ist das auf Teamseite der *Chaining wrapper.* Hier kann nicht mehr fest kodiert sein, welche Rollenmethoden ausgeführt werden. Dies muss nun dynamisch entschieden werden. Dazu dient die *Callin id*. Innerhalb eines Teams

identifiziert eine *Callin id* eindeutig eine Methode einer Basisklasse[1]. Mit ihr kann dann entschieden werden, welche gebundenen Rollenmethoden als *Callins* ausgeführt werden müssen. Gebildet werden sie vom Compiler, vom dem auch die entsprechenden Methoden anhand der *Callin ids* implementiert werden.
Allerdings muss dem *Chaining wrapper* zur Laufzeit bei seinem Aufruf mitgeteilt werden, welche *Callin id* er in diesem Moment bearbeiten soll. Also benötigen die Basisklassen dynamischen Zugriff auf die statisch gebildeten *Callin ids*. Sie sollen deshalb in einer zentralen Instanz, dem `TeamManager` (s. 6.2), gespeichert werden. Dort können sie vom *Team* bei seiner Aktivierung zur Laufzeit abgelegt und von der Basisklasse bei der Ausführung einer Basismethode angefragt werden.

Zusätzlich ist allerdings noch eine Stelle im Team nötig, an der die *Callin ids* vom Compiler abgelegt und vom Team selbst bei seiner Aktivierung wieder ausgelesen werden können. Hierfür bieten sich die Attribute einer Class Datei an (s. 2.1.2).

Die andere Stelle ist die Methode `callOrig`, über die der *Base call* umgesetzt wird. Diese Methode muss wissen, welcher Code ausgeführt werden soll, also von welcher Basismethode der Aufruf ursprünglich ausging. Dazu benutzt sie die *Bound method id*[2]. Diese wird im Gegensatz zur *Callin id* nicht vom Compiler, sondern von der Laufzeitumgebung generiert. Hier ist also keine Übergabe vom Team an die Basisklasse nötig. Allerdings muss auch die *Bound method id* dem *Chaining wrapper* übergeben werden, damit dieser sie im Falle eines *Base calls* an die Methode `callOrig` weiterreichen kann.

In Abbildung 5.1 ist zusammenfassend die Generierung und Verwendung der *Bound method ids* und der *Callin ids* schematisch dargestellt. Dabei sind einige Punkte noch vereinfacht abgebildet, die dann innerhalb dieses Kapitels detailliert erklärt werden.

[1] In 5.3.2 wird diese Bedingung noch ein wenig verändert.
[2] Wie wir in 5.4.2 sehen werden, wird die *Bound method id* auch noch an einer anderen Stelle benutzt.

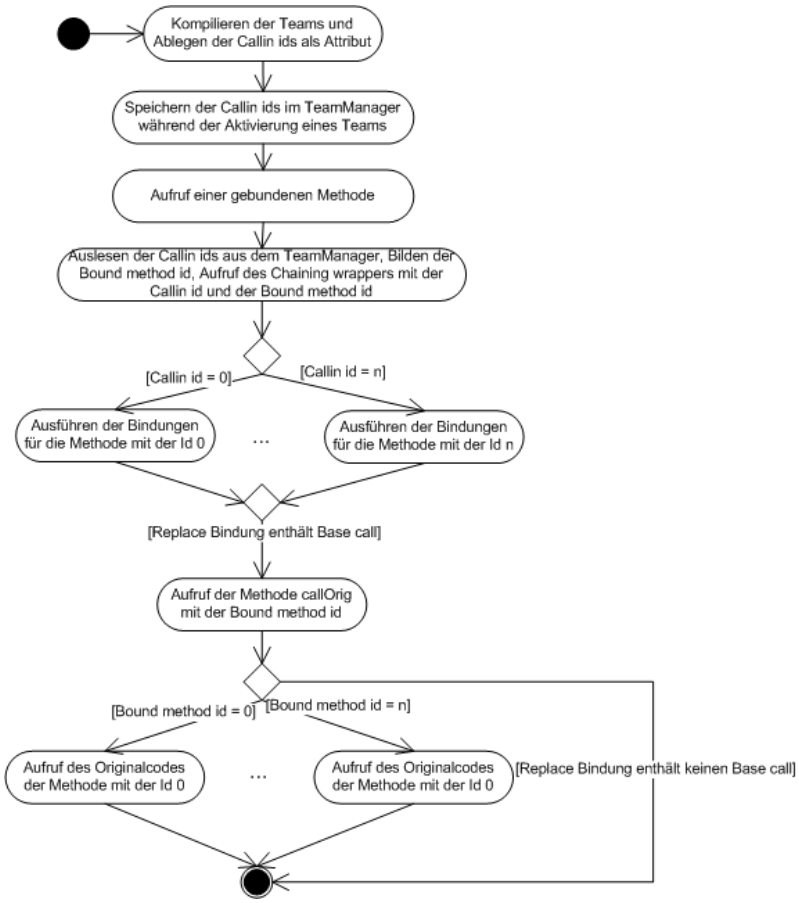

Abbildung 5.1: Generierung und Verwendung der Ids

Bevor nun die genaue technische Umsetzung dieser generischen Schnittstelle erläutert wird, soll an dieser Stelle noch ein allgemeineres Thema behandelt werden, das für die gesamte Implementierung der neuen *OT/J-Laufzeitumgebung* relevant ist.

5.2 Methodenparameter in generischen Methoden

In diesem und auch im nächsten Kapitel werden an vielen Stellen `Object` Arrays benutzt, um Argumente in einer allgemeinen Form übergeben zu können. Dies ist nötig, um die erwähnte generische Schnittstelle umsetzen zu können. Diese kann dann natürlich keine Kenntnis der Parameter einer konkreten Methode haben.

Beim Verpacken von Argumenten in `Object` Arrays bzw. beim Auspacken von Argumenten aus diesen, sind verschiedene Fälle zu beachten. Damit nicht bei jeder Verwendung dieser `Object` Arrays die Fallunterscheidung beschrieben werden muss, soll dies hier an zentraler Stelle erfolgen.

Soll ein beliebiger Wert in Java in ein `Object` Array eingefügt werden, sind folgende Fälle zu unterscheiden:

1. Der Wert hat einen komplexen Typ: Jeder komplexe Typ, also jede Klasse, jedes Interface und jede Enumeration, erben in Java von der Klasse `Object`. Der *Upcast* von dem konkreten Typ nach `Object` geschieht dabei implizit, er muss also nicht explizit geschrieben werden (auch im *Byteocde* nicht). In diesem Fall ist also keine weitere Arbeit nötig, um diesen Typ in das `Object` Array einzufügen.
2. Der Wert hat einen Basistyp: Java kennt die folgenden Basistypen: `int`, `long`, `double`, `float`, `char`, `byte`, `short` und `bool`. Diese Typen sind in Java vordefiniert und sind keine Klassen, erben also auch nicht von `Object`.
   Um mit Basistypen auch in generischen Klassen (wie z.B. `Vector`) arbeiten zu können, bietet Java zu jedem dieser Basistypen eine *Wrapperklasse* an (`Integer`, `Long`, `Double`, `Float`, `Character`, `Byte`, `Short`, `Boolean`). Die Konvertierung in diese *Wrapperklassen* bzw. aus diesen Klassen bezeichnet man als *Boxing* bzw. *Unboxing*. Jede dieser *Wrapperklassen* besitzt dazu einen Konstruktor, der den entsprechenden Basistyp als Argument erhält. Für das *Unboxing* besitzt jede *Wrappeklasse* eine Methode `<Name des Basistyps>Value()` die den Wert als Basistyp zurückliefert. Das Einfügen eines Wertes v von einem Basistyp würde also folgendermaßen aussehen:
   `objects[i] = new` *Wrapperclass*`(v);`

Beim Auspacken von Werten verschiedenen Typs aus einem `Object` Array muss der erste Fall nochmal unterteilt werden:

1. Der Wert hat den Typ `Object`: In diesem Fall kann die Zuweisung direkt erfolgen.
2. Der Wert hat einen anderen komplexen Typ: An dieser Stelle ist ein *Downcast* nötig. Dieser wird von Java nicht implizit durchgeführt, das heißt die folgenden Statements würden einen Compiler Fehler verursachen:
   ```
   Object o = ...
   String s = o;
   ```
   Dieser *Downcast* muss explizit definiert werden:
   ```
   Object o = ...
   String s = (String)o;
   ```
3. Der Wert hat einen Basistyp: Auch dieser Fall muss beim Auspacken wieder beachtet werden. An dieser Stelle muss ein *Unboxing* durchgeführt werden, um die Zuweisung vornehmen zu können, das hier am Beispiel eines Wertes vom Typ `int` gezeigt wird:
   ```
   Object o = objects[i];
   int i = ((Integer)o).intValue();
   ```
   Vor dem *Unboxing* muss also zunächst ein *Downcast* zur *Wrapperklasse* erfolgen.

Wie gesagt muss diese Fallunterscheidung immer vorgenommen werden, wenn mit `Object` Arrays gearbeitet wird. Wo das genau der Fall ist, wird nun im Folgenden gezeigt.

5.3 Die Klasse Team als generische Schnittstelle für Callin Bindungen

Wie in 5.1 beschrieben, soll die Klasse `Team` als generische Schnittstelle für den Aufruf des *Chaining wrappers* aus den Basisklassen heraus dienen.
Im Team soll die Methode `callAllBindings` als allgemeiner *Chaining wrapper* fungieren. Diese wird vom *Initial wrapper*, also einer redefinierten Basismethode, aufgerufen und führt dann alle Bindungen dieses Teams zu dieser Basismethode aus. Da diese Methode nicht von selbstgeschriebenem Code aufgerufen werden soll erhält sie das in 4.1.1 beschriebene Präfix `_OT$`.
Diese Methode kann allerdings nicht, wie das in der bisherigen Implementierung der Fall ist, alle Aufgaben des *Chaining wrappers* selbst übernehmen. Da sie in allgemeiner Form in der Klasse `Team` definiert ist, hat sie keine Kenntnis von den Bindungen konkreter *Teams* und kann dementsprechend auch nicht die gebundenen Rollenmethoden aufrufen. Diese Methode sorgt also lediglich dafür, nacheinander Methoden der konkreten *Teams* aufzurufen, in denen die entsprechenden Rollenmethoden ausgeführt werden.

Bei der Beschreibung des *Chaining wrappers* im Folgenden ist zu beachten, dass mit den Bindungen eines *Teams* immer alle Bindungen gemeint sind. Darunter fallen zum einen natürlich die Bindungen, die ein *Team* selbst deklariert, zum anderen aber auch alle geerbten Bindungen.

Der neue *Chaining wrapper* ist folgendermaßen definiert:

```
1.  public _OT$callAllBindings(IBoundBase base, Team[] teams,
2.          int idx, int[] callinIds, int boundMethodId,
3.          Object[] args) {
4.      this.callBefore(base, callinIds[idx], boundMethodId, args);
5.
6.      Object res = this.callReplace(base, teams, idx, callinIds,
7.             boundMethodId, args);
8.
9.      this.callAfter(base, callinIds[idx], boundMethodId, args,
10.             res);
11.
12.     return res;
13. }
```

Diese Methode muss von den konkreten *Teams* nicht überschrieben werden. Allerdings kann es aus Performancegründen dennoch sinnvoll sein. So muss z.B. ein *Team*, dass keine *Before callin Bindungen* hat, auch nicht die Methode `callBefore` aufrufen.

`callAllBindings` bekommt dabei diverse Parameter übergeben. Einige entsprechen den Parametern des *Chaining wrappers* in der bisherigen Implementierung von *OT/J*.
Zum einen benötigt sie eine Instanz der Basisklasse, um auf ihr den *Base call* durchzuführen. Diese Instanz hat dabei den statischen Typ `IBoundBase`, da die Klasse `Team` ja keine Kenntnis von konkreten Basisklassen hat. Zum anderen bekommt sie die Argumente der Basismethode als `Object` Array übergeben (s. 5.2). Diese können dann für den Aufruf der Rollenmethode bzw. für einen Base Call entsprechend entpackt werden.
Genau wie der *Chaining wrapper* der bisherigen Implementierung der *OT/J-Laufzeitumgebung* arbeitet auch die Methode `callAllBindings` rekursiv die *Teams* ab. Dabei iteriert sie über die *Teams* in dem Array `teams`. In diesem Array sind die *Teams*, die zu einer Basismethode Bindungen haben nach ihrer Priorität (s. 3.4) absteigend gespeichert, damit das *Team* mit der höchsten Priorität seine Bindungen zuerst ausführen kann. Der Parameter `idx` zeigt dabei den Index des aktuellen *Teams* an.
Wie in 5.1.1 beschrieben, braucht der neue *Chaining wrapper* zusätzlich die *Bound method id* und ein Array von *Callin Ids*. Wir werden im Laufe dieses und des nächsten Abschnitts sehen, wo diese Ids genau eingesetzt werden.

Um nun die Bindungen der konkreten *Teams* auszuführen, ruft `callAllBindings` nacheinander die Methoden `callBefore`, `callAfter` und `callReplace` auf. Diese sind dafür zuständig, die verschiedenen Arten von *Callin Bindungen* (*before*, *after* und *replace*) abzuarbeiten. Wenn ein konkretes *Team* Bindungen enthält, muss es, je nach Art der Bindungen, diese Methoden überschreiben und dort den *Dispatch* zu den Rollen vornehmen. Auch diese Methoden erhalten das Präfix _OT$.

```
1.  protected void _OT$callBefore(IBoundBase base,
2.          int callinId, int boundMethodId, Object[] args) {}
3.
4.  protected void _OT$callAfter(IBoundBase base,
5.          int callinId, int boundMethodId, Object[] args) {}
```

Im Gegensatz zu `callBefore` und `callAfter` ist die Methode `callReplace` nicht leer implementiert, sondern enthält eine Default Implementierung.

```
1. protected void callReplace(IBoundBase base, Teem[] teams,
2.          int idx, int[] callinIds, int boundMethodId,
3.          Object[] args) {
4.     return callNext(base, teams, idx, callinIds,
5.              boundMethodId, args);
6.  }
```

Diese ist notwendig, damit die rekursive Abarbeitung der *Teams* fortgeführt wird.
Dies geschieht in der Methode `callNext`:

```
1.  public Object _OT$callNext(IBoundBase base, Teem[] teams,
2.          int idx, int[] callinIds, int boundMethodId,
3.          Object[] args) {
4.      if (idx+1 < teams.length) {
5.          return teams[idx+1].callAllBindings(base, teams,
6.          idx+1, callinIds, boundMethodId, args);
7.      } else {
8.          if (base == null) {
9.              return teams[idx].callOrigStatic(
10.                     callinIds[idx], boundMethodId, args);
11.         } else {
12.             return base.callOrig(boundMethodId, args);
13.         }
14.     }
15. }
```

Falls weitere *Teams* abzuarbeiten sind, wird das nächste *Team* in der Liste aufgerufen (Zeile 3), sonst wird der Originalcode der gebundenen Basismethode ausgeführt (Zeile 8).

Die Methode `callNext` wird in zwei Fällen aufgerufen.

1. Das konkrete *Team* hat keine *Replace callin Bindungen*: Dann wird die Methode implizit durch die Default Implementierung von `callReplace` aufgerufen.
2. Eine Methode, die durch eine *Callin Bindung* gebunden ist, enthält einen *Base call*: Dann wird an dieser Stelle die Methode `callNext` aufgerufen.

Einen Sonderfall stellen statische Basismethoden dar.
In diesem Fall steht keine Instanz des Interfaces `IBoundBase` zur Verfügung, auf der die Methode `callOrig` aufgerufen werden kann. Hierbei muss der Aufruf des Originalcodes mit Kenntnis der konkreten Basisklasse und damit in der konkreten Teamklasse erfolgen. Dafür existiert die Methode `callOrigStatic`. Diese Methode muss dann von den konkreten Teams so überschrieben werden, dass sie, je nach *Callin id*, die Methode `callOrigStatic` der richtigen Basisklasse aufruft.

```
1.  public Object _OT$callOrigStatic(int callinId,
2.          int boundMethodId, Object[] args) {
3.      return null;
4.  }
```

Damit die *Callin Bindungen* eines konkreten Teams greifen, müssen je nach Art der Bindung die Methoden `callBefore`, `callAfter` und/oder `callReplace` geeignet implementiert werden.

5.3.1 Before und After callin Bindungen

Die *Before* und *After callin Bindungen* werden zwar in zwei verschiedenen Methoden, ansonsten aber identisch in der Teamklasse implementiert. Deshalb soll hier auch zusammengefasst beschrieben werden, wie diese zu implementieren sind.

Als Teil des *Chaining wrappers* arbeiten auch diese Methoden in der neuen Implementierung der *OT/J-Laufzeitumgebung* generisch. An dieser Stelle ist also weder bekannt, für welche Klasse, noch für welche Methode Bindungen ausgeführt werden sollen. Dennoch muss hier entscheiden werden können, welche Bindungen ausgeführt werden sollen.

Wie in 5.1.1 beschrieben, geschieht dies anhand der *Callin id*. Zusätzlich zu dem Array `teams`, bekommt der *Chaining wrapper* das Array `callinIds` übergeben. Dort ist genau für jedes *Team* im Array eine *Callin id* gespeichert. Dies ist nötig, da jedes Team für eine Basismethode eine andere Id vergeben kann.
Nun können also in den Methoden `callBefore` und `callAfter` die entsprechenden Rollenmethoden nach folgendem Schema ausgeführt werden.

Zuerst definieren wir die Mengen der *Before/After callin Bindungen* für ein *Team* `T` wie folgt:
Seien $BEFORE_T/AFTER_T$ zwei Mengen von Bindungen:

$$BEFORE/AFTER_T = b_1=(T, R_1, rm_1, B_1, bm_1, before/after), \ldots, b_n=(T, R_n, rm_n, B_n, bm_n, before/after)$$

Hier wird also wieder die Definition einer Bindung aus 4.1.2 verwendet. Zusätzlich benötigen wir jetzt die Abbildung
callinId$_T$: (Class, Method) -> Int,
die, wie oben beschrieben, für das Tupel aus Basisklasse und Basismethode innerhalb eines Teams T eine eindeutige Id vergibt.

Dann wird die Methoden `callBefore`/`callAfter` innerhalb eines Teams `T` folgendermaßen überschrieben (man beachte 5.2)[1]:

[1]An dieser Stelle fehlt noch das *Parameter mapping*. Wie dieses umgesetzt wird, wird in 5.3.3 gezeigt.

```
1.  public void _OT$callBefore/_OT$callAfter(
2.          IBoundBase base, int boundMethodId, int callinId,
3.          Object[] args) {
4.      switch(callinId) {
5.      ∀ b_i=(T, R, rm, B, bm, before/after)∈BEFORE/AFTER_T
6.      case callinId(B, bm):
7.          R r_i = _OT$liftTo$R((B)base);
8.          r_i.rm(args[0], … , args[args.length – 1]);
9.          break;
10. }
```

5.3.2 Replace Callin Bindungen

Ein grundsätzlicher Unterschied zwischen *Replace* und *Before* bzw. *After callin Bindungen* besteht darin, dass die Signatur der Rollenmethoden, die durch eine *Replace callin Bindung* gebunden sind, erweitert werden muss (*Signature enhancement*). Das ist nötig, da diese Methoden einen *Base call* enthalten und damit die Methode `callNext` aufrufen können. Dazu benötigen sie alle Parameter, die für einen Aufruf von `callNext` nötig sind. Die Signatur einer solchen Rollenmethode muss also wie folgt geändert werden:
Sei
Rtype `rm(...)`
eine durch eine *Replace callin Bindung* gebundene Rollenmethode, dann muss die Signatur von `rm` wie folgt erweitert werden:

```
Rtype rm(..., IBoundBase base, Team[] teams,
        int boundMethodId, int[] callinIds, int index,
        Object[] args)
```

Zum anderen muss beachtet werden, dass Rollenmethoden, die durch eine *Replace callin Bindung*, gebunden sind, einen Wert zurückliefern können, der auch im weiteren Verlauf noch benötigt wird, da er den Rückgabewert der Basismethode ersetzen kann.

Die Methode `callReplace` ist grundsätzlich aufgebaut, wie die Methoden `callBefore` und `callAfter`. Auch hier wird ein `switch` Statement benötigt, in dem anhand der *Callin id* entschieden wird, welche Methoden aufzurufen sind

```
1.  public Object _OT$callReplace(IBoundBase base,
2.          Team[] teams, int boundMethodId, int[] callinIds,
3.          int index, Object[] args)  {
4.      switch(callinId) {
5.      }
6.  }
```

Die `case` Fälle dieser Methode unterscheiden sich dagegen von denen der Methoden `callBefore` und `callAfter`. Hier können nicht einfach alle *Replace Bindungen* eines *Teams* zu einer Basismethode hintereinander ausgeführt werden, sondern es darf zunächst nur die Bindung mit der höchsten Priorität (s. 3.4) aufgerufen werden. Weitere *Replace callin Bindungen* greifen nur im Falle eines *Base calls*. Dennoch müssen in dieser Methode grundsätzlich alle *Replace Bindungen* ausgeführt werden können.
Für die Entscheidung, welche Bindung zu welchem Zeitpunkt genau ausgeführt werden muss, kann die *Callin id* (s. 5.1.1) dienen. Hier müssen also für jede *Replace Bindung* zu einer Methode unterschiedliche *Callin Ids* vergeben werden. Dabei bekommt die Bindung mit der höchsten Priorität die niedrigste *Callin id*. So kann entschieden werden, welche Bindung zuerst ausgeführt werden muss.
Für die Definition der Methode `callReplace` genügt die Funktion $callinId_T$ also nicht mehr. Deshalb soll diese so geändert werden, dass sie nun für eine Bindung eine Id liefert:

$callinId_T$: *Binding -> Int*

Diese Id ist für alle *Replace Bindungen* zu einer Basismethode innerhalb eines Teams eindeutig. Bei *Before/After Callin Bindungen* muss sie weiterhin für eine Basismethode eindeutig sein.

Wie oben beschrieben wurde, wird genau für jedes *Team* im Array `teams` eine *Callin id* im Array `callinIds` gespeichert. Wenn nun für eine Basismethode in einem Team mehrere *Callin ids* benötigt werden, muss auch in `teams` das *Team* mehrmals enthalten sein. So wird im Falle eines *Base calls* die Rekursion mit demselben *Team* fortgeführt, wenn es mehrere *Replace callin Bindungen* hat.
Dabei muss beachtet werden, dass die *Before* und *After Bindungen* nur beim ersten Aufruf des *Teams* ausgeführt werden dürfen. Das heißt, in den Methoden `callBefore` und `callAfter` darf für eine Basismethode jeweils nur ein `case` Fall definiert werden. Die *Callin id* muss dabei die Id der *Replace Bindung* mit der höchsten Priorität, also die niedrigste *Callin id*, sein.
Falls zu einer Basismethode keine *Replace Bindung* besteht, so kann für die *Before* und *After Bindungen* eine beliebige (aber einheitliche) Id vergeben werden.

Um nun noch formal Bedingungen für die Vergabe der *Callin ids* definieren zu können, benötigen wir die Abbildung

$prio_T$: *Binding -> Int*

Diese Abbildung ordnet innerhalb eines Teams T jeder Bindung eine Priorität >= 0 zu. Jeder niedriger der Wert, desto höher ist dabei die Priorität der Bindung. Bindungen unterschiedlichen Typs (*before, after, replace*) können dabei auch dieselbe Priorität erhalten. innerhalb eines Typs muss die Priorität jedoch eindeutig sein.

Nun können wir folgende Bedingungen formulieren.

1. $callinId_T(b = (T, R, rm, B, bm, before/after)) =$
   $callinId_T(b' = (T, R', rm', B', bm', before/after)) \rightarrow B = B' \wedge bm = bm'$
   Die *Callin ids* für *Before* und *After Bindungen* werden innerhalb eines Teams eindeutig für das Tupel aus Basisklasse und Basismethode vergeben.

2. $callinId_T(b = (T, R, rm, B, bm, replace) >$
   $callinId_T(b' = (T, R', rm', B, bm, replace) \rightarrow$
   $prio_T(b) > prio_T(b')$
   Bei zwei *Replace Bindungen* mit unterschiedlichen *Callin ids*, wird die Bindung mit der niedrigeren Id zuerst ausgeführt.

3. $callinId_T(b = (T, R, rm, B, bm, before/after)) =$
   $callinId_T(b' = (T, R', rm', B, bm, replace)) \rightarrow$
   $\neg\exists\, b'' = (T, R'', rm'', B, rm, replace): prio_T(b'') < prio_T(b')$
   Die *Before* und *After callin Bindungen* haben dieselbe *Callin id*, wie die *Replace callin Bindung* mit der höchsten Priorität.

Um die `case` Fälle nun genau beschrieben zu können, definieren wir zunächst die Menge $REPLACE_T$ der *Replace Callin Bindungen* eines *Teams*:

$$REPLACE_T = \{b_1 = (T, R_1, rm_1, B_1, bm_1, callinId_1, replace), \ldots, b_n = (T, R_n, rm_n, B_1, bm_1, callinId_n, replace)\}$$

Basismethoden, die durch eine *Replace callin Bindung* gebunden sind, können beliebige Rückgabetypen haben. Das muss bei der Definition der `callReplace` Methode beachtet werden. Abgesehen von den Fällen aus 5.2 muss ein weiterer Fall unterschieden werden:
Die Basismethode hat den Rückgabetyp `void`:

In diesem Fall kann die Methode `callReplace null` zurückliefern, da der Rückgabewert nicht weiter verwendet wird:

```
1. case callinIdT(bi ∈ REPLACET) :
2.      Ri r = _OT$liftTo$Ri((Bi)base);
3.      r.rmi(args[0], … , args[args.length – 1], base, teams,
4.              boundMethodId, callinIds, index, args);
5.      return null;
```

Hat sie einen Rückgabetyp ungleich `void` kann ihr Rückgabewert direkt (nach *Casting* und *Unboxing*) zurückgeliefert werden:

```
1. case callinIdT(bi ∈ REPLACET) :
2.      Ri r = _OT$liftTo$Ri((Bi)base);
3.      return r.rmi(args[0], … , args[args.length – 1], base,
4.              teams, boundMethodId, callinIds, index, args);
```

5.3.3 Parameter mapping

Das *Parameter mapping* ist nicht für eine Rollenmethode, sondern nur für eine *Callin Bindung* eindeutig, d.h. es kann zwei *Callin Bindungen* in einer Rolle geben, die dieselbe Rollenmethode binden, die Parameter aber unterschiedlich mappen. Deshalb darf das *Parameter mapping* nicht in der Rollenmethode selbst geschehen, sondern muss in den Methoden `callBefore`, `callAfter` und `callReplace` direkt vor dem Aufruf einer Rollenmethode erfolgen.

Um die Implementierung des *Parameter mappings* exakt zu beschreiben, ist es zunächst nötig die Definition einer Bindung zu erweitern:

Eine Bindung b ist ein Tupel *(Team, Role, RoleMethod, BaseClass, BaseMethod, Kind, ParameterMapping)*.

Es ist zu der Definition aus 4.1.2 also der Bestandteil *ParameterMapping* hinzugekommen. *ParameterMapping* ist definiert als eine Abbildung von den Namen der Parameter der Rollenmethode (inklusive `result` als Name des Rückgabewertes) auf einen Ausdruck, der den gleichen Typ hat, wie der Parameter. Innerhalb der Methoden `callBefore`, `callAfter` und `callReplace` wird das *Parameter mapping* nun wie folgt implementiert:

Sei `R` eine Rolle mit der Basisklasse `B` und einer Methode `rm`, die durch eine *Callin Bindung* mit der *Callin id callinId1* gebunden ist und sei `rm` folgendermaßen deklariert:
`Rtype rm(`*PType1* `pVal1,` *PType2* `pVal2, … ,` *PTypen* `pValN):`

```
1.  case callinId1:
2.      R r = _OT$liftTo$R((B)base);
3.      PType1 pVal1 = ParameterMapping("pVal1");
4.      PType2 pVal2 = ParameterMapping("pVal2");
5.      .
6.      .
7.      .
8.      PTypen pValN = ParameterMapping("pValN");
9.      return r.rm(pVal1, pVal2, ..., pValN, base, teams,
10.             boundMethodId, callinIds, index, args);
```

Bei Rollenmethoden, die durch eine *Replace callin Bindung* gebunden sind, ist zusätzlich zu beachten, dass diese den Rückgabewert der Originalmethode verändern können. Allerdings ist dies nur in dem Fall möglich, dass die

Basismethode einen Rückgabetyp ungleich `void` und die Rollenmethode den Rückgabetyp `void` hat. In diesem Fall kann ein beliebiger Wert als Rückgabewert verwendet werden.
Dies muss in der Methode `callReplace` umgesetzt werden. In diesem Fall muss diese Methode also nicht den Rückgabewert der Rollenmethode, sondern den durch das *Parameter mapping* definierten Wert zurückliefern:

```
1. case callinId1:
2.       ...
3.       r.rm(pVal1, pVal2, ..., pValN, base, teams,
4.              boundMethodId, callinIds, index, args);
5.       return ParameterMapping("result");
```

5.4 Transformationen der Basisklasse

In der Basisklasse sind, wie in 5.1 beschrieben, beim Bekanntwerden einer Bindung folgende Transformationen nötig. Zum einen muss die Basismethode zum *Initial wrapper* redefiniert werden. Zum anderen muss der Originalcode der Methode an eine Stelle verschoben werden, an der er vom Team aufgerufen werden kann (*Base call*). Wie in 5.1 erwähnt, existiert zu diesem Zweck die Methode `callOrig` des Interfaces `IBoundBase`. Wie nun der *Initial wrapper* und `callOrig` genau implementiert werden können, soll in diesem Kapitel gezeigt werden.

5.4.1 Der Initial Wrapper

Auch beim *Laufzeitweben* soll die Basismethode zum *Initial wrapper* redefiniert werden.
Allerdings muss in der neuen Umsetzung der *OT/J-Laufzeitumgebung* eine zusätzliche Indirektion eingeführt, der *Initial wrappe*r also auf zwei Methoden verteilt werden.
Warum dies nötig ist und wie diese zusätzliche Methode definiert ist, wird in diesem Abschnitt gezeigt.
Es existiert genau ein Fall, für den es nicht ausreicht, alleine die Basismethode zum *Initial wrapper* zu redefinieren:

Seien `B0` und `B1` zwei Klassen, wobei `B1` eine Subklasse von `B0` ist und `R0` eine Rolle, deren Basisklasse `B1` ist. `R0` bindet eine Methode `bm` von `B1`, die nicht in `B1` sondern in `B0` implementiert ist und von `B1` nur geerbt wird. In diesem Fall ist es nicht möglich den Aufruf der Methode `callAllBindings` eines Teams in der Methode `bm` der Klasse `B1` zu implementieren, da die Methode dort nicht definiert ist und auch zur Laufzeit nicht definiert werden kann (s. 2.2.2). Es würde aber auch nicht ausreichen die Methode `bm` der Klasse `B0` zu redefinieren. Wenn nämlich wirklich die Methode `bm` der Klasse `B0` und nicht `B1` aufgerufen wird, sollen natürlich keine Bindungen ausgeführt werden.

Das heißt, es ist nötig in den Basisklassen eine weitere Methode einzuführen, die von der Originalmethode aufgerufen wird. Diese Methode wird in der Klasse B0 so implementiert, dass sie in dem Fall, dass sie von der Methode `bm` aufgerufen wird, direkt die Methode `callOrig` aufruft. `B1` dagegen überschreibt die Methode mit dem tatsächlichen *Dispatch* zur Teamklasse.
Dazu soll die Methode `callAllBindings`[1] dienen.

Das bedeutet, dass die Originalmethode nur den Aufruf der Methode `callAllBindings` übernimmt. Beim Aufruf übergibt sie zwei Argumente an diese Methode. Zum einen, ihre als Array verpackten Argumente (s. 5.2) und zum anderen die *Bound method id* (s. 5.1.1) für diese Methode. Anhand dieser Id kann in `callAllBindings` entschieden werden, ob ein Dispatch zu den Teams erfolgen soll oder nicht. Statische Basismethoden bilden hier eine Ausnahme. Da diese nicht vererbt werden können, ist hier auch kein Aufruf von `callAllBindings` nötig, sondern der Dispatch kann direkt erfolgen.
Die Bedingungen, die an die Generierung der *Bound method id* gestellt werden müssen, werden im Folgenden beschrieben.

5.4.2 Bound method id

Wie in 5.1.1 beschrieben, wird diese Id in den Methoden `callAllBindings` und `callOrig` (s. 5.4.4) verwendet. In `callAllBindings` dient sie dazu, entscheiden zu können, ob für eine Methode ein Dispatch zu den *Teams* stattfindet oder nicht.
Für die Generierung dieser Id sind verschiedene Fälle zu betrachten.
Der triviale Fall ist eine Basismethode, die auch direkt in der gebundenen Klasse implementiert ist und von deren Klasse keine Subklassen existieren. In diesem Fall könnte die *Bound method id* beliebig gewählt werden, solange sie innerhalb dieser Klasse eine Methode eindeutig identifiziert.
Deutlich spannender ist dagegen der Fall, dass die gebundene Methode nicht in der Basisklasse selbst, sondern in einer ihrer Superklassen[2] implementiert ist.
Nehmen wir dazu folgenden Fall an:
`B0` und `B1` sind zwei Klassen, wobei `B0` die Superklasse von `B1` ist. Außerdem soll eine Rolle `R` existieren, die als Basisklasse `B1` hat und über eine *Replace callin Bindung* die Methode `bm` der Klasse `B1` bindet. Wenn nun `B1` die Methode `bm` nicht selbst implementiert, sondern nur von `B0` erbt, dann muss für die *Bound method id* beachtet werden, dass sie in `B0.bm` und nicht in `B1.bm` gebildet wird. Dennoch muss sie in `B1.callAllBindings` richtig ausgewertet werden. `B1` und `B0` müssen also die gleiche Zuordnung von *Bound method id* zu Basismethode bzw. umgekehrt haben. Das heißt, die *Bound method id* muss innerhalb einer Vererbungshierarchie für dieselbe Methode immer gleich gebildet werden .

---

[1] Nicht zu verwechseln mit der Methode `callAllBindings` der *Teams*.

[2] Natürlich kann eine Klasse in Java nur eine Superklasse haben. Gemeint sind hierbei diese direkte Superklasse und ihre Superklasse u.s.w.

Die zwei `callAllBindings` Methoden von B0 und B1 können/müssen dann aber natürlich unterschiedlich auf dieselbe *Bound method id* reagieren. In B0 wird direkt `callOrig` aufgerufen, in B1 dagegen findet der Dispatch zu den Teamklassen statt.

Die *Bound method id* wird dabei von der Laufzeitumgebung bestimmt und bei der Redefinition der Originalmethode in dieser fest kodiert. Hier ist also kein zusätzlicher Methodenaufruf nötig, um diese Id zu bestimmen.
Die Originalmethode wird dann nach folgendem Schema zum ersten Teil des *Initial wrappers* transformiert.
Dafür definieren wir die Abbildung
$boundMethodId_B$: *METHOD -> Integer*
die jeder Methode, unter den beschriebenen Bedingungen, in einer Klasse B eine Id zuordnet:
Sei
RType bm($PType_1$ arg_1, ... , $PType_n$ arg_n*)*
eine Basismethode, dann wird `bm` folgendermaßen redefiniert:

```
1.  RType bm(PType1 arg1, ... , PTypen argn)  {
2.        Object[] args = {arg1, ... , argn};
3.        int boundMethodId = boundMethodIdB(bm);
4.        return callAllBindings(boundMethodId, args);
5.  }
```

Auch hierbei müssen natürlich wieder sämtliche Fälle für die Typen der Parameter und des Rückgabetyps dieser Methode beachtet werden (s. 5.2).

5.4.3 Dispatch zu den Teamklassen

Der eigentliche Dispatch findet dann, wie im vorigen Abschnitt beschrieben, in der Methode

```
Object callAllBindings(int boundMethodId, Object[] args)
```

statt.
Dort müssen also die aktiven Teams und die zugehörigen *Callin ids* für die, anhand der *Bound method id* identifizierte, Methode aus dem `TeamManager` geholt werden.
Dazu wird allerdings eine weitere Id benötigt.
Momentan können wir über die *Bound method id* innerhalb einer Klasse eine Methode und über die *Callin id* innerhalb eines *Teams* eine Basismethode einer Basisklasse eindeutig identifizieren. Der `TeamManager` muss nun aber eine Methode in einer Klasse (also eine *Joinpoint* (s. 1.2)) global eindeutig identifizieren können.

Dazu soll die sog. *Joinpoint id* verwendet werden. Diese wird von der *Laufzeitumgebung* (genau gesagt vom `TeamManager` s. 6.2.1) vergeben und beim Generieren der Methode `callAllBindings` hart kodiert in dieser Methode hinterlegt. Nun können über die Methoden

```
public static Team[] getTeams(int joinpointId)
public static int[] getCallinIds(int joinpointId)
```

des `TeamManagers` die aktiven Teams und die zugehörigen *Callin ids* geholt werden.
Dies passiert natürlich in Abhängigkeit zur *Bound method id*, da für jede dieser Ids eine andere *Joinpoint id* verwendet werden muss.
Um die Methode `callAllBindings` genau beschreiben zu können, definieren wir die Menge $BOUND\_METHODS_B$ als Menge aller gebundenen Methoden der Klasse B und die Abbildung
joinpointId: METHOD -> Integer
die jeder Methode jeder Klasse eindeutig eine *Joinpoint id* zuordnet. Weiterhin verwenden wir wieder die oben definierte Abbildung $boundMethodId_B$.
Nun können wir `callAllBindings`, wie folgt, definieren:

```
1.  private Object callAllBindings(int boundMethodId,
2.          Object[] args) {
3.      switch (boundMethodId) {
4.      ∀ bm∈BOUND_METHODS_B
5.
6.      case boundMethodId_B(bm):
7.          int joinpointId = joinpointId(bm);
8.          int[] callinIds =
9.                  TeamManager.getCallinIds(joinpointId);
10.         Team[] teams = TeamManager.getTeams(joinpointId);
11.         if (teams == null) {
12.             break;
13.         }
14.         return teams[0].callAllBindings(
15.                 (IBoundBase) this, teams, 0, callinIds,
16.                 boundMethodId, args);
17.     }
18.
19.     return callOrig(boundMethodId, args);
20. }
```

Hier wird also anhand der *Bound method id* innerhalb eines `switch` Statements entschieden, mit welcher *Joinpoint id* die Teams und die zugehörigen *Callin ids* aus dem `TeamManager` geholt werden.
Anschließend muss überprüft werden, ob die Arrays der *Teams* bzw. der *Callin ids* gleich `null` sind. Dies kann dann der Fall sein, wenn inzwischen kein Team, das Bindungen zu dieser Klasse hat, aktiv ist, aber zwischenzeitlich mal mindestens eines aktiv war (s. 5.7). Sind die Arrays gleich `null` wird direkt `callOrig`

aufgerufen, da keine Bindungen ausgeführt werden müssen. Sonst wird die Methode `callAllBindings` des ersten *Teams* im Array aufgerufen. Diese erhält als Argumente eine Instanz der aktuellen Basisklasse, die *Teams*, die *Callin ids* zu den *Teams*, die *Bound method id* der Basismethode, die gepackten Argumente der Methode und einen Index, der anzeigt, welches *Team* aus dem Array das aktuelle ist. Beim Aufruf aus der Basismethode ist dieser immer 0.

Nachdem nun der *Initial Wrapper* beschrieben wurde, fehlt noch die Methode `callOrig`, die aus einem Team heraus bei einem *Base call* aufgerufen werden muss. Diese soll im Folgenden beschrieben werden.

5.4.4 Die Basisklasse als generische Schnittstelle

Die Methode

```
Object callOrig(int boundMethodId, Object[] args);
```

wird, wie schon erwähnt, durch das Interface `IBoundBase` definiert. Im Gegensatz zur bisherigen Strategie enthält sie jetzt den Code von allen gebundenen Methoden einer Basisklasse. Um entscheiden zu können, von welcher Basismethode der Aufruf ursprünglich ausging, bekommt sie die *Bound method id* übergeben, die eine Methode innerhalb einer Klasse eindeutig identifiziert.

Um zu zeigen, wie die Methode `callOrig` definiert ist, benötigen wir zum einen wieder die Abbildung $boundMethodId_B$ (s. 5.4.2) und zum anderen die Menge aller Basismethoden einer Klasse B $BOUND\_METHODS_B$ aus dem vorigen Abschnitt.

Dann ist die Methode `callOrig` wie folgt definiert (man beachte 5.1.1):

```
1.  Object callOrig(int boundMethodId, Object[] args) {
2.      switch (boundMethodId) {
3.      ∀ bm_i ∈ BOUND_METHODS_B
4.
5.      case boundMethodId_B(bm):
6.          PType1 arg1_i = args[0];
7.          .
8.          .
9.          .
10.          PTypeN argN_i = args[n];
11.         // execute original code of BaseMethod
12.     }
13.     default:
14.         super.callOrig(boundMethodId, args);
15. }
```

Das heißt, zuerst wird in dem `switch` Statement ermittelt, von welcher Methode der Code ausgeführt werden soll. Danach werden die in dem Array `args` verpackten Argumente dieser Methode ausgepackt und der Originalcode ausgeführt.
Wird kein passender `case` Fall gefunden, so bedeutet dies, dass die Methode in einer Superklasse implementiert sein muss. Deshalb wird im `default` Fall `callOrig` der Superklasse aufgerufen.
Zu beachten ist hierbei der Rückgabetyp der Originalmethode. Dabei sind folgende Fälle zu unterscheiden:

1. Die Methode hat einen Rückgabetyp der von `Object` erbt: In diesem Fall kann der Originalcode 1:1 übernommen werden.
2. Die Methode hat einen Basistyp als Rückgabetyp: In diesem Fall müssen vor allen `return` Statements innerhalb des Originalcodes die Rückgabewerte durch Instanzen der entsprechenden *Wrapperklasse* gekapselt werden.
3. Die Methode hat den Rückgabetyp `void`: In diesem Fall muss am Ende der Methode ein `return` Statement ohne Rückgabewert eingefügt werden.

Analog zu der Methode muss auch die Methode `callOrigStatic` für den Zugriff auf den Originalcode von statischen Methoden implementiert werden.

Nachdem nun dargelegt wurde, wie *Callin Bindungen* in der neuen *OT/J-Laufzeitumgebung* umgesetzt werden können, soll nun beschrieben werden, wie die *Decapsulation* implementiert werden kann.

5.5 Decapsulation

Auch das Prinzip der *Decapsulation* (s. 3.7.2) muss in den Basisklassen umgesetzt werden.
Dazu ist es nötig eine Schnittstelle in der Basisklasse zu schaffen, über die der Zugriff auf nicht sichtbare Felder und Methoden der Basisklassen erfolgen kann. Diese Schnittstelle soll über die Methode `access` abgebildet werden, die wie `callOrig` in dem Interface `IBoundBase` deklariert wird.
Die Methode `access` weißt dabei starke Ähnlichkeiten zu `callOrig` auf. Auch hier wird anhand der *Bound method id*[1] ermittelt, auf welches Feld bzw. auf welche Methode in `access` zugegriffen werden soll.
Allerdings kann die *Bound method id* zum Zugriff auf nicht sichtbare Methoden und Felder nicht exakt nach den gleichen Bedingungen gebildet werden, wie bei der Umsetzung der *Callin Bindungen*.
Wie in 5.4.2, gezeigt muss die *Bound method id* für die Umsetzung der *Callin Bindungen* innerhalb einer Vererbungshierarchie für die gleiche Methode immer identisch gebildet werden. Dies ist nötig, damit eine Superklasse und ihre Subklasse dieselbe *Bound method id* auch auf die gleiche Weise interpretieren.

[1] Der Name *Bound method id* passt hier nicht mehr ganz, da ja auch auf Felder zugegriffen werden kann.

Genau das darf beim Zugriff auf Felder und private oder statische Methoden nicht geschehen, da diese nicht vererbt werden. Hat also eine Subklasse ein Feld oder eine private oder statische Methode mit demselben Namen und derselben Signatur, wie ihre Superklasse so muss in der Basisklasse entschieden werden können, auf welches dieser beiden Felder bzw. Methoden Zugriff genommen werden soll. In diesem Fall müssen die beiden *Bound method ids* also unterschiedlich gebildet werden.

Ein weiterer Unterschied zwischen der Umsetzung der *Callin Bindungen* und der *Decapsulation* besteht darin, dass die *Bound method ids* in der Methode `access` zunächst geholt werden müssen. Im Gegensatz zu `callOrig` muss der Aufruf von `access` nicht ursprünglich von einer Basismethode ausgegangen sein. Die Aufrufe von `access` können an beliebiger Stelle im *Team* definiert sein. Dementsprechend kann auch von der Basisklasse keine *Bound method id* mit übergeben werden.
Allerdings kann der Compiler für ein Team jedem Member, auf das Zugriff genommen werden soll, eine Id zuordnen. Diese Id wird *Access id* genannt.
Diese kann allerdings in der Methode `access` nicht verwendet werden, um zu entscheiden, auf welches Feld Zugriff genommen werden soll.

Stellen wir uns dazu folgenden Fall vor:
`T0` und `T1` sind zwei *Teams*, die beide Zugriff auf das private Feld `f` der Basisklasse `B` nehmen wollen. `T0` vergibt für dieses Feld die *Access Id* 0. Nun kann es aber vorkommen, dass das Team `T1` diesem Feld die Id 1 zuordnet. Nun bezeichnen also zwei verschiedene Ids dasselbe Feld. Genauso könnte es passieren, dass zwei unterschiedliche Member einer Klasse von zwei Teams dieselbe Id erhalten.
Die Basisklasse kann in diesen Fällen nicht entscheiden, auf welches Member zugegriffen werden soll.
Es ist also nötig, die *Bound method id* in Abhängigkeit von der *Access id* und des *Teams* zu bestimmen.

Dazu muss ein Team bei seiner Aktivierung seine *Access ids* im `TeamManager` (s. 6.2.2) ablegen. Wenn die Methode `access` eine Class Instanz des Teams übergeben bekommt, kann sie damit vom `TeamManager` die *Bound method id* für das Member erfahren.
Dazu muss die *Access id* vom Team bezogen auf eine Klasse eindeutig vergeben werden und in einem Attribut der Class Datei (s. 2.1.2) gespeichert werden.

Zusätzlich benötigt die Methode `access` noch zwei weitere Felder. Zum einen das Flag `opKind`, über das entschieden werden kann, ob auf ein Feld lesend oder schreibend zugegriffen werden soll und zum anderen die Argumente für einen Methodenaufruf bzw. das Setzen eines Feldes.

Um die Methode `access` definieren zu können, benötigen wir die Mengen

ACCESSABLE_METHODS$_B$ und *ACCESSABLE_FIELDS$_B$*, die die nicht sichtbaren Methoden bzw. Felder einer Klasse `B` enthalten, auf die aus einem Team heraus zugegriffen werden soll.
Zusätzlich benötigen wir die Abbildung
memberId$_B$: MEMBER -> Integer
die nach dem oben beschriebenen Prinzip, ähnlich wie *boundMethodId$_B$* (s. 5.4.2), den Membern einer Klasse Ids zuordnet.

```
1.  Object access(int accessId, int opKind, Object[] args,
2.          Class<?> caller) {
3.      switch(TeamManager.getMemberId(accessId, caller)) {
4.      ∀ bm_i ∈ ACCESSABLE_METHODS_B
5.      case memberId_B(bm_i):
6.          PType_I arg_Ii = args[0];
7.          .
8.          .
9.          .
10.         PType_n arg_ni = args[n];
11.         // execute original code of BaseMethod
12.         break;
13.     }
14.
15.     ∀ f ∈ ACCESSABLE_FIELDS_B
16.     case memberId_B(f):
17.         if (opKind == 0) {
18.             return f;
19.         } else {
20.             f = args[0];
21.         }
22.         break;
23.     }
24.     default:
25.         super.access(accessId, opKind, args, caller);
26.  }
```

Wie in `callOrig`, wird hier also in einem `switch` Statement anhand der aus dem `TeamManager` geholten *Bound method id*/*Member id* entschieden, auf welches Member zugegriffen werden soll. Wird kein `case` Fall gefunden, ist das Member wieder in der Superklasse definiert.

Analog zum Aufrufen der Originalmethode über `callOrig` existiert auch hier wieder der Fall, dass das Member, auf das zugegriffen werden soll, statisch ist.
Auch hierfür wird wieder eine zusätzlich Methode `accessStatic` benötigt. Im Gegensatz zu `callOrigStatic` muss für diese Methode allerdings kein Gegenstück in der Klasse `Team` deklariert sein, da `accessStatic` nicht von der Klasse `Team`, sondern nur von den konkreten *Teams* aufgerufen wird..
Die Methode `accessStatic` ist dann analog zu `access` definiert.

5.6 Übergabe von Bindungsinformationen

Wie in 5.1.1 beschrieben, werden die *Callin ids* und die *Access ids* vom Compiler generiert und in einem Attribut der Class Datei (s. 2.1.2) abgelegt.Von dort können sie dann zur Laufzeit wieder ausgelesen werden.

Jedes Team muss dabei nicht nur die Ids der Bindungen bereitstellen, die es selbst deklariert, sondern auch die der geerbten.
Es sollen zwei Attribute eingeführt werden, die die Ids beinhalten. Die Attribute werden dabei in der gleichen Form wie in [LY99] beschrieben:

1. Das Attribut *OTDynCallinBindings*, das die *Callin ids* festlegt:

```
1.  CallinBindings_attribute {
2.      u2 bindings_count
3.      {
4.          u2 base_class_index_table;
5.          u2 base_method_name_index_table;
6.          u2 base_method_signature_index_table;
7.          u4 callin_id;
8.      } bindings[bindings_count]
9. }
```

Dieses Attribut hat folgende Bestandteile:
`bindings_count`: Die Anzahl der Bindungen (also die Länge des folgenden Arrays), die in diesem Team definiert werden.
`bindings`: Dieses Array der Länge `bindings_count` enthält die Bindungen. Ein Eintrag dieses Arrays besteht dabei aus drei Komponenten:

1. `base_class_index_table`: Ein Index auf einen *Constant pool* Eintrag, der den Namen der Basisklasse enthält.
2. `base_method_name_index_table`: Ein Index auf einen *Constant pool* Eintrag, der den Namen der Basismethode enthält.
3. `base_method_signature_index_table`: Ein Index auf einen *Constant pool* Eintrag, der die Signatur der Basismethode enthält.
4. `callin_id`: Die *Callin id* dieser Bindung

2. Das Attribut *OTDynAccessBindings*, das die verwendeten *Access ids* speichert.

Dieses Attribut hat dieselbe Struktur wie das Attribut *OTDynCallinBindings*, weist aber die zwei folgenden Unterschiede auf:
Der Name des Attributs ist *OTDynAccessBindings* und nicht *OTDynCallinBindings*. Die Id, die in den Attributen enthalten ist, heißt in diesem Attribut `access_id`, hat die gleichen Eindeutigkeitsbedingungen, wie die `callin_id`, muss aber im Gegensatz zu ihr innerhalb eines Teams immer bei 0 beginnen und fortlaufend vergeben werden.

Wie in 5.1.1 beschrieben, sollen diese Attribute bei der Aktivierung eines Teams vom `TeamManager` wieder ausgelesen werden. Welche Änderungen dazu im Team nötig sind, soll im Folgenden beschrieben werden.

5.7 Teamaktivierung/-deaktivierung

Die Klasse `Team` hat als Oberklasse aller Teams folgende Aufgaben:

- Sie implementiert die Logik zur Aktivierung und Deaktivierung der *Teams*
- Sie bietet nicht-implementierte Methoden zur Abfrage der in einem konkreten *Team* enthaltenen *Rollen* an. Dies Methoden sind zwar nicht abstrakt, müssen aber von den Teams überschrieben werden.
- Sie enthält Interfaces, die von den konkreten *Rollen* implementiert werden können, um ihnen verschiedene Eigenschaften zu geben

An dieser Stelle soll lediglich auf die Funktionsweise der Aktivierung/Deaktivierung bzw. die für die neue Webestrategie nötigen Änderungen an ihr eingegangen werden.
Die Umsetzung der übrigen Aufgaben der Klasse `Team` bleibt erhalten und soll deshalb hier nicht näher beschrieben werden.

Es gibt verschiedene Arten der Aktivierung/Deaktivierung eines Teams, die durch die Klasse `Team` implementiert werden (s. 3.3):

1. Explizite Aktivierung/Deaktivierung für alle Threads
2. Explizite Aktivierung/Deaktivierung für einen Thread
3. Explizite Aktivierung/Deaktivierung innerhalb eines `with`-Blocks
4. Implizite Aktivierung/Deaktivierung

Für die Fälle 1. und 2. existieren die Methoden `activate()`, `deactivate()`, `activate(Thread thread)` und `deactivate(Thread thread)`. Diese Methoden können vom Client Code direkt aufgerufen werden.
Ebenso kann die explizite Aktivierung/Deaktivierung innerhalb eines `within` Blocks geschehen:

```
within(myTeam) { stmts }
```

Dabei werden zwar vom Client direkt keine Methoden zur Aktivierung aufgerufen, allerdings wird dieser Block vom Compiler durch folgenden Statements ersetzt:

```
1.  int oldState = myTeam._OT$saveActivationState();
2.  try {
3.      myTeam.activate();
4.      //stmts
5.  } finally {
6.      myTeam._OT$restoreActivationState(oldState);
7.  }
```

Letztendlich wird also wieder die Methode `activate()` zur Teamaktivierung verwendet. Die Methoden `_OT$saveActivationState()`,

`_OT$restoreActivationState()` dienen dazu den Aktivierungszustand vor dem `within` Block danach wiederherzustellen. Wenn also ein Team vor dem Block schon aktiv war, bleibt es in diesem Zustand, war es allerdings nicht aktiv, wird es danach wieder deaktiviert.

Für die implizite Aktivierung/Deaktivierung eines Teams existieren die Methoden `_OT$implicitlyActivate` und `_OT$implicitlyDeactivate`. Diese können nicht vom Client aufgerufen werden.

Bei der Aktivierung ist es in der bisherigen Umsetzung von *OT/J* nötig, dass sich das Team bei den Basisklassen anmeldet, wenn dies nicht schon geschehen ist. Da das bei allen Arten der Aktivierung geschieht, existiert die Methode `doRegistration`, die wie folgt definiert ist:

```
1.  private void doRegistration() {
2.      if (_OT$registrationState == _OT$UNREGISTERED) {
3.          _OT$registerAtBases();
4.          _OT$registrationState = _OT$REGISTERED;
5.      }
6.  }
```

Hier wird also zuerst überprüft, ob das *Team* sich schon bei den Basisklassen registriert hat. Wenn dies nicht der Fall ist, wird die Methode `_OT$registerAtBases()` aufgerufen, die von den konkreten Teams überschrieben wird und dafür sorgt, dass das *Team* bei allen Basisklassen registriert wird.
Diese Methode muss für die neue Webestrategie geändert werden, da sich die *Teams* nicht mehr bei den Basisklassen anmelden, sondern bei der Klasse `TeamManager` (s. 6.2) . Diese Klasse bietet die Methode

```
public static void handleTeamStateChange(Team t,
        TeamStateChange stateChange);
```

an, über die sich *Teams* registrieren können (`TeamStateChange.REGISTER`). Hier werden dann auch die Ids aus den Attributen der Class Datei des *Teams* ausgelesen.

Das heißt die Methode `_OT$registerAtBases` kann entfallen, da die Registrierung nicht mehr abhängig von einem konkreten *Team* ist, sondern vollständig in der Klasse `Team` implementiert werden kann.
Die Methode `doRegistration()` muss wie folgt geändert werden:

```
1.  private void doRegistration() {
2.      if (_OT$registrationState == _OT$UNREGISTERED) {
3.          TeamManager.handleTeamStateChange(this,
4.                  TeamStateChange.REGISTER);
5.          _OT$registrationState = _OT$REGISTERED;
6.      }
7.  }
```

Analog zur Methode `doRegistration()` muss die Methode `doUnregistration()` zur Abmeldung der *Teams* geändert werden. Die Methode `_OT$unregisterFromBases()` kann entfallen und die Methode `doUnregistration()` muss wie folgt redefiniert werden.

```
1.  private void doUnregistration() {
2.      if (_OT$registrationState == _OT$REGISTERED) {
3.          TeamManager.handleTeamStateChange(this,
4.                  TeamStateChange.UNREGISTER);
5.          _OT$registrationState = _OT$UNREGISTERED;
6.      }
7.  }
```

Nachdem in diesem Kapitel gezeigt wurde, wie *dynamisches Aspektweben zur Laufzeit* für *ObjectTeams/Java* funktionieren kann, soll nun im nächsten Kapitel beschrieben werden, wie die neue *OT/J-Laufzeitumgebung* implementiert werden soll, um das *Laufzeitweben* zu unterstützen.

6 Technische Umsetzung des Laufzeitwebens

Technisch soll die neue Webestrategie mit *JPLIS* umgesetzt werden. Allerdings sind bei diesem Interface wie in 2.2.2 beschrieben, zur Laufzeit nur Transformationen erlaubt, die nicht die Signatur von Klassen, Methoden oder Felder verändern (*schemaverändernde Transformationen*). Erlaubt sind nur *schemaerhaltende Redefinition*. Es kann also nur der Code von Methoden verändert werden.
Da aber auch für die neue Strategie *schemaverändernde Redefinitionen* nötig sind, kann nicht der gesamte Webevorgang zur Laufzeit erledigt werden.
Das Hinzufügen der Methoden `callOrig`, `callOrigStatic`, `callAllBindings`, `access` und `accessStatic` muss also zur Ladezeit der Klasse geschehen. Dazu können diese Methode zunächst leer implementiert sein und zur Laufzeit bei Bekanntwerden der konkreten Bindungen entsprechend verändert werden.
Deshalb wird diese Webestrategie auch als *2-Phasen-Weben* bezeichnet.
In diesem Kapiteln soll nun gezeigt werden, welche Strukturen nötig sind, um das *2-Phasen-Weben* umzusetzen.

6.1 Weben der Klassen zur Ladezeit

Wie in 2.2.1 und 4.3.2 gezeigt, wird für den Einsatz von *JPLIS* ein *Java agent* benötigt. Dieser kann unverändert aus der aktuellen Implementierung der *OT/J-Laufzeitumgebung* übernommen werden, da er lediglich die Aufgabe hat, die Klasse `ObjectTeamsTransformer` als `ClassFileTransformer` zu registrieren und die übergebene Instanz des Interfaces `Instrumentation` zu speichern. Beide Aufgaben werden auch weiterhin erhalten bleiben.

Die Klasse `ObjectTeamsTransformer` soll auch weiterhin das Weben zur Laufzeit übernehmen. Grundsätzlich müssen zur Ladezeit nur die o.g. Methoden leer hinzugefügt werden, damit zur Laufzeit keine *schemaverändernden Transformationen* mehr nötig sind. Allerdings können Bindungen, die zum Ladezeitpunkt einer Klasse schon bekannt sind, auch schon gewoben werden. Diese Fallunterscheidung und entsprechende Bearbeitung des Bytecodes der Klasse müssen also vom `ObjectTeamsTransformer` (bzw. weiteren Klassen, die er aufruft) durchgeführt werden. An dieser Stelle soll noch nicht beschrieben werden, wie der `ObjectTeamsTransformer` genau arbeitet. Dazu ist es zunächst nötig zu erklären, wie Bindungen verwaltet werden. Erst dann wird in 6.3.3 die genaue Funktionsweise des `ObjectTeamsTransformers` dargestellt.

6.2 Verwalten der Bindungsinformationen

Den initialen Einstiegspunkt für die neue Laufzeitumgebung von *OT/J* stellt die Aktivierung eines *Teams* dar. Zu diesem Zeitpunkt werden die Bindungsinformationen aus den Attributen der Class Dateien der *Teams* ausgelesen und entsprechend weiter verwendet.
Wie schon erwähnt ruft ein Team bei seiner Aktivierung die Methode

```
public static void handleTeamStateChange(Team t,
        TeamStateChange stateChange)
```

der Klasse `TeamManager` mit dem `TeamStateChange REGISTER` auf.
Diese Klasse hat im Wesentlichen die Aufgabe die Bindungsinformationen zu verwalten und den Webevorgang anzustoßen. Dabei existieren unterschiedliche Strukturen für *Callin Bindungen* und *Decapsulation Bindungen*. Diese sollen im Folgenden vorgestellt werden.

6.2.1 Verwalten der Bindungsinformationen für Callin Bindungen

Wie schon in 5.1 erwähnt müssen im `TeamManager` sowohl die aktiven Teams als auch ihre *Callin ids* gespeichert werden. Dies muss in Abhängigkeit eines *Joinpoints,* also einer Basismethode in einer Basisklasse, geschehen, da jede Basismethode, die für sie relevanten Teams abfragen können muss. Dazu existieren die beiden Maps

```
private static Map<Integer, List<Teem>> _teams;
private static Map<Integer, List<Integer>> _callinIds;
```

In ihnen werden die Teams und die *Callin ids* über die *Joinpoint id* gespeichert.
Diese können dann über die Methoden

```
public static Team[] getTeams(int joinpointId);
public static int[] getCallinIds(int joinpointId);
```

von den Basisklassen ausgelesen werden.
Gefüllt werden sie bei der Aktivierung eines Teams in der Methode `handleTeamStateChange`. Dort wird für ein Team eine Liste von Objekten der Klasse *Binding* ausgewertet. Wie das im Detail geschieht wird in 6.3.3 gezeigt.

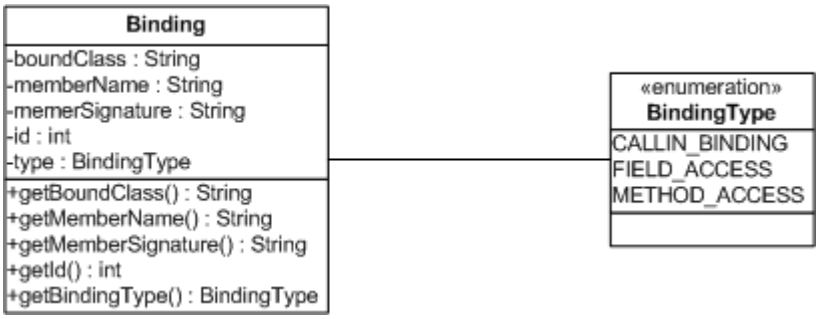

Abbildung 6.1: Die Klasse Binding

Die Klasse `Binding` repräsentiert eine Bindung. Diese kann entweder eine *Callin Bindung* (`BindingType.CALLIN_BINDING`), ein Zugriff auf ein nicht sichtbares Feld (`BindingType.FIELD_ACCESS`) oder eine nicht sichtbare Methode (`BindingType.METHOD_ACCESS`) sein. In dem Fall, dass die Bindung eine *Callin Bindung* ist, wird die *Joinpoint id* aus dem Namen der Basisklasse, dem Namen der Basismethode und der Signatur der Basismethode generiert. Diese bezeichnet den *Joinpoint* global eindeutig.
Das Team und die für diese Bindung im Team gespeicherte *Callin id* werden dann den Listen für diesen *Joinpoint* hinzugefügt.

Beim Aufruf der Methode `handleTeamStateChange` bei der Deaktivierung eines werden sie analog dazu aus diesen Listen wieder entfernt.

Wie die Liste von Bindungen erstellt wird und wo sie gespeichert wird, kann in 6.3.1 bzw. 6.4.2 nachgelesen werden.

6.2.2 Verwalten der Information für Decapsulation

Für die Fälle, dass es sich bei der Bindung um einen Zugriff auf ein nicht sichtbares Feld oder eine nicht sichtbare Methode der Basisklasse handelt, ist in dem `Binding` Objekt die zugehörige *Access id* (s. 5.5) gespeichert. Da diese wie erwähnt nur in einem *Team* eindeutig ein Member einer Basisklasse identifiziert, wird für den *Joinpoint* wie bei den *Callin Bindungen* eine *Joinpoint id* generiert. Diese wird dann in der Struktur

```
private static Map<Teem, List<Integer>> _accessIdMap;
```

gespeichert und kann dann von den Basisklassen über die Methode

```
public static int getMemberId(int accessId, Team teem);
```

ausgelesen werden. Dabei gibt die *Access id* die Position in den Listen an. Wenn also in einem Team `t` ein beliebiges Feld die *Access id* `2` hat, dann kann die zugehörige *Joinpoint id* über den Aufruf `_accessIdMap.get(t).get(2)` ermittelt werden.

Bei der Deaktivierung eines Teams werden die *Access ids* aus der Map wieder entfernt.

Im Folgenden soll nun beschrieben werden, wie die existierenden Bindungen ermittelt und gespeichert werden.

6.3 Abstrakte Repräsentation von Klassen und Teams

Die bisherige Implementierung der *OT/J-Laufzeitumgebung* setzt häufig auf statische, zentrale Strukturen (z.B. zum Verwalten der Bindungsinformationen). Dagegen werden in der neuen *Laufzeitumgebung* die nötigen Informationen der Klassen und *Teams* (also z.B. die Bindungsinformationen) dezentral in Repräsentationen dieser Klassen und *Teams* verwaltet.
Diese Repräsentationen stellen die Klassen `AbstractBoundClass` und `AbstractTeam` dar. Dabei ist `AbstractTeam` eine Subklasse von `AbstractBoundClass`. Das ist nötig, weil auch ein *Team* eine Basisklasse sein kann und deswegen für ein *Team* die gleichen Informationen benötigt werden, wie für eine "normale" Basisklasse. Welche Informationen das sind und wie sie beschafft werden, soll im Folgenden erläutert werden.

6.3.1 Die Klasse AbstractBoundClass und AbstractTeam

Natürlich benötigt jede Instanz der Klasse `AbstractBoundClass` den Namen der Klasse, die sie repräsentiert. Es wichtig an dieser Stelle den *vollqualifizierten Namen (FQN)* zu verwenden. Der *vollqualifizierte Name* enthält nicht nur den Namen der Klasse, sondern auch des Packages, in dem diese Klasse definiert ist. Dies ist nötig um die Klasse eindeutig identifizieren zu können[1].
Auch werden die Namen und Signaturen von Methoden und Feldern einer Klasse benötigt, um die Bindungen richtig zuzuordnen.
Zuletzt werden auch Informationen über die Generalisierungshierarchie, in der sich eine Klasse befindet, benötigt. Es müssen also die Subklassen und die Superklasse gespeichert werden.
Dies ist nötig, da Bindungen auch auf Seite der Basisklassen vererbt werden können (s. 3.6) und somit die Bindung einer Klasse auch auf ihre Subklassen Einfluss haben kann. Dabei ist es allerdings nicht trivial die Subklassen einer Klasse zu ermitteln.

[1] Allerdings reicht auch der *FQN* nicht aus, wie wir in 7.4.2 sehen werden.

Zum einen haben Klasse keine Verweise auf ihre Subklassen. Zum anderen können während der Laufzeit eines Programms beliebig neue Klassen (also auch Subklassen einer schon bekannten Klasse) hinzukommen. Das heißt man kann niemals davon ausgehen alle Subklassen einer Klasse zu kennen.
Wie dieses Problem gelöst werden kann, wird später gezeigt.

Zusätzlich zu diesen Informationen speichert die Klasse `AbstractTeam`, wie schon erwähnt, die Bindungen aller *Rollen*, die zu diesem *Team* gehören.

Um diese Informationen (bis auf die Subklassen einer Klasse) zu bekommen, ist es nötig, den *Bytecode* einer Klasse zu parsen und die nötigen Informationen zu extrahieren. Wie das Arbeiten mit *Bytecode* erfolgen soll, wird in 6.4 gezeigt. Zunächst wird an dieser Stelle angenommen, dass noch nicht bekannt ist, wie genau der *Bytecode* geparsed werden soll.
Das bedeutet, dass auch die Klassen `AbstractBoundClass` und `AbstractTeam` keine Abhängigkeit zu einer konkreten Bibliothek haben sollen. Dies hat gleichzeitig den Vorteil, dass die verwendete *Bytecode library* ausgetauscht werden kann, ohne die Klassen `AbstractTeam` oder `AbstractBoundClass` zu verändern.

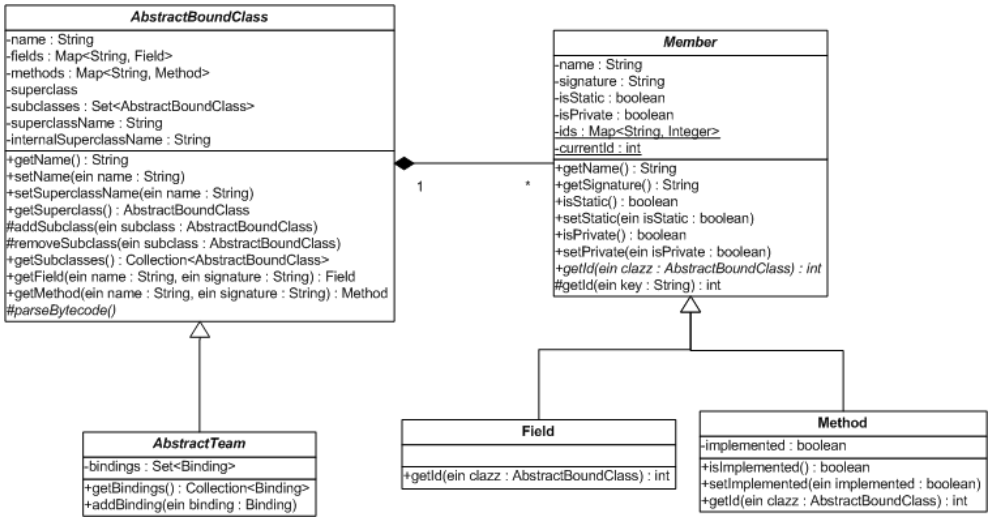

Abbildung 6.2: Die Klassen AbstractBoundClass und AbstractTeam

Um den *Bytecode* parsen zu können, muss eine `AbstractBoundClass` natürlich erstmal Zugriff auf ihn haben. Wie dieser umgesetzt wird soll im Folgenden Kapitel gezeigt werden.

6.3.2 Verwalten des Bytecodes der Klassen

Grundsätzlich kann man sich zwei Ansätze vorstellen, wie auf den *Bytecode* einer Klasse Zugriff erlangt werden kann:

1. *Permanente Verfügbarkeit*: Hierbei wird der *Bytecode* im Speicher gehalten und es kann jederzeit auf ihn zugegriffen werden.

2. *On-Demand Zugriff*: Der *Bytecode* wird jedesmal eingelesen, wenn Zugriff auf ihn benötigt wird.

Leider ermöglichen es beide Zugriffsarten nicht, den Zugriff hinsichtlich Speicherbedarf und Performance zu optimieren.
Im ersten Fall kann sehr schnell auf den *Bytecode* zugegriffen werden. Hierfür ist lediglich der Zugriff auf eine Map (oder ähnliches) nötig. Dafür steigt der Speicherbedarf bei einer Umgebung mit vielen Klassen sehr schnell an.
Die zweite Zugriffsart belastet dagegen den Speicher kaum, braucht aber Zeit um den *Bytecode* jedesmal neu einzulesen.
Natürlich kann man sich auch Mischformen der beiden Arten vorstellen, in denen zum Beispiel nur der *Bytecode* einer bestimmten Anzahl von Klassen im Speicher gehalten wird und der Rest eingelesen werden muss.

Die Frage, welche Strategie verwendet werden soll, hängt vom Kontext (also dem Programm, dass OT/J verwendet) ab. Steht diesem Programm z.B. beliebig viel Speicherplatz zur Verfügung, würde man die erste Strategie verwenden.
Es ist also sinnvoll in der Klasse `AbstractBoundClass` noch nicht festzulegen, wie diese Klasse Zugriff auf den *Bytecode* erlangen soll.
Aus diesem Grund existiert das Interface `IBytecodeProvider`. Dieses enthält ausschließlich die Methoden

```
public byte[] getBytecode(String className);
public byte[] setBytecode(String className, byte[] bytecode);
```

Für dieses Interface kann es nun verschiedene Implementierungen geben, die aber für die `AbstractBoundClass` transparent sind.
Das Parsen des Bytecodes geschieht dann in der Methode:

```
protected abstract void parseBytecode();
```

Da hier noch nicht bekannt ist, welche *Bytecode library* verwendet werden soll, ist die Methode `parseBytecode` abstrakt. Das heißt, es muss eine Subklasse existieren, die mit Kenntnis der Library zum Parsen des Bytecodes diese Methode implementiert.
Es ist also nicht nötig, dass die `AbtractBoundClass` selbst eine Instanz des Interfaces `IBytecodeProvider` erhält, sondern erst ihre konkrete Subklasse. Diese Subklasse kann sich dann mit Hilfe des *Bytecode providers* den *Bytecode* holen.

Neben dem Verwalten von Informationen über Basisklassen hat die Klasse `AbstractBoundClass` auch die Aufgabe die Aspekte zu weben. Wie das funktioniert wird im nächsten Kapitel gezeigt.

6.3.3 Weben der Aspekte

Auch für das Weben der Aspekte existierte in der bisherigen Implementierung von OT/J eine zentrale Instanz. Diese Aufgabe soll nun die Klasse `AbstractBoundClass` übernehmen. Um zu verstehen, wie diese Klasse diese Aufgabe erfüllt, ist es zunächst notwendig zu verstehen zu welchem Zeitpunkt die Klassen transformiert werden.
Zwei Zeitpunkte spielen hierbei eine Rolle:

1. Der Zeitpunkt, zu dem eine Bindung bekannt wird
2. Der Zeitpunkt, an dem die Bindung in die Basisklasse gewoben wird.

Wenn der Weber *Runtime weaving* einsetzt, könnte man annehmen, dass diese beiden Zeitpunkte gleich sind. *Runtime weaving* sagt ja genau aus, dass eine Klasse während der Laufzeit zu beliebigen Zeitpunkten transformiert werden kann. Allerdings ist hierbei zu beachten, dass die Laufzeit einer Klasse erst nach ihrer Ladezeit beginnt, dass sie also nicht redefiniert werden kann, wenn sie noch nicht geladen wurde. Nach ihrer Ladezeit sind die beiden Zeitpunkte dann in der Tat gleich.

Bearbeitung der Bindungen im TeamManager

Wie schon beschrieben, werden die Bindungen eines Teams bei seiner Aktivierung im `TeamManager` ausgewertet. Dort muss also einer `AbstractBoundClass` mitgeteilt werden, dass Bindungen für sie vorliegen.
Dies geschieht über die Methode

```
public void handleAddingOfBinding(Binding binding);
```

der Klasse `AbstractBoundClass`. Der `TeamManager` muss sich also zunächst für jede Bindung eine Instanz der Basisklasse für diese Bindung holen. Wie dies geschieht wird in 6.3.4 gezeigt.
Anschließend kann der `TeamManager` dieser Instanz mitteilen, dass eine neue Bindung für die Klasse, die sie repräsentiert, existiert. Die `AbstractBoundClass` kann dann entscheiden, welche Aktionen nötig sind, um diese Bindung umzusetzen.

Dieser Entscheidungsprozess läuft also für den `TeamManager` transparent ab. Er weiß weder, ob diese Klasse schon geladen wurde, noch, ob die Klasse redefiniert werden muss, um diese Bindung umzusetzen, oder nicht.

Bei der Deaktivierung eines Teams muss die Basisklasse dagegen nicht informiert werden, da einmal gewobene Basisklassen nicht mehr geändert werden sollen. Das Entfernen der Dispatch Strukturen aus den Basisklassen ist nicht nötig und würde nur unnötig Performance kosten. Hier werden lediglich (wie in 6.2 beschrieben) die entsprechenden Datenstrukturen geändert

Bearbeiten der Bindungen in AbstractBoundClass

Um zu beschreiben, wie die Klasse `AbstractBoundClass` genau arbeitet, soll hier zunächst definiert werden, in welchen Fällen welche Transformationen an der Basisklasse und deren Super- bzw. Subklassen durchzuführen sind.
Grundsätzlich können diese Fälle anhand des Typs der Bindung unterschieden werden:

1. *Callin Bindung*
    1. Methode ist statisch
        a) Falls es die erste *Callin Bindung* zu einer statischen Methode dieser Klasse ist, muss in der Methode `callOrigStatic` ein `switch` Statement angelegt werden, dass im `default` Fall `null` zurückliefert.
        b) Verschieben des Originalcodes der Methode in die Methode `callOrigStatic`.
        c) Originalmethode so verändern, dass dort für die *Bound method id* der Methode der *Dispatch* zu den Teams geschieht (*Initial wrapper*).
    2. Methode ist nicht statisch
       In allen folgenden Fällen:
        a) Falls es die erste *Callin Bindung* zu einer nicht statischen Methode dieser Klasse ist, muss in der Methode `callOrig` ein `switch` Statement angelegt werden, dass im `default` Fall `null` zurückliefert.
           Außerdem wird auch in der Methode `callAllBindings` ein `switch` Statement angelegt, dass im `default` Fall die Methode `callOrig` aufruft.
        1. Methode ist in der Basisklasse implementiert:
            a) Der Originalcodes der Methode wird in die Methode `callOrig` verschoben.
            b) Die Originalmethode wird so verändert, dass dort die Methode `callAllBindings` aufgerufen wird (*Initial wrapper*).
            c) Die Methode `callAllBindings` muss so angepasst werden, dass dort für die *Bound method id* der Originalmethode der *Dispatch* zu den Teams geschieht.
            d) Die Bindung an alle Subklassen dieser Klasse weitergeben, wenn die Methode nicht privat ist (s. 3.6).
        2. Methode wird von einer direkten oder indirekten Superklasse geerbt:
            a) Die Methode `callAllBindings` wird so angepasst, dass dort für die *Bound method id* der Originalmethode der *Dispatch* zu den Teams geschieht.
            b) Der Code der Methode wird, in der Superklasse, in der sie implementiert ist in die Methode `callOrig` verschoben
            c) Die Methode wird in der Superklasse so angepasst, dass dort die Methode `callAllBindings` aufgerufen wird (*Initial wrapper*).

   d) Die Bindung an alle Subklassen dieser Klasse weitergeben, wenn die Methode nicht privat ist (s. 3.6).
2. Zugriff auf ein nicht sichtbares Member (*Decapsulation*):
   1. Das Member ist statisch
      1. Falls es der erste Zugriff auf ein statisches Member dieser Klasse ist, muss in der Methode `accessStatic` ein `switch` Statement angelegt werden, dass im `default` Fall die Methode `accessStatic` der Superklasse aufruft.
      2. Der Zugriff auf dieses Member in der Methode `accessStatic` für die *Access id* dieses Members wird erzeugt.
   2. Das Member ist nicht statisch
      1. Falls es der erste Zugriff auf ein nicht statisches Member dieser Klasse ist, wird in der Methode `access` ein `switch` Statement angelegt, dass im `default` Fall die Methode `access` der Superklasse aufruft.
      2. Der Zugriff auf dieses Member in der Methode `access` für die *Access id* dieses Members wird erzeugt.
      3. Die Bindung an alle Subklassen weitergeben.

Dies sind alle Transformationen die zur Laufzeit durchgeführt werden. Wie schon erwähnt müssen vorher zur Ladezeit die Methoden `callOrig`, `callOrigStatic`, `access`, `accessStatic` und `callAllBindings` angelegt worden sein.
In der Methode `handleAddingOfBinding` wird nun zunächst überprüft, welche Transformationen für diese Bindung genau auszuführen sind. Dies hängt natürlich vom Typ, aber auch davon ab, ob für dieses Member schon ein Bindung existiert.
Dazu existieren die Datenstrukturen:

```
private Map<Method, WeavingTask> completedBindingTasks;
private Map<Method, WeavingTask> openBindingTasks;
private Map<Member, WeavingTask> completedAccessTasks;
private Map<Member, WeavingTask> openAccessTasks;
```

Wie hier zu sehen ist, wird innerhalb der `AbstractBoundClass` die Klasse `WeavingTask` benutzt, um zu speichern, welche Methoden wie transformiert werden müssen.

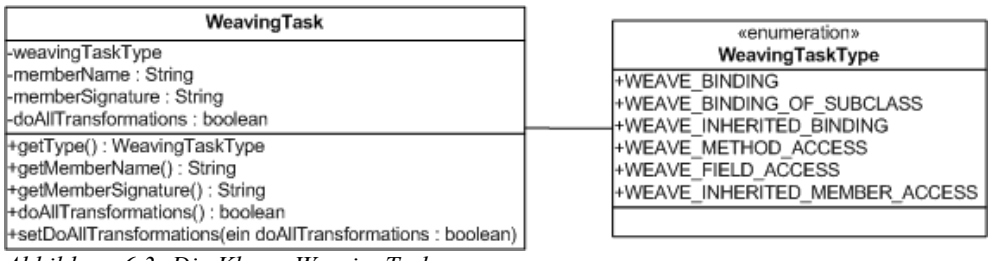

Abbildung 6.3: Die Klasse WeavingTask

Die Klasse `WeavingTask` speichert zum einen genau wie die Klasse `Binding` den Namen und die Signatur des betroffenen Members. Weiterhin verfügt sie über

einen Typ (`WeavingTaskType`). Die drei Typen `WEAVE_BINDING`, `WEAVE_METHOD_ACCESS` und `WEAVE_FIELD_ACCESS` korrespondieren dabei direkt mit den Bindungsarten `CALLIN_BINDING`, `FIELD_ACCESS` und `METHOD_ACCESS`. Zusätzlich existieren die Typen `WEAVE_BINDING_OF_SUBCLASS`, `WEAVE_INERITED_BINDING` `WEAVE_INHERITED_MEMBER_ACCESS`. Diese werden benötigt um die Bindungen auch in den Sub- bzw. Superklassen der gebundenen Klasse umsetzen zu können (s. obige Auflistung). Diese `WeavingTasks` werden zunächst in der Methode `handleAddingOfBindung` analog zu dem übergebenen `Binding` erstellt. Anschließend wird anhand der beschriebenen Datenstrukturen überprüft, ob, und wenn ja, welche Aktionen ausgeführt werden müssen, um eine Bindung umzusetzen.
Zusätzlich zu diesen Informationen enthält die Klasse `WeavingTask` das Attribut `doAllTransformations` vom Typ `boolean`.
Notwendig ist dieses Attribut nur für den Fall 1.2.2, wenn also eine *Callin Bindung* zu einer Basismethode gewoben werden soll, die nicht in der Basisklasse sondern einer direkten oder indirekten Superklasse implementiert ist. In diesem Fall wird zwar die Originalmethode in der Superklasse gewoben, aber nicht die Methode `callAllBindings`. Wenn anschließend eine *Callin Bindung* bekannt wird, die die Superklasse als Basisklasse hat, muss zwar die Methode `callAllBindings` noch gewoben werden, aber nicht mehr die Originalmethode. In diesem Fall ist `doAllTransformations false`. In alle anderen Fällen genügt es mit Hilfe der Maps zu überprüfen, ob schon ein Task des gleichen Typs für ein Member vorliegt. Ist dies der Fall muss nichts mehr getan werden.
Dazu speichern die Maps `openAccessTasks` und `openBindingTasks`, getrennt nach Art des `WeavingTaskTypes`, die noch zu erledigenden Tasks und `completedAccessTasks` und `completedBindingTasks` die schon Erledigten.
`WeavingTasks` mit dem Typ `WEAVE_BINDING` können dabei nur für Methoden gespeichert werden, da es keine *Callin Bindungen* zu Feldern geben kann. Andere `WeavingTasks` können sich dagegen auf Felder oder Methoden beziehen.

Falls bis du diesem Zeitpunkt weder ein noch offener noch ein schon erledigter Task für ein Member vorliegt, wird dieser Task den offenen Tasks hinzugefügt.
Falls diese Klasse schon geladen wurde, wird dieser Task auch gleich abgearbeitet. Ist dies nicht der Fall wird er zunächst nur gespeichert.
Das Abarbeiten der Tasks passiert in der Methode

```
public void handleTaskList();
```

Diese Methode arbeitet genau nach dem oben vorgestellten Schema, weswegen sie hier nicht im Detail vorgestellt werden soll.
Die Transformationen werden letztendlich durch folgende Methoden durchgeführt:

```
1.  void startTransformation();
2.  void endTransformation();
3.  void prepareAsPossibleBaseClass();
4.  void createSuperCallInCallOrig(intboundMethodId);
5.  void createCallAllBindingsCallInOrgMethod(
6.          Method boundMethod, int joinpointId);
7.  void createDispatchCodeInCallAllBindings(int joinpointId,
8.          int boundMethodId);
9.  void moveCodeToCallOrig(Method method, int joinpointId);
10. void prepareForFirstTransformation();
11. void prepareForFirstStaticTransformation();
12. boolean isFirstTransformation();
13. void createDispatchCodeInOrgMethod(Method method,
14.         int joinpointId, int boundMethodId);
15. void prepareForFirstMemberAccess();
16. void weaveFieldAccess(Field field, int accessId);
17. void weaveMethodAccess(Method method, int accessId);
```

Da alle diese Methoden Kenntnis der konkreten *Bytecode library* benötigen und diese Klasse davon unabhängig sein soll, sind sie abstrakt und müssen in einer konkreten Subklasse implementiert werden.
Die Methode `prepareAsPossibleBaseClass` übernimmt dabei die Aufgabe, die nötigen Methoden zur Ladezeit hinzuzufügen. Aufgerufen wird sie, wie alle anderen dieser Methoden, von der Methode `handleTaskList`. Die Methoden `prepareForFirst...` erzeugen die `switch` Statements in den entsprechenden Methoden.
Bevor Transformationen durchgeführt werden können, muss die Methode `startTransformation` aufgerufen werden. Nachdem alle Transformationsmethoden ausgeführt wurden, muss ein Aufruf der Methode `endTransformation` folgen.
Diese Implementierung ermöglicht es, verschiedene Strategien beim Ausführen der Transformation zu verfolgen. So muss nicht jede Transformation einzeln ausgeführt werden, sondern es besteht auch die Möglichkeit zuerst alle nötigen Transformationen zu sammeln und sie erst beim Aufruf der Methode `endTransformation` auszuführen. Die Entscheidung, welche Strategie benutzt werden soll, trifft die konkrete Subklasse.

Die Methode `handleTaskList` wird durch zwei Aktionen getriggert:

1. Hinzufügen einer Bindung, wenn die Klasse schon geladen wurde: Dies passiert, wie eben erläutert, in der Methode `handleAddingOfBinding`.
2. Laden der Klasse: In diesem Fall wird `handleTaskList` von der Methode `transformAtLoadTime` aufgerufen.

`transformAtLoadTime` wird vom `ObjectTeamsTransformer` in der Methode `transform` beim Laden einer Klasse aufgerufen. Dazu holt sich der `ObjectTeamsTransformer` zunächst eine Instanz von `AbstractBoundClass` zu dem übergebenen Namen und ruft auf ihr dann die Methode auf. Anschließend wird nach der Transformation der neue *Bytecode* dieser Klasse zurückgeliefert.

Hier werden also zur Ladezeit der Klassen schon so viele Transformationen vorgenommen, wie möglich. Dies hat den Vorteil, dass diese Transformationen dann nicht mehr zur Laufzeit erledigt werden müssen. Da das Redefinieren zur Laufzeit mehr Zeit kostet als zur Ladezeit, wird somit eine höhere Performance erreicht.

Als nächstes ist die Frage zu klären, an welcher Stelle im Programm Instanzen der Klasse `AbstractBoundClass` erzeugt werden. Dieser Frage widmet sich das nächste Kapitel.

6.3.4 Das Class repository

Grundsätzlich existieren zwei Stellen, an denen Instanzen der Klasse `AbstractBoundClass` bzw. ihrer Subklasse `AbstractTeam` benötigt werden.

1. `TeamManager`
2. `ObjectTeamsTransformer`

Im `TeamManager` werden Instanzen von beiden Klassen benötigt. Für ein Team, das sich registriert, muss ein `AbstractTeam` und für die Klassen, die Basisklassen des Teams sind, jeweils eine `AbstractBoundClass` geholt werden. Dabei steht fest, dass die Klasse des Teams schon geladen wurde, da ja zu seiner Aktivierung eine Instanz des Teams notwendig ist.
Bei den Basisklassen dagegen ist nicht bekannt, ob sie schon geladen worden sind. Dieses Wissen ist aber für den `TeamManager` auch nicht relevant (s. 6.3.3).

Anders verhält es sich in der Klasse `ObjectTeamsTransformer`. Diese weiß, dass die Klassen noch nicht geladen wurden, da sie ja genau beim Laden einer Klasse aufgerufen wird.
Sie hat de Aufgabe der Klasse bzw. dem `BytecodeProvider` den *Bytecode* initial hinzuzufügen.

Die Klasse, die die `AbstractBoundClasses` und die `AbstractTeams` verwaltet und sie diesen beiden Stellen zur Verfügung stellt, ist die Klasse `ClassRepository`.
Diese bietet dazu die folgenden Methoden an:

```
public AbstractBoundClass getClass(String className);
public AbstractBoundClass getClass(String className,
        byte[] byteCode);
public AbstractTeam getTeam(String teamName);
```

Die Methoden `getClass(String className)` und `getTeam` werden vom `TeamManager` benutzt. Über die Methode `getClass(String className, byte[] byteCode)` kann sich der `ObjectTeamsTransformer` eine Instanz von `AbstractBoundClass` holen und gleichzeitig den *Bytecode* setzen.
Dabei gilt, dass pro Klassennamen auch nur eine Instanz der Klasse `AbstractBoundClass` bzw. `AbstractTeam` existiert:

$$\textit{Seien s1 und s2 zwei Strings , dann gilt : } s1.equals(s2) \rightarrow getClass(s1) == getClass(s2)$$

Eigentlich ist es natürlich auch die Aufgabe dieser Klasse Instanzen von `AbstractBoundClass` und `AbstractTeam` zu erzeugen. Wie schon erwähnt sind diese beiden Klassen abstrakt und können deswegen nicht instanziiert werden. Das heißt, das `ClassRepository` müsste Instanzen der konkreten Klassen erzeugen, die von `AbstractBoundClass` bzw. `AbstractTeam` erben und sie vollständig implementieren.

Allerdings soll auch das `ClassRepository` noch keine Kenntnis einer konkreten *Bytecode library* bzw. der Klassen, die sie benutzen, haben. Deshalb besitzt das `ClassRepository` die abstrakte Methode

```
protected abstract AbstractTeam createClass(String name);
```

die eine Instanz von `AbstractTeam` (und damit auch von `AbstractBoundClass`) erzeugt und sie zurückliefert.
Das bedeutet natürlich, dass auch die Klasse `ClassRepository` abstrakt sein muss.
Leider lässt sich die Abhängigkeit zu den Klassen, die Kenntnis der konkreten *Bytecode library* haben, nicht hundertprozentig vermeiden.
Da die Klasse `ClassRepository` ein *Singleton* ist, es also nur eine Instanz von ihr im gesamten Programm geben kann, besitzt sie die Methode

```
public static ClassRepository getInstance();
```

Diese Methode liefert immer die gleiche Instanz zurück.
Hier muss die Instanz einer konkreten Klasse, die von `ClassRepository` erbt, erstellt und zurückgeliefert werden. Diese Methode stellt die Schnittstelle zwischen den Klassen, die Kenntnis einer konkreten *Bytecode library* haben und der übrigen Laufzeitumgebung dar. Die Klärung der Frage, von welcher Klasse in dieser Methode eine Instanz erzeugt wird, wird Teil des nächsten Kapitels sein.

Allerdings sind die Aufgaben der Klasse `ClassRepository` noch nicht vollständig beschrieben, da eine weitere Stelle existiert, an der Instanzen von `AbstractBoundClass` benötigt werden.
Wie schon erwähnt müssen Bindungen zu einer Basisklasse auch an deren Subklassen weitergegeben werden. Dieses stellt wie in 6.3.1 gezeigt ein Problem dar, da man nicht davon ausgehen kann, alle Subklassen einer Klasse zu kennen.
Aus diesem Grund hat die Liste der Subklassen einer `AbstractBoundClas` immer mindestens ein Element, nämlich eine *anonyme Subklasse*. Dieses Element ist eine `AbstractBoundClass` mit dem Namen `"AnonymousSubclass"`. Sie steht stellvertretend für alle noch unbekannten Subklassen. Genau wie "echte" `AbstractBoundClasses` können dieser Instanz Bindungen hinzugefügt werden.
Das `ClassRepository` hat nun die Aufgaben, diese anonymen Subklassen zu erzeugen und entsprechend zu verwalten.
Zu dem Zeitpunkt, an dem eine Klasse geladen wird, kann sie nun schon aus zwei

"Richtungen" `WeavingTasks` haben. Ersten kann die dieser Klasse entsprechende `AbstractBoundClass` schon `WeavingTasks` haben. Diese `WeavingTasks` sind dann ausschließlich direkte Bindungen zu dieser Klasse. Außerdem können für diese Klasse aber auch schon geerbte Bindungen vorliegen. Diese sind dann in einer anonymen Subklasse gespeichert. Wird diese Klasse geladen müssen alle `WeavingTasks` zusammengeführt werden. Diese Aufgabe erledigt das `ClassRepository` in der Methode

```
public void linkClassWithSuperclass(AbstractBoundClass clazz)
```

Damit sich jede Klasse genau eine anonyme Subklasse holen kann, stellt das `ClassRepository` die folgende Methode bereit:

```
protected AbstractBoundClass getAnonymousSubclass(
        AbstractBoundClass abstractBoundClass);
```

Nachdem nun gezeigt wurde, wie Instanzen der Klassen `AbstractBoundClass` verwaltet werden, bleibt noch die Frage, welche Klasse die konkrete Implementierung von `AbstractBoundClass` ist und mit welcher *Bytecode library* sie arbeitet. Diese Frage soll im nächsten Kapitel beantwortet werden.

6.4 Bearbeiten des Bytecodes

Grundsätzlich existieren zwei Ansätze, wie man *Bytecode* parsen bzw. manipulieren kann.

1. Durch speziellen selbstgeschriebenen Code
2. Durch eine *Bytecode library*

Beide Ansätze haben dabei ihre Vor- und Nachteile. Durch Code, der speziell auf eine konkrete Problemstellung zugeschnitten ist, erreicht man im Normalfall eine performantere Lösung als mit Hilfe einer allgemeinen Library. Dafür ist bei der speziellen Lösung sowohl der Entwicklungs- als auch der Wartungsaufwand höher.
Grundsätzlich ist allerdings eine allgemeine Lösung dem speziellen Code vorzuziehen, solange die Performance dieser Lösung nicht allzu stark gegenüber der Speziellen abfällt.
Im Nachfolgenden soll vorgestellt werden, welche *Bytecode libraries* für die Benutzung in der *OT/J-Laufzeitumgebung* in Frage kommen.

6.4.1 Vergleich der Bytecode libraries

Die meisten dieser Bibliotheken arbeiten nach dem Ansatz, den *Bytecode* zu parsen und in ein objektorientiertes Modell zu übertragen. Über dieses Modell können dann Informationen über die Klasse ausgelesen und deren *Bytecode* verändert werden. Vertreter dieser Art sind z.B. *BCEL*, *Javassist* und *Serp*. Die Performance dieser Bibliotheken hängt im Wesentlichen von der Detaillierung des Objektmodells ab, das sie verwenden. Einen anderen Ansatz dagegen verwendet die *Bytecode library ASM* (s. [Asm02]).

ASM

Diese Bibliothek lässt dem Benutzer die Wahl, ob er über ein objektorientiertes Modell (*Tree API*) oder nach dem *Visitor pattern* (*Core API*) arbeiten möchte.

Im Nachfolgenden sollen beide Arbeitsweisen von *ASM* vorgestellt werden.

Core API von ASM

Der Einstiegspunkt der *Core API* von *ASM* ist das Interface `ClassVisitor`. Es deklariert alle Methoden, die beim Parsen einer Klasse aufgerufen werden.

```
1.  public interface ClassVisitor {
2.      void visit(int version, int access, String name,
3.              String signature, String superName,
4.              String[] interfaces);
5.      void visitSource(String source, String debug);
6.      void visitOuterClass(String owner, String name,
7.              String desc);
8.      AnnotationVisitor visitAnnotation(String desc,
9.              boolean visible);
10.     void visitAttribute(Attribute attr);
11.     void visitInnerClass(String name, String outerName,
12.             String innerName, int access);
13.     FieldVisitor visitField(int access, String name,
14.             String desc, String signature, Object value);
15.     MethodVisitor visitMethod(int access, String name,
16.             String desc, String signature,
17.             String[] exceptions);
18.     void visitEnd();
19. }
```

Ein konkreter Visitor kann dieses Interface implementieren und die nötigen Informationen beim Aufruf der Methoden speichern. Soll z.B. der Name der Superklasse einer Klasse gespeichert werden müsste der konkrete Visitor so aussehen:

```
1.  class MyVisitor implements ClassVisitor {
2.      private String superName;
3.
4.      @Override
5.      public void visit (int version, int access,
6.              String name, String signature,
7.              String superName, String[] interfaces) {
8.          this.superName = superName;
9.      }
10.
11.     public String getSuperName() {
12.         return superName;
13.     }
14.
15.    // Es würden die Leerimplementierungen der restlichen
16.    // Methoden folgen.
17. }
```

Das Parsen der Klasse wird über einen `ClassReader` gestartet, dem dieser Visitor hinzugefügt wird:

```
1. ...
2. byte[] bytecode = ...;
3. MyVisitor myVisitor =  new MyVisitor();
4. ClassReader reader = new ClassReader(bytecode);
5. cr.accept(myVisitor, null, 0);
6. String superName = myVisitor.getSuperName();
7. ...
```

Ähnlich wie das Parsen einer Klasse funktioniert auch das Manipulieren ihres Bytecodes.
Dazu existiert in der *Core API* die Klasse `ClassAdapter`, die das Interface `ClassVisitor` implementiert. Alle Klassen, die *Bytecode* manipulieren wollen, müssen von ihr ableiten. Jeder konkrete `ClassAdapter` bekommt im Konstruktor eine Instanz der Klasse `ClassWriter` übergeben. Sollen jetzt Veränderungen am *Bytecode* vorgenommen werden, werden die entsprechenden Methoden dieses `ClassWriters` aufgerufen.
Gestartet wird dieser Vorgang wieder über einen `ClassReader`. Abbildung 6.4 zeigt das Vorgehen exemplarisch für den Fall, dass der Name einer Klasse geändert werden soll.

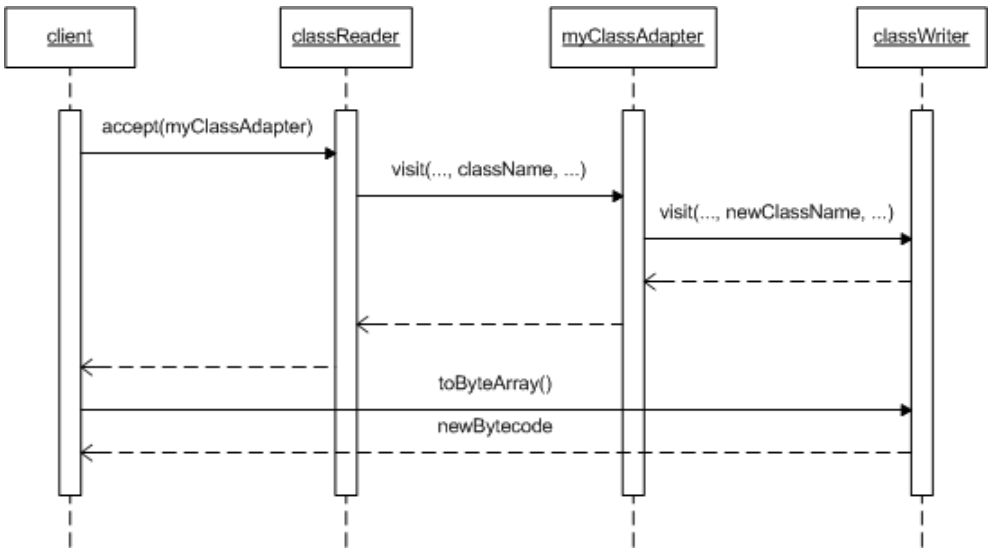

Abbildung 6.4: Ablauf des Manipulierens von Bytecode

Der Vorteil der *Core API* ist, dass diese deutlich performanter arbeitet, als wenn zuerst ein objektorientiertes Modell einer Klasse erstellt werden muss. Allerdings lassen sich mit Hilfe dieser API auch nicht alle Anwendungsfälle umsetzen (wie in 6.4.2 gezeigt wird). Aus diesem Grund bietet ASM auch ein objektorientiertes Modell der Klassen an.

Tree API von ASM

In Dokumenten über ASM wird das objektorientierte Modell auch als *Tree API* bezeichnet.
Man merkt ihr an, dass sie nach der *Core API* implementiert wurde, da auch sie nicht ganz auf Visitors verzichtet. Diese werden hier aber nur beim initialen Parsen einer Klasse benötigt. Danach kann auf der Objektstruktur gearbeitet werden.
Der *Bytecode* einer Klasse wird dabei durch die Klasse `ClassNode` repräsentiert. Sie enthält die allgemeinen Informationen der Klasse, ihre Methoden (`MethodNode`), ihre Felder (`FieldNode`) u.s.w. Um performanter arbeiten zu können sind im Gegensatz zum Standard der *OOP* alle Felder `public` und es existieren keinen Getter oder Setter für diese Felder.
Die Klasse `ClassNode` implementiert wieder das Interface `ClassVisitor`. Dabei implementiert sie die Methode `accept` so, dass dort die Objekt-Struktur aufgebaut wird.
Erzeugt wird ein solcher `ClassNode`, indem er einem `ClassReader` als Visitor übergeben wird.
Anschließend können Informationen ausgelesen oder der *Bytecode* manipuliert werden. Ist Letzteres der Fall, muss dem `ClassNode` abschließend ein `ClassWriter` als Visitor übergeben werden. Aus ihm kann dann der neue *Bytecode* ausgelesen werden.

```
1. byte[] bytecode = ...;
2. ClassNode classNode = new ClassNode();
3. ClassReader reader = new ClassReader(bytecode);
4. reader.accept(node, 0);
5. // get information and manipulate the bytecode
6. ClassWriter writer = new ClassWriter(0);
7. node.accept(writer);
8. bytecode = writer.toByteArray();
```

In der neuen *OT/J-Laufzeitumgebung* gibt es insgesamt drei Anwendungsfälle für die eine *Bytecode library* eingesetzt werden muss.

1. Parsen von Klassen
2. *Loadtime weaving*
3. *Runtime weaving*

In den ersten beiden Anwendungsfällen ist besonders auf die Performance der verwendeten Bibliothek zu achten, da diese für alle Klassen durchgeführt werden müssen und nicht nur für Basisklassen oder Teams. Der dritte Anwendungsfall bezieht sich dagegen nur auf Basisklassen.

Besonders in den ersten beiden Anwendungsfällen bietet sich also der Einsatz der *Core API* von *ASM* an, da diese mit hoher Performance arbeitet. Hier eine Objektstruktur für die Klassen zu erzeugen, wäre unnötiger Overhead. Auch in dem dritten Fall wäre der Einsatz der *Core API* wünschenswert, ist aber leider (wie in 6.4.2 gezeigt wird) nicht bzw. nur mit sehr hohem Aufwand möglich.

Im nächsten Kapitel soll nun gezeigt werden, wie *ASM* in der *OT/J-Laufzeitumgebung* benutzt wird.

6.4.2 Verwendung von ASM in der OT/J-Laufzeitumgebung

Wie schon erwähnt, fehlen der *OT/J-Laufzeitumgebung* bis jetzt noch konkrete Klassen, die von den abstrakten Klassen `AbstractBoundClass` bzw. `AbstractTeam` und `ClassRepository` erben und die fehlenden Methoden implementieren.
Diese Klassen sollen nun mithilfe der *Bytecode library ASM* umgesetzt werden.
Um die fehlenden Funktionen der Klasse `AbstractBoundClass` bzw. `AbstractTeam` zu implementieren, sollen nun zwei konkrete Klassen geschaffen werden. Die eine soll den Anwendungsfall 1, also das Parsen der Klassen, übernehmen, die Andere die beiden übrigen Anwendungsfälle, also das Manipulieren des *Bytecodes* der Klassen.

Parsen des Bytecodes

Die erste Klasse heißt `AsmBoundClass` und implementiert die Methode `parseBytecode` folgendermaßen:

```
1.  protected void parseBytecode() {
2.      if (parsed) {
3.          return;
4.      }
5.
6.      bytecode = bytecodeProvider.getBytecode(getId());
7.      if (bytecode == null) {
8.          return;
9.      }
10.     parsed = true;
11.     AsmClassVisitor cv = new AsmClassVisitor(this);
12.     ClassReader cr = null;
13.     cr = new ClassReader(bytecode);
14.     cr.accept(cv, Attributes.attributes, 0);
15.     bytecode = null;
16. }
```

Die Methode macht also nichts anderes als einen `ClassVisitor` (s. 6.4.1) zu erzeugen und diesem die `this` Referenz zu übergeben. Der `AsmClassVisitor` kann damit die ausgelesenen Informationen direkt eintragen und muss sie nicht zwischenspeichern.
Die Klasse `AsmClassVisitor` erbt von der Klasse `EmptyVisitor`. `EmptyVisitor` wiederum implementiert das Interface `ClassVisitor`, bietet allerdings für alle Methoden nur Leerimplementierungen an. Somit müssen eigene `ClassVisitor` nur die Methoden implementieren, die sie auch wirklich implementieren wollen und nicht alle, die das Interface `ClassVisitor` anbietet.

Im Falle des `AsmClassVisitors` sind das folgende Methoden:

```
1.  public void visit(int version, int access, String name,
2.          String signature, String superName,
3.          String[] interfaces);
4.  public MethodVisitor visitMethod(int access, String name,
5.          String desc, String signature,
6.          String[] exceptions);
7.  public FieldVisitor visitField(int access, String name,
8.          String desc, String signature, Object value);
9.  public void visitAttribute(Attribute attribute);
```

Hier werden also in der Methode `visit` der Name der Superklasse, in der Methode `visitMethod` die Methoden und in `visitField` die Felder der Klasse gelesen und in der Instanz von `AbstractBoundClass` gespeichert.
Die Methoden `visitMethod` und `visitField` speichern die Methoden und Felder der Klasse.
In der Methode `visitAttribute` werden die in 5.6 definierten Attribute ausgelesen. Hier werden Instanzen der Klasse `Binding` erstellt und dem `AbstractTeam` hinzugefügt.

Die Klasse `AsmBoundClass` ist natürlich wieder abstrakt, da sie nur die Methode

`parseBytecode` und nicht die übrigen Methoden implementiert.
Von dieser Klasse leitet die Klasse `AsmWritableBoundClass` ab, die die übrigen Methoden implementiert.

Manipulieren des Bytecodes

Auch wenn alle Methoden, die *Bytecode* manipulieren, in einer Klasse implementiert sind, können sie nach dem 2. und 3. Anwendungsfall, also dem Weben zur Ladezeit bzw. zur Laufzeit unterschieden werden.
Für den 2. Anwendungsfall existieren die Methoden

```
1. public void addEmptyMethod(Method method, int access,
2.          String signature, String[] exceptions);
3. public void addInterface(String name);
```

Wie schon erwähnt wird für diesen Anwendungsfall aus Performancegründen die *Core API* von *ASM* benutzt. Über die Methode `addEmptyMethod` können also leere Methoden einer Klasse hinzugefügt werden. Außerdem kann eine Klasse über die Methode `addInterface` so redefiniert werden, dass sie ein bestimmtes Interface implementiert. Hierbei ist allerdings zu beachten, dass das Interface nur hinzugefügt wird. Die Methoden, die nötig sind, um dieses Interface zu implementieren, müssen zusätzlich hinzugefügt werden. Beide Methoden erzeugen einen `ClassAdapter` (s. 6.4.1). Dieser wird allerdings aus folgendem Grund nicht sofort aufgerufen:
Beim Laden einer Klasse müssen insgesamt 6 Transformationen durchgeführt werden:

1. Hinzufügen der Methode `callOrig`
2. Hinzufügen der Methode `callOrigStatic`
3. Hinzufügen der Methode `callAllBindings`
4. Hinzufügen der Methode `access`
5. Hinzufügen der Methode `accessStatic`
6. Hinzufügen des Interfaces `IBoundBase`

Würde jede dieser Transformationen einzeln durchgeführt, müsste durch die Benutzung der *Core API* auch jedesmal der *Bytecode* durchgegangen werden. Dies würde deutlich Performance kosten, weswegen alle Transformationen in einem Durchlauf erledigt werden sollen. Das ist möglich, da alle Transformationen unabhängig voneinander sind, sich gegenseitig also nicht bedingen oder ausschließen.
Um dies umzusetzen existiert die Klasse `MultiClassAdpater`. Diese erbt wieder von `ClassAdapter` und macht nichts anderes als eine Liste von `ClassAdaptern` zu verwalten und den Aufruf ihrer `visit` Methoden an alle `ClassAdapter` in dieser Liste zu delegieren. So werden alle Transformationen in einem Durchlauf erledigt.

Die Methoden, die den *Bytecode* zur Laufzeit manipulieren, benutzen die *Tree API* von *ASM*. Bis auf einen Fall könnten diese zwar auch die *Core API* nutzen,

allerdings ist Code, der die *Tree API* benutzt übersichtlicher und daher leichter zu warten. Da für einen Fall sowieso die Objektstruktur für die Klassen erzeugt werden muss, würde es die Performance auch nicht erhöhen in allen anderen Fällen die *Core API* zu benutzen.
Dieser eine Fall ist das Verschieben des Originalcodes einer Methode in die Methode `callOrig` oder `callOrigStatic`. Das Problem hierbei ist, dass die *Core API* seriell arbeitet, d.h. sie kann nicht auf zwei Stellen im *Bytecode* (z.B. den beiden Methoden) gleichzeitig arbeiten. Das heißt, zunächst müsste der Originalcode der Basismethode geparsed und gespeichert werden. Dafür müssten allerdings erst Strukturen geschaffen werden, die jede Instruktion speichern könnten.
Außerdem kann nicht sichergestellt werden, dass ein `ClassAdapter` nicht zuerst in der Methode `callOrig`/`callOrigStatic` ankommt und dann erst in der Basismethode. In diesem Fall wären dann zwei Durchläufe nötig.
Aus diesen Gründen ist es sinnvoller für das Manipulieren des Bytecodes zur Laufzeit die *Tree API* von *ASM* einzusetzen.

Wie beim Manipulieren des Bytecodes zur Ladezeit existiert hier wieder für jede mögliche Transformation (s. 6.3.3) eine eigene Klasse und eine Methode in der eine Instanz dieser Klasse erzeugt wird. Jede dieser Klassen erbt dabei von `AbstractTransformableClassNode`. Im Wesentlichen stellt diese Klasse ein Interface mit der Methode

```
protected abstract void transform();
```

dar. In den Implementierungen dieser Methode in den verschiedenen Klassen werden die Transformationen durchgeführt. Die Klasse `AbstractTransformableClassNode` ist nur deswegen kein Interface sondern eine abstrakte Klasse, da sie diverse Hilfsfunktionen für die konkreten `ClassNodes` zur Verfügung stellt.

Die Methoden, in denen Instanzen dieser `ClassAdapter` erzeugt werden, sollen hier nicht alle dargestellt werden, da sie sich sehr ähnlich sind, sondern nur einmal exemplarisch:

```
1. protected void doSpecificTransformation(...) {
2.     nodes.add(new SpecificClassAdapter(...));
3. }
```

Es wird also ein neuer `ClassAdapter` erzeugt und einer Liste hinzugefügt.
Letztendlich werden alle Transformationen in der Methode `endTransformation`() durchgeführt. Zusätzlich wird hier noch der oben beschriebene `MultiClassAdapter` aufgerufen.

Nachdem die Transformationen durchgeführt wurden, wird die Instanz des Interfaces Instrumentation aus dem `ObjectTeamsTransformer` geholt. Auf dieser kann dann mit dem transformierten *Bytecode* die Methode

`redefineClasses` (s. 2.2.2) aufgerufen und so die Klasse redefiniert werden. Die dazu benötigte Class Instanz wird hier über den Aufruf `Class.forName(...)` geholt.

6.5 Thread safety

Eine wichtige Anforderung an die *OT/J-Laufzeitumgebung* wurde bisher völlig außer Acht gelassen, nämlich, dass diese Implementierung mit mehreren Threads noch fehlerfrei arbeitet, also *thread safe* ist. Im Folgenden soll dargestellt werden, was nötig ist, um ein Programm *thread safe* zu implementieren. Allerdings wird dabei nicht auf alle Details eingegangen, da dies den Rahmen dieser Arbeit sprengen würde.

6.5.1 Grundlagen

Um zu überprüfen, ob ein Programm *thread safe* ist, muss zunächst festgestellt werden, ob es Stellen im Code gibt, an denen sich mehrere Threads gegenseitig behindern können. Solche Stellen werden auch *kritische Abschnitte* genannt. Folgendes Beispiel soll dies demonstrieren:

Wir nehmen an, dass es eine Klasse `Stack` gibt, die eine sehr simple Implementierung eines Stacks anbietet, dessen Kapazität auf 10 Elemente begrenzt ist:

```
1.  public class Stack {
2.      private List<Object> stack = new ArrayList<Object>();
3.      public void add(Object element) {
4.          while (stack.size() >= 10) {
5.              // wait until a place for the element is
6.              // available
7.          }
8.
9.          stack.add(element);
10.       }
11.
12.     public Object remove() {
13.         while (stack.isEmpty()) {
14.             //wait until a element is available
15.         }
16.
17.         return stack.remove(stack.size() - 1);
18.     }
19. }
```

Nun sollen zwei Threads existieren, die mit der gleichen Instanz dieses Stacks arbeiten und wie folgt implementiert sind:

```
1. ...
2. Object object = ...
3. Stack stack = ...
4. stack.add(object);
5. ...
```

Der Stack wurde dabei noch nicht benutzt ist also leer. Wird nun nur der erste Thread ausgeführt, wird dem Stack zunächst ein Element hinzugefügt. Wenn dann der zweite Thread ausgeführt wird, fügt er ein weiteres Element hinzu.
Wenn die Reihenfolge der Threads nun aber nicht mehr sichergestellt ist, diese Threads also auch parallel laufen können, kann folgende Situation eintreten:
Beide Threads rufen gleichzeitig die Methode `add` auf. Dies kann dazu führen, dass nur ein Element hinzugefügt wird, obwohl die Methode `add` zweimal aufgerufen wurde. Genauso verhält es sich mit der Methode `remove`. Hier sind also die Methoden `add` und `remove` *kritische Abschnitte*.

Sind die *kritischen Abschnitte* identifiziert, muss sichergestellt werden, dass Threads sich innerhalb dieser *kritischen Abschnitte* nicht gegenseitig stören. Dazu existieren viele Techniken. Unter anderem können diese Abschnitte synchronisiert werden. Zur Synchronisation eines Abschnitts ist ein sogenannter *Monitor* nötig. Ein *Monitor* muss mindestens zwei Methoden anbieten. Die erste (z.B. `wait`) wird von einem Thread aufgerufen, sobald er in einen *kritischen Abschnitt* eintreten will. In dieser Methode wird dann überprüft, ob sich bereits ein Thread in dem Abschnitt befindet. Ist das der Fall, muss dieser Thread warten, bis der andere Thread den kritischen Abschnitt verlassen hat. Verlässt ein Thread den Bereich ruft er die zweite Methode (z.B. `notify`) auf. Diese sorgt dafür, dass ein anderer Thread aufgeweckt wird und den Abschnitt betreten kann.
Das heißt, es können sich auch zwei verschiedene Threads gleichzeitig in einem kritischen Abschnitt befinden, wenn sie sich bei verschiedenen *Monitoren* anmelden.
In Java kann jedes Objekt ein *Monitor* sein, da die Klasse `Object` schon die nötigen Methoden `notify` und `wait` bereitstellt. Diese beiden Methoden müssen allerdings nicht manuell am Anfang bzw. Ende eines *kritischen Abschnitts* aufgerufen werden. Es genügt den Abschnitt mit dem Schlüsselwort `synchronized` zu kennzeichnen. Der Rest wird von der JVM erledigt.
Dieses Schlüsselwort kann sowohl vor einen Block, als auch bei einer Methodendeklaration geschrieben werden:

```
1. public void m() {
2.     synchronized (this) {
3.         // do something
4.     }
5. }
6.
7. public synchronized void m() {
8.     // do something
9. }
```

Im ersten Fall ist immer die explizite Angabe des Monitors nötig. Steht das Schlüsselwort dagegen in einer Methodendeklaration, wird als Monitor bei statischen Methode die zu dieser Klasse zugehörige Class Instanz bzw. bei nicht statischen Methoden die Instanz der Klasse selbst benutzt. Das heißt, die beiden obigen Methoden Definitionen hätten die gleiche Semantik, da in beiden Fällen `this` als Monitor benutzt wird.

Um unser Beispiel des Stacks *thread safe* zu machen, wäre es also nötig die Methoden `add` und `remove` zu synchronisieren:

```
1.  public synchronized void add(Object element) {
2.      while (stack.size() >= 10) {
3.      }
4.
5.      stack.add(element);
6.  }
7.  public synchronized Object remove() {
8.      while (stack.isEmpty()) {
9.      }
10.
11.     return stack.remove(stack.size() - 1);
12. }
```

Nun könnte immer nur ein Thread auf einmal Elemente hinzufügen bzw. entfernen. Das eigentliche Problem ist damit zwar gelöst, allerdings ist ein neues hinzugekommen.
Wir stellen uns wieder zwei Threads vor, die auf dieselbe Instanz des Stacks zugreifen. Einer fügt Elemente hinzu der andere entfernt diese wieder:

Thread 1:

```
1. ...
2. Object object = ...
3. Stack stack = ...
4. stack.add(object);
5. ...
```

Thread 2:

```
1. ...
2. Stack stack = ...
3. Object object = stack.remove();
4. ...
```

In der folgenden Situation führt das zu einem Problem:
Wenn nun Thread 1 ein Element hinzufügen will, der Stack aber schon voll ist, so muss er innerhalb der `while` Schleife warten, bis ein Element entfernt wurde. Thread 2 kann jetzt aber keine Elemente entfernen, da er beim Versuch in die Methode `remove` einzutreten vom Monitor geblockt wird. Solche Situationen, in denen sich zwei Threads gegenseitig blockieren, werden *Dead locks* genannt.
In diesem Beispiel könnte man, um zu verhindern, dass ein *Dead lock* entstehen kann, das Programm so ändern, dass nur die Zugriffe auf die Liste nicht aber die `while` Schleifen synchronisiert sind:

```
1.  public void add(Object element) {
2.      while (stack.size() >= 10) {
3.      }
4.      synchronized (this) {
5.          stack.add(element);
6.      }
7.  }
8.  public Object remove() {
9.      while (stack.isEmpty()) {
10.     }
11.
12.     synchronized (this) {
13.         return stack.remove(stack.size() - 1);
14.     }
15. }
```

Dann würde Thread 1 zwar immer noch in der `while` Schleife warten, bis ein Element entfernt wird, allerdings hätte Thread 2 jetzt die Möglichkeit eines zu entfernen, da sich Thread 1 nicht in einem kritischen Abschnitt befindet.
Es ist also wichtig die synchronisierten Abschnitte nicht zu groß zu wählen, da dadurch *Dead locks* entstehen können und außerdem die Performance des Programms leiden kann.

Im nächsten Kapitel wird gezeigt, wo die kritischen Abschnitte in der neuen OT/J-Laufzeitumgebung sind und wie sie synchronisiert werden können.

6.5.2 Kritische Abschnitte und Synchronisation in der OT/J-Laufzeitumgebung

Grundsätzlich existieren zwei Einstiegspunkte in die Laufzeitumgebung:

1. Die Aktivierung bzw. Deaktivierung eines Teams
2. Das Laden einer Klasse.

Für beide Einstiegspunkte muss überprüft werden, ob sie kritische Abschnitte darstellen oder zu solchen führen.

Laden von Klassen

Bei der Suche nach kritischen Abschnitten macht uns die JVM für den zweiten Einstiegspunkt das Leben dadurch leichter, dass sichergestellt ist, das dieselbe Klasse nicht zweimal (und damit auch nicht zweimal zur gleichen Zeit) geladen wird. Beim Laden der Klassen muss also nur beachtet werden, dass sich nicht zwei Threads gegenseitig stören können, die gerade zwei unterschiedliche Klassen laden.
Die einzigen Aktionen bei denen das passieren kann, ist das Holen der Instanz des `ClassRepositories` und der Instanzen der Klasse `AbstractBoundClass` aus dem `ClassRepository`.
Da sichergestellt werden muss, dass im gesamten Programm nur eine Instanz des `ClassRepositories` existiert muss die Methode `getInstance` dieser Klasse synchronisiert werden.
Beim Holen der Klassen aus dem `ClassRepository` sind zwei Sachen zu beachten: Zum einen wird auf die beiden Maps `classMap` und `anonymousSubclassMap` zugegriffen. Dabei müssen die Schreibenden Zugriffe synchronisiert werden. Zum anderen muss auf die Superklasse der Klasse, die gerade geladen wird, zugegriffen werden, da die aktuelle Klasse ihrer Superklasse als Subklasse hinzugefügt werden muss. Nun könnte es passieren, dass zum selben Zeitpunkt zwei Subklassen einer Superklasse geladen werden. In diesem Fall könnte nicht mehr sichergestellt werden, ob das Hinzufügen der Subklassen fehlerfrei funktioniert.
Da also nahezu die gesamte Methode `getClass` aus *kritischen Abschnitten* besteht, ist es sinnvoll die gesamte Methode mit der `this` Referenz als Monitor zu synchronisieren.

Damit sind für das Laden der Klassen alle kritischen Abschnitte beseitigt. Alle weitere Aktionen, die beim Laden ausgeführt werden beziehen sich immer nur auf eine Instanz von `AbstractBoundClass` und müssen damit nicht synchronisiert werden.

Aktivierung/Deaktivierung von Teams

Beim Aktivieren bzw. Deaktivieren der Teams muss deutlich mehr beachtet werden, da hier nicht mehr sichergestellt ist, dass zwei Threads nicht auf der gleichen Instanz der `AbstractBoundClass` arbeiten. Außerdem können natürlich auch Wechselwirkungen mit dem Laden von Klassen auftreten. So könnte z.B. zum gleichen Zeitpunkt ein Team aktiviert werden und eine Klasse, zu der es Bindungen hat geladen werden.

Zuerst betrachten wir die Klasse `TeamManager`, da hier die Aktivierung/Deaktivierung von Teams vollzogen wird.
Zunächst werden in der Methode `registerTeam` Instanzen der Klasse

`AbstractBoundClass` bzw. `AbstractTeam` für das Team und alle Klassen, die durch das Team gebunden sind, aus dem `ClassRepository` geholt. Auch hier muss im `ClassRepository` wieder beachtet werden, dass nicht für eine Klasse zwei Instanzen von `AbstractBoundClass` erzeugt werden. Deshalb müssen auch die Methoden `getClass(String className)` und `getTeam(String teamName)` synchronisiert sein.

Als nächstes werden aus den `AbstractBoundClasses` die gebundenen Methoden bzw. Felder geholt.
Hierbei ist zu beachten, dass der *Bytecode* der Klasse beim ersten Zugriff auf ihre Felder, Methoden oder Superklasse geparsed wird. Das Parsen ist lazy implementiert, da sonst auch *Bytecode* von ungebundenen Klassen geparsed werden würde, über die gar keine Informationen benötigt werden.
Die Methode `parseBytecode` muss auch synchronisiert werden, da sonst unter Umständen der Klasse Methoden und Felder mehrfach hinzugefügt werden könnten.
Allerdings ist es dennoch nötig auch die Methoden `getField` und `getMethod` zu synchronisieren.
Wenn eine Klasse noch nicht geladen wurde, werden in diesen Methoden zunächst die angefragten Felder und Methoden generiert. Wird die Klasse dann geladen werden diesen Feldern und Methoden die fehlenden Informationen aus dem *Bytecode* hinzugefügt. Das heißt, auch ohne die Methode `parseBytecode` aufzurufen könnten Felder und Methoden mehrfach generiert werden.

Für jede Basismethode, die durch eine *Callin Bindung* dieses Teams gebunden ist, wird dann die *Joinpoint id* geholt. Diese muss wie schon erwähnt für Klasse und Methode eindeutig sein. Auch hierfür muss wieder auf eine Map zugegriffen werden, in der die *Joinpoint ids* gespeichert sind. Dieser Zugriff passiert in der Methode `getJoinpointId` der Klasse `TeamManager`. Diese Methode muss ebenfalls synchronisiert werden.
Auch das Eintragen der Ids und Teams in die entsprechenden Datenstrukturen (s. 6.2) muss synchronisiert werden. Allerdings genügt es hierbei die Aktionen mit den entsprechenden Instanzen der Klasse `Method` bzw. `Field` zu synchronisieren. Es kann also durchaus gleichzeitig auf die Datenstrukturen zugegriffen werden, solange es sich nicht um dasselbe Member derselben Klasse handelt. Durch die Implementierung der `AbstractBoundClass` ist sichergestellt, dass die Methoden `getMethod` und `getField` für denselben Namen und dieselbe Signatur auch immer dieselbe Instanz liefern, sodass diese gefahrlos als Monitor benutzt werden kann.

Nun werden den Instanzen von `AbstractBoundClass` die entsprechenden `WeavingTasks` hinzugefügt.
Dazu wird die Methode `handleAddingOfBinding` der Klasse `AbstractBoundClass` aufgerufen, die dann ihrerseits abhängig von dem Typ des `WeavingTasks` `addAccessWeavingTask` oder `addBindingWeavingTask` aufruft. In diesen beiden Methoden muss dafür gesorgt werden, dass beim Zugriff

auf dasselbe Member nicht gleichzeitig `WeavingTasks` hinzugefügt werden. Aus diesem Grund werden diese Methoden mit dem Member als Monitor synchronisiert.
Als letztes muss die Methode `handleTaskList` synchronisiert werden, da nicht mehrere Threads gleichzeitig eine Klasse weben dürfen.

7 OT/J im Kontext des OSGi Komponentenframeworks

Im letzten Kapitel wurde beschrieben, wie die neue *ObjectTeams/Java-Laufzeitumgebung* funktionieren soll. Damit lassen sich alle Features von *ObjectTeams* im Kontext von Standard Java umsetzen. Allerdings bietet *ObjectTeams* mit *OT/Equinox* auch eine Implementierung, die im Kontext des *OSGi Komponentenframeworks* arbeitet.
In diesem Kapitel soll nun beschrieben werden, was *OSGi* ist, wie *OT/Equinox* arbeitet und welche besonderen Anforderungen dieser Kontext an die Implementierung der Laufzeitumgebung stellt.

7.1 OSGi

Unter dem Begriff *OSGi* kann man zwei Dinge verstehen. Zum einen steht *OSGi* für die *OSGi Alliance*, ein Konsortium von verschiedenen Firmen (z.B. Telekom, Sun, SAP u.s.w.). Außerdem versteht man darunter das *OSGi Framework*, dessen Spezifikation von der *OSGi Alliance* geschaffen wurde. Im Folgenden werden die Bezeichnungen *OSGi* und *OSGi Framework* synonym verwendet.

OSGi stellt ein *Framework* dar, das die Entwicklung von lose gekoppelten dynamischen Komponenten ermöglicht. Die Komponenten werden in *OSGi Bundles* genannt. Ein *Bundle* ist dabei (im Normalfall) ein Jar Archiv[1], das über bestimmte Eigenschaften verfügt. Es wird später noch genauer darauf eingegangen, welche das sind.
OSGi besteht aus verschiedenen Schichten (*Layer*), die u.a. die Verwaltung der *Bundles* übernehmen.
Diese sind in Abbildung 7.1 dargestellt.

[1] Das Jar Archiv kann dabei auch entpackt als Ordner vorliegen.

7.1.1 Layer von OSGi

Das Betriebssystem und die JVM (*Execution environment*) bilden die Basis von *OSGi*. Darauf baut der *Module layer* auf. Dieser ist für die Verwaltung der *Bundles* zuständig. In ihm werden *Bundles* und deren Klassen geladen, Abhängigkeiten zwischen *Bundles* aufgelöst und diese dem *Life cycle layer* zur Verfügung gestellt.

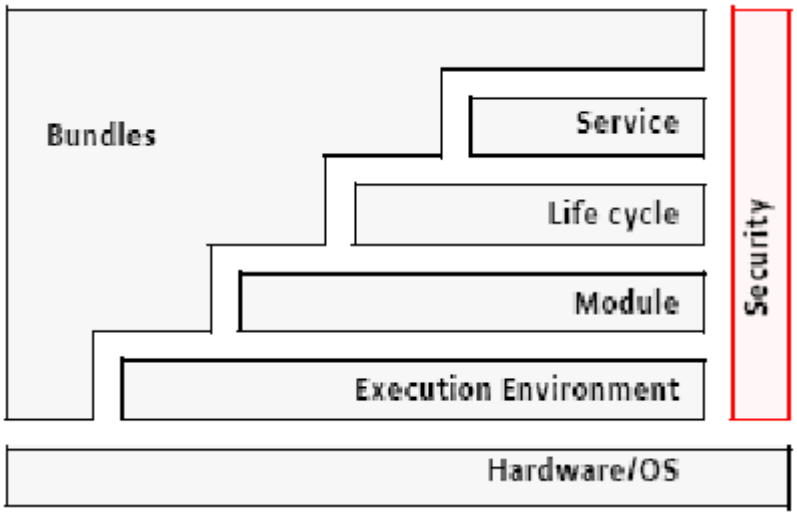

Abbildung 7.1: Layer von OSGi Quelle: [Osg07]

Der *Life cycle layer* übernimmt die Verwaltung des Lebenszyklus der *Bundles*. Ein *Bundle* kann dabei verschiedene Phase durchlaufen, die in Abbildung 7.2 dargestellt sind:

1. *Installed*: Das Jar Archiv oder der Ordner steht zur Verfügung.
2. *Resolved*: Alle Abhängigkeiten wurden aufgelöst und die Klassen des *Bundles* stehen zum Laden bereit.
3. *Starting*: Das *Bundle* wird gestartet.
4. *Active*: Das *Bundle* ist aktiv.
5. *Stopping*: Das *Bundle* wird gestoppt.
6. *Uninstalled*: Das *Bundle* ist jetzt wieder freigegeben und steht nicht mehr zur Verfügung

Der *Service layer* ermöglicht die Verwaltung von *Services*. Ein *Service* kann jede beliebige Java-Klasse sein. Diese kann über ein Interface, das sie implementiert, anderen *Bundles* zur Verfügung gestellt werden. Dazu wird die *Service registry* verwendet. Diese stellt Methoden bereit, um Services anzubieten und abzufragen.
Durch diese *Service registry* benötigt der Klient, der einen Service benutzen will keine Kenntnis über seine Implementierung.
Sowohl der Service als auch das Interface, das er implementiert, sind dabei sogenannte *POJOs* (*Plain Old Java Objects*) bzw. *POJIs* (*Plain Old Java Interfaces*). Mit diesen Begriffen wird ausgedrückt , dass diese Klassen bzw.

Interfaces keine bestimmte Funktionalität bereitstellen müssen (z.B. ein bestimmtes Interface implementieren), um als Service verwendet werden zu können.
Das hat den Vorteil, dass fachliche Implementierungen unabhängig von der Technologie bleiben, die sie benutzen.

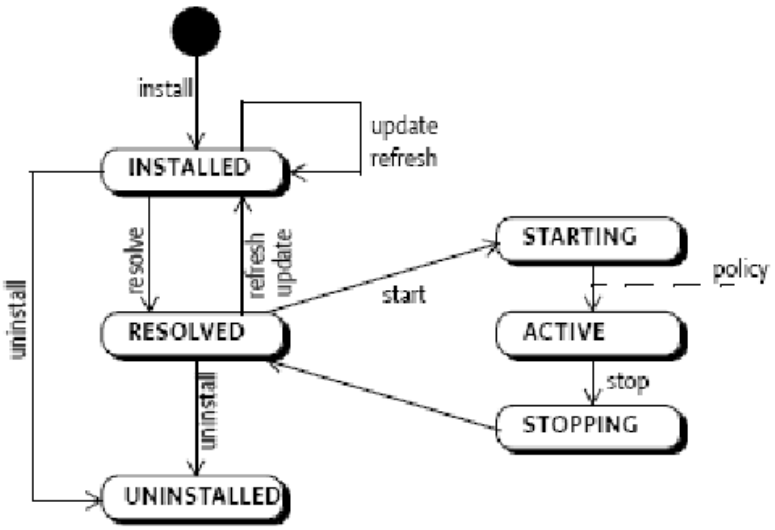

Abbildung 7.2: Lebenszyklus von Bundles Quelle: [Osg07]

7.1.2 Eigenschaften von Bundles

Wie schon erwähnt, sind *Bundles* letztendlich normale Jar-Archive. Sie besitzen allerdings zusätzliche Eigenschaften, die in der Manifest-Datei des Archivs definiert werden. An dieser Stelle sollen die für die Implementierung von *OT/Equinox* relevanten Eigenschaften vorgestellt werden. Eine vollständige Auflistung findet sich in [Osg07].

Allgemeine Informationen von Bundles

Jedes *Bundle* hat einen Namen, der über das Attribut `Bundle-Name` definiert wird. Dieser Name ist nicht als Identifier zu verstehen, sondern stellt eher eine von Menschen gut verständliche Beschreibung des *Bundles* dar. Dieser Name muss nicht eindeutig sein.
Eindeutig identifiziert wird ein *Bundle* über seinen symbolischen Namen und seine Version. Der symbolische Name wird über das Attribut `Bundle-SymbolicName` gesetzt. Grundsätzlich kann er beliebig gewählt werden. Allerdings hat sich als Konvention die aus Java bekannte Package Notation (also z.B. `org.foo.bar`) durchgesetzt. Wenn im folgenden über den Namen eines `Bundles` gesprochen

wird, so ist immer der symbolische Name gemeint.
Die Version eines `Bundles` kann über das Attribut `Bundle-Version` gesetzt werden und folgt dabei folgender Syntax:

```
version ::=
major( '.' minor ( '.' micro ( '.' qualifier )? )? )?
major ::= number // See 1.3.2
minor ::= number
micro ::= number
qualifier ::= ( alphanum | '_' | '-' )+
```
Quelle: [Osg07]

Sie besteht also maximal aus drei Zahlen (*Major version*, *Minor version* und *Micro version*) und einem (nahezu) beliebigen String, die alle durch einen Punkt verbunden sind (also z.B. "`1.0.0.foo`").
Minor version, *Micro version* und der String sind dabei optional (also auch "`1`").

Definition von Abhängigkeiten

Abhängigkeiten zwischen *Bundles* können in *OSGi* auf drei Arten bestehen, die sich im Grad der Kopplung unterscheiden:

1. Definierte Abhängigkeit zu einem anderen *Bundle*
   In der Manifest-Datei kann über das Attribut `Require-Bundle` definiert werden, dass das *Bundle* ein anderes *Bundle* voraussetzt. Hierbei muss der symbolische Name des benötigten *Bundles* angegeben werden. Optional kann auch definiert werden, in welcher Version oder in welchem Versionsbereich das *Bundle* benötigt wird. Hierbei sind die *Bundles* also sehr eng gekoppelt.
2. Definierte Abhängigkeit zu einem Package
   Hier wird über das Attribut `Import-Package` definiert, dass ein bestimmtes Package benötigt wird. Dabei wird nicht festgelegt, aus welchem *Bundle* dieses Package kommen soll. Das bestimmt das Framework beim Auflösen der Abhängigkeiten. Dazu muss mindestens ein *Bundle* existieren, das ein solches Package besitzt und es auch über das Attribut `Export-Package` anderen *Bundles* zur Verfügung stellt. Hier sind die *Bundles* also schon deutlich loser gekoppelt, da der User nicht mehr weiß, zu welchem *Bundle* nun tatsächlich eine Abhängigkeit besteht.
3. Zugriff auf einen Service
   Die loseste Form der Kopplung stellt der Zugriff auf einen Service dar. Hierbei muss das *Bundle* natürlich eine Abhängigkeit zu dem Interface haben, das diesen Service beschreibt. Allerdings muss in keiner Form eine Abhängigkeit zu einer konkreten Implementierung definiert werden. Ein weiterer Unterschied zu den ersten beiden Arten besteht darin, dass die Auflösung dieser Abhängigkeit nicht notwendig ist, um das *Bundle* zu starten.

Wenn in den ersten beiden Fällen die definierte Abhängigkeit nicht aufgelöst werden kann, kann das *Bundle* auch nicht in den Status *resolved* kommen. Im dritten Fall dagegen schon. Hier kann sich also ein *Bundle*, das einen Service benutzen will, nicht darauf verlassen, dass zur Laufzeit eine Implementierung dieses Services zur Verfügung steht.

Bei der Definition der Abhängigkeiten sind auch noch einige Unterformen möglich (O*ptional packages*, *Dynamic imports*), die hier allerdings nicht beschrieben werden sollen, sondern in [Osg07] nachzulesen sind.

Fragment und Extension bundles

OSGi bietet nicht nur die Möglichkeit hoch dynamische Systeme zu entwerfen, bei denen auch zur Laufzeit Komponenten (also *Bundles*) ausgetauscht werden können, sondern erlaubt es Entwicklern auch, das Framework selbst zu verändern.
Dies geschieht über sogenannte *Framework extension bundles*.
Ein weitere Notwendigkeit für *Extension bundles* besteht darin, dass es für bestimmte *Bundles* notwendig sein kann, im *Boot class path*, also vom *Boostrap class loader* (s. 2.2.1), geladen zu werden. Diese Art von *Extension bundles* wird *Boot class path extension bundle* genannt.

Extension bundles sind ein Spezialfall von *Fragment bundles*. Ein *Fragment bundle* stellt eine Erweiterung eines anderen *Bundles* (des sog. *Host bundles*) dar. Es kann ohne Deklarationen auf alle Klassen seines *Host bundles* zugreifen. Das *Host bundle* wird über das Manifest-Attribut `Fragment-Host` definiert.

Für ein *Extension bundle* muss das *Host bundle* das *System bundle*, also das *Bundle*, in dem das Framework selbst implementiert ist, sein.

7.1.3 Verwendung von Class loadern in OSGi

Eine sehr mächtige, wenn auch technisch nicht trivial umzusetzende, Eigenschaft von *OSGi* besteht darin, dass dieses Framework es erlaubt, mehrere Klassen mit gleichem voll qualifizierten Namen (*Fully qualified name*, *FQN*) parallel zu verwenden. Die *FQNs* der Klassen müssen lediglich innerhalb eines *Bundles* eindeutig sein.
Dieses Feature ermöglicht es u.a. mit *Bundles* in unterschiedlichen Versionen parallel, also innerhalb einer Framework Instanz, zu arbeiten.
Auch *Bundles* mit unterschiedlichem Namen können Klassen mit gleichem *FQN* beinhalten.
Um eine Klasse im *OSGi* Kontext eindeutig zu identifizieren ist also die Erweiterung des *FQN* um den Namen des *Bundles*, in dem diese Klasse deklariert ist, und seiner Version nötig.
Im Standard Java Kontext ist es normalerweise nicht möglich Klassen mit gleichem *FQN* zu verwenden, mit einer Ausnahme, die sich *OSGi* zunutze macht:

Voll qualifizierte Klassennamen sind in Java nur innerhalb eines *Class loaders* eindeutig. Wird also für jedes *Bundle* ein eigener *Class loader* verwendet, können in den *Bundles* auch Klassen mit gleichem *FQN* enthalten sein.
Auf diese Besonderheit von *OSGi* ist die *OT/J-Laufzeitumgebung* aber bis jetzt nicht eingerichtet, da hier Klassen nur über ihren FQN identifiziert werden. Wie dieses Problem gelöst werden kann wird in 7.4.2 gezeigt.

Weiterhin wird diese *Class loader* Konstruktion für das *Dependency management* eingesetzt. Der *Class loader* eines *Bundles* kennt nur die Klassen dieses *Bundles* und kann auch nur diese selbst laden. Wird allerdings eine Abhängigkeit zu einem anderen *Bundle* definiert, dann (und nur dann) kann der *Class loader* dieses *Bundles* die Anfrage, eine Klasse zu laden, an den *Class loader* des anderen *Bundles* delegieren. So wird sichergestellt, dass ein *Bundle* keine Klassen aus anderen *Bundles* laden kann, zu denen nicht in irgendeiner Form eine Abhängigkeit definiert wurde.
Der *Parent class loader* aller *Bundle class loader* ist standardmäßig der *Bootstrap class loader*. Dieses lässt sich allerdings durch die *System property* `osgi.parentClassloader` auch ändern, was, wie wir in 7.4.3 sehen werden, noch von Bedeutung sein wird.

7.2 Equinox

Equinox ist eine Open source Implementierung des *OSGi Komponentenframeworks* von Sun. Unter anderem bildet *Equinox* seit Version 3.0 die Basis von *Eclipse*.
Als Implementierung von *OSGi* bietet *Equinox* alle Features, die in der Spezifikation von *OSGi* beschrieben sind. Darüber hinaus besitzt es allerdings noch weitere Funktionalität, die in diesem Kapitel beschrieben werden soll. Vorher sei noch darauf hingewiesen, dass *Bundles* in *Equinox Plug-ins* genannt werden. Im folgenden werden diese Begriffe synonym verwendet.

7.2.1 Hooks

Über sogenannte *Hooks* greift *Equinox* das Konzept der *Framework extension bundles* auf und erweitert es so, dass es komfortabel benutzt werden kann, um die Funktionalität des *Frameworks* zu erweitern.
Hooks sind dabei Erweiterungen des `FrameworkAdaptors`.
Der `FrameworkAdaptors` übernimmt die Anbindung an die zugrunde liegende Laufzeitumgebung, also die *JVM*. Über ihn wird das Framework gestartet und gestoppt, *Bundles* installiert und deinstalliert, Klassen geladen u.s.w.
Dabei bietet er diverse vordefinierte Stellen, an denen das Framework über die *Hooks* erweitert werden kann. Technisch sind *Hooks* Implementierungen von bestimmten Interfaces, die vom `FrameworkAdaptors` bereitgestellt werden.

Über die Datei `config.ini` kann definiert werden, welche *Hooks* der `FrameworkAdaptors` an welchen Stellen benutzen soll. Diese Datei existiert nur einmal global für das gesamte Framework. Der `FrameworkAdaptors` kann dann über die `HookRegistry`, in der alle *Hooks* verwaltet werden, abfragen zu welchen Stellen welche *Hooks* existieren.

Es können dabei folgende Arten von *Hooks* unterschieden werden:

1. `AdaptorHook`: Über diesen *Hook* kann das Starten und Stoppen des Frameworks erweitert werden.

2. `BundleFileFactoryHook`: Standardmäßig liegen *Bundles* entweder als Ordner oder als Jar-Archiv vor. Über diesen *Hook* ist es möglich *Bundles* auch aus anderen Formaten zu lesen.

3. `BundleFileWrapperFactoryHook`: Implementierungen dieses *Hooks* fungieren als Wrapper für die *Bundles*. Sie können den Zugriff auf *Bundles* beeinflussen.

4. `BundleWatcher`: Dieser *Hook* ermöglicht es auf Statusübergänge von *Bundles* zu reagieren.

5. `ClassLoadingHook`: Dieser *Hook* wird beim Laden einer Klasse aufgerufen. Über ihn kann z.B. der *Bytecode* einer Klasse verändert werden. Dieser *Hook* spielt, wie wir in 7.3.2 sehen werden, eine wichtige Rolle in *OT/Equinox*.

6. `StorageHook`: Implementierungen dieses *Hooks* können das Speichern und Laden von *Bundles* beeinflussen.

7. `ClassLoadingStatsHook`: Dieser *Hook* bietet die Möglichkeit, Statistiken über das Laden von Klassen zu erstellen.

7.2.2 Extensions und Extension points

Extensions basieren auf den Services von *OSGi*. Sie sind insofern mit den eben beschriebenen *Hooks* zu vergleichen, dass sie Erweiterungen einer bestehenden Implementierung ermöglichen. Im Gegensatz zu *Hooks* bieten *Extensions* aber nicht nur die Möglichkeit, an fest definierten Stellen das Framework zu erweitern, sondern jedes *Bundle* kann sogenannte *Extension points*, also Stellen, an denen es erweitert werden kann, definieren und auch *Extensions* zu *Extensions points* anderer *Bundles* bereitstellen.
Die *Extension points* werden in der Datei `plugin.xml` eines *Bundles* definiert. Diese Datei hat folgende Struktur: Das Root tag ist immer `<plugin>`. Darin können über das Tag `<extension-point>` beliebig viele *Extension points* definiert werden. Ein *Extension point* hat dabei immer eine Id, die ihn eindeutig

definiert, einen für Menschen gut verständlichen Namen und einen Verweis auf ein XML Schema. Dieses Schema legt fest, welche Informationen eine *Extension* für diesen *Extension point* anbieten muss und in welcher XML Struktur sie beschrieben werden.

Für diese *Extension points* können dann in anderen *Bundles Extensions* definiert werden. Dies geschieht ebenfalls in der Datei `plugin.xml` über das Tag `<extension>`. Als Attribut dieses Tags muss die Id des *Extension points*, den diese *Extension* erweitert, angegeben werden. Die weitere Struktur innerhalb dieses Tags wird vom XML Schema definiert, das im *Extension point* festgelegt wurde.
Wie genau diese *Extensions* dann in dem *Bundle*, das den *Extension point* definiert, verwendet werden, bestimmt das *Bundle* selbst. Es wäre also auch möglich, dass diese *Extensions* überhaupt keine Relevanz haben.

Über die `ExtensionRegistry` können zu einem *Extension point* alle *Extensions* abgefragt werden.

Nachdem nun die grundlegenden Eigenschaften von *OSGi* und *Equinox* beschrieben wurden, soll dargelegt werden, wie die *OT/J-Laufzeitumgebung* über *OT/Equinox* in *OSGi* bzw. *Equinox* integriert wird.

7.3 OT/Equinox

Die Laufzeitumgebung für *OT/Equinox* besteht aus drei *Plug-ins*:

1. `org.objectteams.otdt.pde.core.lib`
2. `org.objectteams.eclipse.transformer.hook`
3. `org.objectteams.otequinox`

Deren Funktionsweise soll im Folgenden erklärt werden.

Das *Plug-in* `org.objectteams.otequinox` stellt den Einstiegspunkt für *OT/Equinox* dar. Es übernimmt die Aufgabe die Teams zu laden. Dabei werden zwei Bedingungen überprüft, die durch die *Extension points* `aspectBindings` und `aspectBindingNegotiators` bzw. deren *Extensions* ausgedrückt werden.

7.3.1 Extension points von OT/Equinox

Jedes *Plug-in*, das Teams enthält, die Klassen von anderen *Plug-ins* adaptieren, muss eine *Extension* zu dem *Extension point* `aspectBindings` definieren. Eine gültige *Extension* zu diesem *Extension point* könnte z.B. so aussehen:

```
1.  <extension
2.          point="org.objectteams.otequinox.aspectBindings">
3.      <aspectBinding>
4.          <basePlugin id="basebundle">
5.              <forcedExports>
6.                  basebundle.foo,basebundle.bar
7.              </forcedExports>
8.              <requiredFragment id="basebundle.fragment"/>
9.          </basePlugin>
10.         <team class="foo.bar.T1" superclass="foo.bar.T0"
11.                 activation="ALL_THREADS">
12.         </team>
13.         <team class="bar.foo.T1" superclass="bar.foo.T0"
14.                 activation="NONE">
15.         </team>
16.     </aspectBinding>
17. </extension>
```

Das Tag `<aspectBinding>` beschreibt dabei jeweils eine 1:n Beziehung zwischen dem *Base bundle*, also dem *Bundle*, das adaptiert wird, und Teams, von denen es adaptiert wird. Zunächst wird im Tag `<basePlugin>` durch das Attribut `id` das *Bundle* definiert, das adaptiert wird. Dieses Attribut muss den symbolischen Namen des *Base bundles* enthalten. In dem Fall, dass das *Base bundle* ein *Fragment bundle* ist, muss hier der Name des *Host bundles* angegeben werden. Das adaptierte *Fragment bundle* wird dann über das Tag `<requiredFragment>` festgelegt.
Das Tag `<forcedExports>` führt das Konzept *Decapsulation* (s. 3.7.2) konsequent in *OSGi* weiter. Hierüber können Packages des *Base bundles* festgelegt werden, die vom *Aspect bundle* auch dann benutzt werden können, wenn das *Base bundle* sie nicht exportiert.

Ist das *Base bundle* definiert, können beliebig viele Teams angegeben werden, die dieses *Base bundle* adaptieren. Dies geschieht über das Tag `<team>`. In ihm wird über das Attribut `class` der voll qualifizierte Name des Teams definiert. Zusätzlich muss über das Attribut `superclass` die Superklasse des Teams angegeben werden, wenn diese ungleich `Team` ist. Weiterhin kann mit dem Attribut `activation` festgelegt werden, ob das Team beim Start von *Equinox* für alle Threads (`ALL_THREADS`) , nur für einen Thread (`THREAD`) oder gar nicht (`NONE`) aktiviert werden soll.
Werden von diesem *Aspect bundle* noch weitere *Base bundles* adaptiert, muss das Tag `<aspectBinding>` wiederholt werden.
Dieser *Extension point* hat zwei Aufgaben. Zum einen macht er die Architektur der Aspekte ausgehend vom *Base bundle* explizit sichtbar. Dadurch kann man sich schnell einen Überblick verschaffen, welche *Plug-ins* von einem *Aspect Bundle* adaptiert werden. Dazu muss jede Bindung zu einem *Base bundle* über eine Extension in dem *Aspect bundle* definiert werden. Versucht ein Team eine Klasse aus einem *Plug-in* zu adaptieren und ist diese Bindung nicht in der Extension definiert, so führt dies zu einer Fehlermeldung der Laufzeitumgebung.

Zum anderen ermöglicht er die technische Umsetzung von *OT/Equinox*. Durch ihn kann sichergestellt werden, das Teams vor ihren Basisklassen geladen werden(s. 4.3.3).

Der *Extension point* `aspectBindingNegotiators` schafft die Möglichkeit, Policies über die Bindungen zu definieren. Dazu kann in *Extensions* dieses *Extension points* eine Klasse angegeben werden, die für eine Bindung überprüft, ob sie zulässig ist oder nicht. Beim Laden eines Teams werden dann alle diese Klassen aufgerufen, um zu überprüfen, ob diese Bindung erlaubt ist oder nicht. Ist dies nicht der Fall führt dies wiederum zu einer Fehlermeldung in der Laufzeitumgebung.

Extensions zu diesem *Extension point* werden an einer zentralen Stelle im *Plug-in* `org.objectteams.otequinox`, nämlich in der Klasse `TransformerPlugin`, ausgewertet. Wie das im Detail geschieht soll im nächsten Abschnitt beschrieben werden.

Auswerten der Extensions

In der Klasse `TransformerPlugin` werden alle *Extensions* zu den *Extension points* `aspectBindings` und `aspectBindingNegtiator` ermittelt und ausgewertet.
Dazu kann, wie in 7.2.2 beschrieben, die `ExtensionRegistry` verwendet werden.
Diese definiert die Methode

```
public IConfigurationElement[]
        getConfigurationElementsFor(
        String namespace, String extensionPointName)
```

Diese Methode liefert für jede Extension ein `IConfigurationElement` zurück. Aus den `IConfigurationElements` können dann alle Daten der *Extension* ausgelesen werden.
An dieser Stelle soll nicht detailliert auf die Datenstrukturen eingegangen werden, die momentan benutzt werden, um die Bindungsinformationen zu speichern, da diese in der neuen Implementierung von *OT/Equinox* so nicht mehr erhalten bleiben.

Ein Punkt soll hier allerdings noch erläutert werden.
Die Informationen, die aus den *Extensions* ausgelesen werden, werden auch im *Plug-in* `org.objectteams.eclipse.transformer.hook` benötigt (s. nächster Abschnitt). An sich müsste das *Bundle* `org.objectteams.eclipse.transformer.hook` nun also eine Abhängigkeit zu dem *Bundle* `org.objectteams.otequinox` definieren, um auf die Informationen zugreifen zu können. Dies ist allerdings aus folgendem Grund nicht möglich:

Das *Bundle* `org.objectteams.eclipse.transformer.hook` ist ein *Framework extension bundle* (s. 7.1.2). Es ist also ein *Fragment* des *System bundles*[1]. Wie erwähnt, müssen alle *Bundles*, auch `org.objectteams.otequinox`, eine Abhängigkeit zu dem *System bundle* haben. Da *Fragment bundles* und ihre *Host bundles* quasi eine Einheit bilden, würde eine Abhängigkeit von `org.objectteams.eclipse.transformer.hook` zu `org.objectteams.otequinox` allerdings eine zyklische Abhängigkeit bedeuten. Diese sind zwar von *OSGi,* nicht aber von Java erlaubt. Es muss also eine andere Möglichkeit gefunden werden, die Bindungsinformationen zu übergeben.

Diese Funktionalität kann über *Services* ermöglicht werden (s. 7.1.2). Da hierbei der *Service client* keine Abhängigkeit zu dem Server haben muss, entsteht hierbei keine zyklische Abhängigkeit. Das Interface `IAspectRegistry` stellt die Beschreibung dieses Services dar. Hier können alle Methoden deklariert werden, die von `org.objectteams.eclipse.transformer.hook` für den Zugriff auf Bindungsinformationen benötigt werden. Die Implementierung dieses Interfaces ist dabei die Klasse `TransformerPlugin` selbst.

7.3.2 Hooks zur Umsetzung des Aspektwebens

Das *Plug-in* `org.objectteams.eclipse.transformer.hook` implementiert verschiedene *Hooks,* um sicherzustellen, dass die Aspekte greifen. Der wesentliche Teil dieses *Plug-ins* sind die Klassen `TransformerHook`, `OTStorageHook` und `HookConfigurator`.
Die Klasse `OTStorageHook` implementiert das Interface `StorageHook` (s. 7.2.1). In der Methode `getManifest`, die aufgerufen wird, wenn auf die Manifest Datei eines *Plug-ins* zugegriffen wird, wird überprüft, ob dieses *Bundle* ein *Base bundle* ist und ob `forcedExports` für dieses *Bundle* existieren. Ist das der Fall werden sie dem Manifest hinzugefügt.

Im Gegensatz zur Implementierung der *OT/J-Laufzeitumgebung* (s. 4.3) setzt *OT/Equinox* weder *JMangler* noch *JPLIS* ein um den *Bytecode* von Klassen zur Ladezeit manipulieren zu können. Dies wird dagegen durch die Implementierung des *Hooks* `ClassLoadingHook` durch die Klasse `TransformerHook` umgesetzt.
Die Methode `processClass` dieses *Hooks* wird beim Laden einer Klasse aufgerufen, bekommt den ursprünglichen *Bytecode* dieser Klasse übergeben und ermöglicht es, den veränderten *Bytecode* zurückzuliefern.
In der Implementierung dieser Methode in der Klasse `TransformerHook` wird zunächst überprüft, ob zu dieser Klasse Bindungen existieren oder nicht. Ist das der Fall wird das Weben der Aspekte an die *OT/J-Laufzeitumgebung* delegiert. Außerdem wird hier sichergestellt, dass die *Plug-ins*, die *Teams* enthalten, vor ihren *Base plugins* geladen werden (was allerdings nicht in allen Fällen funktioniert). Wie dieser Mechanismus funktioniert soll hier nicht im Detail erklärt werden, da er in der neuen Umgebung von *OT/Equinox* nicht mehr nötig ist.

---

[1] Für *Equinox* ist das *Bundle* `org.eclipse.osgi` das *System bundle*.

Nun muss dem Framework noch mitgeteilt werden, dass diese Klassen als *Hooks* verwendet werden sollen. Das geschieht über das Framework Interface `HookConfigurator`:

```
1. public interface HookConfigurator {
2.     public void addHooks(HookRegistry hookRegistry);
3. }
```

eine Klasse die dieses Interface implementiert wird beim Start des Frameworks aufgerufen und kann beliebige *Hooks* in der `HookRegistry` registrieren.
In *OT/Equinox* wird dieses Interface durch die gleichnamige Klasse `HookConfigurator` implementiert. Diese registriert dann alle nötigen *Hooks*.

Das *Plug-in* `org.objectteams.otdt.pde.core.lib` hat nur einen einzigen Zweck: In *OSGi* bzw. *Equinox* können *Extension bundles* keine Jar-Archive referenzieren. Dies ist aber eigentlich nötig, um auf die *OT/J-Laufzeitumgebung* und die aktuell benutzte *Bytecode library* zuzugreifen. Aus diesem Grund ist es nötig, alle Klassen, die eigentlich in Jar Archiven liegen würden in ein eigenes *Plug-in* auszulagern. Dieses *Plug-in* ist `org.objectteams.otdt.pde.core.lib`.

7.4 Weben von Aspekte zur Laufzeit für OT/Equinox

Die im vorherigen Kapitel beschriebene aktuelle Implementierung von *OT/Equinox* muss nun natürlich an das Konzept des *Aspektwebens zur Laufzeit* und die neue Implementierung der *OT/J-Laufzeitumgebung* angepasst werden.

Dabei soll das Laden der Basisklasse nicht mehr erzwungen werden, da es zu unerwünschten und für den Nutzer nicht durchschaubaren Seiteneffekten führen kann. Man stelle sich z.B. vor, dass das Laden einer Klasse zu einer Exception führt. Wenn die Klasse im Programm gar nicht benutzt wird aber durch ein Team adaptiert ist, wird sie dennoch geladen. Für den Nutzer ist dann nicht verständlich, weswegen diese Exception ausgelöst wird.

Im Folgenden soll nun beschrieben werden, welche Schritte notwendig sind, um *OT/Equinox* an die neue Webestrategie anzupassen.

7.4.1 Aufruf des Aspektwebers zur Ladezeit der Klassen

Wie auch in der bisherigen Implementierung wird das *Aspektweben zur Ladezeit* der Klassen in der Methode `processClass` Klasse `TransformerHook` angestoßen. Dazu wird zunächst eine Instanz von `AbstractBoundClass` für diese Klasse aus dem `ClassRepository` geholt. Anschließend wird die Methode `transform` des `ObjectTeamsTransformers` aufgerufen. Die weiteren Schritte

sind analog zum Vorgehen in der *OT/J-Laufzeitumgebung* (s. 6.1). Auch in der neuen Implementierung wird also kein *Java agent* benötigt, um die Klassen zur Ladezeit zu weben[1].

7.4.2 Existenz von mehreren Klassen mit gleichem FQN

Bis jetzt wurde in der alten und neuen Implementierung der *OT/J-Laufzeitumgebung* davon ausgegangen, dass Klassen über ihren voll qualifizierten Namen im gesamten Programmkontext eindeutig identifiziert werden können. Wie wir gesehen haben ist dies im *OSGi* Kontext nicht der Fall. Das heißt, es ist nötig eine Möglichkeit zu finden, wie Klassen eindeutig identifiziert werden können.

In *OSGi* ist diese Möglichkeit schnell gefunden. Wie in 7.1.3 erwähnt können Klassen hier eindeutig über die Kombination aus dem symbolischen Namen ihres *Bundles*, der Version ihres *Bundles* und ihrem *FQN* identifiziert werden.
Nun muss die *OT/J-Laufzeitumgebung* so angepasst werden, dass sie im Standard-Java-Kontext die *FQN* und im *OSGi* Kontext die eben beschriebene Kombination als eindeutigen Identifier einer Klasse verwendet.

Dazu soll das Interface `IClassIdentifierProvider` dienen, dass wie folgt definiert ist:

```
1.  public interface IClassIdentifierProvider {
2.      public String getBoundClassIdentifier(Class<?> teem,
3.              String boundClassname);
4.
5.      public String getSuperclassIdentifier(String classId,
6.              String superclassName);
7.
8.      public String getClassIdentifier(Class<?> clazz);
9.  }
```

Damit auf eine Instanz dieses Interfaces in der gesamten Laufzeitumgebung zugegriffen werden kann existiert die Klasse `ClassIdentifierProviderFactory`. Diese speichert eine Instanz dieses Interfaces und ermöglicht über die Methode `getClassIdentifierProvider` Zugriff auf diese. Ist die Instanz `null` so wird in dieser Methode ein Objekt des Typs `DefaultClassIdentifierProvider` erzeugt.
Die Klasse `DefaultClassIdentifierProvider` stellt eine Implementierung dieses Interfaces dar, in der davon ausgegangen wird, dass der voll qualifizierte Name einer Klasse auch als ihr Identifier verwendet werden kann. Sie liefert also immer nur den Klassennamen zurück. Diese Implementierung soll im Standard-Java-Kontext verwendet werden.
Im *OSGi* Kontext dagegen darf der `DefaultClassIdentifierProvider` nicht verwendet werden. Dazu besitzt die Factory die Methode `setClassIdentifierProvider`, mit der die Instanz von

[1] Allerdings wird ein *Agent* für das Laufzeitweben benötigt, wie in 7.4.3 gezeigt wird.

`IClassIdentifierProvider` gesetzt werden kann. Welche Implementierung des Interfaces in *OT/Equinox* verwendet wird, wird im nächsten Abschnitt gezeigt.
Diese Implementierung muss nun in der Factory gesetzt werden. Dies muss so früh wie möglich geschehen, damit der `ClassIdentifierProvider` in der gesamten Implementierung eingesetzt werden kann. Dazu eignet sich am besten der `HookConfigurator` (s. 7.3.2). Von allen Klassen in *OT/Equinox*, wird diese Klasse als erste aufgerufen. Dort kann dann der `ClassIdentifierProvider` in der Factory gesetzt werden.

Das Interface `IClassIdentifierProvider` bietet drei Methoden, deren Zweck und Einsatzort im Folgenden erläutert werden.
Die Methode `getClassIdentifier` kann überall dort aufgerufen werden, wo eine Class Instanz existiert und bekannt ist und ein Identifier benötigt wird. In der *OT/J-Laufzeitumgebung* ist dies im `TeamManager` in der Methode `registerTeam` (s. 6.3.3) der Fall, wenn ein Team aktiviert wird. Hier muss für das Team eine Instanz der Klasse `AbstractTeam` aus dem `ClassRepository` geholt werden. Da die Methode `handleTeamStateChange` eine Instanz des Teams übergeben bekommt, kann sie sich darüber die Class Instanz für das Team holen. Mit dieser Class Instanz ruft sie die Methode `getClassIdentifier` des `ClassIdentifierProviders` auf und bekommt den Identifier für das Team zurückgeliefert.
Außerdem müssen in der Methode `registerTeam` auch noch die Basisklassen als `AbstractBoundClasses` aus dem `ClassRepository` geholt werden. Für diese stehen allerdings keine Class Instanzen zur Verfügung, sondern nur ihr Name und die Class-Instanz des Teams. Für diesen Zweck existiert die Methode `getBoundClassIdentifier`, die anhand dieser beiden Informationen den Identifier für die Basisklasse bestimmt.
Der dritte Anwendungsfall für dieses Interface ist das Bestimmen der Superklasse einer Klasse. Dies geschieht in der Methode `getSuperclass` von `AbstractBoundClass`. An dieser Stelle steht nur der Name der Superklasse (dieser wird aus dem *Bytecode* ausgelesen) und der Identifier der Subklasse zur Verfügung. Hierfür bietet der `ClassIdentifierProvider` die Methode `getSuperclassIdentifier` an.

Implementierung von IClassIdentifierProvider in OT/Equinox

In *OT/Equinox* wird das Interface `IClassIdentifierProvider` durch die Klasse `OTEquinoxClassIdentifierProvider` implementiert. Um die nötige Funktionalität bereitstellen zu können, implementiert sie zusätzlich das *Hook* Interface *ClassLoadingStatsHook* (s. 7.2.1). Benötigt wird von diesem *Hook* lediglich die Methode

```
void recordClassDefine(String name, Class clazz,
        byte[] classbytes, ClasspathEntry classpathEntry,
        BundleEntry entry, ClasspathManager manager);
```

Diese Methode wird direkt nach dem Definieren einer Klasse im *Class loader*

aufgerufen. Als einzige *Hook* Methode in *OSGi* bekommt sie eine Class Instanz übergeben. Außerdem bekommt sie eine Instanz der Klasse `ClasspathManager` übergeben. Diese Klasse bietet über die Methode `getBaseData` Zugriff auf die Daten eines *Bundles*. In diesem `BaseData` Objekt sind sowohl der symbolische Name als auch die Version des *Bundles* gespeichert.
Damit haben wir in dieser Methode alle Informationen zur Verfügung, die wir benötigen, um erstens den Identifier für eine Klasse zu bilden und ihn zweitens in Abhängigkeit von der Class Instanz zu speichern. Der Identifier wird in der Map

```
private Map<Class<?>, String> classIdentifiers;
```

gespeichert und kann von dort in der Methode `getClassIdentifier` ausgelesen werden.

Identifier von gebunden Klassen

Das Bereitstellen der Identifier der Basisklassen eines Teams ist dagegen nicht trivial. Auch die dafür nötigen Informationen werden in der Methode `recordClassDefine` gespeichert.
Wird ein Team geladen müssen zunächst die Namen der Basisklassen, die im Team gespeichert sind ausgelesen werden. Dazu holt sich die Methode eine Instanz von `AbstractTeam` zu diesem Team und geht seine Bindungen durch.

Für jede Basisklasse muss nun das *Bundle* ermittelt werden, in dem sie definiert ist. Um das *Bundle* der Basisklasse bestimmen zu können, muss für alle *Bundles*, die von diesem Team adaptiert werden, überprüft werden, ob dieses *Bundle* diese Basisklasse enthält. Wie zu einem *Team* alle *Base bundles* bestimmt werden können, zeigt der nächste Abschnitt.
Für jede Basisklasse muss nun überprüft werden, ob sie in einem dieser *Bundles* enthalten ist. Dies geschieht über die Methode

```
public URL getResource(String name)
```

des *Bundles*. Liefert diese für den Namen der Basisklasse einen Wert ungleich `null` zurück, steht fest, dass dieses *Bundle* die Basisklasse enthält. Mit dem *Base bundle* und dem Namen der Basisklasse, kann nun der Identifier dieser Klasse bestimmt und gespeichert werden.
Dazu existiert die Map

```
Map<Class<?>, Map<String, String>> baseClassIdentifiersByTeam
```

Für die Class Instanz eines Teams und den Namen der Basisklasse, kann über sie der Identifier der Basisklasse abgefragt werden. Somit ist auch für die Implementierung der Methode `getBoundClassIdentifier` gesorgt.

Im nächsten Abschnitt soll nun beschrieben werden, wie die *Base bundles* für ein *Team* ermittelt werden können.

Übergabe von Bindungsinformationen

Wie auch in der bisherigen Implementierung, soll für den Zugriff auf die Bindungsformationen aus dem *Bundle* `org.objectteams.eclipse.transformer.hook` heraus der Service `IAspectRegistry` dienen. In der neuen Implementierung soll dieses Interface dazu die Methode

```
public Collection<Bundle> getBaseBundles(String teamId)
```

anbieten, über die alle *Base bundles* zu der Id eines *Teams* ermittelt werden können. Wie in 7.3.1 beschrieben, müssen dazu zunächst alle *Extensions* zu dem *Extension point* `aspectBindings` ausgewertet werden. Dies geschieht in der Klasse `TransformerPlugin` des *Bundles* `org.objectteams.otequinox`. Da diese Klasse gleichzeitig die Implementierung des Interfaces `IAspectRegistry` darstellt, muss sie die Methode `getBaseBundles` implementieren. Dazu speichert sie in der Map

```
private static Map<String, Set<Bundle>> baseBundlesByTeam
```

zu einem *Team* Identifier einen Set von *Base bundles*.
Diese Map wird beim Auswerten der *Extensions* gefüllt. Da durch die *Extension* lediglich der symbolische Name des *Base bundles* und nicht das *Bundle* selbst bekannt ist, muss dieses zunächst ermittelt werden. `Equinox` bietet dazu den Service `PackageAdmin` an. Dieser deklariert die Methode

```
public Bundle[] getBundles(String symbolicName,
        String versionRange);
```

Mit dieser Methode können für einen symbolischen Namen alle *Bundles* ermittelt werden. Falls dieses *Bundle* in mehreren Versionen vorliegt, liefert `getBundles` alle *Bundles* in den verschiedenen Versionen zurück. Dabei ist das *Bundle* mit der höchsten Version immer an erster Stelle im Array.

An dieser Stelle gehen wir davon aus, dass das *Bundle* mit der höchsten Version vom Team adaptiert wird[1]. Nun können wir das *Base bundle* in Abhängigkeit vom *Team* Identifier in der Map `baseBundlesByTeam` speichern.
Allerdings müssen die *Base bundles* für ein *Team* in dieser Map sortiert sein. Warum dies nötig ist und wie sie sortiert werden müssen, soll anhand eines Beispiels gezeigt werden:

[1] Diese Annahme muss nicht korrekt sein. Hier sind also noch zusätzliche Überprüfungen nötig (s. 8.2).

Nehmen wir an es existieren drei *Bundles* `basebundle1`, `basebundle2` und `aspectbundle`. Diese *Bundles* enthalten dabei folgende Klassen:

```
basebundle1: foo.bar.C0
basebundle2: foo.bar.C0, foo.bar.C1
```

Das *Bundle* `aspectbundle` enthält das Team `T0` und deklariert über eine Extension, dass es die beiden *Bundles* `basebundle1` und `basebundle2` adaptiert. Das Team `T0` soll die beiden Rollen `R0` und `R1`, deren Basisklasse `C0` für `R0` und `C1` für `R1` sind, enthalten.
An dieser Stelle ist es nun nicht mehr trivial zu bestimmen, ob `R0` die Klasse `C0` aus dem *Bundle* `basebundle1` oder aus `basebundle2` als Basisklasse hat. Dazu muss bekannt sein, welche Strategie *Equinox* beim Auflösen der Abhängigkeiten verfolgt. Die Spezifikation von *OSGi* schreibt dabei nicht vor, welche Strategie verwendet werden muss. Sie gibt dem Benutzer also nicht die Möglichkeit zu erfahren, warum eine bestimmte Reihenfolge der *Bundles* beim Auflösen der Abhängigkeiten benutzt wurde. Allerdings ermöglicht sie es zumindest zu erfahren, welche Reihenfolge verwendet wird. Dazu hat jedes *Bundle* eine eindeutige Id gespeichert. Das *Bundle* mit der niedrigsten Id hat dabei Priorität (s. [Osg07]).
Wenn die *Base bundles* für ein *Team* in der Map `baseBundlesByTeam` also aufsteigend nach ihrer Id sortiert sind, ist sichergestellt, dass beim Suchen des *Bundles* für eine Basisklasse (s. 7.4.2) immer das richtige *Bundle* gefunden wird.

Nachdem somit sichergestellt ist, dass für die Basisklasse eines *Teams* der Identifier bestimmt werden kann, ist es nun noch nötig, eine Möglichkeit zu schaffen, wie für eine Klasse der Identifier ihrer Superklasse ermittelt werden kann (s. 7.4.2). Wie dies möglich ist, zeigt der nächste Abschnitt.

Identifier von Superklassen

Zuletzt muss noch die Methode `getSuperclassIdentifier` implementiert werden.
Auch hierfür wird eine Map verwendet. Diese ist wie folgt definiert:

```
Map<String, String> superClassIdentifiersBySubclassIdentifiers;
```

Sie bietet also über den Identifier einer Klasse Zugriff auf den Identifier ihrer Superklasse. Diese Map könnte eigentlich recht trivial gefüllt werden. Es würde genügen in der Methode `recordClassDefine` die Instanz der Superklasse durch den Aufruf `getSuperclass` auf der Class Instanz der Subklasse zu holen. Nun müsste lediglich noch das *Bundle* der Superklasse ermittelt werden, was leicht möglich ist.

Allerdings reicht dieses Vorgehen aus folgendem Grund nicht aus:
Das Problem an der Stelle ist der Zeitpunkt des Zugriffs auf die Superklasse einer Klasse. Dieser geschieht beim Setzen des Bytecodes für eine Klasse im `ClassRepository` (s. 6.3.4). Angestoßen wird das Setzen im *OSGi* Kontext in

der Methode `processClass` im `TransformerHook` (s. 7.4.1).
Das bedeutet, zu dem Zeitpunkt, an dem die Methode `processClass` für eine Klasse aufgerufen wird, muss der Identifier bereits bekannt sein. Um zu erläutern, warum das ein Problem darstellt, soll hier zunächst kurz anhand eines Sequenzdiagramms (s. Abbildung 7.3) dargestellt werden, in welcher Reihenfolge die Methoden `processClass` und `recordClassDefine` für eine Klasse und ihre Superklasse aufgerufen werden. Dabei wird nur der Fall betrachtet, dass die Subklasse vor ihrer Superklasse im Programm verwendet wird. Ist die Reihenfolge umgekehrt können wir das oben beschriebene triviale Vorgehen anwenden.

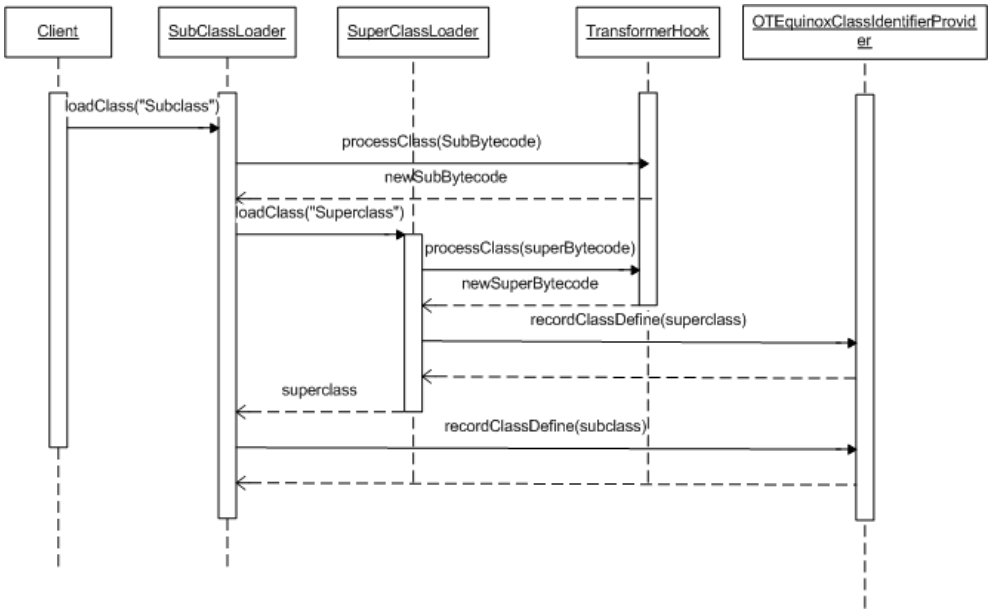

Abbildung 7.3: Classloader und Hooks

Die Abläufe sind in diesem Diagramm stark vereinfacht dargestellt. Es werden aber die wesentlichen Zusammenhänge dargestellt. `SubClassLoader` bezeichnet dabei den *Class loader* der Subklasse, `SuperClassLoader` den der Superklasse.
Zunächst wird an einer beliebigen Stelle im Programm (`Client`) eine Class Instanz vom `SubClassLoader` angefordert. Dieser lädt den *Bytecode* dieser Klasse und ruft mit diesem die Methode `processClass` des `TransformerHook` auf. Diese Methode kann jetzt den *Bytecode* manipulieren. Anschließend wird die Superklasse geladen. Dies wird von Java so erzwungen. Auch für die Superklasse wird nun der *Bytecode* geladen und an `processClass` übergeben. Nachdem diese Methode beendet ist, wird `recordClassDefine` des `OTEquinoxClassIdentifierProviders` für die Superklasse aufgerufen. Anschließend gibt der `SuperClassLoader` die Class Instanz der Superklasse an den `SubClassLoader` zurück. Dieser ruft dann `recordClassDefine` für die Subklasse auf.
Anhand dieses Diagramms ist zu sehen, dass zu dem Zeitpunkt, an dem die

Methode `processClass` für eine Subklasse aufgerufen wird, noch kein Identifier für ihre Superklasse, weder in `processClass` noch in `recordClassDefine`, gespeichert worden sein kann.
Das heißt es ist nötig, dass Laden der Superklasse vor dem Laden der Klasse zu erledigen, damit ein Identifier für die Superklasse bestimmt werden kann.
Das geschieht beim Aufruf der Methode `processClass` für die Subklasse. Dort wird zunächst eine Instanz von `AbstractBoundClass` für die Subklasse geholt, um den Namen der Superklasse bestimmen zu können. Dieser wird dann von der `AbstractBoundClass` abgefragt. Anschließend wird die Superklasse durch den Aufruf `loadClass` auf dem `ClassLoader` der Subklasse geladen. Da die Superklasse sowieso geladen worden wäre, ändert dieser Aufruf nichts außer der Ladereihenfolge. Es wird also kein Laden von Klassen erzwungen, die nicht sowieso geladen worden wären.
Nach diesem Aufruf werden nun die Methoden `processClass` und `recordClassDefine` für die Superklasse aufgerufen. In `recordClassDefine` kann jetzt der Identifier der Superklasse gespeichert werden. Anschließend wird die Methode `processClass` für die Subklasse fortgesetzt. An dieser Stelle kann nun mittels der Class Instanz der Superklasse ihr Identifier aus dem `ClassIdentifierProvider` abgefragt werden und diesem die Relation zwischen den Identifiern der Sub- und Superklasse durch den Aufruf der Methode

```
protected void addSuperclassIdentifier(String classId,
        String superClassId)
```

hinzugefügt werden.

7.4.3 Einführung eines Java agents für OT/Equinox

Wie schon erwähnt wird für das Weben der Klassen zur Ladezeit kein *Agent* mehr benötigt, da dies über *Hooks* von *Equinox* erledigt wird. Allerdings ist in der neuen Implementierung von *OT/Equinox* dennoch ein *Java agent* nötig.

Der Grund dafür ist, dass der *Agent* als einzige Instanz Zugriff auf eine Instanz des Interfaces `Instrumentation` aus der *JPLIS API* hat. Diese muss also auch in *OT/ Equniox* vom *Agent* gespeichert und der *OT/J-Laufzeitumgebung* zur Verfügung gestellt werden.
Grundsätzlich stellt die Verwendung eines *Java agents* für *OSGi* keine Probleme dar, da *OSGI* bzw. *Equinox* und der Agent unabhängig voneinander arbeiten. Ein Problem stellt allerdings die Verwendung von *Class loadern* in *OSGi* dar. Wie in 7.1.3 dargestellt hat in *OSGi* jedes *Bundle* seinen eigenen *Class loader.* Der Parent von allen *OSGi Class loadern* ist standardmäßig der *Bootstrap class loader* (s. 2.2.1). Der *Java agent* wird dagegen vom *Application class loader* geladen.
Beim Start des Programms wird nun zunächst vom *Application class loader* eine Class Instanz der *Agent* Klasse bereitgestellt. In dieser wird auch die Instanz von `Instrumentation` gespeichert. Wenn jetzt aber aus *OSGi* heraus der Agent referenziert wird, wird die Agent Klasse von dem *Class loader* des *Bundles* nicht

gefunden, da der *Application class loader* nicht in seiner *Class loader* Hierarchie auftaucht. Dadurch kann kein Zugriff auf die gespeicherte Instanz von Instrumentation genommen werden.
Allerdings ermöglicht es *OSGi* über das Setzen der *System property* `osgi.parentClassloader` den *Parent class loader* für die *Class loader* der *Bundles* festzulegen. Folgende Werte sind für diese Property möglich:

1. `app`: Der *Application class loader* wird als Parent verwendet
2. `ext`: Der *Extension class loader* wird als Parent verwendet
3. `boot`: Der *Bootstrap class loader* wird als Parent verwendet
4. `fwk`: Der *Framework class loader* wird als Parent verwendet

Der Wert `app` erzeugt also genau die Hierarchie, die wir benötigen. Nun hat jedes *Bundle* den *Application class loader* in seiner *Class loader* Hierarchie und kann auf die bereits geladenen Klasse des *Agents* zugreifen.

Jede Klasse aus der *OT/J-Laufzeitumgebung,* die vom Agent referenziert wird, muss dabei allerdings auch im Agent enthalten sein, da diese dem *Application class loader* sonst nicht bekannt sind. Sie werden erst vom *Class loader* des *Bundles* `org.objectteams.pde.core.lib` geladen. Würde die Klasse `otreAgent` (s. 6.1) nun auch als Agent für *OT/Equinox* verwendet, so müssten alle Klassen, die von diesem Agent direkt oder indirekt referenziert werden auch im Agent enthalten sein. Das sind allerdings (indirekt über `ObjectTeamsTransformer`) alle Klassen der *Laufzeitumgebung*. Das würde den Agent unnötig vergrößern und soll deshalb vermieden werden.
Da ja, wie gesagt, die Aufgabe des *OT/Equinox Agents* im Gegensatz zum *OT/J Agent* nur noch darin besteht, die Instanz des Interfaces `Instrumentation` zu speichern, ist es aber auch gar nicht nötig, dass der *OT/Equinox Agent* den `ObjectTeamsTransformer` (und damit weitere Klassen) referenziert.
Es wird also eine neue Klasse benötigt, die als Agent für *OT/Equinox* dient.
Diese Aufgabe soll die Klasse `OTEquinoxAgent` übernehmen, die im *Plug-in* `org.objectteams.otequninox` folgendermaßen implementiert ist:

```
1.  public class OTEquinoxAgent {
2.      private static Instrumentation instCopy;
3.
4.      public static void premain(String options,
5.              Instrumentation inst) {
6.          instCopy = inst;
7.      }
8.
9.      public static Instrumentation getInstrumentation() {
10.         return instCopy;
11.     }
12. }
```

Die Benutzung eines anderen *Agents* für *OT/Equinox* stellt allerdings ein Problem dar. Bis jetzt wurde der *OT/J Agent* in der Klasse `AsmWritableBoundClass` benutzt, um dort die Methode `redefineClasses` des Interfaces

`Instrumentation` aufrufen zu können (s. 6.4.2). Hier wurde die Klasse `otreAgent` fest referenziert. Das ist nun nicht mehr möglich, da jetzt in `AsmWritableBoundClass` nicht mehr bekannt ist, in welchem Kontext die Laufzeitumgebung benutzt wird und welcher Agent in diesem Kontext verwendet werden muss. An dieser Stelle muss also eine Indirektion eingeführt werden.

Dazu soll das Interface `IRedefineStrategy` dienen, das wie folgt definiert ist:

```
1. public interface IRedefineStrategy {
2.     public void redefine(Class<?> clazz, byte[] bytecode)
3.             throws ClassNotFoundException,
4.             UnmodifiableClassException;
5. }
```

Von diesem Interface können dann zwei Implementierungen existieren. Die eine, `OtreRedefineStrategy`, ruft den `otreAgent` auf, die andere, `OTEquinoxRedefineStrategy`, den `OTEquinoxAgent`.
Natürlich muss die `AsmWritableBoundClass` die Möglichkeit haben sich eine Instanz dieses Interfaces zu holen.
Dies ermöglicht die Klasse `RedefineStrategyFactory`:

```
1.  public class RedefineStrategyFactory {
2.      private static IRedefineStrategy redefineStrategy;
3.
4.      public static IRedefineStrategy
5.              getRedefineStrategy() {
6.          if (redefineStrategy == null) {
7.              redefineStrategy = new
8.                      OtreRedefineStrategy();
9.          }
10.         return redefineStrategy;
11.     }
12.
13.     public static void setRedefineStrategy(
14.             IRedefineStrategy redefineStrategy) {
15.         RedefineStrategyFactory.redefineStrategy =
16.                 redefineStrategy;
17.     }
18. }
```

Diese Factory arbeitet analog zu der `ClassIdentifierProviderFactory` (s. 7.4.2). Wird hier keine Instanz gesetzt, so liefert sie die `OtreRedefineStrategy` zurück, die den `otreAgent` aufruft. Für *OT/Equinox* muss dagegen eine Instanz von `OTEquinoxRedefineStrategy` gesetzt werden. In dieser Klasse wird der `OTEquinoxAgent` benutzt.
Das Setzen der Instanz geschieht genau wie bei der `ClassIdentifierProviderFactory` im `HookConfigurator`.

Im Gegensatz zum `otreAgent` kann im `OTEquinoxAgent` die benötigte Class Instanz allerdings nicht mehr durch den Aufruf `Class.forName` geholt werden, da die zu redefinierende Klasse und die `OTEquinoxRedefineStrategy` unterschiedliche *Class loader* haben.
Um dennoch Zugriff auf eine Class Instanz bekommen zu können, soll die Class Instanz im `ClassIdentifierProvider` gespeichert werden. Dies ist in der Methode `recordClassDefine` (s. 7.4.2) problemlos möglich. Von diesem können sie dann in der `OTEquinoxRedefineStrategy` geholt werden.

Nachdem nun in diesem Kapitel die Anpassung von *OT/Equinox* an die neue *OT/J-Laufzeitumgebung* beschrieben wurde, soll nun im letzten Kapitel die Arbeit noch einmal zusammengefasst und bewertet werden.

8 Zusammenfassung und Ausblick

In diesem Kapitel soll nun noch einmal zusammenfassend beschrieben werden, welche Ziele dieser Arbeit eingehalten werden konnten, welche offenen Punkte noch existieren und in welche Richtung sich die OT/J-Laufzeitumgebung in Zukunft entwickeln kann.
Ziel dieser Arbeit war es aufzuzeigen, wie dynamisches *Aspektweben* zur Laufzeit für *OT/J* praxistauglich umgesetzt werden kann. Auch im Kontext des *OSGi Komponentenframeworks* sollte es die neue Laufzeitumgebung ermöglichen, Aspekte dynamisch zu weben.
Ein großer Teil der Ziele wurde in dieser Arbeit auch eingehalten. Der größte Nachteil der bisherigen *OT/J-Laufzeitumgebung* wurde behoben, *Teams* und Basisklassen können jetzt in beliebiger Reihenfolge geladen werden.

8.1 Die neue OT/J-Laufzeitumgebung

Zunächst wurde das Konzept für *dynamisches Laufzeitweben* aus [Flü06] so verfeinert, dass es technisch umgesetzt werden kann.
Anschließend wurde eine neue Laufzeitumgebung für OT/J entwickelt, die das Laufzeitweben umsetzt.
Hierbei wurde auf zwei Dinge besonderer Wert gelegt:
Zum einen sollte die Implementierung performant genug sein, um dem Benutzer einen produktiven Einsatz von OT/J zu ermöglichen. Dazu wurde sichergestellt, dass

- Laufzeitweben so selten wie möglich durchgeführt wird. Soweit wie möglich werden alle Klassen schon zur Ladezeit transformiert.
- der *Bytecode* der Klassen nicht für jede Transformation neu eingelesen werden muss.
- Synchronisation so selten wie möglich durchgeführt wird

Zum anderen sollte die neue *OT/J-Laufzeitumgebung* auch so implementiert sein, dass sie optimal und mit wenig Aufwand weiterentwickelt und gewartet werden kann. Auch sollte sie eine leichte Einarbeitung ermöglichen. Dazu wurde(n)

- Klassen mit vielen statische Strukturen weitestgehend vermieden
- Klassen und Methoden möglichst klein gehalten
- die Implementierung so weit wie möglich unabhängig von der verwendeten *Bytecode library* gehalten (lediglich eine Klasse dient als Schnittstelle zwischen Klassen mit bzw. ohne Kenntnis der *Bytecode library*)
- Bezeichner einheitlich und aussagekräftig gewählt
- der Code gut dokumentiert

8.1.1 Offene Punkte

In dieser Arbeit war es nicht mehr möglich eine detaillierte Untersuchung der Performance der neuen *OT/-Laufzeitumgebung* durchzuführen. Dieses muss in weiteren Arbeiten noch erfolgen.
Weiterhin ist auch die Testabdeckung der neuen *OT/J-Laufzeitumgebung* noch nicht voll zufriedenstellend. Hier muss noch detaillierter getestet werden, welche Features in der neuen Laufzeitumgebung bereits umgesetzt sind und welche nicht.

Im Folgenden sollen die Features beschrieben werden, bei denen bekannt ist, dass sie von der neuen Implementierung noch nicht umgesetzt worden sind.

Super calls in Basismethoden

Enthält eine gebundene Basismethode eine *Super call*, kann es zu Problemen bei der Ausführung der *Callin Bindungen* kommen. Dies soll durch ein Beispiel näher erläutert werden.
Nehmen wir an, es existieren folgende *Teams*, *Rollen* und Klassen:

```
1.  class B0 {
2.      public void bm() {
3.          // do something
4.      }
5.  }
6.
7.  class B1 extends B0 {
8.      @Override
9.      public void bm() {
10.         //do something
11.         super.bm();
12.         //do something
13.     }
14. }
15.
16. public team class T0 {
17.     protected class R0 playedBy B0 {
18.         callin void rm() {
19.             //do something
20.             base.rm();
21.             //do something
22.         }
23.     }
24.     rm <- replace bm;
25. }
```

Wenn nun das `T0` aktiv ist und die Methode `bm` von `B1` aufgerufen wird, würde es zu folgender Aufrufreihenfolge kommen:
1. Aufruf von `B1.bm()`
2. `B1.bm()` wird ersetzt durch `R0.rm()` durch die *Callin Bindung*
3. Aufruf von `B1.bm()` durch den *Base call*
4. Aufruf von `B0.bm()` durch den *Super call*
5. `B0.bm` wird ersetzt durch `R0.rm()` durch die *Callin Bindung*
6. Aufruf von `B1.bm()` durch den *Base call*
7. Fortsetzung bei Schritt 4.

Diese Konstellation würde also zu einer Endlosrekursion führen. Aus diesem Grund müssen *Super calls* innerhalb von Code, der in die Methode `callOrig` verschoben wird, durch den Aufruf der Methode `callOrig` der Superklasse ersetzt werden, wenn diese Basismethode in der Superklasse Bindungen hat.

Einen Spezialfall davon stellen die sog. *Wicked super calls* dar. Um diese zu erklären nehmen wir wieder obiges Beispiel an. Die Klasse `B1` ist nun allerdings folgendermaßen definiert:

```
1. class B1 extends B0 {
2.       @Override
3.       public void bm() {
4.          //do something
5.          bm2();
6.          //do something
7.       }
8.
9.       private void bm2() {
10.          //do something
11.          super.bm();
12.          //do something
13.      }
14. }
```

Hierbei wird also nur indirekt die Supermethode von `bm` aufgerufen. Allerdings würde auch dieser Fall zu der oben erwähnter Endlosrekursion führen. Auch dieser *Super call* muss also ersetzt werden.

In beiden Fällen werden von der neuen *OT/J-Laufzeitumgebung* die *Super calls* noch nicht ersetzt.

Erhalten der Instanzen von gebundenen Rollen

Jede *Rolle*, die durch eine `playedBy` Beziehung an eine Basisklasse gebunden ist, stellt eine konzeptionelle Einheit mit der Basisklasse dar. Dies bedeutet u.a., dass eine Instanz der Rolle solange nicht vom *Garbage collector* gelöscht werden darf, wie die zugehörige Instanz der Basisklasse existiert. Gleichzeitig muss aber auch sichergestellt sein, dass die Instanz der *Rolle* nicht das Löschen der Basisinstanz verhindert.
In jeder Instanz einer gebundenen *Rolle* ist die zugehörige Instanz der Basisklasse gespeichert. Zusätzlich hält jedes *Team* in internen Strukturen die Instanzen aller *Rollen*, die in diesem *Team* definiert sind. Grundsätzlich würde diese Struktur zwar verhindern, dass eine Rolleninstanz gelöscht wird, solange das *Team* aktiv ist, allerdings würde auch das Löschen der Basisinstanz verhindert. Aus diesem Grund hält ein *Team* nur schwache Referenzen auf die Rolleninstanzen. Schwache Referenzen ermöglichen es, eine Referenz auf ein Objekt zu halten, das dennoch gelöscht werden kann. In Java werden schwache Referenzen durch die Klasse `WeakReference` abgebildet.
Nun können zwar die Basisinstanzen wieder gelöscht werden, allerdings ist jetzt nicht mehr sichergestellt, dass die Rolleninstanzen erhalten bleiben, solange die zugehörige Basisinstanz existiert.
Um auch das sicherstellen zu können, enthalten in der bisherigen *OT/J-Laufzeitumgebung* die Basisinstanzen Instanzen aller Rollen, von denen sie gebunden werden. Rollen können ihre Instanzen der Basisinstanz über die Methoden

```
void _OT$addRole(Object aRole);
void _OT$removeRole(Object aRole);
```

hinzufügen oder wieder entfernen.

Dieses Verhalten ist in der neuen *OT/J-Laufzeitumgebung* noch nicht implementiert.

Smart lifting

Um das in 3.6.1 beschriebene *Smart lifting* umsetzen zu können, muss innerhalb eines Teams entschieden werden können, zu welcher Rolle eine Basisklasse geliftet werden soll. Dazu ist es nötig den dynamischen Typ des Basisobjekts zu ermitteln. Dies könnte über den `instanceof` Operator erfolgen. Da die Verwendung dieses Operators relativ teuer ist, wurde in der bisherigen *OT/J-Laufzeitumgebung* allerdings eine andere Variante gewählt.
In jedem *Team* wird über das Attribut `BaseClassTags` eine Zuordnung von Ids zu den Namen aller Basisklassen dieses *Teams* gespeichert. Die Laufzeitumgebung liest dazu die `BaseClassTags` Attribute des Teams aus und kodiert die Ids in den Basisklassen. In der Basisklasse wird für ein Team *T* dann die Methode
`short get`*`T`*`_OT$Tag();`
definiert, über die die Id für diese Basisklasse in dem Team *T* wieder ausgelesen werden kann.
Diese Ids, und damit auch die Methoden zum auslesen, sind Team abhängig. Das heißt, eine Basisklasse muss für jedes Team, von dem sie gebunden wird eine solche Methode implementieren. Das wäre allerdings beim Einsatz des *Laufzeitwebens* nicht mehr möglich, da hier zur Laufzeit keine Methoden hinzugefügt werden können.

Es müsste also eine generische Methode in den Basisklassen geschaffen werden, die das Auslesen der Ids ermöglicht oder es müsste ein anderer Weg gefunden werden, wie das *Smart lifting* ermöglicht werden kann.

Layering

Wie schon in 3.2 erwähnt, können auch *Rollen* als Basisklassen anderer *Rollen* fungieren, wenn die *Rollen* in verschiedenen *Teams* definiert sind. Dieses wird als *Layering* bezeichnet.
Da *Rollen* nicht statisch sein können, wird, um eine Rolle ansprechen zu können, eine Instanz des *Teams* benötigt, in dem diese *Rolle* definiert ist. *OT/J* schreibt es zusätzlich vor, dass diese Teaminstanz als `final` deklariert sein muss. Die Benutzung einer *Rolle* als Basisklasse müsste also folgendermaßen implementiert werden:

```
1.  public team class T0 {
2.      final T1 other;
3.      public T0(T1 other) {
4.          this.other = other;
5.      }
6.      protected class R0 playedBy R1<@other> {
7.          ...
8.      }
9.  }
10.
11. public team class T1 {
12.     protected class R1 {
13.         ...
14.     }
15. }
```

Wenn jetzt `R0` *Callin Bindungen* zu `R1` hat, dürfen diese nur ausgeführt werden, wenn die Instanz von `T1` aktiv ist, die in `T0` referenziert wird. Ist dagegen eine andere Instanz dies Teams `T1` aktiv, dürfen die Bindungen nicht greifen.

Auch dieses Verhalten ist in der neuen Implementierung der *OT/J-Laufzeitumgebung* noch nicht implementiert.

Bindungsvererbung bei kovarianter Methodenredefinition

In Java können Methoden, die eine Methode der Superklasse überschreiben, einen spezielleren Rückgabetyp haben, als die Supermethode. Diese Konstellation wird als *Kovarianz* genannt.

Dies muss bei der Bindungsvererbung berücksichtigt werden. Auch kovariant überschreibende Methoden der Subklasse müssen die Bindungen der Supermethode erben.

In der neuen *OT/J-Laufzeitumgebung* wird aber bei der Umsetzung der Bindungsvererbung in den Subklassen nach einer Methode gesucht, die exakt denselben Rückgabetyp hat, wie die Methode der Superklasse.
Bindungen würden in diesem Fall also nicht vererbt.

Decapsulation von Methoden mit der Sichtbarkeit protected

Versucht eine *Rolle* auf ein nicht sichtbares Member ihrer Basisklasse Zugriff zu nehmen, wird momentan davon ausgegangen, dass immer das Member von genau dieser Klasse angesprochen werden muss. Wenn also das Basisobjekt als dynamischen Typ eine Subklasse der Basisklasse hat, wird dennoch auf das Member der Basisklasse zugegriffen. Für Felder und private und statische Methoden, ist dieses Verhalten auch richtig, da diese nicht vererbt werden können. Hier soll also nicht in der Subklasse nach dem Member gesucht werden.

Handelt es sich bei dem Member dagegen, um eine nicht statische Methode mit der Sichtbarkeit `protected`, die in der Subklasse redefiniert wird, führt dieses Verhalten allerdings zu einem Fehler.
Hier muss dann auf die Methode der Subklasse und nicht auf die Methode der direkten Basisklasse zugegriffen werden.

8.2 Anpassung von OT/Equinox an die neue Laufzeitumgebung

OT/Equinox konnte in dieser Arbeit erfolgreich an die neue Laufzeitumgebung angepasst werden. Dabei ermöglicht es die neue Implementierung zum ersten Mal auch mit mehreren Klassen mit gleichem voll qualifizierten Namen umgehen zu können. Dies ist nun sowohl für *Teams* als auch für Basisklassen möglich.

In einem Fall funktioniert auch die neue Implementierung von *OT/Equinox* noch nicht wie gewünscht. Wenn in einem *Bundle* durch eine *Extension* des *Extension points* `AspectBindings` eine Bindung von diesem *Bundle* zu einem *Base bundle* definiert ist, wird davon ausgegangen, dass das *Base bundle* entweder nur in einer Version vorliegt oder das die höchste Version des *Base bundles* von dem *Aspekt bundle* gebunden wird. Diese Konstellation soll anhand eines Beispiels nochmal verdeutlicht werden:

Es existieren drei *Bundles:*

Symbolischer Name	Version
`basebundle`	1.0.0
`basebundle`	2.0.0
`aspectbundle`	1.0.0

Das *Bundle* `aspectbundle` definiert durch eine *Extension*, dass es das *Bundle* mit dem Namen `basebundle` bindet. Gleichzeitig wird im *Manifest* von `apsectbundle` eine Abhängigkeit zu `basebundle` in der Version 1.0.0 definiert. Weiterhin wird angenommen, dass in beiden Versionen von `basebundle` dieselben Klassen existieren.
In diesem Fall würde die neue Implementierung von *OT/Equinox* die Aspekte fälschlicherweise in das *Bundle* `basebundle` in Version 2.0.0 weben.

Es muss also in *OT/Equinox* noch eine Überprüfung implementiert werden, ob ein *Aspekt bundle* im *Manifest* eine Abhängigkeit zu einem *Bundle* oder Package in einer bestimmten Version definiert. Ist dies der Fall müssen die Aspekte in das *Bundle* mit der richtigen Version eingefügt werden.

8.3 Ausblick

8.3.1 Weiterentwicklung von JPLIS

Von der Java Community wird schon lange gefordert, dass die Beschränkungen von *JPLIS* zur Redefinition von Klassen zur Laufzeit (zumindest teilweise) aufgehoben werden. Dazu existiert seit dem 21.08.2003 ein *Request for enhancement* (*RFE*) bei Sun (s. [Sun03]).
Wenn die Beschränkungen von *JPLIS* bei der Redefinition von Klassen zur Laufzeit tatsächlich wegfallen würden, könnte man die Implementierung der *OT/J-Laufzeitumgebung* noch deutlich effizienter gestalten. Es wäre nun nicht mehr nötig, jede Klasse zur Ladezeit zu präparieren, sondern es würde genügen, die nötigen Methoden erst zur Laufzeit hinzuzufügen. Gerade bei großen Systemen mit tausenden von Klassen könnte hiermit ein enormer Performancegewinn erreicht werden.
Allerdings ist nicht bekannt, wann dieser *RFE* von Sun umgesetzt wird.

Anhang A

Technische Instruktionen

In diesem Anhang wir beschrieben, welche Schritte nötig sind, um die neue *OT/J-Laufzeitumgebung* zu benutzen. Dazu soll zunächst die Struktur des Subversion Repositories dargestellt werden, in dem die nötigen Projekte abgelegt sind.

Struktur des Repositories

Alle Artefakte, die beim Schreiben dieser Diplomarbeit entstanden sind (auch die Diplomarbeit selbst), liegen in dem Subversion Repository https://svn.objectteams.org/diplom/ofrank/. Das Repository hat die Struktur, die in Abbildung A.1 dargestellt ist.

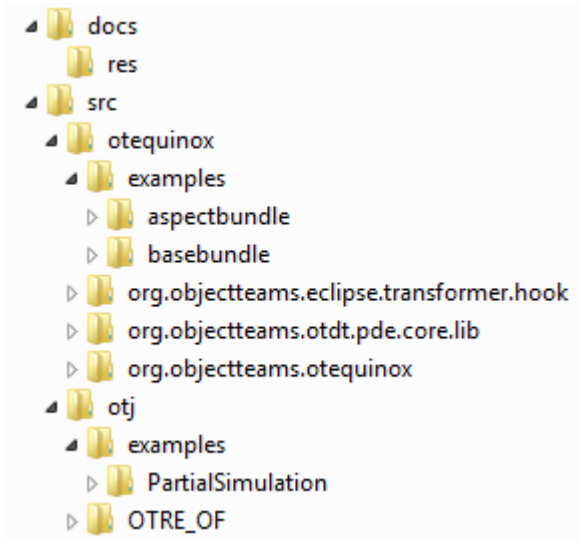

Abbildung A.1: Repository Struktur

Im Ordner `docs` befindet sich zum einen die Diplomarbeit selbst und zum anderen, im Unterordner `resource`, alle Grafiken, die in der Arbeit verwendet wurden.

Im Ordner `src` findet sich die Implementierung, die im Laufe dieser Arbeit erstellt wurde. Im Unterordner `otj` findet sich das Projekt `OTRE_OF`. Dieses Projekt stellt die neue Laufzeitumgebung dar. Im Unterordner `otequinox` des Ordners `src` finden sich die Projekte `org.objectteams.otequinox`, `org.objectteams.otdt.pde.core.lib` und `org.objectteams.eclipse.transformer.hook`. Diese Projekte bilden die neue Implementierung von *OT/Equinox*.
Zusätzlich existiert in den Ordnern `otj` und `otequinox` jeweils der Ordner `examples`. In diesem Ordner sind Beispielprojekte enthalten, die demonstrieren, wie die Implementierungen zu benutzen sind. Für die *OT/J-Laufzeitumgebung* ist das das Projekt `PartialSimulation`, für *OT/Equinox* die Projekte `basebundle` und `aspectbundle`.

Im nächsten Abschnitt soll nun beschrieben werden, wie die neue *OT/J-Laufzeitumgebung* und die neue Implementierung von *OT/Equinox* eingesetzt werden können.

Benutzung der neuen Implementierungen

Um das Beispielprojekt `PartialSimulation` mit der neuen *OT/J-Laufzeitumgebung* zu starten, sind folgende Schritte nötig. Dabei wird davon ausgegangen, dass *Eclipse* in der Version 3.4.1 als Entwicklungsumgebung verwendet wird. *OT/J Plugin-ins* dürfen in dieser *Eclipse* Instanz dabei nicht installiert sein, da sonst z.B. normale Klassen nicht von der Klasse `Team` erben dürfen:
Zuerst müssen die Projekte `OTRE_OF` und `PartialSimulation` in *Eclipse* aus dem Repository ausgecheckt werden.
Anschließend muss der *OT/J Agent* erzeugt werden. Dazu muss in der Datei `OTRE_OF/otre_agent.jardesc` "`<OTRE_OF_PATH>`" durch den absoluten Pfad des Projekts `OTRE_OF` ersetzt werden. Anschließend kann mit dieser Datei der *Agent* erstellt werden.
Solange der OT/J Compiler das neue Laufzeitweben noch nicht unterstützt, können die Bindungen nicht aus den Attributen der Teams ausgelesen, sondern müssen hart kodiert werden. Dies geschieht am Ende der Methode `parseBytecode` in der Klasse `org.objectteams.bytecode.asm.AsmBoundClass`. Dort müssen Bindungen nach folgendem Muster deklariert werden:

```
1.  // For every adapting team:
2.  if (getName().equals(<TEAM_NAME>)) {
3.      //For every tupel (Baseclass, Basemethod, CallinId) {
4.      Binding binding = new Binding(<BASE_CLASS_NAME>,
5.              <BASE_METHOD_NAME>, <BASE_METHOD_SIGNATURE>,
6.              <CALLIN_ID>, BindingType.CALLIN_BINDING);
7.      addBinding(binding);
8.      //}
9.
10.     // For every decapsulation binding to a field {
11.     Binding binding = new Binding(<BASE_CLASS_NAME>,
12.             <BASE_FIELD_NAME>, <BASE_FIELD_SIGNATURE>,
13.             <ACCESS_ID>, BindingType.FIELD_ACCESS);
14.     // }
15.
16.     // For every decapsulation binding to a method {
17.     Binding binding = new Binding(<BASE_CLASS_NAME>,
18.             <BASE_METHOD_NAME>, <BASE_METHOD_SIGNATURE>,
19.             <ACCESS_ID>, BindingType.METHOD_ACCESS);
20.     // }
21.
22.     setTeam(true);
23. }
```

Um das Projekt `PartialSimulation` zu starten, müsste diese Deklaration beispielsweise so aussehen:

```
1. if (getName().compareTo("T0") == 0) {
2.     Binding binding = new Binding("C0", "test", "()V",
3.             0, BindingType.CALLIN_BINDING);
4.     addBinding(binding);
5.     setTeam(true);
6. }
```

Nun kann das Projekt mit der Launch configuration `PartialSimulation/launch/PartialSimulation.launch` gestartet werden. Wurde alles richtig konfiguriert, sollte die Ausgabe folgendermaßen aussehen:

```
Method "roletest" in role "R0"
Method "test" in class "C0"
```

Für die Benutzung von *OT/Equinox* muss zunächst, wie oben beschrieben, die *OT/J-Laufzeitumgebung* ausgecheckt werden. Zusätzlich werden noch die Projekte `org.objectteams.otequinox`, `org.objectteams.otdt.pde.core.lib`, `org.objectteams.eclipse.transformer.hook`, `basebundle` und `aspectbundle` benötigt. Zusätzlich muss noch das *System bundle* `org.eclipse.osgi` lokal vorliegen. Anschließend muss der *OT/Equinox Agent* erzeugt werden. Hierfür muss in der Datei `org.objectteams.otequinox/otequinox_agent.jardesc`

"<ORG_OBJECTTEAMS_OTEQUINOX_PATH>" durch den absoluten Pfad des Projekts org.objectteams.otequinox ersetzt werden. Nun kann der *Agent* mit dieser Datei erstellt werden.
Anschließend müssen, wie oben beschrieben, die Bindungen in AsmBoundClass definiert werden. Für das Beispielprojekt aspectbundle entspricht diese Definition der Definition der Bindungen für das Projekt PartialSimulation.
Nun kann das Projekt mit der Launch configuration aspectbundle/launch/AspectBundle.launch gestartet werden. Auch hier sollte die Ausgabe folgendermaßen aussehen:

```
Method "roletest" in role "R0"
Method "test" in class "C0"
```

Literaturverzeichnis

[Asj01] AspectJ Website , 2001, http://www.eclipse.org/aspectj/

[Asm02] ASM Website , 2002, http://asm.ow2.org/

[Aus00] Austermann, Michael: Ladezeittransformation von Java-Programmen, Rheinische Friedrich-Wilhelms-Universität Bonn, Diplomarbeit, 2000

[Flü06] Flüh, Michael: Schemaerhaltende Bytecodetransformationen zum Aspektweben zur Programmlaufzeit, Technische Universität Berlin, Diplomarbeit, 2006

[Fun09] Funke, Holger: Das OSGi Framework In: Java Magazin(2009), 5.2009

[GJSB05] Gosling, James; Joy, Bill; Steele, Guy; Bracha, Gilad: The Java(tm) Language Specification, 2005, Addison-Wesley Longman, ISBN: 0321246780

[Her02] Herrmann, Stephan: Object Teams: Improving Modularityfor Crosscutting Collaborations, 2002

[HHM09] Herrmann, Stephan; Hundt, Christine; Mosconi, Marco; : ObjectTeams/Java Language Definition version 1.2, 2009

[HS07] Haupt, Michael; Schippers, Hans: A Machine Modelfor Aspect-Oriented Programming, 2007

[Hun03] Hundt, Christine: Bytecode-Transformationen zur Laufzeitunterstützung von aspekt-orientierter Modularisierung mit ObjectTeams/java, Technische Universität Berlin, Diplomarbeit, 2003

[KR91] Kiczales, Gregor; Des Rivieres, Jim: The Art of the Metaobject Protocol, 1991, MIT Press, ISBN: 0262610744

[LB98] Liang, Sheng; Bracha, Gilad: Dynamic Class Loading in the Java Virtual

Machine, 1998

[LS08] Lippert, Martin; Seeberger, Heiko: Getting Hooked on Equinox In: Java Magazin(2008), 3.2008

[LY99] Lindholm, Tim; Yellin, Frank: The Java(tm) Virtual Machine Specification, 1999, Addison-Wesley Longman, ISBN: 0201432943

[Mey97] Meyer, Bertrand: Object Oriented Software Construction, 1997, Prentice Hall International, ISBN: 0136291554

[Osg07] OSGi Alliance: OSGi Service Platform Core Specification Version 4.1, 2007

[Sun03] Sun RFE Website , 2003, http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4910812

[Sun03] Sun Website , 2003, http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4910812

[Sun04] Sun: Java(tm) Platform Debugger Architecture (JPDA), 2004

[Wik07] Wikipedia Website , 2007, http://en.wikipedia.org/wiki/Aspect-oriented_software_development

[Wun07] Wunderlich, Lars: Java Classloader In: Java Magazin(2007), 9.2007