Paul Häder

Benchmarking und Optimierung der Aspektwebestrategie von ObjectTeams/Java

– Diplomarbeit –

Berlin, 23. August 2006

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

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

"When there were no computers, programming was no problem. When we had a few weak computers, it became a mild problem. Now that we have gigantic computers, programming is a gigantic problem."

Edsger Dijkstra

1.1 Motivation

Die steigende Komplexität von Software zu beherrschen ist eine der größten Herausforderungen der Softwaretechnik. Den zentralen Ansatz stellt in diesem Zusammenhang die Modularisierung von Softwaresystemen dar. Das der Modularisierung zu Grunde liegende Prinzip wird mit dem ursprünglich von Edsger Dijkstra geprägten Begriff *separation of concerns* [Dij82] bezeichnet. Dieser beschreibt die Zerlegung eines Systems in unterschiedliche Einheiten, die sich funktionell möglichst wenig überschneiden. *Separation of concerns* fördert ein gutes Softwaredesign und eine höhere Wiederverwendbarkeit der Implementierung. Die Objektorientierung realisiert dies mit ihren Konzepten in weitem Maße, jedoch nicht erschöpfend. Aspektorientierte Programmierung [KLM+97] erweitert die Anwendung des Prinzips auf Eigenschaften, die quer zur Klassenstruktur eines Softwaresystems angesiedelt sind, so genannte *crosscutting concerns*.

Software kann mit Hilfe von Aspektorientierter Programmierung *strukturell* verbessert – und dadurch ihre Wartbarkeit, Verständlichkeit und Wiederverwendbarkeit erhöht werden. Es bleibt jedoch die wichtige Frage nach den Auswirkungen auf die Performance aspektorientierter Programme, da diese sich von der rein objektorientierter Programme auf Grund der zusätzlichen Konzepte unterscheidet. Das Problem der Performance[1] wird in dieser Arbeit am Beispiel der aspektorientierten Programmiersprache ObjectTeams/Java [Her02, OT] untersucht. Die Frage dabei ist, ob der Overhead, der durch ObjectTeams/Java verursacht wird, gegenüber einer rein

[1] Unter Performance wird nachfolgend die Ausführungsgeschwindigkeit verstanden. Der Speicherverbrauch soll nicht im Mittelpunkt dieser Arbeit stehen.

objektorientierten Lösung vertretbar ist bzw. wie man ihn minimieren kann.

1.2 Struktur

Die Arbeit ist wie folgt aufgebaut. Kapitel 2 gibt zunächst einen Überblick über aspektorientierte Programmierung im Allgemeinen und die Konzepte der Programmiersprache ObjectTeams/Java im Speziellen. Kapitel 3 enthält eine Übersicht über die eingesetzten Verfahren zur Performance-Messung. In Kapitel 4 werden die verschiedenen Performance-Messungen vorgestellt, die im Rahmen der Arbeit vorgenommen wurden. Es folgt eine Beschreibung der durchgeführten Optimierung des Webeprozesses und des von ObjectTeams/Java generierten Codes in Kapitel 5. Kapitel 6 beschreibt den angestellten Vergleich der Performance von ObjectTeams/Java mit der anderer aspektorientierter Sprachen. Kapitel 7 gibt schließlich eine zusammenfassende Bewertung der erlangten Erkenntnisse sowie einen Ausblick über mögliche weiter gehende Arbeiten.

Kapitel 2

Aspektorientierte Programmierung mit ObjectTeams

Die Aspektorientierung ist ein immer noch relativ neues Paradigma, das Gegenstand einer ganzen Reihe von Forschungen ist, ihren Eingang in die industrielle Softwareentwicklung jedoch erst allmählich vollzieht. Zunächst als aspektorientierte *Programmierung* entwickelt, sind die Bemühungen inzwischen dahingehend, die Aspektorientierung auf den gesamten Entwicklungsprozess anzuwenden [CB05, JN04]. So hat sich der Begriff der aspektorientierten Softwareentwicklung (AOSD) etabliert. Man kann z.B. bereits während der Anforderungsanalyse *Aspekte* identifizieren und sie schon im Entwurf mit berücksichtigen. In diesem Zusammenhang spricht man auch von *early aspects* [EA], also Aspekten, die bereits in frühen Entwicklungsphasen eine Rolle spielen. Die Aspektorientierung kann darüber hinaus auch beim Testen zum Einsatz kommen, beispielsweise für Überdeckungsmessungen oder speziell beim Unit-Testen [VS05]. Die derzeit am weitesten verbreitete aspektorientierte Programmiersprache ist AspectJ [KHH+01, AJ].

Folgend werden die grundlegenden Konzepte der aspektorientierten Programmierung erläutert und anschließend die Sprache ObjectTeams/Java vorgestellt.

2.1 Aspektorientierte Programmierung

Das zentrale Modularisierungskonzept der Objektorientierung ist bei Analyse und Design wie auch in der Implementierung die *Klasse*. In ihr werden Funktionalitäten gekapselt und sie definiert das Verhalten der Objekte. Ein Objekt kann nur über klar definierte Schnittstellen angesprochen werden. Die Vorgänge innerhalb des Objekts bleiben jedoch nach außen hin verborgen. Auf diese Weise gelingt es der Objektorientierung, die Komplexität eines Systems ein Stück weit zu „verstecken“. Dieses Verbergen von Implementierungsdetails wird als Kapselung bezeichnet. Durch *Vererbung* wird zudem die Wiederverwendung von Teilen des Codes ermöglicht.

2.1.1 *Crosscutting concerns* und Aspekte

Wie in der Einleitung bereits kurz angesprochen, ist der Grad der Modularisierung, der mit Hilfe der Objektorientierung erzielt werden kann, jedoch in einigen Fällen nicht ausreichend. Bestimmte Anforderungen an eine Software (*crosscutting concerns*) lassen sich in der Regel nicht vollständig in einzelnen funktionalen Einheiten konzentrieren, sondern sind über weite Teile des Systems verstreut. Beispiele hierfür sind Logging, Security, Verteilung oder Persistenz. Das Phänomen, dass Quellcode, der eine Systemanforderung betrifft, über viele Module verteilt ist, wird als *code scattering* bezeichnet. Beinhalten einzelne Module Quellcode, der mehrere verschiedene Funktionen erfüllt, spricht man von *code tangling*. *Code scattering* führt dazu, dass bei Änderungen der betreffenden Funktionalität diese an vielen verschiedenen Stellen im Code vorgenommen werden müssen, was die Wartung und Evolution der Software deutlich verkomplizieren kann. *Code tangling* verkörpert gewissermaßen das Gegenteil von *separation of concerns*. Die Wiederverwendbarkeit eines Moduls, in dem mehrere verschiedene Funktionen implementiert sind, ist stark eingeschränkt. *Code scattering* und *tangling* beeinträchtigen darüber hinaus die Verständlichkeit des Codes. Beide sind also im Sinne einer guten Modularisierung zu vermeiden.

Die Aspektorientierung stellt dazu ein zusätzliches Modularisierungskonzept neben die Klasse: den *Aspekt*. Um die Modularisierung auch für *crosscutting concerns* zu ermöglichen, werden diese in Aspekten gekapselt. Damit kommt zu dem Basiscode, in dem die eigentliche Kernfunktionalität implementiert wird, der so genannte Aspektcode hinzu. Aspektcode und Basiscode können separat voneinander entwickelt und zur Kompilier-

oder gar erst zur Laufzeit miteinander „verwoben“ werden. So werden *code scattering* und *tangling* verhindert und man erreicht die Lokalität von Änderungen, die die *crosscutting concerns* betreffen.

2.1.2 Joinpoints und Pointcuts

Da Aspekte getrennt implementiert werden, müssen sie anschließend in die Basisapplikation integriert werden. Dazu ist es zunächst notwendig, auf programmiersprachlicher Ebene zu definieren, an welchen Punkten in der Ausführung des Basisprogramms welcher Aspektcode hinzugefügt werden soll. Diese Punkte werden *Joinpoints* genannt. Mögliche Joinpoints sind z.B. der Aufruf oder die Ausführung einer Methode, der lesende oder schreibende Zugriff auf ein Attribut oder das Werfen bzw. Abfangen einer Exception. Darüber hinaus ist es möglich, Joinpoints auf einen bestimmten Kontrollfluss oder auf bestimmte Klassen bzw. Methoden zu beschränken.

Joinpoints werden in einigen aspektorientierten Programmiersprachen (z.B. AspectJ und JAsCo [SV03, Jas]) zu Pointcuts zusammengefasst. Pointcuts repräsentieren also Mengen von Joinpoints. Die Möglichkeit, Aussagen über Mengen von Joinpoints zu machen, wird nach Filman und Friedman *Quantification* genannt und als eins von zwei charakteristischen Merkmalen der aspektorientierten Programmierung angesehen [FF00]. Diese Quantifikationen werden deklarativ spezifiziert, wobei die konkrete Syntax von der verwendeten Programmiersprache abhängt.

Es gibt eine Reihe verschiedener Kategorien von Pointcuts. Grundsätzlich kann man entweder über die statische Programmstruktur oder über das dynamische Verhalten des Programms quantifizieren. Dementsprechend unterscheidet man statische von dynamischen Pointcuts, abhängig davon, ob sie bereits vor oder erst während der Laufzeit zugeordnet werden können. Außerdem gibt es zusammengesetzte Pointcuts. Diese bestehen wiederum aus Pointcuts, die mit logischen Operationen miteinander verknüpft werden. Schließlich existieren so genannte lexikalische Pointcuts, mit denen Joinpoints auf einzelne Klassen oder Methoden beschränkt werden können.

Das zweite charakteristische Prinzip der Aspektorientierung ist die Tatsache, dass beim Entwickeln des Basiscodes keinerlei Vorkehrungen für eine eventuelle Adaptierung durch einen Aspekt vorgenommen werden müssen. Der Entwickler der Kernfunktionalität soll mögliche Aspekte nicht

berücksichtigen müssen. Dies wird im Englischen als *Obliviousness*[1] bezeichnet [FF00]: Der Basiscode „weiß“ nichts von dem Aspektcode. Genau genommen schafft er jedoch immer die Voraussetzungen für eine Adaption, indem er mögliche Joinpoints zur Verfügung stellt, die von einem Aspekt genutzt werden können. Aus diesem Grund wurde stattdessen der Begriff der Nichtinvasivität vorgeschlagen, der beschreibt, dass das Basisprogramm adaptiert werden kann und der Code dabei trotzdem „unangetastet“ bleibt.

2.1.3 Aspektweben

Der Vorgang, der den Aspektcode an den definierten Stellen des Basiscodes integriert, wird Aspektweben genannt. Hierbei gibt es drei Möglichkeiten. Der Aspektcode kann vor, nach oder an Stelle des Basiscodes ausgeführt werden. Bei der letzt genannten Variante gibt es meist die Möglichkeit, den Originalcode aufzurufen. In AspectJ geschieht dies beispielsweise mit einem Aufruf von `proceed()`.

Hinsichtlich des Zeitpunktes gibt es drei Ansätze, wann dieser Webevorgang stattfinden kann: Weben zur Kompilierzeit (*static weaving*), zur Ladezeit (*loadtime weaving*) oder zur Laufzeit (*runtime weaving*, auch dynamisches Weben genannt).

Static weaving

Hinsichtlich aspektorientierter Systeme, die mit statischem Weben arbeiten, sind zwei Ansätze möglich. Entweder werden Basisprogramm und Aspekte auf der Ebene des Quellcodes verwoben und anschließend mit dem herkömmlichen Compiler der jeweiligen Programmiersprache kompiliert (*source code weaving*), oder es wird erst das Basisprogramm kompiliert und anschließend der Bytecode mit den Aspekten verwoben (*bytecode weaving*). Beim *source code weaving* muss im Falle von Änderungen am Aspektcode das gesamte Programm neu kompiliert werden. Außerdem wird der Quellcode des Basisprogramms benötigt. Mit *bytecode weaving* hingegen können Aspekte auch in ein Basisprogramm eingewoben werden, das beispielsweise nur als jar-Datei vorliegt. In den Abbildungen 2.1 und 2.2 sind beide Arten des statischen Webens schematisch dargestelllt.

[1] Die deutsche Übersetzung „Vergesslichkeit“ ist eher ungeeignet, um zu beschreiben, was mit diesem Begriff gemeint ist.

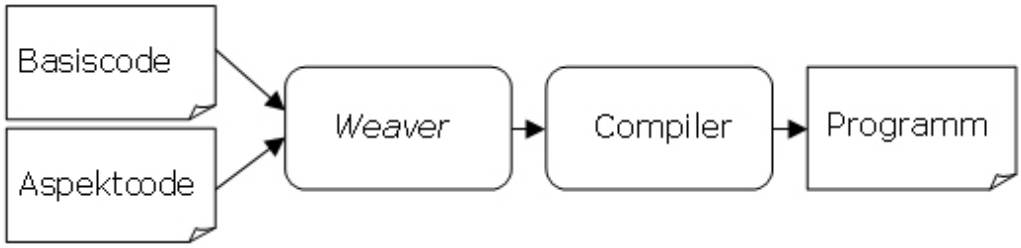

Abbildung 2.1: *Source Code Weaving*

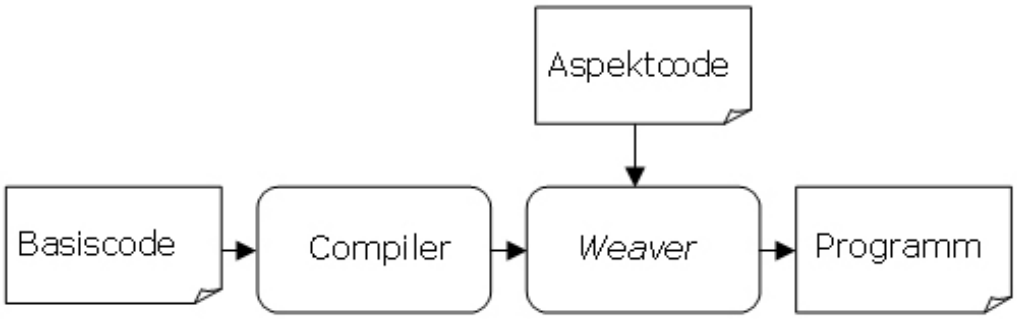

Abbildung 2.2: *Bytecode Weaving*

Häufig kommt es vor, dass eine Applikation sowohl mit als auch ohne eingewobene Aspekte gestartet werden soll. Das ist zum Beispiel der Fall, wenn die Aspekte zum Testen oder Debuggen verwendet werden. Beim statischen Weben ergibt sich generell der Nachteil, dass dafür zwei Versionen des Kompilats verwaltet werden müssen. Im Hinblick auf die Performance hat dieses Vorgehen jedoch den Vorteil, dass nur ein sehr geringer Overhead zur Laufzeit produziert wird.

Loadtime Weaving

Beim *loadtime weaving* wird das Laden der Basisklassen in die *Virtual Machine* (VM)[2] dazu genutzt, den Bytecode entsprechend zu modifizieren. Der tatsächlich ausgeführte Code liegt also nur in der VM, jedoch nicht in Dateiform vor. Dadurch entfällt die Notwendigkeit, zwei Versionen des Bytecodes zu verwalten und auch hier wird der Quellcode des Basisprogramms nicht benötigt. Neben AspectJ benutzt z.B. auch ObjectTeams/Java *loadtime weaving*.

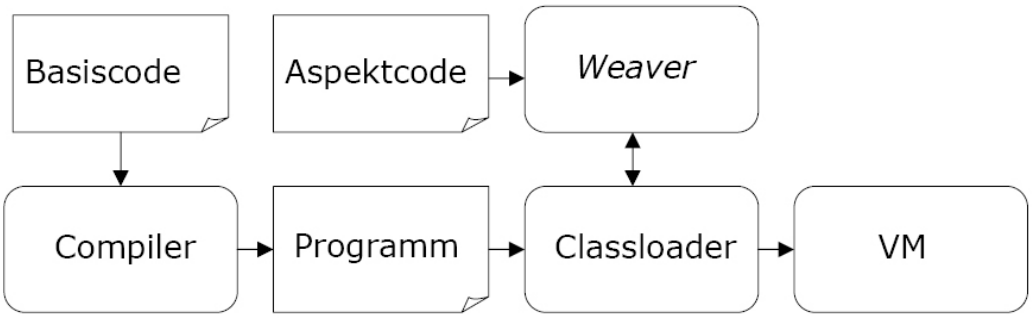

Abbildung 2.3: *Loadtime Weaving*

Die genannten Vorteile müssen durch gewisse Einbußen in der Performance erkauft werden. Das Weben zur Ladezeit wirkt sich negativ auf die Laufzeit des Programms aus. Insbesondere die Zeit für das Starten wird durch den Webevorgang beeinträchtigt.

[2] Die JVM von Java bzw. die CLR von .NET.

Runtime Weaving

Da jede Klasse nur einmal geladen wird, kann sie beim *loadtime weaving* auch nur einmal verändert werden. Beim *runtime weaving* dagegen können Aspekte zur gesamten Laufzeit hinzugefügt, geändert oder entfernt werden. Man muss eine laufende Applikation nicht beenden und neu starten, um sie mit zusätzlichen Aspekten zu versehen. Die Sprache JAsCo arbeitet z.B. mit dieser Methode.

Das dynamische Weben bietet also die größte Flexibilität der Ansätze. Allerdings ist auch hier zu berücksichtigen, dass der Webevorgang die Performance der Applikation negativ beeinflusst, da er während der gesamten Programmlaufzeit ausgeführt werden kann und weil außerdem zusätzlicher Mehraufwand aufzubringen ist, wenn Klassen eines laufenden Systems verändert werden müssen.

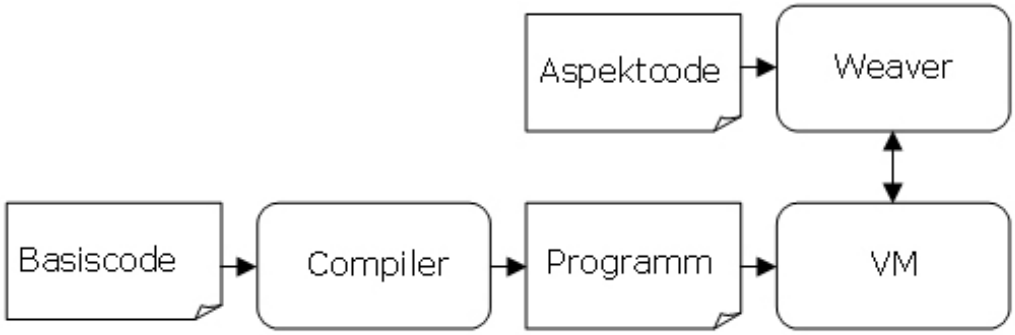

Abbildung 2.4: *Runtime Weaving*

Fazit

Offensichtlich existiert ein gewisser Zielkonflikt zwischen Dynamizität und Flexibilität auf der einen und optimaler Performance auf der anderen Seite. Den Laufzeit-Overhead zu minimieren, der vor allem durch *loadtime* und *runtime weaving* verursacht wird, ist daher eine wichtige Herausforderung an die Forschung auf dem Gebiet der aspektorientierten Programmierung. Dementsprechend existiert eine große Anzahl von Arbeiten [PAG03, HH04, VS04, AE05, BHMM05], die sich mit der Performance der verschiedenen Aspektwebestrategien beschäftigen und auf die an dieser Stelle verwiesen wird.

2.2 ObjectTeams

Die Sprache ObjectTeams/Java wurde (und wird) an der TU Berlin entwickelt. Sie ist eine Erweiterung der Programmiersprache Java und hat ihre Wurzeln in dem Konzept der *Aspectual Components*, das von Lieberherr et al. vorgestellt wurde [LLM99]. Die Grundidee besteht in der Kapselung von Kollaborationen, also von Interaktionen zwischen verschiedenen Objekten.

2.2.1 Teams

Das zentrale Konzept von ObjectTeams ist das Team. Ein Team beinhaltet mehrere Klassen, die als Rollen bezeichnet werden, und kapselt so die Kollaboration mehrerer Rollenobjekte. Es bildet quasi ein für die Sprache namensgebendes „Team von Objekten“.

Teams sind ihrerseits selbst Klassen und können somit beerbt und instanziiert werden. Außerdem besitzen sie Attribute und Methoden zur Koordination der Rollenobjekte untereinander und für die Interaktion mit Klassen außerhalb des Teams (vgl. Entwurfsmuster *Facade* [GHJV95]). Teams werden durch das Schlüsselwort `team` gekennzeichnet.

2.2.2 Rollen

Jede innere Klasse eines Teams ist eine Rolle. Rollen können an Basisklassen gebunden werden, um diese mit zusätzlicher Funktionalität anzureichern bzw. die ursprüngliche zu modifizieren. Diese Bindung geschieht deklarativ mit dem `playedBy`-Schlüsselwort:

```
protected class MyRole playedBy MyClass {...}
```

Die Methoden einer so gebundenen Rolle können an Methoden der Basisklasse gebunden werden. Dabei gibt es zwei Arten von Methodenbindungen: *Callins* und *Callouts*.

Callins

ObjectTeams/Java weist neben den Modularisierungskonzepten *Team* und *Rolle* auch aspektorientierte Konzepte auf. Mit Hilfe von *Callins* ist es

nämlich möglich, die Ausführung einer Basismethode als Joinpoint zu definieren. Das Basisprogramm muss von dieser Bindung keinerlei Kenntnis haben. Die gebundene Rollenmethode kann dann entweder vor, nach oder an Stelle der Methode ausgeführt werden, an die sie gebunden ist. Dies wird mit den Schlüsselwörtern `before`, `after` bzw. `replace` festgelegt. Wenn beispielsweise die Rollenmethode `rm()` nach der Basismethode `bm()` ausgeführt werden soll, wird dies in ObjectTeams/Java folgendermaßen deklariert:

```
void rm() {...}
void rm() <- after void bm();
```

Innerhalb einer Rollenmethode `rm()`, die per *replace callin* an die Basismethode `bm()` gebunden ist, kann mit der Anweisung `base.rm()` die Basismethode aufgerufen werden. Dieser Aufruf wird in ObjectTeams/Java *base call* genannt.

Callouts

Callouts sind Methodenbindungen, die in der entgegengesetzten Richtung funktionieren. Beim Aufruf einer gebundenen Rollenmethode wird dieser an die entsprechende Basismethode weitergeleitet. Ein *Callout* wird auf die folgende Weise deklariert:

```
void rm() -> void bm();
```

Ein Aufruf der Methode `rm()` an einem Rollenobjekt hat also einen Aufruf der Basismethode `bm()` zur Folge.

Aktivierung und Deaktivierung

ObjectTeams/Java bietet die Möglichkeit, Teams beliebig zu aktivieren und zu deaktivieren. Dieses Feature erhöht die Flexibilität, mit der zur Laufzeit von deren Funktionalität Gebrauch gemacht werden kann. So ist es beispielsweise möglich, verschiedene Programmmodi mit Hilfe von Aktivierung bzw. Deaktivierung verschiedener Teams zu realisieren.

Der Effekt der Aktivierung eines Teams ist das Wirksamwerden sämtlicher seiner *Callin*-Methodenbindungen. Mit Hilfe der Methoden `activate()` und `deactivate()` kann jedes Teamobjekt explizit aktiviert bzw. deaktiviert werden. In ObjectTeams/Java wird zudem zwischen thread-lokaler Aktivierung

und globaler Aktivierung unterschieden. Die *Callins* sind dadurch wahlweise nur für einen bestimmten Thread oder für alle aktiven Threads der Applikation wirksam. Im letzteren Fall wird die Aktivierung des Teams mit dem Aufruf `activate(ALL_THREADS)` vorgenommen.

Neben dieser Art der Teamaktivierung kann man in ObjectTeams/Java die Ausführung von *Callins* auch von Prädikaten abhängig machen, die zur Laufzeit ausgewertet werden. Diese so genannten *Guard Predicates* [HHMW05] können auf den folgenden vier Ebenen auftreten:

- Teamklasse
- Rollenklasse
- Rollenmethode
- *Callin*-Methodenbindung

Die *Guard Predicates* haben prinzipiell die Form `when(`*<Boolscher Ausdruck>*`)`. Die *Callins* im Geltungsbereich des *Guards* werden nur dann aktiv, wenn die Auswertung des Boolschen Ausdrucks zur Laufzeit `true` ergibt.

Besitzen mehrere Teams Rollen, die *Callin*-Bindungen mit derselben Basismethode aufweisen, so entscheidet die Reihenfolge der Aktivierung der Teams über die Reihenfolge der Ausführung der einzelnen Rollenmethoden. Dabei gilt, dass das zuletzt aktivierte Team die größte Priorität erhält. Das bedeutet, dass seine *before-* und *replace callins* zuerst und seine *after callins* zuletzt ausgeführt werden.

Mit Sicht auf die Performance ist schließlich wichtig zu bemerken, dass erst zur Laufzeit ermittelt werden kann, in welcher Reihenfolge mehrere solcher „konkurrierenden“ *Callins* ausgeführt werden müssen. Außerdem muss die ObjectTeams-Laufzeitumgebung die aktiven Teams speichern. Ferner speichert jede nicht global aktive Teaminstanz alle Threads, für die sie aktiv ist. Aus der Möglichkeit, Teams zur Laufzeit beliebig zu aktivieren und zu deaktivieren, folgt also zwangsläufig zusätzlicher Laufzeit-Overhead, der ohne dieses Feature nicht anfallen würde.

2.2.3 Webestrategie

Damit *Callins* den Programmfluss der Basisapplikation wie gewünscht verändern können, müssen die entsprechenden Anweisungen, die dieses

Verhalten realisieren, in das Programm eingewoben werden. Auch für im Programm vorkommende *base calls* ist eine zusätzliche Modifikation des Programms notwendig.

Wie bereits in Abschnitt 2.1.3 erwähnt, benutzt ObjectTeams *Loadtime Weaving* als Webestrategie. Hierfür wird das JMangler-Framework [KCA01, JM] verwendet, das die Transformation des Bytecodes zum Zeitpunkt des Ladens der Klassen in die JVM ermöglicht. Derzeitige Bemühungen sind allerdings auf das Ziel gerichtet, das flexiblere *Runtime Weaving* zu unterstützen [Flü06]. Zu diesem Zweck wird JMangler durch die *Java Programming Language Instrumentation Services* (JPLIS) ersetzt, die Bestandteil der Java SE 5 API sind und das Aspektweben während der gesamten Laufzeit erlauben.

Sowohl JMangler als auch die Laufzeitumgebung von ObjectTeams/Java benutzen außerdem das Framework BCEL [BCE]. BCEL steht für *Byte Code Engineering Library*. Es bietet eine abstraktere Sicht auf Java-Bytecode und ermöglicht so das Erzeugen und Manipulieren desselben. Dieses Framework wird beispielsweise auch von AspectJ genutzt.

Die beiden Ansätze, mit JMangler bzw. mit JPLIS zu arbeiten, haben gemeinsam, dass der Webevorgang erst nach dem Programmstart, also zur Laufzeit stattfindet und diese somit negativ beeinflusst. In Kapitel 4 soll u.a. näher untersucht werden, wie genau diese Beeinflussung zu quantifizieren ist. Die Minimierung des Overheads, der durch das Weben verursacht wird, ist Gegenstand von Kapitel 5.

Kapitel 3

Performance-Messverfahren

ObjectTeams reichert wie in Kapitel 2 beschrieben die Programmiersprache Java mit zusätzlichen, aspektorientierten Konzepten an. Das damit verbundene Aspektweben führt zu zwei Arten von Overhead. Zum einen beansprucht die Ausführung des Webevorgangs selbst eine gewisse Zeit. Zum anderen muss von ObjectTeams zusätzlicher Code generiert werden, der die Integration von Basisprogramm und Aspekten ermöglicht. Dieser Code gehört also weder zur Basisapplikation, noch ist er dem Aspektcode zuzurechnen. Seine Ausführung stellt daher ebenfalls einen Overhead dar.

Dieses Kapitel der Arbeit untersucht beide Formen des Overheads dahingehend, wie stark sie die Performance von ObjectTeams/Java-Programmen im Vergleich zu reinen Java-Applikationen beeinflussen. Die Mittel hierfür waren *Micro-Benchmarks*, *Macro-Benchmarks* sowie *Profiling* und werden folgend näher besprochen.

3.1 Micro-Benchmarks

Micro-Benchmarking verfolgt das Ziel, die Performance relativ kleiner Programmstücke zu messen und sie gegebenenfalls mit der alternativer Implementierungen zu vergleichen. Dabei ist es wichtig, die Aussagekraft der Benchmarks richtig einzuschätzen, da diese immer einen bestimmten Teil des Programms aus dem normalen Anwendungskontext herauslösen und isoliert betrachten. Es ist daher entscheidend, die Micro-Benchmarks in einer Umgebung und mit Eingabedaten auszuführen, die der realen Anwendungssituation möglichst gut entsprechen.

Im Zusammenhang mit aspektorientierten Programmiersprachen können

Micro-Benchmarks einen Beitrag dazu leisten, den Overhead zu quantifizieren, der durch die Benutzung von Aspekten verursacht wird. Zu diesem Zweck haben Haupt und Mezini einen Katalog von Micro-Benchmarks für dynamische AOP-Systeme vorgeschlagen [HM04]. Darunter werden solche aspektorientierten Systeme verstanden, die das Aspektweben zur Laufzeit unterstützen, also mit *Loadtime Weaving* oder mit *Runtime Weaving* arbeiten. Der Katalog umfasst die folgenden drei Arten von Messungen:
Messungen

1. des Webevorgangs, bzw. dessen Umkehr,

2. der Ausführung eines *Joinpoint Shadows*[1],

3. der Verarbeitung von Joinpoint-Kontext-Information.

Die Ausführung des *Joinpoint Shadows* (2.) sollte [HM04] zufolge sowohl ohne als auch mit einem Aspekt erfolgen, der an den entsprechenden Joinpoint gebunden ist. Im letzteren Fall soll jede Art von Bindung berücksichtigt werden. Bei der Sprache ObjectTeams/Java wären dies also *before*, *after-* und *replace-Callins*. Der 3. Punkt umfasst beispielsweise die Bindung von Parametern von Basismethoden an den Aspekt. In ObjectTeams/Java wird dies als *Parameter Mapping* bezeichnet.

Die ersten beiden Punkte des in [HM04] vorgeschlagenen Katalogs von Messungen bildeten die Grundlage für die im Rahmen dieser Arbeit erstellten Micro-Benchmarks zum Benchmarking von ObjectTeams/Java. Der dritte Punkt war nicht im Fokus dieser Arbeit. Die konkreten Benchmarks und deren Ergebnisse werden in Abschnitt 4.2 besprochen.

3.2 Macro-Benchmarks

Während Micro-Benchmarks Messungen an kleinen Programmabschnitten vornehmen, wird bei einem Macro-Benchmark die Performance einer gesamten Applikation betrachtet. Macro-Benchmarks können folglich weniger Aufschluss über die Ausführungsgeschwindigkeit einzelner Programmteile geben, sondern vielmehr ein System als Ganzes beurteilen

[1] Die tatsächliche Stelle im Programmcode, die potentiell zur Laufzeit ein Joinpoint wird, z.B. ein Methodenaufruf [MKD02].

helfen. Daraus ergibt sich eine höhere Aussagekraft der Macro-Benchmarks über das Verhalten des untersuchten Programms im realen Anwendungskontext. Das im Rahmen dieser Arbeit durchgeführte Macro-Benchmarking diente folglich in erster Linie dazu, den Umfang des Object-Teams-spezifischen Overheads in einer realen Anwendung zu ermitteln. In Abschnitt 3.4.2 wird das Vorgehen beim Macro-Benchmarking genauer beschrieben.

3.3 Profiling

Wie gezeigt wurde, kann mit Hilfe des Micro-Benchmarkings die Laufzeit einzelner konkreter Programmabschnitte gemessen werden. Auf diese Weise ließen sich die verschiedenen Arten des ObjectTeams-spezifischen Overheads quantitativ bestimmen. Dahingegen liefert das Profiling diverse Daten, die über die gesamte Programmlaufzeit gesammelt werden. Im folgenden Abschnitt werden zunächst die Grundlagen des Profilings erläutert und anschließend seine Rolle bei der Optimierung der Performance von ObjectTeams/Java beschrieben.

3.3.1 Einführung

Einer der wichtigsten Grundsätze der Performance-Optimierung ist die so genannte 80/20-Regel. Diese besagt, dass eine Applikation erfahrungsgemäß in etwa 80% der Programmlaufzeit nur ca. 20% des Codes ausführt. Daraus folgt, dass sich eine Optimierung auf jene 20% des Programmcodes konzentrieren muss, da ein Tuning des übrigen Codes nahezu keinen Einfluss auf die Performance hätte.

Profiling-Tools können unter anderem dafür benutzt werden, die Stellen eines Programms zu ermitteln, die am meisten einer Optimierung bedürfen. Sie identifizieren die so genannten *Performance Hotspots*, also diejenigen Methoden, die während eines Programmablaufs die meiste Zeit verbrauchen. Dabei werden inklusive von exklusiven Zeiten unterschieden, je nachdem ob die Ausführungszeit einer Methode auch der aufrufenden Methode zugerechnet wird oder nicht. Ist beispielsweise die inklusive Zeit einer Methode relativ lang, die exklusive jedoch eher kurz, so ist dies ein Zeichen dafür, dass die Methode hauptsächlich andere Methoden aufruft, selbst jedoch keine zeitintensiven Operationen ausführt. Typischerweise

ist die main-Methode einer Applikation ein Beispiel für eine solche Situation.

3.3.2 Techniken

Für die Messung der Zeit, die ein Programm in seinen einzelnen Methoden verbraucht, unterstützen Profiling-Tools prinzipiell zwei Messtechniken, welche jeweils unterschiedliche Vor- und Nachteile mit sich bringen.

Instrumentierung Die eine Möglichkeit besteht darin, mittels einer Bytecode-Instrumentierung vollständige Daten über die Ausführungszeiten zu sammeln. Die Vollständigkeit ist gleichzeitig der Vorteil dieser Methode, da während des Programmablaufs alle Methodenaufrufe sowie deren Beendigung bei der Messung berücksichtigt werden. Nachteilig ist jedoch, dass die Instrumentierung die Performance des ausgeführten Programms stark beeinflussen kann, so dass die Ergebnisse durch das Profiling verfälscht werden und Skalierungsprobleme auftreten können.

Stichprobenmethode Der andere Ansatz vermeidet eine Beeinflussung der Performance während der Messung weitgehend dadurch, dass nicht der komplette Programmablauf überwacht wird. Stattdessen analysiert der Profiler in regelmäßigen Abständen (beispielsweise alle 10 Millisekunden) den aktuellen Stack und errechnet die Ausführungszeiten der Methoden auf Basis dieser Stichproben. Dies führt unvermeidbar zu einer geringeren Präzision der Messergebnisse. Kleinere Methoden können ganz „unter den Tisch fallen“, wenn sie genau zwischen zwei Stichproben ablaufen. Programme mit einer relativ kurzen Ausführungszeit können insgesamt nur ungenau analysiert werden.

Die Stichprobenmethode sollte daher vor allem bei langen Programmlaufzeiten eingesetzt werden, während bei eher kurzen Programmen die Instrumentierung vorzuziehen ist. Da die ObjectTeams/Java-Programme, die im Rahmen dieser Arbeit einem Profiling unterzogen wurden, verhältnismäßig klein sind, wurde in allen Fällen mit einer Instrumentierung gearbeitet.

3.3.3 HPROF und HPjmeter

HPROF

Für das Profiling wurde das Tool HPROF [O'H04] benutzt, welches als Teil von Java SE 5 ausgeliefert wird. Der Name HPROF steht für *Heap Profiler*. Das Programm arbeitet auf Basis des JVMTI[2] und ist frei erhältlich. Im Gegensatz zu manchen anderen Tools eignet es sich ohne Weiteres zum Profiling von ObjectTeams-Programmen, da es einfach mit Hilfe eines Parameters der Java Virtual Machine gestartet wird. Andere Tools wie z.B. JProfiler [JPRb] oder JProbe [JPRa] versuchen meist, selbst festzustellen, wann die JVM gestartet wird, um dann per JVMPI[3] bzw. JVMTI mit der JVM zu kommunizieren und an die Profiling-Informationen zu gelangen. Da der Start von ObjectTeams/Java-Programmen anders erfolgt als es bei reinen Java-Programmen der Fall ist, hat sich die Benutzung anderer Tools als problematisch erwiesen.

HPROF bietet die wichtigsten Funktionen zur Analyse von Zeit- und Speicherverbrauch eines Programms. So lässt sich für die Messung der Zeiten, die die Methoden verbrauchen, zwischen den oben erläuterten Alternativen Instrumentierung und Stichproben wählen, wobei im letzteren Fall das Zeitintervall einstellbar ist. Standardmäßig ist dieser auf 10ms eingestellt.

HPjmeter

Die Ausgabe der Daten, die HPROF bei einem Profiling sammelt, erfolgt in einer Textdatei. Diese kann schnell eine erhebliche Größe erreichen und ist für einen Menschen nur schwer lesbar. Für die Aufbereitung der Profiling-Daten wurde HPjmeter 2.0 von Hewlett Packard [HPJ] verwendet, das ebenfalls frei erhältlich ist. Dieses Tool wertet die Informationen aus der von HPROF erstellten Textdatei aus und stellt diese graphisch dar. So lassen sich z.B. leicht diejenigen Methoden identifizieren, die die meiste Zeit verbraucht haben, und zwar sowohl inklusive als auch exklusive. Durch Anzeigen des *Call Graph Tree* (s. Abb 3.1) kann darüber hinaus abgelesen werden, von wo aus diese Methoden aufgerufen wurden. Diese Information ist besonders interessant für Methoden aus Klassen, die zur Java API gehören. Da diese Methoden selbst nicht zu optimieren sind,

[2] Java Virtual Machine Tool Interface [JVMb].

[3] Java Virtual Machine Profiler Interface [JVMa], mit der Java-Version 1.5 durch das JVMPI abgelöst und nunmehr als *deprecated* eingestuft.

kann eine Optimierung nur durch eine Reduzierung der Zahl ihrer Aufrufe erfolgen.

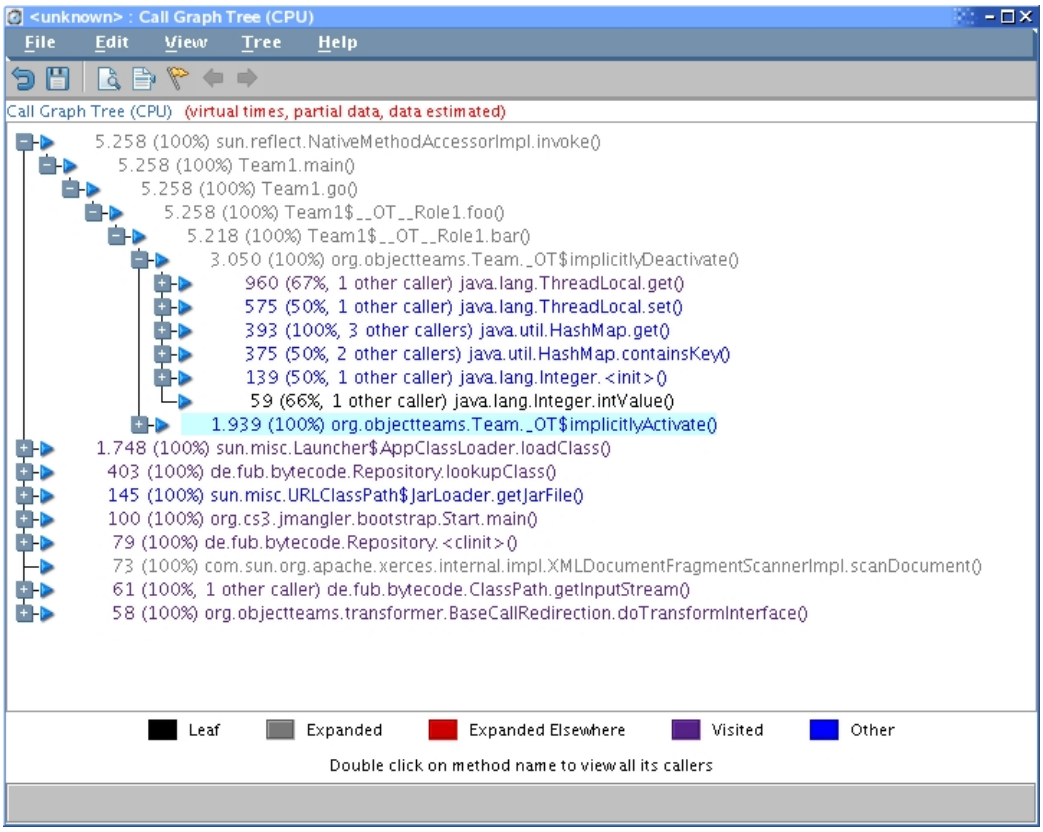

Abbildung 3.1: *Call Graph Tree* in HPjmeter 2.0

3.4 Messtechniken

3.4.1 Hard- und Software

Grundlage des Benchmarkings war jene Version von ObjectTeams/Java, die als Teil der Entwicklungsumgebung OTDT 0.9.10 verfügbar ist [OTD]. Sämtliche Messungen wurden auf einem Rechner mit Intel Pentium 4 Prozessor, 2,4 GHz und 1 GB RAM unter Debian-Linux mit der Kernelversion 2.4.22 ausgeführt. Alle Programme liefen unter der *Java Runtime Environment* (JRE) 5.0 Update 7.

3.4.2 Vorgehen

Rauschen

Um die Ergebnisse nicht zu verfälschen, wurde bei allen Benchmarks darauf geachtet, dass während der Messungen möglichst wenige andere Programme im Hintergrund liefen, insbesondere jedoch keine, die in erheblichem Umfang CPU-Zeit erfordert hätten. Zudem wurde non jederlei Benutzeraktionen wie Mausklicks oder Tastendrücken während der Messungen abgesehen. Trotzdem ist Rauschen eines gewissen Ausmaßes nicht völlig vermeidbar. Quellen dieses Rauschens können u.a. das Betriebssystem oder die Desktop-Umgebung, aber auch die JVM sein. Bei der JVM führen die *Garbage Collection* oder auch andere Umstände [GVG04] zu verrauschten Messdaten. Aus diesen Gründen wurden bei der Durchführung der Benchmarks immer jeweils mehrere Messungen durchgeführt und anschließend der Mittelwert der einzelnen Ergebnisse berechnet. Auf diese Weise sollte der Effekt von Rauschen bei der Messdatenaufnahme minimiert werden. „Ausreißer“ wurden dabei nicht festgestellt[4]. Zu den mathematischen Details dieser Vorgehensweise sei hier auf [Loe00] verwiesen. Dort wird gezeigt, dass der Effekt des Rauschens bei einer Messdatenaufnahme reduziert werden kann, indem der Mittelwert mehrerer Werte berechnet wird.

Messung

Folgend werden kurz die Techniken erläutert, mit deren Hilfe die einzelnen Messungen vorgenommen wurden.

Micro-Benchmarks Wie oben bereits erläutert, messen Micro-Benchmarks die Ausführungszeit relativ kleiner Programmstücke. Prinzipiell erfolgt diese Messung, indem jeweils vor und nach dem Programmabschnitt von Interesse die Systemzeit ermittelt wird. Dies kann in Java mit der Methode `System.nanoTime()` geschehen. Durch Subtraktion ergibt sich schließlich die verbrauchte Zeit (siehe auch Listing 3.1).

Oft ist der zu betrachtende Programmabschnitt so kurz, dass die Zeit einer einmaligen Ausführung kaum messbar wäre. In diesem Fall wird der Code in einer Schleife mit einer großen Zahl von Wiederholungen ausgeführt.

[4] Als Ausreißer wird im Allgemeinen ein Messwert verstanden, der um mehr als das Dreifache der Standardabweichung vom Mittelwert abweicht.

Listing 3.1: Prinzipielles Vorgehen beim Micro-Benchmarking

```
...
long tBefore = System.nanoTime();
// zu messender Code-Abschnitt
long elapsed = (System.nanoTime() - tBefore) / 1000000L;
System.out.println("elapsed time: " + elapsed + "ms");
...
```

Die Maßgabe, Micro-Benchmarks mit realistischen Eingabedaten auszuführen, kann insofern als eingehalten angesehen werden, als dass die „Eingabe“ hier aus Programmkonstrukten besteht, die in derselben Form auch in realen Anwendungen auftreten (Methodenaufrufe, *Callins, Callouts*, etc.).

Webezeit Hinsichtlich der Webezeit können zwei Arten von einander unterschieden werden: Brutto- und Netto-Webezeit. Um den Laufzeit-Overhead zu messen, der zur Ladezeit durch den Webevorgang entsteht, kamen dementsprechend zwei verschiedene Verfahren zum Einsatz. Im Folgenden werden beide Arten der Webezeit definiert und die Methode ihrer jeweiligen Messund vorgestellt.

Brutto-Webezeit Für die Messung der Brutto-Webezeit wurde ein indirekter Ansatz gewählt. Dabei wurden jeweils zwei relativ kleine Beispielprogramme erstellt, ein ObjectTeams/Java-Programm und ein äquivalentes reines Java-Programm. Äquivalent bedeutet in diesem Fall, dass beide Programme aus derselben Anzahl von Klassen bestehen sowie dieselbe Anzahl von Konstruktor- und Methodenaufrufen ausführten etc. Da die Methoden meist leer waren oder nur aus Methodenaufrufen bestanden, können diese Programme als äquivalent angesehen werden. Sie unterschieden sich also im Wesentlichen nur in dem Punkt, dass jeweils das eine Programm dem Webevorgang von ObjectTeams/Java unterzogen werden musste. Mit Hilfe des Unix-Befehls `time` konnte die komplette Ausführungszeit beider Programme gemessen werden. Dieser Ansatz misst also nicht nur die inklusive Ausführungszeit der jeweiligen `main`-Methode. Es werden darüber hinaus z.B. auch die Zeiten gemessen, die das Starten und Beenden der JVM sowie das Laden der einzelnen Klassen benötigen. Die jeweilige Differenz der Ausführungszeiten beider Programme ergibt ein gutes Maß für den Overhead, der durch das Aspektweben von ObjectTeams zur Ladezeit verursacht wird. Die so gemessene Zeit kann auch als Brutto-Webezeit bezeichnet werden, da sie neben der Zeit, welche die ei-

gentliche Bytecode-Transformation benötigt, beispielsweise auch das Einklinken in den Klassenladevorgang durch das JMangler-Framework oder das Laden der zusätzlich notwendigen Klassen in die JVM beinhaltet. Auf die beschriebene Weise lassen sich jedoch nur relativ kleine Programme mit vertretbarem Aufwand messen, da für dieses Verfahren immer jeweils auch eine äquivalente reine Java-Version des untersuchten Programms benötigt wird. Der Overhead, der zur Laufzeit durch ObjectTeams/Java-Konstrukte wie *Callins* und *Callouts* verursacht wird, ist um mehrere Größenordnungen geringer als die gemessene Webezeit der Benchmarks und kann daher in diesem Zusammenhang vernachlässigt werden.

Netto-Webezeit Das andere Verfahren zur Webezeitmessung bestand darin, die ObjectTeams/Java-Laufzeitumgebung so zu instrumentieren, dass die Ausführungszeiten sämtlicher Methoden, die für die Bytecode-Transformationen zuständig sind, zu messen und aufzuaddieren. Anders als beim ersten Ansatz wird so also nur die reine Zeit der Transformation gemessen. Die Zeit für das Laden von Klassen, die lediglich für den Transformationsvorgang benötigt werden, bleibt so z.B. unberücksichtigt. Die so gemessene Webezeit heißt entsprechend auch Netto-Webezeit, da sie allein die Dauer des eigentlichen Webevorgangs betrachtet. Der Vorteil dieses Vorgehens besteht darin, dass nur eine ObjectTeams/Java-Version des untersuchten Programms benötigt wird und deshalb auch beliebig große Anwendungen mit relativ geringem Aufwand untersucht werden können.

Macro-Benchmarks Die Messtechnik bei Macro-Benchmarks besteht im Wesentlichen darin, die Laufzeit der `main`-Methode des untersuchten Programms zu messen. Erfordert das Programm Benutzereingaben, ist es notwendig, diese zu automatisieren, um immer gleich schnelle Eingaben zu ermöglichen. Ein menschlicher Benutzer würde sonst die Ergebnisse verfälschen. Im Falle von Anwendungen mit graphischer Benutzerschnittstelle (GUI) bietet sich hierfür die Benutzung eines so genannten „GUI-Roboters“ an, der automatisch Benutzeraktionen wie z.B. Mausklicks erzeugen kann. Von dieser Methode wurde auch bei dem vergleichenden Macro-Benchmark Gebrauch gemacht, der in Abschnitt 6.3 beschrieben wird. Für die Messung kann analog zum Micro-Benchmarking eine Methode wie `System.nanoTime()` genutzt werden.

Kapitel 4

Performance-Benchmarking

4.1 Übersicht

Bevor in den Abschnitten 4.2 bis 4.5 erläutert wird, wie die in Kapitel 3 vorgestellten Messtechniken auf ObjectTeams/Java angewendet wurden, soll Abb. 4.1 zunächst die Zusammensetzung des ObjectTeams-spezifischen Overheads verdeutlichen. Die Abbildung zeigt die drei Arten von Overhead in der zeitlichen Abfolge. Durch Fettdruck hervorgehoben sind außerdem die Abschnitte, in denen die Messungen für die jeweilige Phase behandelt werden.

4.2 Micro-Benchmarking des Laufzeit-Overheads

Als ein zentrales Ziel dieser Arbeit sollte die aktuelle Webestrategie von ObjectTeams/Java anhand ihrer Performance bewertet werden. Zu diesem Zweck wurde die Ausführungszeit des gewobenen Codes gemessen. In den folgenden Abschnitten werden die einzelnen Micro-Benchmarks vorgestellt, die dafür erstellt wurden. Im Einzelnen sind dies Benchmarks für *Callins*, *Callouts*, Teamaktivierung sowie den Webevorgang.

4.2.1 *Callins*

Callins sind Methodenbindungen zwischen Basis- und Rollenmethoden. Der Aufruf einer Basismethode wird dabei durch den zusätzlichen Aufruf einer Rollenmethode ergänzt.

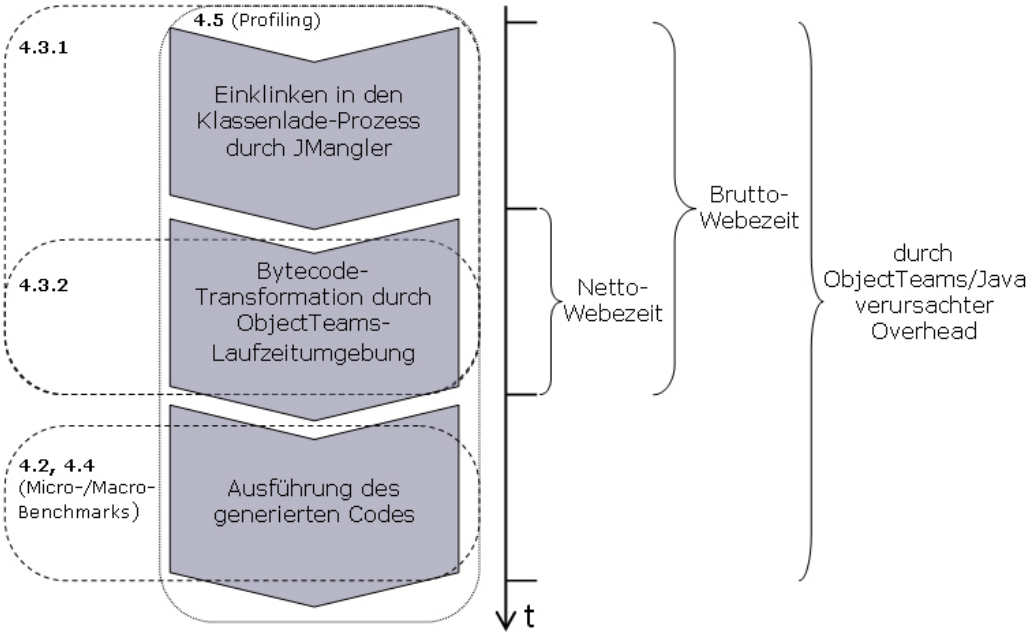

Abbildung 4.1: Arten von Overhead in ObjectTeams/Java

Callin-Benchmark 1

Der erste Micro-Benchmark vergleicht die Zeit für einen herkömmlichen Methodenaufruf mit der, die benötigt wird, wenn die aufgerufene Methode mit einem *after-Callin* an eine Rollenmethode gebunden ist. Listing 4.1 zeigt ein reines Java-Programm, welches die Zeit misst, die für 10 Millionen Aufrufe einer leeren Methoden benötigt wird. Die Methode ist deshalb leer, weil das Interesse der Ausführungszeit der Methoden*aufrufe* und der *Callins* gilt. Anweisungen innerhalb der Methode sind für diesen Zweck unnötig.

Der Benchmark wurde sowohl gänzlich ohne ein Team als auch mit einer aktiven Instanz des Teams ausgeführt, das in Listing 4.2 zu sehen ist. Dieses besitzt eine Rollenklasse, die obige Basisklasse adaptiert und die Methode `nop()` mit Hilfe eines *after-Callins* an die Methode `m()` bindet. Nach `m()` wird also zusätzlich `nop()` ausgeführt. Der Zeitunterschied zwischen beiden Versionen gibt Aufschluss über den Laufzeit-Overhead, der durch ein *after-* bzw. *before-Callin* verursacht wird[1].

[1] Hinsichtlich der Performance gibt es wegen der nahezu identischen Implementierung keinen Unterschied zwischen *before-* und *after-Callins*.

Listing 4.1: Callin-Benchmarks: Java-Klasse

```
public class BaseClass {
   public static void main(String[] args) {
      final int nCalls = 10000000;
      BaseClass b = new BaseClass();
      long tBefore = System.nanoTime();
      for(int i = 0; i < nCalls; i++) {
         b.m();
      }
      long elapsed = (System.nanoTime()-tBefore) / 1000000L;
      System.out.println("elapsed time: " + elapsed + "ms");
   }
   void m() {}
}
```

Listing 4.2: Callin-Benchmark 1: Team-Klasse

```
public team class Team1 {
   public class Role1 playedBy BaseClass {
      nop <- after m;
      void nop() {}
   }
}
```

Ohne das Team betrug die gemessene Zeit durchschnittlich 26,2 ms, mit aktiviertem Team wurden 7792,7 ms gemessen. Die Ausführung der *Callins* verlängert demnach die Laufzeit des Programms etwa um den Faktor 300. Der Webevorgang ist nicht Teil des Overheads, da er bereits zur Ladezeit der Klassen und damit vor dem Beginn der Zeitmessung vorgenommen wird.

Abbildung 4.2 zeigt den Ablauf bei der Ausführung eines *after-Callins* in einem vereinfachten UML-Sequenzdiagramm. Der Overhead kommt folgendermaßen zustande. Erst nach dem Ermitteln der für den aktuellen Thread aktiven Teams (in diesem Fall eins) wird die ursprüngliche Basismethode (`bm_orig()`) aufgerufen. Anschließend muss dem Basisobjekt das dazugehörige Rollenobjekt zugeordnet werden. Dieser Vorgang wird in Object-Teams/Java als *Lifting* bezeichnet. Ist für ein Basisobjekt noch kein Rollenobjekt bekannt, so muss dies zunächst instanziiert werden. Bei diesem Benchmark geschieht dies beim ersten Schleifendurchlauf. Danach kann dann schließlich auf diesem Objekt die Rollenmethode aufgerufen wer-

den, die an die Basismethode gebunden wurde (`rm()`). Am Anfang und am Ende der Rollenmethode wird außerdem die so genannte implizite Aktivierung bzw. Deaktivierung des Teams `Team1` vorgenommen. Nach § 5.3 der ObjectTeams/Java Sprachdefinition [OTJ] findet diese Art der Aktivierung immer dann statt, wenn der Kontrollfluss von außen an ein Team oder an eine seiner Rollen übergeht. In diesem Zusammenhang stellte sich heraus, dass die Laufzeitumgebung die implizite Aktivierung nicht nur beim Aufruf von `public`-Methoden veranlasst, wie es laut Sprachdefinition nötig wäre, sondern darüber hinaus bei Methoden mit default oder `protected` Sichtbarkeit. D.h, in jede Rollenmethode, die nicht als `private` deklariert ist, generiert die Laufzeitumgebung die entsprechenden Aufrufe hinein. Da die implizite Teamaktivierung einen signifikanten Anteil am Overhead ausmacht, kann diese Erkenntnis als ein wichtiges Ergebnis dieser Arbeit angesehen werden.

Zum Vergleich wurde der Benchmark auch in einer Abwandlung ausgeführt, in der die Methode `nop() private` war. Dort ergab die Messung 4976,2 ms. Die Differenz von rund 2800 ms ist also die Zeit, die das implizite Aktivieren und Deaktivieren bei der ersten Messung benötigte.

Callin-Benchmark 2

Ein weiterer Benchmark untersucht *replace-Callins*. Dieselbe Basisklasse wie in Listing 4.1 wurde nun durch folgendes Team ergänzt:

Listing 4.3: Callin-Benchmark 2: Team-Klasse

```
public team class Team2 {
   public class Role1 playedBy BaseClass {
      nop <- replace m;
      callin void nop() {
         base.nop();
      }
   }
}
```

Die *callin*-Methode enthält lediglich einen *Basecall*, also einen Aufruf der Basismethode. Bei diesem Benchmark ergab die Messung eine Durchschnittszeit von 8450,5 ms, und damit eine knapp 10% längere Laufzeit als bei Benchmark 1. Hier kommen im Vergleich zum dortigen *after-Callin* zusätzlich mehrere Methodenaufrufe hinzu, um den *BaseCall* zu realisieren.

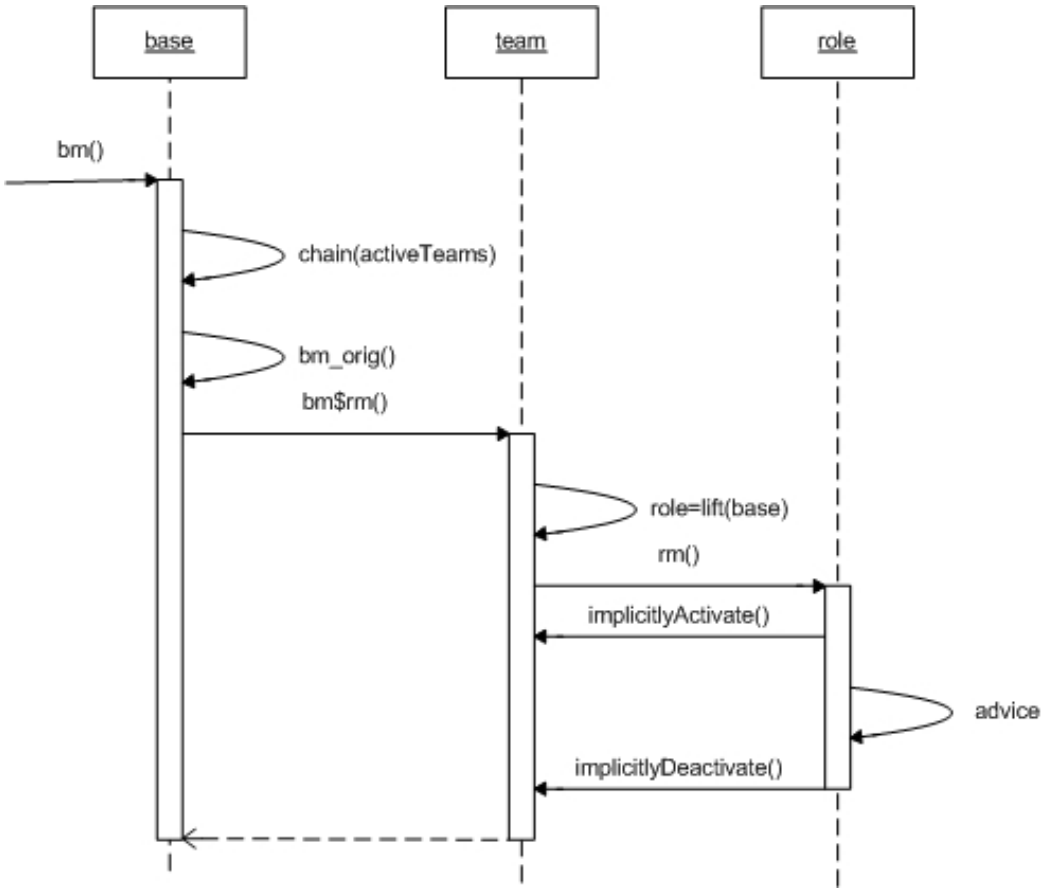

Abbildung 4.2: Ausführung eines after-Callins als UML-Sequenzdiagramm (vereinfacht)

Callin-Benchmark 3

Die letzten beiden Callin-Benchmarks untersuchen, wie es sich auf die Laufzeit auswirkt, wenn mehrere *Callins* auf dieselbe Basismethode wirken. Callin-Benchmark 3 gleicht Benchmark 1, hat jedoch zehn identische Teams statt eines einzelnen. Seine Laufzeit betrug 75345,5 ms. Das ist knapp das Zehnfache im Vergleich zu Benchmark 1, in dem nur ein Team aktiv war. Weitere Messungen mit unterschiedlicher Anzahl von aktiven Teams deuten darauf hin, dass der Overhead erwartungsgemäß linear mit der Anzahl der Teams ansteigt.

Callin-Benchmark 4

Beim 4. Callin-Benchmark kommen ebenfalls bei jedem Aufruf der Basismethode zehn Callins zur Ausführung. Hier sind diese jedoch in einem einzelnen Team mit nur einer Rolle realisiert. Die `precedence`-Anweisung gibt dabei die Reihenfolge vor, in der die *Callins* ausgeführt werden sollen.

Listing 4.4: Callin-Benchmark 4: Team-Klasse

```
public team class Team4 {
   public class Role1 playedBy BaseClass {
      callin1: nop <- after m;
      callin2: nop2 <- after m;
      // Callins 3-10 analog

      void nop() {}
      void nop2() {}
      // Methoden nop3()-nop10() analog

      precedence callin1, callin2, callin3, callin4,
       callin5, callin6, callin7, callin8, callin9,
       callin10;
   }
}
```

Für diesen Benchmark ergaben die Messungen eine durchschnittliche Laufzeit von 65032,8 ms. Das ist 14% schneller als die Version mit 10 Teams. Der so genannte *Chaining-Wrapper*, also die generierte Methode, die zunächst die Originalversion der Basismethode und anschließend die entsprechende Teammethode aufruft (s.a. Abb 4.2), muss für jedes Team erneut aufgerufen werden, was die längere Laufzeit von Callin-Bechmark 3 gegenüber Benchmark 4 erklärt.

4.2.2 *Callouts*

Callouts sind Methodenbindungen, bei denen der Aufruf einer Rollenmethode an die gebundene Basismethode weitergeleitet wird. Analog zu den Callin-Benchmarks 1 und 2 stellen die Callout-Benchmarks die Ausführungszeiten herkömmlicher Methodenaufrufe und der von *Callouts* gegenüber. Während die Basis-Klasse bei beiden Messungen dieselbe war (s. Listing 4.5), wurde einmal mit der reinen Java-Klasse aus Listing 4.6 und einmal mit dem Team gemessen, das in Listing 4.7 zu sehen ist.

Listing 4.5: Callout-Benchmarks: Basis-Klasse

```
public class BaseClass {
   public void bm() {}
}
```

Listing 4.6: Callout-Benchmark 1: Java-Klasse

```
public class C1 {
   void foo() {
      final int nCalls = 10000000;
      BaseClass b = new BaseClass();
      long tBefore = System.nanoTime();
      for (int i = 0; i < nCalls; i++) {
         b.bm();
      }
      long elapsed = (System.nanoTime()-tBefore) / 1000000L;
      System.out.println("elapsed time: " + elapsed + "ms");
   }
   public static void main(String[] args) {
      new C1().foo();
   }
}
```

Die Laufzeit der reinen Java-Klasse, also die Zeit für 10 Millionen herkömmliche Aufrufe einer Methode, betrug im Schnitt 27ms. Die gemessene Zeit für das Team, das ebenso viele *Callouts* ausführte, war 3876,1 ms. Der Unterschied kommt im Wesentlichen durch die implizite Aktivierung bzw. Deaktivierung des Teams zustande. Diese wird laut § 5.3 der ObjectTeams/Java Sprachdefinition auch dann vorgenommen, wenn eine Rolle mittels *Callout* einen Methodenaufruf an ihre Basis weiterleitet. Dadurch wird sichergestellt, dass die *Callin*-Bindungen der Rolle während eines *Callouts* aktiv sind.

4.2.3 Team-Aktivierung und -Deaktivierung

Neben den Messungen des Overheads durch *Callins* und *Callouts* wurden auch Benchmarks für die Aktivierung von Teams erstellt. Dabei wurde sowohl die thread-lokale als auch die globale Aktivierung betrachtet. Auch die Anzahl der an das zu aktivierende Team gebundenen Basisklassen wurde variiert. Die Klasse in Listing 4.8 bildete dabei die Grundlage aller Aktivierungs-Benchmarks.

Listing 4.7: Callout-Benchmark 2: Team-Klasse

```
public team class Team1 {
   public class Role1 playedBy BaseClass {
      void foo() {
         final int nCalls = 10000000;
         long tBefore = System.nanoTime();
         for(int i = 0; i < nCalls; i++) {
            bar();
         }
         long elapsed=(System.nanoTime()-tBefore)/1000000L;
      }
      void bar() -> void bm();
   }
   void go() {
      new Role1(new BaseClass()).foo();
   }
   public static void main(String[] args) {
      new Team1().go();
   }
}
```

Listing 4.8: Aktivierungs-Benchmarks: Main-Klasse

```
public class Main {
   public static void main(String[] args) {
      Team1 t = new Team1();
      final int nCalls = 10000000;
      long tBefore = System.nanoTime();
      for (int i = 0; i < nCalls; i++) {
         t.activate();
         t.deactivate();
      }
      long elapsed = (System.nanoTime()-tBefore) / 1000000L;
      System.out.println("elapsed time: " + elapsed + "ms");
   }
}
```

Die Benchmarks messen also die Zeit, die benötigt wird, um ein Team 10 Millionen mal zu aktivieren und anschließend zu deaktivieren.

Aktivierungs-Benchmark 1

Beim ersten Aktivierungs-Benchmark war `Team1` ein leeres Team ohne Rollen oder Methoden. Die Messung ergab hierfür eine durchschnittliche Laufzeit von 3203,6 ms. Derselbe Benchmark mit globaler statt thread-lokaler Team-Aktivierung benötigte 17325,0 ms, also mehr als das Fünffache. Dieser signifikante Unterschied steht in einem Missverhältnis zu der Komplexität der globalen Aktivierung, die sich nicht wesentlich von der der thread-lokalen Aktivierung unterscheidet. Der Grund für den großen Mehraufwand der globalen Aktivierung bei diesem Benchmark konnte daher nicht endgültig ermittelt werden.

Aktivierungs-Benchmark 2

In Aktivierungsbenchmark 2 besitzt das Team eine Rolle, die eine *played-By*-Beziehung zu einer Basisklasse hat. Er benötigte 5954,3 ms für seine Ausführung. Das sind etwa 85% mehr im Vergleich zum leeren Team von Benchmark 1. Auch hier benötigt die globale Aktivierung mit 16484,0 ms deutlich länger als die thread-lokale Aktivierung.

Aktivierungs-Benchmark 3

Der dritte Aktivierungsbenchmark untersucht schließlich ein Team, das 10 Rollen umfasst, die alle an verschiedene Basisklassen gebunden sind. Seine Laufzeit betrug 21261,4 ms bei thread-lokaler und 16822,4 ms bei globaler Aktivierung.

Listing 4.9: Aktivierungs-Benchmark 3: Team-Klasse

```
public team class Team1 {
    public class Role1 playedBy BaseClass1 {}
    public class Role2 playedBy BaseClass2 {}
    // Rollen 3 – 10 analog
    precedence Role1, Role2, Role3, Role4,
     Role5, Role6, Role7, Role8, Role9,
     Role10;
}
```

4.3 Benchmarking des Webevorgangs

Das prinzipielle Vorgehen bei der Messung des Webevorgangs wurde bereits in Abschnitt 3.4.2 beschrieben. Es wurde dabei die Brutto- von der Nettowebezeit unterschieden. Nachfolgend werden erst die drei erstellten Benchmarks zur indirekten Messung der Brutto-Webezeit mit Hilfe von äquivalenten ObjectTeams- und reinen Java-Programmen vorgestellt. Anschließend wird es um die direkte Messung der Netto-Webezeit mittels Instrumentierung der ObjectTeams-Laufzeitumgebung gehen.

4.3.1 Brutto-Webezeit

Webezeit-Benchmark 1

Der erste Benchmark besteht aus einem Team, dessen einzige Rolle an eine Basisklasse gebunden ist. Es existiert eine *Callin*- und eine *Callout*-Methodenbindung, die jeweils einmal zur Ausführung kommt. Von der Gesamt-Laufzeit dieses Programms wurde die Zeit subtrahiert, die das äquivalente Java-Programm benötigte. Diese Differenz und damit die approximierte Brutto-Webezeit betrug 649,4 ms.

Webezeit-Benchmark 2

Auch bei der Messung der Webezeit galt das Interesse der Frage, wie sich eine größere Anzahl Klassen auf den Overhead auswirkt. Webezeit-Benchmark 2 umfasst daher 10 Teams, die analog zu Benchmark 1 jeweils in einer *playedBy*-Beziehung mit derselben Basisklasse stehen und je eine *Callin*- und eine *Callout*-Bindung besitzen. Erwartungsgemäß ergab die Messung der Brutto-Webezeit mit 863,3 ms eine längere Laufzeit als beim ersten Benchmark.

Webezeit-Benchmark 3

Der dritte Benchmark zur Messung der Webezeit ist komplexer als der erste, besteht jedoch aus weniger Klassen als der zweite. Abbildung 4.3 zeigt ihn in Form eines UML-Klassendiagramms. Es wurde versucht, möglichst viele der ObjectTeams/Java-Konzepte gleichzeitig in diesen Benchmark einzuarbeiten, um so verschiedene Vorgänge bei der Code-Trans-

[4] 10 Millionen Methodenaufrufe, s. Listing 4.1
[5] 10 Millionen Methodenaufrufe, s. Listings 4.5 und 4.6

formation in die Messung mit einzubeziehen. Dementsprechend enthält er alle drei Arten von *Callin*-Methodenbindungen, eine *Callout*-Bindung und einen *Basecall*. Eine weitere *Callin*-Bindung besteht zwischen zwei Methoden, die `static` deklariert wurden, da sich der Webevorgang bei diesen Methoden vom nicht-statischen Fall unterscheidet. Hier betrug der gemessene Overhead 680,5 ms.

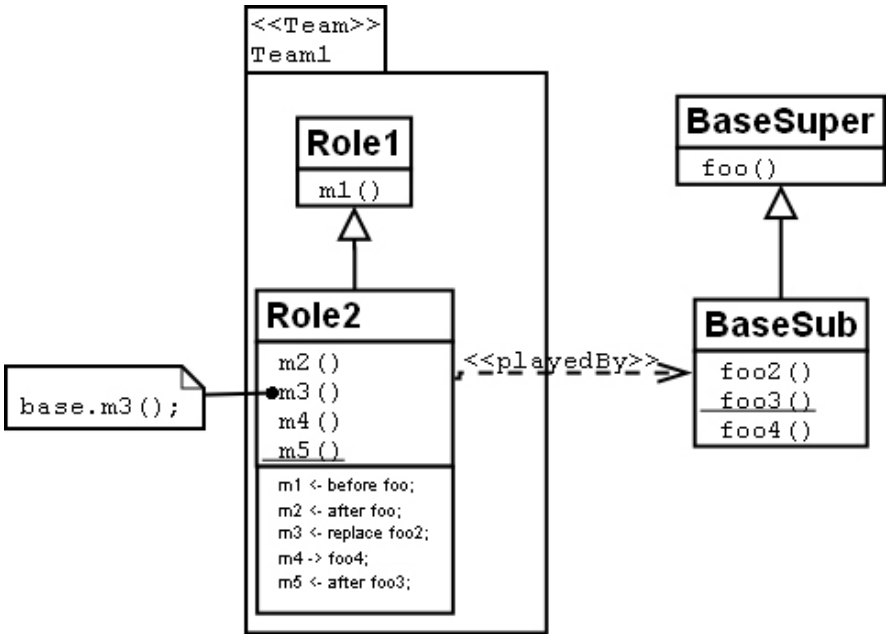

Abbildung 4.3: Webezeit-Benchmark 3 als UML-Klassendiagramm

4.3.2 Ergebnisse

Tabelle 4.4 fasst abschließend die Ergebnisse aller Micro-Benchmarks zusammen. Hier zeigt sich noch einmal, dass vor allem der Overhead durch *Callins* signifikante Ausmaße annimmt. Aber auch die anderen ObjectTeams/Java-Konzepte verursachen einen deutlich messbaren Overhead.

4.3.3 Netto-Webezeit

Als Grundlage für die Messung der Netto-Webezeit wurden drei in ObjectTeams/Java geschriebene Beispielprogramme ausgewählt: das Stop-

Benchmark	**Laufzeit**
Callin-Vergleichsmessung[2]	26,2 ms
Callin 1	7792,7 ms
Callin 2	8450,5 ms
Callin 3	75345,5 ms
Callin 4	65032,8 ms
Callout-Vergleichsmessung[3]	27,0 ms
Callout 1	3876,1 ms
Aktivierung 1 (thread-lokal)	3203,6 ms
Aktivierung 2 (thread-lokal)	5954,3 ms
Aktivierung 3 (thread-lokal)	21261,4 ms
Aktivierung 1 (global)	17325,0 ms
Aktivierung 2 (global)	16484,8 ms
Aktivierung 3 (global)	16822,4 ms
Brutto-Webezeit 1	649,4 ms
Brutto-Webezeit 2	863,3 ms
Brutto-Webezeit 3	680,5 ms

Abbildung 4.4: Die Resultate der Micro-Benchmarks

watch-, das Flightbonus- und das Hierarchy-Beispiel. Es handelt sich dabei um kleinere beispielhafte Anwendungen, deren konkrete Funktionalität hier weniger relevant ist.

Ergebnisse Abbildung 4.5 zeigt die Messergebnisse der Netto-Webezeit bei den genannten Beispielprogrammen. Der Vegleich mit den Messungen der Brutto-Webezeit (siehe Abschnitt 4.3.1) macht deutlich, dass der Anteil der Netto-Webezeit an der gesamten Webezeit den weitaus geringeren Teil ausmacht. Das Stopwatch-Beispiel, das ein Team und eine Basisklasse enthält, ist mit dem Webezeit-Benchmark 1 vergleichbar. Die Netto-Webezeit beträgt nur etwa ein Sechstel der Brutto-Webezeit des besagten Micro-Benchmarks.

Dieses Verhältnis zwischen Brutto- und Netto-Webezeit erschwert die Optimierung des Webevorgangs insofern, als dass die ObjectTeams/Java-Laufzeitumgebung ausschließlich Einfluss auf die Netto-Webezeit hat. Der

Programm	Netto-Webezeit
StopWatch	103,2 ms
Flightbonus	135,6 ms
Hierarchy	278,6 ms

Abbildung 4.5: Die Resultate der Messungen der Netto-Webezeit

größere Teil der Zeit wird von den Frameworks JMangler bzw. BCEL verbraucht.

4.4 Macro-Benchmarking

Die bisher durchgeführten Micro-Benchmarks messen bestimmte Arten von ObjectTeams-spezifischem Overhead. Sie beachten jedoch nicht die Frage, wie groß der Anteil dieses Overheads in einer realen Anwendung eigentlich ist. Deshalb wurde zusätzlich ein Macro-Benchmarking vorgenommen. Bei der untersuchten Anwendung handelt es sich um eine einfache Verwaltung von universitären Sportkursen. Kurse können gebucht werden, Teilnehmer werden bei Krankheit des Kursleiters benachrichtigt. Außerdem gibt es ein Bonusprogramm, bei dem die Sportler Punkte sammeln können. Diese Anwendung wurde für eine Lehrveranstaltung entwickelt und existiert sowohl in einer ObjectTeams- als auch in einer reinen Java-Version. Diese beiden Versionen können zwar nicht als äquivalent im Sinne von Abschnitt 3.4.2 (Webezeit) angesehen werden, erfüllen jedoch exakt dieselbe Funktionalität. So wurde beispielsweise die Krankheitsbenachrichtigung in der reinen Java-Version mit dem *Observer*-Pattern gelöst, wohingegen die ObjectTeams-Version mit Methodenbindungen arbeitet.

Es wurde die Laufzeit eines beispielhaften Programmdurchlaufs gemessen, in dem mehrere Personen und Sportkurse angelegt werden. Danach werden einige Kurse gebucht und schließlich erkrankt ein Kursleiter. Während die reine Java-Version für die Ausführung 21 ms benötigte, brauchte die in ObjectTeams/Java geschriebene Version des Programms mehr als das Zehnfache, nämlich 225 ms.

Dieses Ergebnis zeigt, dass der Overhead, den ObjectTeams/Java zur Laufzeit verursacht, nicht nur in Micro-Benchmarks, sondern auch in ei-

ner regulären Anwendung deutlich messbar ist. Im konkreten Fall mag dem Benutzer ein Laufzeitunterschied von 200 ms nicht auffallen[4]. Aufgrund des relativ hohen Faktors, um den sich die Versionen unterscheiden, ist jedoch anzunehmen, dass bei größeren Anwendungen spürbare Performance-Einbußen auftreten. So wäre z.B. ein Laufzeit-Unterschied zwischen einer Sekunde und zehn Sekunden vom Benutzer klar zu bemerken.

4.5 Profiling

4.5.1 Vorgehen

Der Zweck des Profilings war zum einen die Aufschlüsselung des Overheads, der durch ObjectTeams/Java zur Programmlaufzeit verursacht wird. Daran anknüpfend sollte es des Weiteren Aufschluss über mögliche Optimierungspunkte geben, um diesen Overhead schließlich minimieren zu können. Um diese Ziele zu erreichen, wurden zwei verschiedene Arten von Programmen einem Profiling unterzogen. Einerseits wurden die in Abschnitt 3.4 beschriebenen Benchmarks untersucht. Diese bestehen fast ausschließlich aus Overhead, da sie genau diesen messen sollen. Die Profilings sollten also in erster Linie Aufschluss über die Zusammensetzung der verschiedenen Arten des Overheads geben. Zum zweiten wurden mehrere der Beispielprogramme untersucht, die in der OTDT-Distribution [OTD] enthalten sind. Der Anteil des Overheads dieser Programme ist deutlich geringer als im ersten Fall. Hier liefert das Profiling jedoch Informationen darüber, wie sich der Overhead in realistischen Anwendungen darstellt.

4.5.2 Ergebnisse

Nachfolgend wird beschrieben, mit welchen Ergebnissen die einzelnen Profilings durchgeführt wurden und wie diese zu konkreten Optimierungen beigetragen haben.

[4] Die Performance eines Systems in der Wahrnehmung des Benutzers wird auch mit dem Schlagwort *Perceived Performance* bezeichnet.

Micro-Benchmarks

Zunächst soll anhand des Profilings zweier Micro-Benchmarks exemplarisch gezeigt werden, wie mit Hilfe des Profilings der spezifische Overhead verschiedener ObjectTeams-Features reduziert werden konnte.

Callin-Benchmark 3 Callin-Benchmark 3 (s. Abschnitt 3.4.1) beinhaltet 10 Teams mit je einer an dieselbe Basisklasse gebundenen Rolle. Jede Rolle besitzt eine Callin-Methodenbindung mit derselben Basismethode. Mit dem Profiling dieses Benchmarks konnte zunächst festgestellt werden, dass die mit Abstand am längsten ausgeführte Methode eine Methode aus der Java-API war, nämlich `java.lang.ref.ReferenceQueue.remove()`. Ihre Ausführung machte etwa 28% der Gesamtlaufzeit aus. Diese Methode wird von Javas *Garbage Collection* benutzt, um nicht mehr referenzierte Objekte wieder freizugeben. Die Tatsache, dass sie einen so hohen Anteil an der Ausführungszeit hat, deutet darauf hin, dass das Programm eine große Anzahl von Objekten erzeugt, die anschließend wieder freigegeben werden müssen. Die Auswertung der Profiling-Daten ergab weiterhin, dass fast 80% der erzeugten Objekte vom Typ `java.lang.Integer` waren. Diese Instanziierungen fanden wiederum ausschließlich in zwei Methoden statt: zum Einen in `org.objectteams.Team._OT$implicitlyActivate()` sowie zum Anderen in `_OT$implicitlyDeactivate()` aus derselben Klasse. Die beiden Methoden beinhalten die Implementierung der impliziten Aktivierung und Deaktivierung von Teams.

Die `Integer`-Objekte wurden dabei als Zähler benutzt, um die Anzahl der stattgefundenen impliziten Aktivierungen und Deaktivierungen festzuhalten, damit die „letzte“ mehrerer geschachtelter impliziter Deaktivierungen wieder zur tatsächlichen Deaktivierung des jeweiligen Teams führte. Da dieser Zähler für jeden Thread separat geführt werden muss, ist er innerhalb eines `ThreadLocal`-Objekts gekapselt. Aus diesem Grund ist es auch nicht möglich, stattdessen den Typ `int` zu benutzen, was deutlich performanter wäre.

Bei jeder Veränderung des Zählerwertes wurde ursprünglich mit einem Aufruf von `new Integer()` ein neues Objekt erzeugt. Diese relativ aufwendige Operation kann dadurch optimiert werden, dass anstelle des Konstruktors die statische Methode `Integer.valueOf()` benutzt wird, die ebenfalls ein `Integer`-Objekt liefert. Sie realisiert zudem die Wiederverwendung bereits benutzter Objekte. Da Objekte vom Typ `Integer` (wie z.B. auch Strings) unveränderlich sind, ist es in der Regel unnötig, ein neues

Objekt zu erzeugen, wenn ein anderes mit demselben Wert bereits existiert. Allein diese Optimierung führte bereits zu einer messbaren Verbesserung der Callin-Benchmarks.

Aktivierungs-Benchmark 3 Dieser Benchmark misst die Zeit für die Aktivierung eines Teams mit zehn Rollen, die jeweils an verschiedene Basisklassen gebunden sind. Wie bereits bei Callin-Benchmark 3 nahm auch hier die Methode `java.lang.ref.ReferenceQueue.remove()` mit Abstand die meiste Zeit in Anspruch. Über 37% der Laufzeit verbrachte der Benchmark in dieser Methode. Also verursachte auch bei dieser Messung die große Anzahl der erzeugten Objekte den größten Teil des Overheads. Die mit 12,2% und 11,7% am häufigsten erzeugten Objekte waren vom Typ `java.util.HashMap$Entry` bzw. `Boolean`. Bei ersteren handelt es sich um Einträge in jener Hashmap, in der das Team die Threads speichert, für die es aktiv ist. Die boolschen Werte werden benutzt, um anzuzeigen, ob es sich um eine explizite oder implizite Aktivierung handelt. Beide Objekterzeugungen lassen sich in derselben Programmzeile der Methode `Team.activate()` lokalisieren, in der die explizite Team-Aktivierung implementiert ist:

```
_OT$activatedThreads.put(thread, new Boolean(true));
```

Ein denkbarer Optimierungsansatz wäre hier die Benutzung einer anderen *Collection*-Klasse. Allerdings ist `java.util.HashMap` bereits die schnellste Implementierung des `Map`-Interface, so dass diese Möglichkeit nicht besteht. Auch von der grundsätzlichen Notwendigkeit, die Threads zu speichern, für welche die Aktivierung des Teams gilt, kann nicht abgesehen werden. Die Art, wie die `Boolean`-Objekte erzeugt werden, ist jedoch optimierungsfähig. Da auch Objekte von diesem Typ unveränderlich sind, ist es selten notwendig, explizit ein neues Objekt zu erzeugen. Stattdessen kann auf die Konstante `Boolean.TRUE` zurückgegriffen werden. So kann der Overhead bei der Teamaktivierung bereits messbar verringert werden.

Fazit Auf die beschriebene Weise wurden sämtliche der erstellten Micro-Benchmarks untersucht. Insbesondere die große Anzahl der erzeugten Objekte hat sich dabei als einer der Hauptfaktoren erwiesen, welche die Performance der Benchmarks am stärksten beeinträchtigen. Da das Profiling die Identifikation so genannter *Performance Hotspots* ermöglicht, können bereits relativ kleine Optimierungen derselben deutlich messbare Ergebnisse liefern.

Reale Anwendungen

Während das Profiling der Micro-Benchmarks Aufschluss über die Zusammensetzung bestimmter Arten von Overhead gibt, sagt es nur wenig darüber aus, welche Bedeutung der Overhead in realen Anwendungen besitzt. Aus diesem Grund wurden auch vollständige ObjectTeams/Java-Programme einem Profiling unterzogen. Dafür wurde auf Beispiel-Programme zurückgefgriffen, die in der OTDT-Distribution enthalten sind. Exemplarisch werden hier die Ergebnisse des Profilings des Flightbonus-Beispiels besprochen.

Flightbonus Das Flightbonus-Beispiel besteht aus einer Basisanwendung, die ein sehr einfaches Flugbuchungssystem realisiert, und einer aspektorientierten Erweiterung, in der die Basisapplikation um ein Bonusmeilenprogramm erweitert wird. Für das Profiling wurde ein einfacher Programmdurchlauf ausgeführt. Nach dem Registrieren eines Benutzers für Flugbuchungs- und Bonusmeilensystem wird ein Flug gebucht.

Das Profiling dieses Programmdurchlaufs ergab zunächst, dass aufgrund der Kürze der Laufzeit der gemessene Overhead im Wesentlichen durch die Bytecode-Transformation zur Ladezeit zustande kommt. Ferner stellte sich heraus, dass die beiden Methoden, die zusammen mit über 80% die meiste Zeit verbrauchten, zum einen `java.io.FileInputStream.read()` und zum anderen erneut `java.lang.ref.ReferenceQueue.remove()` waren. Mit der erstgenannten Methode werden die Klassen durch JMangler mit Hilfe von BCEL eingelesen. Das Einlesen von Klassen ist prinzipiell nötig, wenn diese unter Benutzung von JMangler transformiert werden sollen. Die einzige denkbare Optimierung besteht deshalb darin, die Frameworks JMangler und BCEL zu ersetzen. Am Umstieg von JMangler auf JPLIS wird derzeit bereits gearbeitet (s. Abschnitt 4.6), auf eine Alternative zu BCEL wird im Ausblick in Abschnitt 7.2 eingegangen.

Die zweitgenannte Methode erfordert erneut einen Blick auf die am häufigsten erzeugten Objekte. Diese sind im vorliegenden Fall zum einen vom Typ `java.util.HashMap$KeyIterator` und `java.util.HashMap$KeySet` mit knapp 40% sowie `java.util.AbstractMap` und `java.util.AbstractSet` mit zusammen 21%. Diese Objekte werden ebenfalls von Methoden des JMangler-Frameworks erzeugt.

Das Profiling dieses Beispiels bestätigt die Ergebnisse der Webezeit-Messungen. Das Einklinken in den Klassenladeprozess durch JMangler ver-

braucht nicht nur den größten Teil der Webezeit, sondern sogar der gesamten Laufzeit einer kleinen Anwendung wie dem Flightbonus-Beispiel.

4.6 Umstellung auf JPLIS

Bei allen bisher beschriebenen Messungen wurde Version 0.9.10 von ObjectTeams/Java benutzt. Diese verwendet das JMangler-Framework für das „Einklinken“ in den Klassenladeprozess, um der Laufzeitumgebung die Transformation des Bytecodes zu ermöglichen. Wie bereits in Abschnitt 2.2.3 erwähnt, findet derzeit jedoch eine Umstellung von JMangler auf die *Java Programming Language Instrumentation Services* (JPLIS) statt. JPLIS ist seit Java SE 5 Bestandteil der API. Durch die Umstellung kann daher die bisherige Abhängigkeit von einem Drittanbieter beendet werden. Der Hauptgrund für den Wechsel ist jedoch die Tatsache, dass mit Hilfe von JPLIS auch *Runtime Weaving* in ObjectTeams/Java möglich wird. Die JPLIS-Funktionalität wird mittels eines so genannten Agenten angestoßen. Dabei handelt es sich um eine Java-Klasse, die eine `premain`-Methode enthält und in einer `jar`-Datei enthalten sein muss. Deren Name wird beim Start per Kommandozeilenargument an die *Virtual Machine* übergeben.

Dieser Abschnitt enthält eine kurze Untersuchung der Auswirkungen besagter Umstellung auf die Performance des Webevorgangs. Von der Umstellung ausschließlich betroffen ist die Brutto-Webezeit (s. Abb. 4.1). Die eigentliche Bytecode-Transformation bleibt unberührt. Daher wurden einige[5] der Messungen zur Brutto-Webezeit zusätzlich auch mit JPLIS statt mit JMangler ausgeführt.

Das Resultat dieser Messungen war eine um 30 bis 40% längere Brutto-Webezeit mit JPLIS. Etwa die Hälfte dieses Zeitunterschiedes benötigt JPLIS für das Laden und Starten des Agenten, der die Bytecode-Transformation anstößt. Die dafür verbrauchte Zeit ließ sich feststellen, indem die Ausführungszeit eines reinen Java-Programms zunächst ohne und anschließend mit einem Agenten gemessen wurde, dessen `premain`-Methode leer war. Dieser Agent führte also keinerlei Code aus. Die Differenz beider Messungen kommt demnach ausschließlich durch das Laden des Agenten zustande. Die Ursache für den restlichen Zeitunterschied konnte

[5] Der Funktionsumfang von ObjectTeams mit Verwendung von JPLIS ist derzeit noch eingeschränkt.

nicht abschließend geklärt werden, da weder eine detailliertere Dokumentation über JPLIS noch der Quellcode für den Agentenmechanismus zur Verfügung stehen. Die Vermutung, dass mit JPLIS mehr Klassen verwendet werden und deren Ladezeit den Prozess verlangsamt, hat sich nicht bestätigt. Im Gegenteil, bei den vorgenommenen Messungen wurden mit JPLIS rund 170 Klassen weniger geladen als mit JMangler.

Zusammenfassend ist festzustellen, dass die Umstellung von JMangler auf JPLIS mit deutlichen Performance-Einbußen in der Brutto-Webezeit einhergeht. Den eingangs erwähnten Vorteilen stehen diese Einbußen somit gegenüber. Eine abschließende Bewertung des Umstiegs kann jedoch erst erfolgen, wenn dieser auch vollständig abgeschlossen ist.

Kapitel 5

Optimierung der Webestrategie

Der praktische Teil dieser Arbeit bestand in der Optimierung der Aspektwebestrategie von ObjectTeams/Java bezüglich ihrer Performance. Diese Optimierung umfasste zum einen den Webevorgang selbst und zum anderen den Code, der von ObjectTeams-Compiler und -Laufzeitumgebung zusätzlich generiert wird.

5.1 Optimierung des Webevorgangs

Die Optimierung der Performance des Webevorgangs lässt sich in die folgenden drei Kategorien einteilen: Sourcecode-Optimierungen, Strategische Optimierungen und sonstige Optimierungen.

Im folgenden werden die einzelnen Optimierungen des Webevorgangs erläutert, die im Rahmen dieser Arbeit vorgenommen wurden.

5.1.1 Sourcecode-Optimierungen

Unter Sourcecode-Optimierungen werden solche Optimierungen verstanden, welche die Performance auf Ebene des Quellcodes verbessern. Sie verändern dabei nicht das Verhalten des Programms. Diese Eigenschaft teilen sie mit dem Refactoring [FBB+99, Ref]. Martin Fowler unterscheidet dennoch Sourcecode-Optimierungen von Refactorings, da beide mit unterschiedlichen Zielen angewendet werden. Beim Refactoring stehen Verständlichkeit und Wartbarkeit des Codes im Mittelpunkt, bei der Optimierung hingegen dessen Performance [Fow04].

Der erste Teil der Optimierungsarbeit bestand darin herauszufinden, welche Teile des Quellcodes einer Optimierung bedürfen. Aufschluss über die zu optimierenden Stellen im Code gaben zum einen die gewonnenen Erkenntnisse aus dem Profiling (s. Abschnitt 3.3) und zum anderen verschiedene Publikationen, welche die Performance-Optimierung von Java-Programmen thematisieren [Shi03, WK00, Bul00].

Logging

Um bei der Entwicklung der ObjectTeams-Laufzeitumgebung zu Testzwecken Meldungen auf der Konsole ausgeben zu können, existiert eine Logging-Methode. Diese bekommt die jeweilige Meldung als String übergeben. Sie prüft ein entsprechendes *Flag* und gibt – falls dieses gesetzt ist – die Meldung auf der Konsole aus.

Das Problem bei dieser Herangehensweise besteht darin, dass die Meldungen meist aus mit „+“ zusammengesetzten Strings bestehen. Die aus Sicht der Performance relativ aufwendige String-Konkatenation wird also in jedem Fall ausgeführt, insbesondere auch, wenn das Logging ausgeschaltet ist. Um dies zu verhindern, wurde die Überprüfung des *Flags* aus der Logging-Methode herausgenommen und vor jeden einzelnen ihrer Aufrufe gestellt. Da die Anzahl derselben etwa 60 beträgt, war allein diese Optimierung deutlich messbar.

Strings

Ein Datentyp, der von der Laufzeitumgebung besonders häufig verwendet wird, sind Strings. Namen und Signaturen von generierten Methoden, *Modifier*, Namen von Typen sowie vom Compiler generierte Bytecode-Attribute sind Beispiele für den vielfachen Gebrauch von Strings seitens der ObjectTeams-Laufzeitumgebung. In Java gehören Strings zu denjenigen Objekten, die unveränderlich sind. Dies bedeutet, dass der Speicherplatz des Objekts nie mit einem anderen Wert überschrieben wird, was dahingegen bei veränderlichen Objekten möglich ist. Besonders beim Umgang mit Strings gab es daher einige Aspekte, die in Bezug auf die Performance zu berücksichtigen waren. Diese seien hier nur kurz zusammengefasst.

Die Klasse `StringBuffer` kann benutzt werden, um Strings effizient zu bearbeiten, da ihre Objekte veränderlich sind. Soll beispielsweise ein String innerhalb einer Schleife iterativ zusammengesetzt werden, ist ein `String-`

`Buffer` gegenüber dem „+“-Operator zu bevorzugen, da bei dessen Benutzung bei jedem Schleifendurchlauf ein neues Objekt erzeugt werden muss.

Seit Java SE 5 gibt es die Klasse `StringBuilder`. Sie bietet dieselbe Funktionalität wie `StringBuffer`, ist jedoch schneller, da ihre Methoden nicht synchronisiert sind. Wenn Thread-Sicherheit nicht benötigt wird, ist also der `StringBuilder` dem `StringBuffer` vorzuziehen.

Objekte vom Typ `StringBuffer` bzw. `StringBuilder` besitzen eine bestimmte Kapazität für die Länge des Strings, den sie repräsentieren. Standardmäßig beträgt diese Kapazität 16 Zeichen. Wird der String länger als die aktuelle Kapazität, muss diese erhöht werden. Da dieser Vorgang Zeit kostet, sollte beim Aufruf des Konstruktors – sofern bekannt – die voraussichtlich benötigte Kapazität angegeben werden.

Um größere Strings in Teilen zu bearbeiten, kann ein Objekt der Klasse `StringTokenizer` verwendet werden. Dieses liefert immer den Teilstring bis zum nächsten Vorkommen eines vordefinierten Zeichens (sog. *Delimiter*, z.B. Leerzeichen o.ä.). Dies ist zwar eine komfortable Möglichkeit, mit Strings zu arbeiten, für die Performance ist ein `StringTokenizer` jedoch von Nachteil. Während die Erzeugung eines Objektes ohnehin eine relativ lange Zeit in Anspruch nimmt, ist das bei dieser Klasse im Besonderen der Fall, da intern ein beträchtlicher Aufwand für die Realisierung der Funktionalität nötig ist. Folglich sollte überall dort, wo die Performance eine Rolle spielt, von der Benutzung der Klasse `StringTokenizer` Abstand genommen werden. Bei kleineren Strings genügen oft die Methoden der Klasse `String` wie z.B. `indexOf()` oder `substring()`. Wo diese Methoden nicht genügen, kann auch auf die Methode `String.split()` zurückgegriffen werden, die eine ähnliche Funktion hat wie ein `StringTokenizer`, diese jedoch deutlich performanter ausführen kann.

Attributzugriff

Der Zugriff auf die Attribute eines Objektes geschieht in der Regel nicht direkt. Um die Kapselung der Daten zu bewahren, werden meist so genannte Setter- und Getter-Methoden benutzt. Diese setzen den Wert des Attributs bzw. liefern ihn zurück. Ein Objekt, das auf eigene Attribute zugreift, kann ebenfalls diese Methoden benutzen. Wenn jedoch die Performance der Applikation eine Rolle spielt, sollte ein Objekt auf seine eigenen Attribute direkt zugreifen. Dies verletzt die Kapselung nicht und spart bei

jedem Zugriff einen Methodenaufruf ein. Deshalb wurde dieses Vorgehen an sämtlichen betroffenen Stellen im Code der Laufzeitumgebung realisiert.

Schleifen

Schleifen sind bei der Sourcecode-Optimierung ebenfalls von großer Bedeutung. Anweisungen, die innerhalb von Schleifen stehen, werden in der Regel sehr oft ausgeführt. Hier kann sich also eine Optimierung besonders lohnen, wobei vor allem darauf zu achten ist, dass Anweisungen, die nicht wiederholt ausgeführt werden müssen (z.B. Zuweisungen), außerhalb der Schleife stehen. Insbesondere sollten im Schleifenkopf nach Möglichkeit keine Methodenaufrufe stehen.

Datenstrukturen

Java verfügt über ein relativ großes *Collections*-Framework [JCF] mit einer Reihe verschiedener Klassen. Alle diese Klassen implementieren eins der Interfaces `Set`, `Map` und `List`, die für unterschiedliche Verwendungen vorgesehen sind. Die Wahl des Datentyps kann auch einen großen Einfluss auf die Performance haben. So ist eine `ArrayList` beispielsweise deutlich performanter als eine `LinkedList`, ein `TreeSet` ist langsamer als ein `HashSet`. Neben der konkreten Anwendung der Datenstruktur sollte also auch die Performance ein Kriterium für die Auswahl der richtigen *Collection*-Klasse sein.

Innere Klassen

Die Implementierung der ObjectTeams-Laufzeitumgebung enthält auch mehrere innere Klassen. Innere Klassen können in Java entweder als Eigenschaften der umgebenden Klasse oder eines konkreten Objekts implementiert werden. Im ersten Fall werden sie mit dem Schlüsselwort `static` versehen. Relevant für die Performance ist die Tatsache, dass im letzteren Fall Instanzen der inneren Klasse immer eine Referenz auf das Objekt speichern müssen, das sie erzeugt hat. Dadurch werden diese Instanzen größer und der Umgang mit ihnen langsamer. Benötigt eine innere Klasse gar keine Referenz auf das erzeugende Objekt – handelt es sich also ausschließlich um eine Eigenschaft der Klasse – so sollte sie immer als `static` deklariert werden, um so eine bessere Performance zu erreichen.

Fazit

Zusammenfassend lässt sich feststellen, dass die Sprache Java meist mehrere Möglichkeiten bietet, eine bestimmte Funktionalität zu implementieren. Daraus ergibt sich eine Vielzahl von potentiellen Sourcecode-Optimierungen, von denen ein Teil beschrieben wurde. Sämtliche der genannten Verfahrensweisen und weitere wurden bei der Optimierung der ObjectTeams-Laufzeitumgebung berücksichtigt. Die entsprechenden Ergebnisse werden in Abschnitt 5.3 besprochen.

5.1.2 Strategische Optimierungen

Folgend werden die Optimierungen besprochen, die über reine Quellcode-Optimierungen hinausgehen. Sie ändern das Verhalten der ObjectTeams-Laufzeitumgebung bei der Bytecode-Transformation, um diese effizienter zu gestalten.

Überspringen von Klassen

Bei der Ausführung eines ObjectTeams-Programms wird jede Klasse zum Zeitpunkt des Ladens in die JVM von der Laufzeitumgebung dahingehend untersucht, ob sie Teil einer Rolle-Basis-Beziehung ist. Ist dies der Fall, so werden die nötigen Transformationen vorgenommen. Anderenfalls muss der Bytecode der jeweiligen Klasse nicht verändert werden. Wenn für bestimmte Klassen von vornherein ausgeschlossen ist, dass sie transformiert werden müssen, so ist diese Überprüfung unnötig. Die Klassen der ObjectTeams-Laufzeitumgebung sowie der Frameworks JMangler und BCEL dienen der Bytecodetransformation und damit letztendlich der Realisierung der ObjectTeams-spezifischen Sprachkonzepte zur Laufzeit. Dass sie selbst von einer Rollenklasse adaptiert werden, ist nicht vorgesehen. Aus diesem Grund wurde eine Abfrage vorangestellt, die bewirkt, dass diese Klassen übersprungen werden. Es wurde ermittelt, dass sie beispielsweise beim in der OTDT-Distribution [OTD] enthaltenen *Flightbonus*-Beispiel etwa 15% der Klassen ausmachen, die während der Programmlaufzeit in die JVM geladen werden.

5.1.3 Sonstige Optimierungen

Die Klassen der ObjectTeams-Laufzeitumgebung wie auch die des BCEL-Frameworks werden, wie bei Java-Programmen üblich, in jar-Dateien aus-

geliefert. Diese enthalten den Bytecode der Klassen als class-Dateien in komprimierter Form. Zur Ladezeit müssen diese Dateien also wieder dekomprimiert werden bevor sie von der JVM benutzbar sind. Der Dekomprimierungsvorgang verlangsamt also das Laden dieser Klassen und damit die Laufzeit des Programms. Werden die jar-Dateien allerdings mit der Option `-0` erzeugt, so entfällt die Komprimierung. Dadurch wird die Datei zwar größer, das Laden der in ihr enthaltenen Klassen wird jedoch beschleunigt. Aus diesem Grund wurden die jar-Dateien der Laufzeitumgebung und von BCEL auf die unkomprimierte Form umgestellt.

5.2 Optimierung des generierten Codes

In ObjectTeams/Java wird zusätzlicher Code sowohl zur Kompilier-Zeit vom Compiler als auch zur Ladezeit von der Laufzeitumgebung generiert. Auch an diesem Code konnten kleinere Sourcecode-Optimierungen vorgenommen werden, die im Folgenden knapp besprochen werden.

5.2.1 Compiler

Der Vorgang, bei dem einem Basisobjekt ein entsprechendes Rollenobjekt zugeordnet wird, wird in ObjectTeams/Java *Lifting* genannt. Diese Aufgabe erfüllt die vom Compiler generierte Methode `_OT$liftTo<Rollenname>`. Sie bekommt ein Basisobjekt und liefert ein Rollenobjekt zurück. Zunächst wird z.B. mit `Role1 myRole = null;` das Rollenobjekt initialisiert. Unter Umständen kann das Argument der Methode `null` sein. In diesem Fall gibt sie daraufhin auch `null` zurück. Die Optimierung bestand darin, die Initialisierung erst vorzunehmen, *nachdem* sichergestellt ist, dass das Argument nicht `null` ist. Immer da wäre diese sonst unnötigerweise vorgenommen worden. Dies stellt nur eine kleine Optimierung dar, die jedoch durchaus einen Beitrag zur Verbesserung der Performance leisten kann, da das Lifting – wie in Kapitel 3 gesehen – einen erheblichen Teil des ObjectTeams-spezifischen Overheads ausmacht.

Eine zweite Optimierung des vom Compiler generierten Codes betrifft die reflexive Team-Methode `public Object[] getAllRoles()`, die ein Array mit allen Rollenobjekten des Teams zurückliefert. Da die Rollenobjekte in einer `WeakHashMap` gespeichert werden, muss diese in ein Array umgewandelt werden. Das folgende Listing zeigt die bisherige Implementierung dieser Methode für den Fall, dass das betreffende Team nur über eine

Rolle verfügt.

```
public Object[] getAllRoles() {
   Collection first_result = new ArrayList();
   first_result.addAll(_OT$cache_OT$Role1.values());
   return first_result.toArray();
}
```

Um die unnötige Benutzung der lokalen Variable `first_result` zu vermeiden, wurde diese Implementierung durch die äquivalente Programmzeile `return _OT$cache_OT$Role1.values().toArray();` ersetzt. Diese kleine Optimierung wirkt sich nur auf Programme aus, die diese Art von *Reflection* in signifikantem Maße benutzen.

5.2.2 Laufzeitumgebung

Eine Sourcecode-Optimierung in dem Code, der von der Laufzeitumgebung generiert wird, wurde im so genannten *Chaining-Wrapper* vorgenommen. Diese Methode wird zunächst an Stelle einer gebundenen Basismethode ausgeführt. Sie enthält Aufrufe der jeweiligen Rollenmethoden sowie der eigentlichen Basismethode. Ähnlich wie bei der Optimierung der *Lifting*-Methode wurde auch hier eine lokale Variable initialisiert, die eventuell gar nicht benötigt wird, da die Methode u.U. vor ihrer ersten Benutzung terminiert. Daher wurde die Initialisierung hier ebenfalls auf einen späteren Zeitpunkt verschoben, so dass sie nicht mehr unnötigerweise vorgenommen wird. Ebenfalls analog zur *Lifting*-Methode kann diese kleinere Optimierung trotzdem ins Gewicht fallen, da der *Chaining-Wrapper* in ObjectTeams/Java-Programmen gewöhnlich relativ häufig zur Ausführung kommt.

5.3 Ergebnisse der Optimierungen

Der Erfolg der vorgenommenen Optimierungen wird an verschiedenen Stellen deutlich. Sowohl die Micro-Benchmarks, die in Abschnitt 3.4 beschrieben wurden, als auch weitere Messungen zeigen in unterschiedlichem Maße, wie sehr die Performance von ObjectTeams durch die Optimierung verbessert werden konnte.

5.3.1 Micro-Benchmarks

Zunächst folgt eine graphische Übersicht der Micro-Benchmarks, die deren Laufzeiten vor und nach der Optimierung gegenüberstellt. Die Abkürzung „Webez.“ steht dabei für Webezeitbenchmark, „Akt.“ für Aktivierungsbenchmark. „l“ und „g“ bedeuten thread-lokale bzw. globale Teamaktivierung.

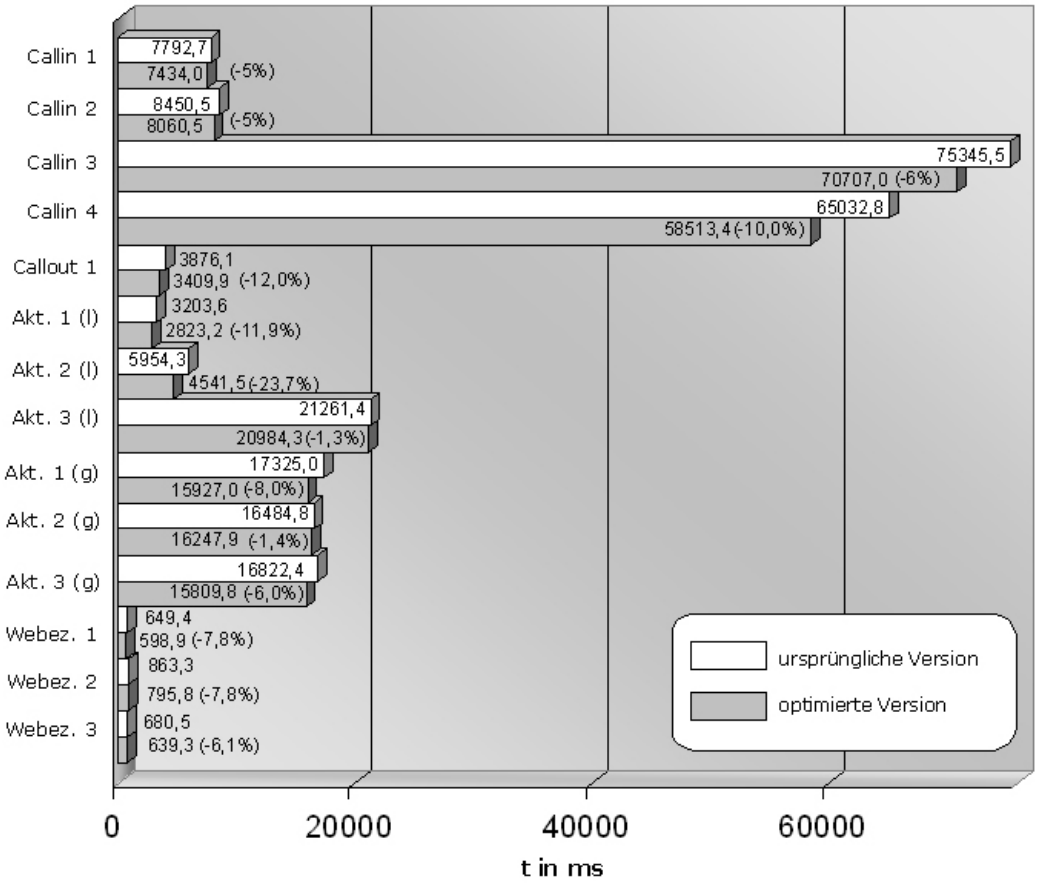

Abbildung 5.1: Ergebnisse der Micro-Benchmarks vor und nach der Optimierung

Wie in der Abbildung zu erkennen ist, liegt die Größenordnung der Performance-Verbesserungen je nach Benchmark zwischen gut einem und knapp 24 Prozent. Besonders bemerkbar macht sich die Optimierung bei

dem *Callout*-Benchmark sowie bei zwei der Aktivierungs-Benchmarks.

5.4 Verwandte Arbeiten

Dieser Abschnitt gibt einen kurzen Überblick über einige andere Arbeiten mit dem Ziel der Optimierung aspektorientierter Systeme.

5.4.1 Envelope-Based Weaving

Bockisch et al. beschreiben ein Verfahren, mit dessen Hilfe der Webevorgang für bestimmte Arten von Pointcuts beschleunigt werden kann [BHMM05]. Das so genannte *Envelope-Based Weaving* eignet sich für das Weben bei Methoden*aufrufen* sowie bei lesenden und schreibenden Feldzugriffen. Für diese Arten von Pointcuts muss normalerweise an allen Stellen im Code gewoben werden, an denen die entsprechenden Methodenaufrufe bzw. Feldzugriffe auftreten. Für eine optimale Performance des Webens wäre es jedoch wünschenswert, die wiederholte Ausführung desselben Webevorgangs zu vermeiden. Das Prinzip des *Envelope-Based Weaving* besteht in der Einführung einer Indirektion, die die beschriebene Redundanz auflöst. Für jede Methode, deren Aufruf an einen Aspekt gebunden ist, wird ein so genannter *Envelope* generiert. Dabei handelt es sich um eine neue Methode, welche die eigentliche Basismethode aufruft. Außerdem werden alle Aufrufe dieser Methode durch Aufrufe des *Envelopes* ersetzt. Bei Feldzugriffen wird analog verfahren. Dadurch wird erreicht, dass der Webevorgang anschließend nur noch an einer Stelle ausgeführt werden muss, nämlich innerhalb des *Envelopes*. Der Laufzeit-Overhead, der durch die zusätzliche Indirektion verursacht wird, ist laut Angaben der Autoren nur minimal.

5.4.2 *Runtime Weaving* mit JVM-Unterstützung

Dynamisches Weben – auch *Runtime Weaving* genannt – ermöglicht das Hinzufügen und Entfernen von Aspekten während der gesamten Laufzeit einer Applikation. In ObjectTeams/Java soll dies zukünftig mit Hilfe von JPLIS unterstützt werden. JPLIS ist Teil des seit Java SE 5.0 existierenden *Java Virtual Machine Tool Interface* (JVMTI). Es handelt sich dabei um eine Schnittstelle, die es Tools erlaubt, mit der JVM zu interagieren.

Eine andere Möglichkeit zur Realisierung von dynamischem Weben ist ein Ansatz, bei dem die Unterstützung dafür direkt in der *Virtual Machine* implementiert wird [Hau06]. Systeme, die diesen Ansatz verfolgen, sind z.B. Steamloom [BHMO04, Ste] und PROSE [PGA02, Pro]. Beide Implementierungen basieren auf der *Jikes Research JVM* von IBM [AA+05, Jik]. Wie ihr Name bereits andeutet, ist diese *Virtual Machine* speziell für Forschungszwecke konzipiert und eignet sich daher besonders gut für Erweiterungen, die *Runtime Weaving* unterstützen.

Dynamische Ansätze arbeiten oft so, dass sie an allen potentiellen Joinpoints so genannte *Hooks* einfügen, an die sich Aspekte „anhängen“ können. Soll ein bestimmter Aspekt also zur Laufzeit gewoben werden, so muss das System alle jene *Hooks* aktivieren, die durch den Aspekt betroffen sind. Um den Overhead gering zu halten, ist es wichtig, dass die *Hooks* nur einen minimalen Einfluss auf die Laufzeit haben. Dieser Ansatz wird daher auch *Minimal Hook Weaving* genannt. Popovici et al. beschreiben ihr Konzept der so genannten *Just-In-Time Aspects* [PAG03]. Hier werden die *Hooks* nicht auf der Ebene des Bytecodes sondern auf der Ebene des *Just-In-Time* Compilers gewoben. Dadurch wird ein geringerer Overhead sowohl mit als auch ohne eingewobenen *Advice* erzielt, da der Maschinencode deutlich besser optimiert wird.

5.4.3 Aspect Bench Compiler

Die Arbeiten des AspectBench Compiler-Projektes [ABC] wie zum Beispiel [DGH+04] und [ACH+05] beschreiben das Vorgehen der Autoren bei der Optimierung der Performance von AspectJ. In [ACH+05] wird speziell die Optimierung von *Around-Advice* und *cflow* besprochen.

Kapitel 6

Vergleich

Nachdem in Kapitel 3 das Benchmarking und in Kapitel 4 die Optimierung der Webestrategie besprochen wurde, soll in diesem Kapitel der Frage nachgegangen werden, wie die Performance von ObjectTeams/Java im Vergleich zu anderen aspektorientierten Programmiersprachen einzuordnen ist.

6.1 Die Vergleichssprachen

6.1.1 AspectJ

Als Vergleichssprache wurde zum einen AspectJ in der Version 1.5.1a ausgewählt, da AspectJ derzeit als De-facto-Standard unter den aspektorientierten Sprachen angesehen werden kann. AspectJ verfügt über eine umfangreiche Pointcut-Sprache. Außerdem bietet es die Möglichkeit, bestehende Klassen um zusätzliche Attribute oder Attribute zu erweitern (so genannte *intertype declaration*), sowie weitere Features. *Callouts* finden jedoch keine Entsprechung in AspectJ und auch die Aktivierung bzw. Deaktivierung von Aspekten ist nicht möglich.

6.1.2 CaesarJ

Für die zweite Vergleichssprache fiel die Wahl auf CaesarJ [MO03, CJ] in der Version 0.8.5. Der Grund dafür war vor allem die relativ große Ähnlichkeit der Sprachen, die beide ihre Wurzeln in *Aspectual Components* [LLM99] haben. Die Konzepte der Teams und Rollen finden sich in den so genannten *Caesar classes* wieder. Während die aus ObjectTeams/Java bekannten *Callouts* eine ähnliche Entsprechung in CaesarJ haben, setzt

die Sprache an Stelle von *Callins* auf Pointcut-Definitionen in AspectJ. Aktivierung und Deaktivierung sind ähnlich wie in ObjectTeams/Java ebenfalls möglich. Eine Unterscheidung zwischen threadlokaler und globaler Aktivierung bietet CaesarJ jedoch nur eingeschränkt.

6.2 Microbenchmarking

Eine Möglichkeit, die Performance von ObjectTeams/Java mit der anderer aspektorientierter Sprachen zu vergleichen, ist die Benutzung der Microbenchmarks, die in Abschnitt 3.4 vorgestellt wurden. Für den Vergleich wurden diese zusätzlich – soweit möglich – in den Vergleichssprachen AspectJ und CaesarJ implementiert. Am Beispiel von Callin-Benchmark 1 (siehe Abschnitt 3.4.1) soll kurz gezeigt werden, wie dabei vorgegangen wurde. Während die Basisklasse unverändert bleibt, da es sich ja um eine reine Java-Klasse handelt, sieht der entsprechende Aspekt in AspectJ folgendermaßen aus:

Listing 6.1: Callin-Benchmark 1 in AspectJ

```
public aspect Aspekt1 {
   pointcut executeBaseMethod() :
      execution(void BaseClass.bm());
   after() : executeBaseMethod() { }
}
```

Die *Callin*-Methodenbindung von ObjectTeams/Java findet ihre Entsprechung in AspectJ in einem *execution-Joinpoint*. *Basecalls* wurden durch `proceed()`-Aufrufe ersetzt und mehrere Teams in ObjectTeams entsprechen mehreren AspectJ-Aspekten.

6.2.1 Callin-Benchmarks

Da CaesarJ an Stelle einer eigenen Pointcut-Sprache auf AspectJ-Pointcuts zurückgreift, wurden die Callin-Benchmarks zusätzlich nur in AspectJ implementiert.

Listing 6.2 zeigt das Ergebnis des Webevorgangs von AspectJ am Beispiel der Basismethode `bm()`. Der Inhalt der Methode wird von einem `try`-Block umgeben. (Da `bm()` ursprünglich leer war, ist auch der `try`-Block leer.)

Aspekte werden in AspectJ mit dem Singleton-Pattern realisiert. Die Methode `aspectOf()` liefert die Singleton-Instanz des Aspekts. Auf diesem Objekt wird schließlich die Methode aufgerufen, die den Advice enthält.

Listing 6.2: Basismethode nach dem Weben von AspectJ

```
private void bm() {
   try {}
   catch(Throwable throwable) {
      Aspekt.aspectOf().ajc$after$Aspekt1$1$6661efa8();
      throw throwable;
    }
    Aspekt.aspectOf().ajc$after$Aspekt1$1$6661efa8();
}
```

Da AspectJ keine Möglichkeit bietet, Aspekte dynamisch zu aktivieren und zu deaktivieren, entfällt die Notwendigkeit, zur Laufzeit zu ermitteln, ob ein Aspekt gerade aktiv ist. Wie in Abschnitt 4.2.3 dargelegt, führt die Unterscheidung von threadlokaler und globaler Aktivierung in ObjectTeams/Java zu zusätzlichem Overhead.

Während in ObjectTeams die Aktivierungsreihenfolge mehrerer Teams über die Ausführungsreihenfolge entscheidet, wird diese in AspectJ statisch festgelegt. Dies kann mit einer `declare precedence`-Anweisung geschehen. Z.B. `declare precedence : Aspekt1, Aspekt2;`. Fehlt eine solche Anweisung, so ist die Reihenfolge nicht-deterministisch. Schließlich besteht in AspectJ anders als in ObjectTeams/Java keine Notwendigkeit, jedem Basisobjekt eine bestimmte Instanz des Aspekts zuzuordnen, da von jedem Aspekt nur eine Instanz existiert.

Benchmark	**ObjectTeams/Java (optimiert)**	**AspectJ**
Callin 1	7434,0 ms	212,1 ms (2,9 %)
Callin 2	8060,5 ms	63,9 ms (0,8 %)
Callin 3	70707,0 ms	233,5 ms (0,3 %)
Callin 4	58513,4 ms	235,2 ms (0,4 %)

Abbildung 6.1: Ergebnisse der Callin-Benchmarks in ObjectTeams/Java und AspectJ

Abb. 6.1 enthält die Ergebnisse des Vergleichs, die deutliche Performance-Vorteile für AspectJ zeigen.

6.2.2 Callout-Benchmarks

In ObjectTeams/Java sind *Callouts* Methodenbindungen, die den Aufruf einer Rollenmethode an das entsprechende Basisobjekt weiterleiten. Während AspectJ über keine direkte Entsprechung hierzu verfügt, gibt es ein ähnliches Konstrukt in CaesarJ. Dort kann im Kontext einer gebundenen Caesar-Klasse mit dem Schlüsselwort `wrappee` auf das Basisobjekt zugegriffen werden. Auf diese Weise ist es möglich, jede sichtbare Methode aufzurufen. *Callouts* in ObjectTeams funktionieren hingegen auch mit Basismethoden, die nicht sichtbar sind, weil sie z.B. als `private` deklariert wurden.

Während Callout-Benchmark 1, also die Vergleichsmessung mit normalen Methodenaufrufen, 27 ms benötigte, betrug die Laufzeit von Benchmark 2 3409,9 ms. Der Overhead besteht hier hauptsächlich aus der impliziten Team-Aktivierung bzw. -Deaktivierung. Da CaesarJ diese Form der Aktivierung nicht vorsieht, ist der Overhead dort deutlich geringer. Es wurden lediglich 33 ms gemessen. Der geringe Unterschied zum herkömmlichen Methodenaufruf erklärt sich dadurch, dass CaesarJ auf das Basisobjekt nicht direkt, sondern mit einer Getter-Methode zugreift, so dass dort sogar eine kleine Optimierung des generierten Codes möglich wäre.

6.2.3 Aktivierung und Deaktivierung

Benchmark	**ObjectTeams/Java**	**CaesarJ**
Aktivierung 1	17325,0 ms	774,8 ms (4,5 %)
Aktivierung 2	16484,8 ms	772,4 ms (4,7 %)
Aktivierung 3	16822,4 ms	770,1 ms (4,6 %)

Abbildung 6.2: Ergebnisse der Aktivierungs-Benchmarks in ObjectTeams/Java und CaesarJ

Die Werte von ObjectTeams/Java in Abb. 6.2 beziehen sich auf die globale Aktivierung. Die geringfügigen Unterschiede bei den Messergebnissen mit CaesarJ sind vermutlich auf Messfehler zurückzuführen. Die Zeit für

die Aktivierung einer CaesarJ-Klasse ist also offensichtlich konstant, d.h. unabhängig davon, ob sie an keine, eine oder an mehrere Basisklassen gebunden ist. In ObjectTeams/Java dagegen ist dies nicht der Fall.

6.3 Macro-Benchmarking

6.3.1 Das *Hierarchy*-Beispiel

Für einen Vergleich der Programmlaufzeiten von CaesarJ- und ObjectTeams-Programmen wurde das *Hierarchy display example* [CJE] ausgewählt. Dieses Programm ist eine Beispiel-Anwendung in CaesarJ. Es besteht aus einer Basisapplikation, die die Personal-Hierarchie eines kleinen Unternehmens modelliert, und einem aspektorientierten Teil. Dieser implementiert eine Visualisierung dieser Hierarchie. Benutzereingaben werden an das Basisprogramm weitergeleitet, und mit Hilfe von Pointcuts bzw. Methodenbindungen wird die Visualisierungskomponente über Änderungen im Modell benachrichtigt. Außerdem bietet das Beispiel verschiedene Darstellungsvarianten an, die in Aspekten implementiert und mittels Aktivierung bzw. Deaktivierung derselben ein- und ausgeschaltet und miteinander kombiniert werden können.

6.3.2 Vorgehen

Um die Performance beider Sprachen vergleichen zu können, wurde zusätzlich eine ObjectTeams/Java-Version des Beispiels angefertigt. Beide Versionen waren im Wesentlichen identisch. Nur die CaesarJ-spezifischen Konstrukte wurden in die entsprechende ObjectTeams-Syntax überführt. So wurden die Caesar-Klassen (mit dem Schlüsselwort `cclass` versehene Klassen) in Teams bzw. Rollen umgewandelt; die auch in CaesarJ verwendeten AspectJ-Pointcuts wurden durch *Callin*-Methodenbindungen ersetzt, usw.

Dieses Vorgehen ermöglichte einen Vergleich anhand von zwei Versionen desselben Programms, die in beiden Sprachen vorlagen. Da es sich um eine Anwendung mit graphischer Benutzerschnittstelle (GUI) handelt, musste außerdem eine Möglichkeit gefunden werden, die Benutzereingaben (Anwahl von Menüpunkten per Mausklick) zu automatisieren. Anderenfalls wäre ein Vergleich nicht möglich gewesen, da ein menschlicher Benutzer nie mit immer derselben Geschwindigkeit bestimmte Aktionen

ausführen kann. Zu diesem Zweck wurde Abbot [Abb] benutzt, ein Testframework für GUI-Anwendungen, das als Erweiterung von JUnit [Jun] implementiert wurde. Mittels *Reflection* bietet es Zugriff auf alle benötigten GUI-Komponenten sowie außerdem Methoden, um Mausklicks auf denselben zu simulieren. Auf diese Weise ermöglicht es Abbot, Sequenzen von Benutzeraktionen automatisiert auszuführen. Dementsprechend wurde die Ausführungszeit einer Sequenz von 20 Aktionen gemessen. Jede einzelne hatte die Ausführung von gebundenen Basismethoden und die Aktivierung bzw. Deaktivierung von Aspekten zur Folge. Die gesamte Prozedur wurde zehn mal wiederholt, um schließlich das jeweilige Mittel der Laufzeiten zu berechnen.

6.3.3 Ergebnis

Die Messungen haben ergeben, dass die Ausführung des *Hierarchy*-Beispiels in CaesarJ um knapp 30% schneller ist als mit ObjectTeams/Java. Die Micro-Benchmarks in Abschnitt 6.2 hatten bereits gezeigt, dass CaesarJ Performance-Vorteile gegenüber ObjectTeams besitzt. Diese werden nun durch das Macro-Benchmarking bestätigt. Der Unterschied in der Performance macht sich also auch in einer realen Anwendung bemerkbar. Die Gründe für den geringeren Overhead von CaesarJ wurden zum Teil schon im Zusammenhang mit dem Micro-Benchmarking besprochen, zum Teil müssen sie auch noch untersucht werden. In diesem Zusammenhang sei hier auf den Ausblick am Ende dieser Arbeit in Abschnitt 7.2 verwiesen.

Kapitel 7

Zusammenfassung und Ausblick

Im letzten Kapitel dieser Arbeit werden die erlangten Erkenntnisse zunächst noch einmal kurz zusammengefasst. Abschließend folgt ein Ausblick, der mögliche Anknüpfungspunkte für weiterführende Arbeiten herausstellt.

7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde erstmals die Performance der aspektorientierten Programmiersprache ObjectTeams/Java weitreichend untersucht. Für die dafür nötigen Messungen kamen Micro- und Macro-Benchmarking, Profiling sowie im Falle der Webezeitmessung auch eigene Messverfahren zur Anwendung.

Mit Hilfe der erstellten Micro-Benchmarks konnte der Overhead gemessen werden, der zur Laufzeit durch die verschiedenen ObjectTeams-Konzepte verursacht wird. Hier wurde nach *Callins*, *Callouts* und Team(de)aktivierung unterschieden. Die Ausführung von *Callins* verursachte dabei den erheblichsten Overhead. Sowohl bei *Callins* als auch bei *Callouts* ist die implizite Teamaktivierung bzw. -deaktivierung ein wesentlicher Bestandteil des Overheads. Auch bei der impliziten Aktivierung trägt diese neben der Unterscheidung von threadlokaler und globaler Aktivierung zum Overhead bei.

Im Rahmen des Macro-Benchmarking wurden zwei funktionsgleiche Anwendungen untersucht, die zum einen in ObjectTeams/Java und zum anderen in purem Java implementiert waren. Dabei wurde festgestellt, dass die Laufzeit des ObjectTeams-Programms etwa um den Faktor 10 größer

war als die der reinen Java-Applikation. Der Overhead ist also auch in realen Anwendungen deutlich messbar.

Hinsichtlich der Webezeit wurde zwischen Brutto- und Netto-Webezeit unterschieden. Die Zeit für die eigentliche Bytecode-Transformation (Netto-Webezeit) war bei den getesteten Programmen deutlich geringer als die Dauer für das Einklinken in den Klassenladevorgang durch JMangler.

Die durchgeführten Optimierungen haben sich sowohl positiv auf den Overhead zur Laufzeit als auch auf die Webezeit ausgewirkt. Auf weitere mögliche Optimierungen in der Zukunft wird im Ausblick (Abschnitt 7.2) näher eingegangen.

Trotz der Optimierungen hat der Vergleich mit den Sprachen AspectJ und CaesarJ ergeben, dass ObjectTeams/Java zur Laufzeit einen zum Teil deutlich größeren Overhead aufweist als die gewählten Vergleichssprachen. Der Grund dafür sind vor allem einige Konzepte von ObjectTeams/Java, die in den anderen Sprachen nicht oder nur zum Teil existieren. Dazu gehört insbesondere die Möglichkeit, Teams zur Laufzeit zu aktivieren und zu deaktivieren, wobei dies sowohl threadlokal als auch global möglich ist. Ferner verursacht die implizite Aktivierung und Deaktivierung von Teams weiteren Overhead. Schließlich führt auch das Weben zur Ladezeit zu einer messbaren Beeinträchtigung der Programmlaufzeit, die mit einer statischen Webestrategie – wie sie die beiden Vergleichssprachen verfolgen – nicht auftritt.

Sprachen, die ausschließlich mit statischem Weben arbeiten, oder solche ohne die Möglichkeit Aspekte zur Laufzeit zu aktivieren bzw. deaktivieren, bieten deutlich weniger Flexibilität im Umgang mit Aspekten. Die genannten Features von ObjectTeams/Java haben also auch Vorteile. Eine vollständige Auflösung des Zielkonfliktes zwischen dem Nutzeffekt dynamischer Konzepte und einer optimalen Performance ist nicht möglich. Um die Performance-Einbußen jedoch zu minimieren, müssen zwei Strategien verfolgt werden. Zum einen sollte der Laufzeit-Overhead durch Optimierung minimiert werden. Zum anderen erscheint es sinnvoll, neben die dynamischen auch statische Konzepte zu stellen, auf die selektiv zurückgegriffen werden kann. In Anwendungsfällen, in denen keine Dynamik benötigt wird, kann so zugunsten der Performance auf diese verzichtet werden. Im folgenden Ausblick wird auf beide Strategien eingegangen.

7.2 Ausblick

Im Verlauf dieser Arbeit konnte die Performance von ObjectTeams/Java sowohl in Hinblick auf den Webeprozess zur Ladezeit als auch auf den ObjectTeams-spezifischen Overhead zur Programmlaufzeit verbessert werden. Dennoch besteht weiterhin Potenzial für zusätzliche Optimierungen. Diese sind zum Teil abhängig von Arbeiten, die derzeit noch nicht abgeschlossen sind. Dazu gehören die Entwicklung des Joinpoint-Modells für ObjectTeams/Java und die Umstellung vom JMangler-Framework auf JPLIS. Zum Teil wären sie auch über den Rahmen dieser Diplomarbeit hinausgegangen.

Die im Folgenden vorgeschlagenen Optimierungen lassen sich entsprechend der in Abschnitt 7.1 genannten Strategien in zwei Kategorien fassen.

7.2.1 Optimierung der dynamischen Konzepte

Envelope-Based Weaving

In Abschnitt 5.4.1 wurde mit dem so genannten *Envelope-Based Weaving* [BHMM05] ein Ansatz vorgestellt, mit dessen Hilfe durch Einführung von Indirektionen (*Envelopes*) der Webevorgang für bestimmte Arten von Joinpoints optimiert werden kann. ObjectTeams/Java verfügt zum gegenwärtigen Zeitpunkt noch nicht über eine Joinpoint-Sprache. Die einzige derzeit unterstützte Art von Joinpoints ist daher die Methoden*ausführung*. Sobald auch Methoden*aufrufe* sowie lesende und schreibende Feldzugriffe als Joinpoints definiert werden können, ist auch die Implementierung von *Envelope-Based Weaving* in Betracht zu ziehen.

Lifting

Das *Lifting* – also die Zuordnung eines Rollenobjektes zu einem bestimmten Basisobjekt – bildet einen bedeutenden Teil des Overheads, der bei der Ausführung von *Callins* entsteht. Eine Optimierungsöglichkeit ergibt sich hier bei Programmen, in denen eine Rolle an mehrere Basismethoden gebunden ist. In diesem Fall kann es hilfreich sein, das Rollenobjekt in einem *Cache* zu speichern, um so ein wiederholtes Lifting desselben Objektes zu vermeiden.

ASM

Wie in der Arbeit bereits mehrfach erwähnt, wird das BCEL-Framework sowohl von JMangler als auch von der ObjectTeams/Java-Laufzeitumgebung für die Manipulation und Erzeugung von Bytecode genutzt. Mit ASM [BLC02] existiert ein sehr ähnliches Framework. Obwohl es deutlich leichtgewichtiger implementiert ist, weist es im Wesentlichen dieselbe Funktionalität wie BCEL auf. Während die aktuell benutzte Version von BCEL 400kB groß ist, umfasst ASM lediglich 33kB. Daraus resultiert nicht nur eine geringere Download-Größe, sondern vor allem auch eine bessere Performance. Laut Angaben auf den Webseiten von ASM liegt der Overhead für Bytecode-Transformationen zur Ladezeit bei 60% im Vergleich zu 700% mit BCEL [ASM]. Zwar wird nicht näher darauf eingegangen, wie genau dies gemessen wurde. Untersuchungen bestätigen jedoch die deutlich bessere Performance von ASM gegenüber BCEL [Sos05].

Da auch JMangler BCEL benutzt, wäre die vollzogene Umstellung auf JPLIS eine notwendige Voraussetzung für die vollständige Unabhängigkeit vom BCEL-Framework. Ein dann eventuell vorzunehmender Umstieg von BCEL auf ASM verspricht also eine signifikante Verringerung der Brutto-Webezeit.

Neben der zu erwartenden Performance-Verbesserung gibt es noch weitere Gründe, die für einen Umstieg auf ASM sprechen. Um beispielsweise in ObjectTeams/Java auch mit *Generics* arbeiten zu können, wird derzeit eine eigens modifizierte Version von BCEL verwendet, da die ursprüngliche Version diese Möglichkeit nicht bietet. ASM hingegen unterstützt *Generics* von vornherein [Sos06]. Schließlich ist das ASM-Projekt aktiver als dies bei BCEL der Fall ist. So gab es in den letzten drei Jahren nur ein neues *Release* von BCEL, während es von ASM regelmäßige Updates gibt.

Gegen eine Umstellung würde lediglich der zu erwartende Implementierungsaufwand sprechen. Da ein großer Teil der ObjectTeams/Java-Laufzeitumgebung Bytecode mit Hilfe von BCEL bearbeitet, wäre der Umstieg auf ASM eine relativ umfangreiche Aufgabe. Ein existierendes Eclipse-Plugin könnte diese Arbeit jedoch erleichtern. Der so genannte *ASMifier* kann für ein gegebenes Stück Java-Quellcode die nötigen ASM-Anweisungen generieren, die wiederum zur Laufzeit den gewünschten Code erzeugen. Zum anderen fehlen einige wenige Features in ASM, über die BCEL verfügt. Vor einer Umstellung wäre zu prüfen, ob diese von der Laufeitumgebung für die Bytecode-Transformation benötigt werden.

Insgesamt erscheint eine Umstellung von BCEL auf ASM vor allem aus Gründen der Performance dringend angeraten.

Runtime Weaving

In Abschnitt 5.4.2 wurden mit Steamloom und PROSE zwei aspektorientierte Systeme vorgestellt, die das Weben zur Laufzeit mit JVM-Unterstützung realisieren. Dieser Ansatz bietet deutliche Performance-Vorteile gegenüber anderen dynamischen aspektorientierten Systemen. Vor diesem Hintergrund erscheint es sinnvoll, auch für ObjectTeams/Java ein solches Vorgehen in Erwägung zu ziehen.

Ein Nachteil beim JVM-unterstützten dynamischen Weben ist die Abhängigkeit von der verwendeten *Java Virtual Machine*. Die Auswahl der zu benutzenden JVM ist daher eine wichtige Entscheidung. Zum eine würde – wie bei Steamloom und PROSE – die Jikes Research JVM für eine solche Erweiterung in Frage kommen. Als spezielle *Forschungs*-JVM wäre Jikes gut geeignet. Im Hinblick auf die Akzeptanz seitens der Benutzer von ObjectTeams/Java wäre jedoch eine Erweiterung der *HotSpot VM* von Sun Microsystems eventuell von Vorteil. Sun arbeitet derzeit daran, weite Teile von Java als Open-Source freizugeben. Im Rahmen dessen soll auch die *Virtual Machine* quelloffen gemacht werden und würde somit ebenfalls für die Erweiterung in Frage kommen.

Reihenfolge der Transformationen

JMangler ermöglicht die Transformation von Klassen zur Ladezeit. Die so genannten Transformatoren, also jene Klassen, die die eigentliche Bytecode-Transformation durchführen, werden mit Hilfe einer XML-Datei an das Framework übergeben. Die Reihenfolge, in der die Transformatoren ausgeführt werden, ist dabei nicht vorhersagbar. Die Methode `checkReadClassAttributes()` aus der Superklasse aller ObjectTeams-Transformatoren liest vom Compiler abgelegte Informationen aus Bytecode-Attributen aus. Für mehrere Transformatoren ist es entscheidend, dass diese Informationen initial vorliegen. Da jedoch keine Aussage über die Ausführungsreihenfolge der Transformatoren getätigt werden kann, muss diese Methode von allen betreffenden Klassen am Anfang der Transformation aufgerufen werden. Mit der Umstellung auf JPLIS wird auch die Reihenfolge der einzelnen Transformationen steuerbar. Infolgedessen wird es dann möglich sein, die redundanten Aufrufe der Methode „wegzuoptimieren“.

Teamaktivierung

Der Vergleich mit CaesarJ ergab, dass die Aktivierung und Deaktivierung von Aspekten in CaesarJ performanter ist als die Teamaktivierung und -deaktivierung in ObjectTeams/Java (s. Abschnitt 6.2.3). Das Konzept der dynamischen Aktivierung ist in beiden Sprachen trotz geringer Unterschiede prinzipiell vergleichbar. Daher wäre eine genauere Untersuchung der implementierungstechnischen Realisierung in CasarJ Erfolg versprechend in Hinsicht auf die Minimierung des Overheads, den ObjectTeams/Java durch die Teamaktivierung verursacht.

7.2.2 Statische Konzepte

Die zweite Optimierungsstrategie, um den Overhead durch die dynamischen aspektorientierten Konzepte zu reduzieren, stellt neben diese zusätzlich statische Konzepte. Dies kann in verschiedener Weise geschehen, wie im Folgenden dargestellt wird.

Statische Analyse

Die Messungen des ObjectTeams-spezifischen Overheads haben gezeigt, dass die implizite Aktivierung und Deaktivierung von Teams einen signifikanten Anteil an demselben hat. Ein Teil dieser Aktivierungen wird jedoch ohne Effekt vorgenommen. Das liegt daran, dass zur Laufzeit oft nicht mehr festgestellt werden kann, ob eine implizite Aktivierung überhaupt notwendig ist. Dies kann z.B. in folgenden Situationen der Fall sein:

- Beim Aufruf von Rollenmethoden, die als `public` deklariert sind, wird nach § 5.3 der ObjectTeams-Sprachdefinition das umgebende Team immer imiplizit aktiviert. Dies geschieht derzeit auch dann, wenn die Methode durch ein *Callin* aufgerufen wird. In diesem Fall ist das Team also bereits aktiv, da die Methodenbindung sonst nicht wirksam wäre. Eine implizite Aktivierung ist hier also unnötig.

- Für die Dauer eines *Callouts* wird das betreffende Team ebenfalls implizit aktiviert, damit eventuelle *Callin*-Methodenbindungen wirksam werden. Dies ist unabhängig davon, ob es während des *Callouts* eventuell ohnehin bereits explizit aktiviert ist. Ferner wird nicht geprüft, ob im Kontrollfluss des *Callouts* überhaupt Basismethoden enthalten sind, die durch das aufrufende Team gebunden sind. In beiden Situationen ist die implizite Aktivierung also ebenfalls unnötig.

Die geschilderten Programmsituationen könnten durch eine statische Analyse zur Kompilierzeit identifiziert werden. Unnötige implizite Teamaktivierungen wären so zu vermeiden. Es ist allerdings anzunehmen, dass eine solche statische Analyse die Kompilierzeit deutlich negativ beeinflusst. Ideal wäre daher ein Option des Compilers, die es erlaubt, die Optimierung wahlweise auszuführen oder nicht. Während der Entwicklungsarbeit könnte auf die statische Analyse zugunsten der Kompilierzeit verzichtet werden. Die tatsächlich ausgelieferte Software würde dann jedoch die Optimierungen enthalten.

Permanente Teamaktivierung

Die implizite Teamaktivierung ist auch immer dann unnötig, wenn ein Team während seiner gesamten Lebenszeit aktiviert bleibt. Mit den derzeit existierenden Sprachkonstrukten wäre für die Identifikation solcher Teams eine statische Analyse notwendig, wie sie oben kurz beschrieben wurde.

Wenn der Entwickler eines Teams vorsieht, dass dieses permanent aktiv sein soll, könnte dies mittels einer Annotation `@active` deklariert werden. Das Team wäre dann während der gesamten Laufzeit aktiv und könnte auch nicht deaktiviert werden. Für ein so deklariertes Team könnte dann die implizite Aktivierung generell entfallen. In CaesarJ gibt es mit dem Schlüsselwort `deployed` ein ähliches Feature, das hier *static deployment* genannt wird.

Literaturverzeichnis

[AA+05] B. Alpern, S. Augart, , S. M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K. S. McKinley, M. Mergen J. E. B. Moss, T. Ngo, V. Sarkar, and M. Trapp. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Systems Journal*, 44(2), 2005.

[Abb] Abbot Java GUI Test Framework. `http://abbot.sourceforge.net/`.

[ABC] Homepage des AspectBench Compiler-Projektes. `http://www.aspectbench.org`.

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. *SIGPLAN Notices*, 40(6):117–128, 2005.

[AE05] Sufyan Almajali and Tzilla Elrad. Coupling Availability and Efficiency for Aspect Oriented Runtime Weaving Systems. In Filman et al. [FHH05], pages 47–55.

[AJ] AspectJ Homepage. `http://www.eclipse.org/aspectj/`.

[Akş03] Mehmet Akşit, editor. *Proceedings of AOSD '03: 2nd international conference on Aspect-oriented software development*. ACM Press, March 2003.

[ASM] Homepage von ASM. `http://asm.objectweb.org/`.

[BCE] Homepage von BCEL. `http://jakarta.apache.org/bcel/`.

[BHMM05] Christoph Bockisch, Michael Haupt, Mira Mezini, and Ralf Mitschke. Envelope-based Weaving for Faster Aspect Compilers. In Robert Hirschfeld, Ryszard Kowalczyk, Andreas Polze, and Mathias Weske, editors, *NODe/GSEM*, volume 69 of *LNI*, pages 3–18. GI, 2005.

[BHMO04] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In Lieberherr [Lie04], pages 83–92.

[BLC02] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, November 2002.

[Bul00] Dov Bulka. *Java Performance and Scalability, Volume 1*. Addison-Wesley, 2000.

[CB05] Siobhán Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison Wesley, March 2005.

[CJ] CaesarJ Homepage. `http://www.caesarj.org`.

[CJE] CaesarJ Beispiele. `http://caesarj.org/index.php/Caesar/Examples`.

[DGH+04] Bruno Dufour, Christopher Goard, Laurie Hendren, Oege de Moor, Ganesh Sittampalam, and Clark Verbrugge. Measuring the Dynamic Behaviour of AspectJ Programs. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 150–169. ACM Press, October 2004.

[Dij82] Edsger W. Dijkstra. *Selected Writings on Computing: A Personal Perspective*, chapter On the role of scientific thought, pages 60–66. Springer Verlag, 1982.

[EA] Early Aspects Homepage. `http://www.early-aspects.net`.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison Wesley, June 1999.

[FF00] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Proceedings of the Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Minneapolis, October 2000.

[FHH05] Robert Filman, Michael Haupt, and Robert Hirschfeld, editors. *Proceedings of the Second Dynamic Aspects Workshop*, March 2005.

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

[Fow04] Martin Fowler. Is Optimization Refactoring. `http://www.martinfowler.com/bliki/IsOptimizationRefactoring.html`, September 2004.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[GVG04] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code Layout as a Source of Noise in JVM Performance. Technical Report 2004-8, McGill University, School of Computer Science, Sable Research Group, October 2004.

[Hau06] Michael Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. Dissertation, Software Technology Group, Darmstadt University of Technology, 2006.

[Her02] Stephan Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In Mehmet Akşit and Mira Mezini, editors, *Proceedings of Net.ObjectDays*, October 2002.

[HH04] Erik Hilsdale and Jim Hugunin. Advice Weaving in AspectJ. In Lieberherr [Lie04], pages 26–35.

[HHMW05] Stephan Herrmann, Christine Hundt, Katharina Mehner, and Jan Wloka. Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation. In Filman et al. [FHH05], pages 93–101.

[HM04] Michael Haupt and Mira Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. In Mathias Weske and Peter

Liggesmeyer, editors, *Proceedings of Net.ObjectDays*, volume 3263 of *Lecture Notes in Computer Science*, pages 81–96. Springer, September 2004.

[HPJ] Homepage von HPjmeter. `http://www.hp.com/products1/unix/java/hpjmeter/`.

[Jas] JAsCo Homepage. `http://ssel.vub.ac.be/jasco/`.

[JCF] Java Collections Framework. `http://java.sun.com/j2se/1.5.0/docs/guide/collections/index.html`.

[Jik] Jikes^{TM} RVM Homepage. `http://jikesrvm.sourceforge.net/`.

[JM] Homepage des JMangler-Projektes. `http://roots.iai.uni-bonn.de/research/jmangler/`.

[JN04] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, December 2004.

[JPRa] Homepage von JProbe. `http://www.quest.com/jprobe/`.

[JPRb] Homepage von JProfiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.

[Jun] Homepage von JUnit. `http://www.junit.org/`.

[JVMa] Java^{TM} Virtual Machine Profiler Interface (JVMPI). `http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html`.

[JVMb] JVM^{TM} Tool Interface (JVMTI). `http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/`.

[KCA01] Günter Kniesel, Pascal Costanza, and Michael Austermann. JMangler - a framework for load-time transformation of Java class files. In *First IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001)*, pages 100–110. IEEE Computer Society Press, November 2001. ISBN 0-7695-1387-5.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of ECOOP*

2001, LNCS 2072, pages 327–353. Springer-Verlag, June 2001.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings of ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997*, volume 1241 of *Lecture Notes in Computer Science*. Springer, 1997.

[Lie04] Karl Lieberherr, editor. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, March 2004.

[LLM99] Karl Lieberherr, David H. Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA 02115, April 1999.

[Loe00] E. B. Loewenstein. Reducing the Effects of Noise in a Data Acquisition System by Averaging. Application Note 152, National Instruments, April 2000.

[MKD02] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation Semantics of Aspect-Oriented Programs. In *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 17–26, March 2002.

[MO03] Mira Mezini and Klaus Ostermann. Conquering Aspects with Caesar. In Akşit [Akş03].

[O'H04] Kelly O'Hair. HPROF: A Heap/CPU Profiling Tool in J2SE 5.0. `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`, November 2004.

[OT] ObjectTeams Homepage. `http://www.objectteams.org`.

[OTD] Object Teams Development Tooling Download Page. `http://www.objectteams.org/distrib/single.html#otdt.html`.

[OTJ] ObjectTeams/Java Sprachdefinition. `http://www.objectteams.org/def/0.9/`.

[PAG03] Andrei Popovici, Gustavo Alonso, and Thomas Gross. Just-In-Time Aspects: Efficient Dynamic Weaving for Java. In Akşit [Akş03], pages 100–109.

[PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic Weaving for Aspect-Oriented Programming. In Gregor Kiczales, editor, *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 141–147. ACM Press, April 2002.

[Pro] Homepage des PROSE-Projektes. `http://prose.ethz.ch/Wiki.jsp`.

[Ref] Refactoring Homepage. `http://www.refactoring.com`.

[Shi03] Jack Shirazi. *Java Performance Tuning*. O'Reilly, 2nd edition, 2003.

[Sos05] Dennis Sosnoski. Classworking toolkit: ASM classworking. `http://www-128.ibm.com/developerworks/java/library/j-cwt05125/index.html`, May 2005.

[Sos06] Dennis Sosnoski. Classworking toolkit: Generics with ASM. `http://www-128.ibm.com/developerworks/java/library/j-cwt02076.html`, February 2006.

[Ste] Steamloom Homepage. `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/AORTA/Steamloom.jsp`.

[SV03] Davy Suvée and Wim Vanderperren. JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development. In Akşit [Akş03], pages 21–29.

[VS04] Wim Vanderperren and Davy Suvee. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In Robert Filman, Michael Haupt, Katharina Mehner, and Mira Mezini, editors, *DAW: Dynamic Aspects Workshop*, pages 120–134, March 2004.

[VS05] Matthias Vösgen and Dehla Sokenou. Aspektorientierte Programmiertechniken im Unit-Testen. *Informatik - Forschung und Entwicklung*, 20(1-2):57–71, 2005.

[WK00] Steve Wilson and Jeff Kesselman. *Java Platform Performance*. Addison-Wesley, 2000.

Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, 23. August 2006

(Paul Häder)