ERWEITERUNG EINER JAVA VIRTUAL MACHINE UM DELEGATIONSBASIERTE ASPEKTAUSFÜHRUNGSMECHANISMEN ZUR OPTIMIERUNG VON OBJECTTEAMS/JAVA

Diplomarbeit

von
Julian Bischof
Matr. Nr. 178246

Gutachter:

Prof. Dr. Sabine Glesner

Prof. Dr. Stefan Jähnichen

Betreuung durch: Dipl. Inf. Christine Hundt

Berlin, den 02. August 2010

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

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, den

_____

Inhalt

Kapitel 9

Der Delegationproxy
- Callinausführung in OTRun **79**

Kapitel 10

OTRun outruns...
- Zusammenfassung und Benchmarks **102**

Kapitel 11

OTRun, quo vadis?
- Ausblick und Schluss **106**

Einleitung

Aspektorientierung und Delegation

- Adaptabilität für Computerprogramme

1.1 Einleitung - Ein Wunsch nach Adaptabilität?

Die Durchdringung unseres Alltags mit Produkten der Informationstechnik – mit eingebetteten Systemen – hat in den letzten Jahren in den großen Industrienationen bedeutend zugenommen. Handys, Smartphones, Navigationsgeräte, PDAs, Netbooks – iPhones, iPods und iPads, um nur exemplarisch die Apple-Produktpalette zu nennen, werden zunehmend zu Alltagsgegenständen und „Lifestyle"-Artikeln. Um dies festzustellen reicht ein Blick auf die den öffentlichen Raum dominierenden Werbeplakate für entsprechende Produkte und der Gang durch einen Multimediamarkt. Aber auch Alltagsgeräte wie Waschmaschinen, Geschirrspüler, Mikrowellen, Kühlschränke und Herde sowie die unterschiedlichsten Multimediaplattformen sind zunehmend mit leistungsstarker Elektronik ausgestattet und zu kleinen Computern geworden. Und Produkte der Augmented Reality, wie „intelligente" Kleidungsstücke oder Brillen, die unsere visuelle Wahrnehmung mit computergestützten Informationen überlagern sind bereits in der Entwicklung und warten nur darauf, die Berührungspunkte mit eingebetteten Systemen in unserem Alltag auf ein bisher nicht gekanntes Maß zu steigern. Eingebettete Systeme sind schon längst nicht mehr ausschließlich in Industriemaschinen anzutreffen.

Die Geschwindigkeit mit der neue Produkte der Informationstechnik auf diesem Markt erscheinen – und sogleich auch wieder vom Markt verschwinden oder durch Nachfolgeprodukte ersetzt werden - ist beeindruckend. Allerdings stellt sie den Benutzer und auch den Entwickler dieser Geräte und Systeme vor die schwierige Aufgabe, mit dieser Entwicklung Schritt zu halten. Für den Anwender erhöht sich dadurch vor allem der Anpassungsdruck, sich immer wieder auf neue Bedienoberflächen und Applikationen einstellen zu müssen. Der Entwickler dagegen steht vor einer unüberschaubaren Anzahl von Geräten mit sehr

unterschiedlicher Ausstattung und Leistungsfähigkeit und muss Sorge dafür tragen, möglichst viele davon unterstützen zu können.

Bleiben wir aber zunächst beim Anwender. Dieser ist damit konfrontiert, sich immer wieder an neue Produkte gewöhnen zu müssen – und er muss Unpässlichkeiten in Kauf nehmen, wenn das eine Gerät nicht mit dem anderen harmonieren will. Daher ist es nicht immer leicht festzustellen, ob der Mensch die Technik oder die Technik den Menschen beherrscht. Schön wäre es dagegen, wenn Software einfach an die Bedürfnisse des Anwenders angepasst werden könnte. Dieser benötigt also Software, die so weich ist, dass sie umgeformt, angepasst werden kann. So wäre er nicht gezwungen, sich den Eigenwilligkeiten einer Maschine zu beugen. Der Anwender benötigt eine Software, die den Lernaufwand, den er in die Beherrschung seines computerisierten Umfeldes im Laufe seines Lebens investiert hat, wertschätzt. Eine Software, auf die er seine Erfahrungen und Vorlieben in der Bedienung, der Präsentation, dem Funktionsumfang und der Handhabung übertragen kann.

In der Praxis entsteht der soeben beschriebene Anpassungsdruck recht häufig. Besitzen zwei Applikationen neben ihren jeweiligen Spezialfunktionalitäten einen gemeinsamen Kern an Funktionalität , so ist es z.B. möglich, mit Eclipse und mit Visual Studio Programme zu entwickeln oder mit Cubase und mit Logic Audio Musikstücke zu komponieren. Diese Applikationen sind jedoch jeweils sehr umfangreich und ihre effiziente Bedienung muss von einem Benutzer über Jahre hinweg erlernt und trainiert werden. Es wäre schön, wenn ein Anwender diese Erfahrung und sein Wissen von einer Applikation zu einer anderen mitnehmen könnte. Bestehende Softwaresysteme erlauben dies bisher jedoch nur im Ansatz. Sie sind exakt so konfigurierbar, wie es der Entwickler vorausgesehen hat oder wie es das Framework erlaubt, in dem die Applikation abläuft. Besitzen sie dann ein umfangreiches Konfigurationssystem, so ist die Beherrschung der Konfiguration oftmals eine Wissenschaft für sich und die Entwicklung des Systems kostspielig, aufwändig und vor allem sehr komplex.
Wäre Software leichter umzuformen, wäre es praktikabel, dass der Hersteller solche Anpassungen der öffentlichen Nutzergemeinde eines Produktes überlässt und sie darin bestmöglich unterstützt. Der Entwickler könnte sich dann auf seine Kernaufgaben konzentrieren und bräuchte sich nicht mehr um die Vielzahl verschiedener Eingabegeräte und Bildschirmauflösungen oder um die Kompatibilität zum Gerät der Vorserie zu sorgen. Die Nutzergemeinde wiederum würde unabhängig von den Produktphilosophien einzelner Hersteller und könnte selbstständig darüber befinden, welche Form ein Produkt letztendlich besitzen soll.

So ist der Anwender, der eine anpassungsfähige Software verwenden kann, z.B. darin flexibel geworden, dass er die Bedienphilosophie, die er bei einem Produkt kennen gelernt hat, auf ein anderes Produkt übertragen kann - oder als Dienstleistung übertragen lassen könnte, selbst wenn es sich bei dem neuen Gerät um ein Produkt der Konkurrenz handelt. Er könnte sich sogar eine Benutzeroberfläche maßschneidern lassen, die ihn dann sein Leben lang begleiten würde und die er sogar schrittweise an seine im Alter nachlassende Wahrnehmung anpassen könnte. Auch für Menschen, die über bestimmte Wahrnehmungssinne oder Bedienmöglichkeiten nicht oder nur eingeschränkt verfügen können, mag es

hilfreich sein, wenn es direkt möglich ist, die von ihnen verwendeten Computerprogramme und elektronischen Geräte an ihre individuellen Bedürfnisse anzupassen, indem ihre Benutzeroberfläche beispielsweise um akustische Rückmeldungen erweitert wird. Viele Haushaltsgeräte sind bereits kleine eingebettete Systeme, die dies unterstützen könnten, doch ist die Gruppe derjenigen, die solche Sonderfunktionen benötigen, zu gering um marktwirtschaftlich für viele Hersteller relevant zu sein, so dass sie in das Pflichtenheft des Softwareentwicklers keinen Eingang finden. Daher wäre es gut, wenn sich die Nutzergemeinschaft eines solchen Gerätes wenigstens selbst zu den Features verhelfen könnte, die sie wirklich braucht.

Nicht zuletzt führt die zunehmende Integration und Verbreitung von eingebetteten Systemen im Alltag – nicht nur als unsichtbare Helferlein, sondern auch als Statussymbol und multimediale Visitenkarte, dazu, dass sich ein Bedürfnis herausbildet, diesen Geräten eine persönliche, individuelle Note zu geben.

Es gibt also viele Gründe für den Wunsch nach anpassbarer Software, die vom Benutzer neu und frei gestaltet werden kann. Die Existenz der OpenSource- und freien Software-Bewegung demonstriert das Bestehen dieses Wunsches bereits eindrücklich . Aber kann der Wunsch nach weitreichenderer Anpassbarkeit von Software auch zur Wirklichkeit werden?

1.2 Grenzen der Objektorientierung

Die Realisierung dieses Wunsches liegt bisher ausschließlich in den Händen der Entwickler von Applikationen. Diese sind beispielsweise mit dem ausgeprägt heterogenen Markt mobiler eingebetteter Systeme konfrontiert und müssen für die unterschiedlichen Wünsche der Anwender und des Marktes praktikable, zuverlässige Lösungen finden. Das Problem zunehmender Komplexität, das sich ihnen dadurch stellt, ist in der Softwaretechnik generell kein neues Problem. Es ist aber auch kein gelöstes Problem, da keine Silberkugel[1] in Sichtweite ist, mit der alle Komplexität mit einem Schuss erschlagen werden könnte. Der Werwolf Komplexität schickt sich in immer wieder neuer Gestalt an, den Informatiker in Schrecken zu versetzen und sein Umfeld in Entsetzen zu stürzen, wenn es mit den Produkten leben muss, die dieser zu verantworten hat.

[1] <Siehe auch „No Silver Bullet“, [Br95]. Dieser Artikel stellt die Suche nach einer allumfassenden Lösung zur Beherrschung der Softwarekrise als den Wunsch nach einer Silberkugel dar, die einen Werwolf erschlagen soll.

Edsger W. Dijkstra schreibt dazu bereits 1972 in seiner Turingpreis-Rede[2]:
„as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them, it has created the problem of using its products. To put it in another way: as the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between ends and means. The increased power of hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible, that the programmer had not dared to dream about a few years before. And now, a few years later, he had to dream about them, and, even worse, he had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis? No, certainly not, and as you may guess, it was even predicted well in advance; but the trouble with minor prophets, of course, is, that it is only five years later that you really know that they had been right."

Aber es gibt auch so manchen Hoffnungsschimmer, der durch das Dickicht der Programmzeilen und Anforderungen scheint. So hilft das Prinzip der Modularisierung den Wald aus Methoden und Variablen auszulichten und in einen wohlgeordneten Park zu transformieren, in dem der Entwickler auf breiten Alleen flanieren kann und nur bei Bedarf einen Abstecher in das nebenan liegende Labyrinth eines Moduls zu tätigen braucht, das hinter hohen Hecken vor all zu neugierigen Blicken verborgen bleibt.

Doch leider will es nicht gelingen, die ganze Welt in solch kleinteilige, modularisierte Kästchen, in Klassen zu verpacken, dazu ist die Welt zu vielgestaltig. Schon bei kleinen Aufgabenstellungen, wie sie für die Informatik so sehr typisch sind, wie z.B. der Frage danach, ob im Rahmen einer Verwaltungssoftware ein Mensch nun ein Student ist, oder ein Mitarbeiter, fällt dies schwer. Nicht alle Menschen sind Studenten und nicht alle Menschen sind Mitarbeiter. Manche sind beides, manche sind eine Zeitlang das eine und dann eine Zeitlang das andere - und manche sind ihr ganzes Leben lang sogar weder das eine noch das andere, aber natürlich dennoch Menschen. Menschen spielen, wie in diesem kleinen Beispiel zu erkennen ist, in ihrem Leben also ganz verschiedene Rollen. Und sie spielen diese Rollen in ganz unterschiedlichen Kontexten. Student sind sie an der Universität, und das in der Regel nur für eine bestimmte Zeitperiode, Mitarbeiter sind sie vielleicht an der Universität oder aber an einem anderen Ort – und Mensch sind sie glücklicherweise ihr ganzes Leben lang. Hinzu kommen Multiplizitäten. Vielleicht arbeitet der Student an zwei Fachbereichen als studentische Hilfskraft. Oder er studiert zwei Fächer. Vielleicht hat er auch ein Fach studiert und studiert nun ein anderes? Viele Begrifflichkeiten der Realität oder eines abstrakten Modells lassen sich nicht eindeutig einer bestimmten Kategorie zuordnen. Eine klare Kategorisierung von Objekten in Klassen fällt also in bestimmten Fällen sehr schwer, insbesondere dann, wenn auch noch variable Multiplizitäten in diese Modellierung einer Welt hineinspielen. Diese müssen bei

[2] aus "The humble programmer", [Dij72].

der Implementierung dann durch abstrakte Datenstrukturen realisiert werden, wodurch sich die innere Komplexität einer Klasse weit von ihrer formell noch eleganten Modellierung entfernt. Daher gelangt auch das Prinzip der Modularisierung und das Prinzip der Spezialisierung durch Vererbung, wie es in der Objektorientierung verwendet wird, mitunter an seine Grenzen, obwohl es nach wie vor an vielen Stellen hervorragende Arbeit leistet, um die Komplexität eines Systems auf ein überschaubares Maß zu reduzieren. Wir werden im 2. Kapitel sehen, wie die aspektorientierte Programmiersprache ObjectTeams/Java diese Grenze überwinden will.

Denn die bisher weit verbreitete Lösung, mit solchen Grenzfällen bei der Modellierung einer Software umzugehen, besteht nun darin, aus den einzelnen Zutaten, die alle ihren eigenen Geschmack mitbringen, einen Eintopf zu kochen, und darauf zu hoffen, dass er dann jedem im jeweiligen Kontext schmeckt. Klassen werden so durch Aggregation mit anderen Klassen und der Erweiterung ihrer Interfaces soweit aufgebläht, bis sich aus dieser Suppe eine Eierlegende Wollmilchsau erhebt, oder die Suppe gänzlich versalzen ist. Beides ist möglich, die Gefahr für letzteres jedoch allgegenwärtig, „Featureitis“ eine Erkrankung die so manche Applikation, so manche schön konstruierte Vererbungshierarchie, so manche Klasse mit zunehmendem Lebensalter dahingerafft hat.

Das Paradigma der Aspektorientierung will nun ein Gegenmittel gegen diese Krankheit bereitstellen.

1.3 Aspektorientierung

Die Problemsituation, der sich die Aspektorientierung annehmen will, lässt sich anschaulich an dem Beispiel der „Eierlegenden Wollmilchsau“ demonstrieren, jener Chimäre, als die man eine Klasse bezeichnen könnte, die mehrere Probleme auf einmal lösen soll. Da bei einer solchen Chimäre niemand so recht weiß, woran er ist, bleibt es schwer, sie zu beherrschen und schwierig diesen Geist, den man da als alchemistisch tätiger Zauberlehrling aus seinem Anforderungskatalog zu sich gerufen hat, wieder loszuwerden, wenn er das System mit so mancher Abhängigkeit, so mancher potenziellen Fehlerquelle zu überschwemmen droht.

Eine Lösung des Problems bestünde darin, den einzelnen Aufgabenstellungen wieder ihren eigenen Freiraum geben zu können und so statt auf eine Eierlegende Wollmilchsau, auf Hühner, Kühe und Schafe zu setzen. Jedes dieser Tiere erfüllt dann genau eine Aufgabe und stellt zugleich für den Bauern auch nur genau eine Aufgabe dar, einen einzelnen *concern*, den er artgerecht pflegen kann. Die Pflege einer Chimäre, eines Tieres, das mehrere Tiere in sich vereint, einer Klasse, die verschiedene Anforderungen erfüllen soll, also *Crosscutting Concerns* beinhaltet, wie sie in der Welt der Aspektorientierung genannt werden, fällt da nicht nur einem Bauern, sondern auch einem Softwareentwickler bedeutend schwerer.

Deshalb ist es das Ziel der Aspektorientierung, Werkzeuge anzubieten, mit denen verhindert werden kann, dass in bestimmten Modellierungssituationen Klassen mit Crosscutting Concerns, Chimären, ins Leben gerufen werden müssen. Zugleich

will sie Werkzeuge anbieten, wie ein bestehendes System modular um Crosscutting Concerns erweitert werden kann, ohne dass es dabei zur Bildung von Chimären kommt.

Die Aspektorientierung möchte dies durch neuartige Konzepte zur Modularisierung erreichen, die dort ansetzen, wo das objektorientierte Paradigma keine wirkungsvollen Werkzeuge zur Software-Modellierung anbietet. In [FF00] wird dazu der Satz geprägt, dass es sich bei der Aspektorientierung um „*Quantification and Obliviousness*“ handelt. Um *Quantification*, weil das Modularisierungskonzept der Aspektorientierung daraus besteht, an einem Punkt des Programms alle Stellen zu benennen, d.h. zu *quantifizieren*, an denen eine bestimmte Funktionalität in das System eingefügt werden soll. Um *Obliviousness*, weil die Stelle, an der die Änderung erfolgt, von dieser Änderung nichts wissen soll und auch nicht vom Programmierer manuell verändert zu werden braucht. Stattdessen wird ein *Crosscutting Concern* in einer modularen Einheit, dem *Aspekt*, gekapselt. Darin enthaltene *Pointcuts* quantifizieren die Orte der Veränderung, die *Joinpoints*. *Advices* beschreiben schließlich die Funktionalität, die an diesen Orten nun zusätzlich eingefügt werden soll.

1.4 Delegation

Nun bedarf es jedoch noch eines Mechanismus, der die Advices mit den passenden Joinpoints verbindet, damit die Advices auch zur Laufzeit ausgeführt werden können. Dafür existieren zwei Alternativen: Bei der einen fügt ein *Weaving*-Mechanismus an den quantifizierten Stellen neuen Programmcode ein, bevor der Programmfluss diese Stelle erreicht. Dies kann statisch beim Kompilieren des Programms geschehen oder während ein Programm geladen wird. Auch zur Laufzeit könnte ein bereits bestehendes Programmfragment rekompiliert und dabei um Advices ergänzt werden.

Die andere Alternative kann man als *delegationsbasierten* Mechanismus bezeichnen. Im Gegensatz zum Weaving fügt dieser Mechanismus einem Programm keine neuen Codezeilen oder ein maschinensprachliches Äquivalent hinzu, sondern verknüpft dagegen datengesteuert Programmelemente miteinander. Die beiden Mechanismen Weaving und Delegation verhalten sich damit zueinander in etwa so, wie eine Collage zu einem Hypertext. Während bei einer Collage einzelne Bildteile zerschnitten und mit neuen Bildteilen kombiniert wieder zusammengeklebt werden, so werden bei einem Hypertext Verknüpfungen zu neuen Texten eingefügt, die wiederum Verknüpfungen zu weiteren Texten enthalten können. Zwar ist auch dabei eine Form von Weaving notwendig, um die Verknüpfung selbst einzufügen, dabei wird jedoch nicht der neue Inhalt selbst – also der Advice – eingefügt, sondern nur ein Mechanismus, der es erlaubt, neue Inhalte an dieser Stelle miteinander verknüpfen zu können. Der besondere Vorteil eines delegationsbasierten Mechanismus liegt daher vor allem in seiner Flexibilität. Ein Programm kann mit Hilfe von delegationsbasierten Mechanismen zur Laufzeit mit nur geringem Aufwand verändert werden, so dass Software verformbar wird, reformierbar. Und dies ist eine Voraussetzung, um den Wunsch nach von dem Benutzer anpassbarer Software irgendwann erfüllen zu können.

1.5 ObjectTeams/Java

Die Programmiersprache ObjectTeams/Java, die wir im nächsten Kapitel kennen lernen werden, ist eine aspektorientierte Programmiersprache, die sich dem Problemfeld der Crosscutting Concerns annimmt. Sie erlaubt die Anpassbarkeit bestehender Software-Programme, indem die Funktionalität einzelner Klassen eines Programms nachträglich dynamisch und modularisiert verändert werden kann. ObjectTeams/Java ist damit zwar keine Silberkugel, die den Werwolf der Komplexität erschlägt, bietet aber mit ihren Modularisierungskonzepten eine feste Burg mit guter Wehr und Waffe gegen Chimären.

ObjectTeams/Java besitzt zurzeit jedoch zwei Eigenschaften, die den Einsatz im embedded Bereich – und den Einsatz zur Realisierung von anpassbarer, veränderbarer Software – erschweren. Zum einen sind das ihre geringe Effizienz und ihr hoher Ressourcenverbrauch, wie in [Hä06] gezeigt wurde, zum anderen ihre Inflexibilität, Teamklassen dynamisch zur Laufzeit nachladen zu können. Daher ist es bisher nicht direkt möglich, Applikationen mit ObjectTeams/Java zu entwickeln, bei denen erst der Anwender zur Laufzeit einer Applikation entscheidet, welche Klassen er dazu laden möchte. Mithilfe von delegationsbasierten Mechanismen soll in dieser Arbeit nun versucht werden, die Programmiersprache ObjectTeams/Java zu optimieren. Zugleich soll sie mit diesen Mechanismen einen kleinen Beitrag dazu leisten können, den Wunsch nach anpassbarer Software, die von der Nutzergemeinde flexibel umgestaltet werden kann und der dieser Arbeit vorangestellt wurde, irgendwann Realität werden zu lassen.

Mit Hilfe der vorgestellten delegationsbasierten Mechanismen soll es schließlich ermöglicht werden, die aspektorientierten Modularisierungsmöglichkeiten von ObjectTeams/Java zusammen mit ihren Möglichkeiten zur Entwicklung anpassbarer, wieder verwendbarer und gut wartbarer Software auch im vielgestaltigen Segment der eingebetteten Systeme einsetzen zu können, die oftmals besonderen Einschränkungen bezüglich ihrer Ressourcen unterliegen.

1.6 Aufbau der Arbeit

Im nun folgenden Kapitel 2 soll zunächst die Sprache ObjectTeams/Java in ihren Grundzügen vorgestellt werden. Anschließend betrachten wir in Kapitel 3 die Java Virtual Maschine, die technische Grundlage zur Ausführung von ObjectTeams/Java-Programmen. Dies geschieht am konkreten Beispiel der JamVM, einer kompakten, portablen virtuellen Maschine, die sich gut zum Einsatz in eingebetteten Systemen eignet. Sie stellt das Framework da, in das die Optimierungen für ObjectTeams/Java, die in dieser Arbeit vorgestellt werden, integriert werden. In Kapitel 4 wird die gewählte Lösungsstrategie, die Verwendung delegationsbasierter Mechanismen zur Optimierung, vorgestellt, mit der eine Verbesserung der Laufzeit von ObjectTeams/Java-Applikationen erreicht

werden soll. Die daran anschließenden Kapitel betrachten diese Lösungsstrategie und ihre Realisierung mit Hilfe der dazu im Rahmen dieser Arbeit entwickelten JamVM-Erweiterung OTRun im Detail. In Kapitel 5 wird erläutert, wie die Kommunikation zwischen ObjectTeams/Java und der virtuellen Maschine bzw. OTRun umgesetzt werden kann. Kapitel 6 erläutert, wie der Klassenladevorgang erweitert wurde, um der neuen Laufzeitumgebung OTRun alle benötigten Informationen über ein ObjectTeams/Java-Programm zur Verfügung stellen zu können. Kapitel 7, 8 und 9 betrachten jeweils die Umsetzung eines der grundlegenden Mechanismen von ObjectTeams/Java. In Kapitel 7 wird das Lifting, in Kapitel 8 die Teamaktivierung und in Kapitel 9 schließlich die Callin-Ausführung mit Hilfe des *DelegationProxy* ausführlich dargestellt. Kapitel 10 fasst dann den Lösungsansatz noch einmal zusammen und stellt Benchmarkergebnisse vor, anhand derer beurteilt werden kann, ob das Optimierungsziel erreicht wurde. Kapitel 11 ist den noch offen gebliebenen Punkten der Umsetzung gewidmet und Kapitel 12 betrachtet schließlich das wissenschaftliche Umfeld dieser Arbeit und beendet sie mit einem Fazit.

Kapitel 2

Die Sprache ObjectTeams/Java

- Aspektorientierte Modularisierung

2.1 Modularisierung

Die Softwaretechnik hat in ihrer kurzen Geschichte bereits viele Konzepte hervorgebracht, um durch kleine Schritte den Krisenherd Komplexität so gut es jeweils geht, bestmöglich einzudämmen. Eines dieser Konzepte ist das Konzept der Modularisierung. Indem Software in voneinander möglichst unabhängige Module aufgeteilt wird, sinkt die Komplexität des Systems. Das System wird wieder beherrschbar. Erreicht wird dies durch eine möglichst lose und geringe Koppelung der einzelnen Komponenten, bewirkt durch festgelegte Schnittstellen und Zugriffskontrollen. Diese schirmen den Zugriff auf interne Elemente ab und erreichen es dadurch, den Inhalt eines Moduls syntaktisch und semantisch vom restlichen System zu entkoppeln. Ein System mit einer hohen Kopplungsrate, welches zum Beispiel 32 boolesche Variablen besitzt, die je nach Inhalt und Kombination das gesamte System auf sehr unterschiedliche Art beeinflussen können, kann sich in insgesamt 2^{32}, über 4 Milliarden Zuständen befinden. Dies ist nicht beherrschbar. Nun könnte man das System in Module aufteilen, z.B. in zwei voneinander unabhängige Module mit jeweils 16 booleschen Variablen. Damit kann dann garantiert werden, dass diese 16 Variablen niemals den Zustand des jeweils anderen Moduls beeinflussen werden, da zwischen ihnen keine Bindung besteht. Jedes Modul für sich betrachtet kann sich nun nur noch in einem von 2^{16}, also ca. 64000 Zuständen befinden. Möchte man nun alle Zustände beider Module testen, so reichen 2^{17}, ca. 128000 Tests aus. Im Vergleich zu den 4 Milliarden Möglichkeiten eines nicht modularisierten Systems ist dies ein erheblicher Gewinn.

Durch eine hierarchische Gliederung des Systems lässt sich diese Komplexität weiter reduzieren, da jedes Modul dadurch erneut in eine überschaubare Menge modularer Subsysteme aufgeteilt werden kann. Noch weiter steigern lässt sich diese Komplexitätsreduktion, indem bereits bestehende und getestete Module wieder verwendet werden, bestehende Module also in neuer Kombination zu einem anderen System zusammengefügt werden. Schließlich können Module durch

Veränderung oder Erweiterung auch an neue Aufgabenstellungen angepasst werden. Durch eine lose Kopplung ist nun garantiert, dass diese Änderungen sich zu jedem Zeitpunkt nur auf einen eingegrenzten Teil des Gesamtsystems auswirken können.

Das Paradigma der Objektorientierung erlaubt eine solche Modularisierung und erweitert zudem die Softwareentwicklung um eine leicht verständliche Metapher, der Rede von „*Objekten*“ und von „*Klassen*“. Diese Metapher ist allgemeinverständlich und kann daher auch im Gespräch mit Auftraggebern alltagssprachig angewendet werden um ein System daraufhin zu untersuchen, aus welchen Objekten es besteht, welche Eigenschaften diese haben und wie diese Objekte miteinander interagieren.

Mit der Vorstellung des Begriffs der Crosscutting Concerns haben wir im ersten Kapitel eine Situation kennen gelernt, in der diese Möglichkeiten der Modellierung allein nicht ausreichend sind. In solchen Situationen degeneriert eine Klasse zu einer „Chimäre“, einem Container für semantisch sehr unterschiedliche Konzepte, die nur noch durch diese Klasse selbst zusammengehalten werden. Damit wird sie selbst zu einem nichtmodular aufgebauten Subsystem mit einer hohen, unüberschaubaren Menge von Zuständen. Da eine Klasse in der Objektorientierten Programmierung jedoch die kleinstmögliche Modularisierungs-Entität darstellt, fehlt hier eine Möglichkeit feingranularer, auf Subklassenebene modellieren zu können. Zusätzlich fehlt der Sprache Java ein ausgereiftes Konzept, um aus Klassen eine Modulhierarchie zu formen.

2.2 ObjectTeams/Java

Die aspektorientierte Programmiersprache ObjectTeams/Java, die nun kurz vorgestellt werden soll, erweitert Java um die Möglichkeit der Modellierung auf Subklassenebene und ein Konzept zur klassenübergreifenden Modularisierung in beliebig vielen Hierarchieebenen. Ein Entwickler kann seine Sicht auf das System so von der mikroskopischen Ebene bis zu einer selbst gewählten makroskopischen Ebene beliebig skalieren und besitzt Werkzeuge um das System auf jeder dieser Ebenen modellieren zu können. Darüber hinaus ergänzt die Sprache ObjectTeams/Java die objektorientierte Modellierung aber auch um eine neue Metapher, um die Sprache von „*Rollen*“ und „*Teams*“, mit der Crosscutting Concerns auf einfache Weise umgangssprachlich erfasst werden können.

2.2.1 Die Team- und Rollen-Metapher

Viele Entitäten der Welt spielen je nach dem Kontext, in dem sie sich befinden, eine jeweils andere Rolle. Ein Mensch kann an der Universität ein Student sein und im Rahmen einer Betätigung als studentische Hilfskraft ein Mitarbeiter. Jeder Rolle können Informationen zugeordnet sein, die für die jeweilige Rolle von spezifischer Bedeutung sind. So besitzt ein Student eine Matrikelnummer, als Mitarbeiter mag dagegen seine Versicherungsnummer von größerem Interesse sein, die wiederum im Studierenden-Kontext keine Rolle spielt. Mit Rollen lassen sich auch Multiplizitäten ausdrücken. So kann es sein, dass ein Student in mehreren Studiengängen eingeschrieben ist. Dann besitzt er für jeden Studiengang einen eigenen Satz von Leistungsnachweisen. Solche Multiplizitäten werden in ObjectTeams/Java über Teaminstanzen modelliert. So kann jeder Studiengang als ein Team bestehend aus Lehrkräften, Räumen, Verwaltungsangestellten, studentischen Hilfskräften, Studenten etc. modelliert werden. Unser Beispielstudent würde dann in zwei dieser Teaminstanzen, in zwei Studiengängen eine Rolle spielen – auf ObjectTeams/Java zurückübertragen würde das bedeuten, dass eine Rolleninstanz existiert, die seinen Studentenstatus in dem einen Studiengang beschreibt – und eine Rolleninstanz mit den Daten für den anderen Studiengang. Für Studiengänge, in denen er sich nie immatrikuliert hat, existiert dagegen kein Datensatz, dieser wird schließlich erst bei der Immatrikulation angelegt. So verhält sich auch ObjectTeams/Java. Eine Teamklasse Studiengang würde eine Methode immatrikuliere(Person as Student pAsSt) anbieten. Sobald eine Person das Immatrikulationsbüro bzw. die entsprechende Methode verlässt, besitzt sie den passenden Datensatz um in Zukunft die Rolle des Studierenden in dem entsprechenden Studiengang spielen zu können. Besucht dieser Student das nächste Mal einen Kurs, so sind der entsprechenden Fakultät sofort alle Informationen aus seiner Immatrikulation wieder verfügbar und er kann einen Leistungsnachweis erwerben, der dann in diesem Datensatz gespeichert wird. Zu Hause hat er jedoch keinen Zugriff darauf, kann also seine Noten nicht fälschen, sie sind sicher vor ihm verwahrt. Dies ist dadurch gewährleistet, dass Rolleninstanzen nur demjenigen Kontext zugänglich sind, der sie auch erzeugt hat, nicht aber auch automatisch demjenigen, der die Rolle spielt. Auf diese Weise ist es ObjectTeams/Java möglich, auf chimärenartige Klassen verzichten zu können. Zwar besitzt eine Entität nun je nach Kontext sehr verschiedene Eigenschaften, diese bleiben aber sauber voneinander getrennt, so dass sich die Kontextinformationen nicht versehentlich mischen können. Sie stehen jeweils nur einem konkreten Kontext direkt zur Verfügung. ObjectTeams/Java erlaubt damit die schichtweise Modellierung von Objekten – jeder Kontext, in dem sich das Objekt aufhält, fügt dem Objekt eine weitere Schicht hinzu. Wir konnten soeben am Beispiel des Studierenden sehen, dass die Team- und Rollenmetapher dort eine ganz natürliche, leicht nachzuvollziehende Modellierung erlaubte. Was aber sind nun Teams und Rollen im Kontext der Programmiersprache ObjectTeams/Java?

2.2.2 Team-, Rollen- und Basisklassen

Ein Team, d.h. eine Teamklasse ist in ObjectTeams/Java eine Containerklasse für Rollenklassen, die das Zusammenspiel einzelner Rollenklassen orchestriert und

dirigiert. Da eine Teamklasse selbst eine Rollenklasse sein kann, können Teams als Mittel zur hierarchischen Komposition angewendet werden. Sie erfüllen damit die Aufgabe eines Moduls. Bei Rollenklassen handelt es sich um innere Klassen einer Teamklasse. Sie besitzen in ObjectTeams/Java die spezielle Fähigkeit an Basisklassen *binden* zu können. Dazu stellt ObjectTeams/Java das Schlüsselwort `playedBy` zur Verfügung. Eine Bindung an eine Basisklasse hat den Effekt, dass die Basisklasse um die Funktionalität und um die Felder der Rollenklasse anwächst. Diese neue Funktionalität und die neu hinzugewonnenen Felder der Basisklasse sind jedoch nur im Kontext der entsprechenden Team- oder Rollenklasse direkt zugänglich. Dies wird auch als *Encapsulation* bezeichnet. Umgekehrt enthält eine Rollenklasse über *Callouts* jedoch vollen Zugriff auf alle Felder und Methoden der Basisklasse. Sie kann sich dabei auch freiwillig auf eine Teilmenge beschränken, indem sie nur die Callouts deklariert, die sie auch benötigt. So sind die Abhängigkeiten zwischen einer Rollenklasse und einer Basisklasse ganz unmittelbar durch die Liste aller Callout-Deklarationen ersichtlich. Die Verwendung von Callouts wird auch als *Decapsulation* bezeichnet. Für den Rest des Systems erscheint die Basisklasse nach einer Rollenklassenbindung an sie jedoch zunächst unverändert. Erst durch Callins, die in Abschnitt 2.2.3 vorgestellt werden, ist auch das Verhalten einer Basisklasse beeinflussbar.
Nun soll für diese Arbeit noch eine sprachliche Abkürzung eingeführt werden. Wird von den Basisklassen eines Teams oder von den Basisklassen, an die ein Teamklasse bindet, gesprochen, so meint dies die Basisklassen, die von den Rollenklassen der Teamklasse gebunden werden.

Das kleine Studierenden-Beispiel aus Abschnitt 2.2.1 sieht in der Sprache ObjectTeams/Java in etwa so aus:

```
public team class Studiengang
{
      Person[]      Lehrkräfte;
      Person[]      Angestellte;
      Person[]      Studenten;
      int           AktuelleMatrikelnummer;

   protected class Student playedBy Person
   {  protected int        Matrikelnummer;
      Studienleistung[]    Unterlagen;

      protected void archiviereUnterlage(Studienleistung Formular){}
      callin void schreibeHausarbeit(){}
      schreibeHausarbeit <- replace geheZurParty;
   }

  protected class Mitarbeiter playedBy Person
  {   protected int Versicherungsnummer; }

  void immatrikuliere(Person as Student PasSt)
  {   PasSt.Matrikelnummer = AktuelleMatrikelnummer++; }

  void stelleHilfskraftEin(Person as Mitarbeiter stmb, int vn)
  {   stmb.Versicherungsnummer = vn; }
```

```
int besucheKurs(Person as Student PaSt)
  {   Studienleistung Unterlage = new Studienleistung();
      PaSt.archiviereUnterlage(Unterlage);
      return(Unterlage.Note);
  }
}

public class Person { public void geheZurParty(){} }

public void Studentenleben(Person Student)
{
  Informatik.immatrikuliere(Student);
  Theologie.immatrikuliere(Student);
  Informatik.stelleHilfskraftEin(Student, 12345);

  Note = Informatik.besucheKurs(Student);
  Student.geheZurParty();
}
```

Der Student, der dieses Studentenleben führt, wäre nun in zwei Studiengängen eingeschrieben und würde zusätzlich die Rolle einer Hilfskraft im Studiengang Informatik spielen. Die Instanz Student der Klasse Person wäre dadurch mit zwei Studenten-Rolleninstanzen für den jeweiligen Studiengang assoziiert und mit einer Mitarbeiter-Rolleninstanz. Die Unterlagen seiner Studienleistungen werden sicher vor ihm in der Studenten-Rolleninstanz aufbewahrt. Die Person Student erfährt nur ihre Note. Für den Kontextwechsel zwischen einer Basisinstanz und einer an sie gebundenen Rolleninstanz führt die Programmiersprache ObjectTeams/Java das Konzept des *Translation Polymorphism* ein. Dieses beschreibt die Korrektheit von Abbildungen zwischen Rollen- und Basisklassen und wie diese Abbildungen mit Hilfe des *Liftings* und des *Lowerings* für Instanzen dieser Klassen realisiert werden können. In Kapitel 7 werden diese Konzepte ausführlich dargestellt.

2.2.3 Callins

Die kleinste Modularisierungseinheit, die in Java genutzt werden kann, ist eine einzelne Klasse. Durch die Bindung von Rollenklassen an Basisklassen ist es in ObjectTeams/Java bereits möglich, Klassen als eine Menge von disjunkten Schichten zu modellieren, die in unterschiedlichen Kontexten Verwendung finden. Mit Hilfe der Aspektorientierung kann das Konzept der Modularisierung nun auch noch auf einzelne Methoden angewendet werden, indem einzelne Methoden einer Klasse durch andere Methoden ersetzt werden. Durch Overriding wäre dies zwar prinzipiell auch in Java selbst möglich, wäre dann jedoch auf den Zugang zu einer „*Factory*“[3] beschränkt, die passende Objekte einer anzugebenden Subklasse erzeugt. Die möglichen Erweiterungspunkte, *hooks* genannt, müssten also bereits zur Entwicklungszeit des Programms eingeplant werden. Dies ist jedoch nicht immer möglich, insbesondere dann nicht, wenn ein System mit agiler Methodik entwickelt wird und die Zielvorgaben des Programms unbekannt sind , da sie sich erst im Verlauf des Projektes, möglicherweise sogar erst im Verlauf seines Lebenszyklus herausbilden.

[3] Siehe auch [GHJV95]

Rollenklassen besitzen daher die Möglichkeit, auch das Verhalten einer Basisklasse zu verändern, indem sie *Callins* deklarieren. Damit kann das Verhalten einer Basisklasse auch nachträglich, auf Subklassenebene von außen verändert werden, ohne dass der Programmcode der Klasse umgeschrieben werden müsste. Callins stellen das ObjectTeams/Java-Äquivalent zu Pointcuts in anderen aspekt-orientierten Sprachen dar. Sie erlauben jedoch nur eine eingeschränkte Form der Quantifizierung, weshalb die Menge der Joinpoints, die ein Callin beschreibt, übersichtlich bleibt. Callins können nur das Verhalten von Methoden derjenigen Basisklasse verändern, an die ihre Rollenklasse gebunden ist. Dies umfasst allerdings auch alle Methoden, welche die Basisklasse sowohl erbt als auch vererbt, da Callin-Bindungen auch die Subklassen einer Basisklasse betreffen. Auf die Superklasse einer Basisklasse haben Callins jedoch keine Auswirkung.

Zur Veränderung des Verhaltens von *Basismethoden* stehen drei Varianten zur Verfügung. Über *BEFORE-Callins* können Methoden der Rollenklasse vor der Ausführung einer bestimmten Basismethode ausgeführt werden. Analog dazu können *AFTER-Callins* nach dem Ausführen einer Basismethode aufgerufen werden. Schließlich können mit *REPLACE-Callins* Methoden der Basisklasse auch komplett ersetzt werden. Die REPLACE-Methode entscheidet dann durch die Ausführung eines *Basecalls*, der die Basismethode aufruft, darüber, ob die Basismethode nur von dem Programmcode der REPLACE-Methode umrahmt wird oder ob die REPLACE-Methode sie komplett ersetzt.

Durch die Verwendung eines Callins gewinnt der Student aus dem Beispiel im Abschnitt 2.2.2 bei seiner Immatrikulation nicht nur neue Rolleninstanzen hinzu, sondern sie hat auch noch weitreichendere Folgen für ihn, denn sein Verhalten ändert sich dadurch nun. Vielleicht weil mit der Immatrikulation das Verantwortungsgefühl für seine neue Rolle als Student anstieg... jedenfalls hat die Immatrikulation im Beispiel zur Folge, dass unser Beispielstudent nun immer dann, wenn er als Person eigentlich zu einer Party gehen will, als Student, der er nun ist, lieber zu Hause bleibt und an seinen Hausarbeiten schreibt. Im Beispiel aus Abschnitt 2.2.2 wird dies durch die folgende Zeile realisiert:

```
schreibeHausarbeit <- replace geheZurParty;
```

Es handelt sich dabei um ein REPLACE-Callin ohne *Basecall*. Da die Methode `schreibeHausarbeit()` keinen Aufruf der durch sie ersetzten Methode enthält, keinen Basecall, wird diese nun auch nicht mehr ausgeführt. Jede Partyeinladung bleibt für den armen Studenten in der Folge also wirkungslos. Um dem Studenten auch weiterhin das Besuchen von Parties zu erlauben, ließe sich nun die REPLACE-Methode mit Hilfe eines Basecalls so abändern, dass er doch noch zu Parties gehen kann, wenn er mit seiner Arbeit fertig ist:

```
callin void schreibeHausarbeit()
{     if(SeitenzahlFürHeuteGeschafft())
      { base.schreibeHausarbeit();}
}
```

Das Schlüsselwort **base** leitet nun einen Basecall ein, der – obwohl der Name der REPLACE-Methode dazu verwendet wird - die *Basismethode* aufruft, diejenige

Methode, die durch die Callin-Definition gebunden wird. Eine Alternative dazu wäre in diesem Beispiel auch die Verwendung eines BEFORE-Callins gewesen:

```
schreibeHausarbeit <- before geheZurParty;
```

oder aber auch ein AFTER-Callin, falls der Student nach der Party noch zum Schreiben einer Hausarbeit in der Lage sein sollte...

```
schreibeHausarbeit <- after geheZurParty;
```

2.2.4 Teaminstanzen und Teamaktivierung

Callin-Bindungen werden nur für aktive Teaminstanzen ausgeführt. Dies ermöglicht es einer Applikation, ihr Verhalten während der Laufzeit zu verändern. Realisiert wird dies durch die Methode `activate()`, within-Blöcke und impliziter Teamaktivierung. In Kapitel 8 werden diese Möglichkeiten im Detail vorgestellt. Durch die Verwendung von `activate(Thread t)` kann die Teamaktivierung auch auf einzelne Threads eingeschränkt werden, so dass eine Teaminstanz in einem Thread aktiv sein kann, in einem anderen jedoch nicht.

So wird auch nur ein aktiver, lebendiger Studiengang, der den Studierenden aus 2.2.2 mitreißen kann, den in 2.2.3 beschriebenen Effekt erzielen, dass der Student lieber seine Hausarbeit schreibt, als zu einer Party zu gehen. Damit das also geschieht und das entsprechende Callin aktiv ist, müsste das Beispielprogramm aus 2.2.2 auch die folgende Zeile enthalten:

```
Informatik.activate();
```

2.2.5 Vererbungsbeziehungen in ObjectTeams/Java

Die Sprache ObjectTeams/Java führt eine weitere Vererbungshierarchie für Rollenklassen ein. Diese können nun explizit von ihrer Superklasse erben – und zusätzlich implizit von einer Rollenklasse gleichen Namens aus der Superklasse ihrer Teamklasse. Analog zum Schlüsselwort super existiert das Schlüsselwort tsuper, um Zugriff auf die implizite Vererbungshierarchie zu erhalten. Realisiert wird diese Form der Vererbung, indem alle Methoden der gleichnamigen tsuper-Rollenklasse vom Compiler in die erbende Rollenklasse kopiert und durch einen zusätzlichen Parameter erweitert werden, um sie von den implizit geerbten und den von ihr selbst definierten Methoden unterscheiden zu können. Außerdem werden in ObjectTeams/Java Callin-Bindungen zusammen mit Rollen-Basis-Bindungen vererbt. Dies wird in Kapitel 6 besprochen.

2.3 Weitere Spracheigenschaften

Es folgt nun eine kurze Aufzählung weiterer Spracheigenschaften von ObjectTeams/Java, die im Rahmen dieser Arbeit jedoch nicht näher behandelt werden. In Kapitel 11 wird auf sie jedoch noch einmal kurz eingegangen, wenn ein Ausblick darauf gegeben wird, wie die in dieser Arbeit vorgestellte VM-Erweiterung OTRun in Zukunft weitere Spracheigenschaften von ObjectTeams/Java unterstützen könnte.

Die Deklaration von Callin-Bindungen kann durch eine Reihe von Spracheigenschaften ergänzt werden. Mit Hilfe von *Guards* kann die Ausführung einer Callin-Bindung an ein Prädikat gebunden werden, dass vor jeder Ausführung einer Callin-Methode zunächst evaluiert wird. Vom Wahrheitswert des Prädikats hängt es dann ab, ob die Callin-Methode auch tatsächlich aufgerufen wird.

Mit *Parametermappings* können Callin-Methoden an eine Basismethode gebunden werden, obwohl sich ihre Signaturen unterscheiden. Das Parametermapping erlaubt es dann, auf jeden Parameter der Callin-Methode einen beliebigen Parameter der Basismethode abzubilden – oder auf eine Abbildung ganz zu verzichten. AFTER-Callins kann so auch der Rückgabewert der Basismethode als Parameter übergeben werden.

Durch die Verwendung einer *precedence*-Deklaration kann die Reihenfolge der Callin-Ausführung innerhalb einer Rollenklasse und zwischen den Rollenklassen einer Teamklasse festgelegt werden. Enthält eine Rollenklasse zwei Callin-Bindungen, die an die selbe Basismethode binden, so wird durch die Precedence-Deklaration bestimmt, welche dieser beiden Callin-Bindungen zuerst ausgeführt wird. Analog gilt dies für Rollenklassen, wenn aus zwei Rollenklassen einer Teamklasse eine Callin-Methode an die gleiche Basismethode gebunden wird.

Mit dem Konzept des *Confinement* bietet ObjectTeams/Java einen zusätzlichen Schutzmechanismus zur Kapselung von Modulen an. Indem die Superklassen-hierarchie einer Klasse nicht mehr bei `Object` endet, sondern direkt mit der Klasse die Confinement einsetzt, kann verhindert werden, dass über `Object`-Typecasts ungewollt Referenzen aus einem Modul nach außen gelangen können.

2.4 Die Laufzeitumgebung OTRE

Für die Umsetzung der ObjectTeams/Java-Funktionalität ist neben dem ot-Compiler, der ObjectTeams/Java-Programme übersetzen kann, auch eine spezielle Laufzeitumgebung notwendig, das OTRE. Dieses verwendet einen Ladezeit-Webemechanismus, um Basisklassen so zu transformieren, dass sie unter anderem Lifting, Teamaktivierung und Callin-Ausführung unterstützen. Ein weiterer Teil des OTRE ist eine kleine Klassenbibliothek. Diese enthält Klassen, die für die Ausführung eines ObjectTeams/Java Programms benötigt werden. Dazu gehören speziell angepasste Hashmaps für das Lifting, Klassen, die in neuen, mit ObjectTeams/Java dazugekommenen Ausnahmezuständen benötigt werden, und

vor allem die Klasse Team. Diese Klasse ist die Superklasse aller Teamklassen und vererbt an alle Teamklassen eine Reihe von Methoden, mit denen die Funktionalität von ObjectTeams/Java von einem Programm genutzt werden kann, wie z.B. die activate()-Methode.

Damit soll die kurze Einführung in die Programmiersprache ObjectTeams/Java beendet sein. In diesem Kapitel wurden die umfangreichen Modularisierungsmechanismen von ObjectTeams/Java beschrieben, die Team- und Rollen-Metapher anhand eines Beispiels erklärt, sowie die Möglichkeiten zur aspektorientierten Adaption bestehender Basisklassen vorgestellt. Weitere Details zu den Spracheigenschaften Lifting, Teamaktiverung und Callin-Ausführung sind in den Kapiteln 7, 8 und 9 zu finden. Eine vollständige Definition der Spracheigenschaften liefert die ObjectTeams/Java Language Definition [OTJLD].

Im nächsten Kapitel wird nun die Funktionsweise einer Java Virtual Machine vorgestellt. Dies liefert uns die noch fehlenden Grundlagen, die notwendig sind, um in Kapitel 4 die Aufgabenstellung dieser Arbeit und den in ihr vorgestellten Lösungsansatz darstellen zu können: die delegationsbasierte Integration der Spracheigenschaften Lifting, Teamaktivierung und Callin-Ausführung in eine Java Virtual Machine zur Optimierung von ObjectTeams/Java.

Kapitel 3

Java und die Java Virtual Machine

- Einblicke in die Funktionsweise der JamVM

3.1 Java und die Java Virtual Machine

Programme, die in der Programmiersprache Java geschrieben werden, können vom Java-Compiler zu einer plattformunabhängigen Maschinensprache, dem Java Bytecode, kompiliert werden. Java Bytecode umfasst dabei alle Maschinenbefehle, die man typischerweise in einer stackbasierten Maschinensprache erwarten würde (Ladeinstruktionen, Arithmetik, Kontrollfluss-Steuerung) und insgesamt alle Befehle[4], die notwendig sind, um die Programmiersprache Java auf eine Maschinensprache abbilden zu können.

Gespeichert wird der Java Bytecode in einem eigenen binären Format, den .class-Dateien. Diese stellen das für die Java Virtual Machine lesbare Äquivalent zu den in Java geschriebenen Quelldateien dar, wobei jeder .java-Datei, die genau eine Klasse enthält, genau eine .class-Datei entspricht. Mehrere .class-Dateien können zu einer einzelnen Datei, einem .jar-Paket, zusammengefasst werden.

Java-Programme werden von der Java Virtual Machine (VM) ausgeführt. Diese lädt zuerst eine ihr zum Start übergebene .class-Datei, um dann in dieser nach einer Main-Methode zu suchen. Anschließend beginnt die VM mit der Ausführung des Programms, indem sie den Bytecode der Main-Methode entweder interpretiert oder durch einen „Just in time“-Compiler (JIT) in die Maschinensprache der Plattform übersetzt, auf der die VM ausgeführt wird.

Die .class-Datei der Main-Methode enthält – wie alle .class-Dateien – im so genannten *Constantpool* utf8-codierte Zeichenketten, die unter anderem den Namen aller Methoden, ihrer Signaturen und Klassen codieren, die innerhalb der Main-Klasse verwendet werden. Sie darf aber auch individuelle Attribute enthalten, die von einem Compiler generiert werden. Der ot-Compiler nutzt dies,

[4] Für Details siehe [JVMSpec].

um statische Informationen über Team- und Rollenklassen an die Laufzeitumgebung OTRE weitergeben zu können.

Sobald die VM beim Abarbeiten der Main-Methode auf einen Bytecode stößt, der auf ein solches Symbol im Constantpool verweist, wird dieser Eintrag in einem Vorgang, der *resolve* genannt wird, ausgewertet, um die Klasse, die zu dem referenzierten Element gehört, in die VM zu laden. Die VM kann dadurch dynamisches, spätes Binden verwenden, was bedeutet, dass Klassen erst zur Laufzeit des Programms miteinander verknüpft und nach Bedarf geladen werden. Dies wird auch als *lazy loading* bezeichnet und der Standard für die Java Virtual Machine verlangt, dass dies bestmöglich von einer VM umgesetzt wird, das Laden und Binden einer Klasse also so spät wie möglich zu erfolgen hat. Wurde eine Klasse geladen, kann anschließend mit Hilfe der Informationen aus dem Constantpool geprüft werden, ob ihr Inhalt typkonform mit dem angeforderten Inhalt der Klasse ist, die den Ladevorgang auslöste. Eine Übereinstimmung beendet erfolgreich den resolve-Vorgang.

Zusätzlich zum Ausführen von Bytecodes und zur Verwaltung der Klassen eines Java-Programmes fällt auch das Speichermanagement in den Aufgabenbereich der VM. Java-Objekte werden bei Ausführung des new-Bytecodes von einem Heap allokiert, der mit Hilfe eines Garbage-Collector-Algorithmus durch die VM verwaltet wird.

In diesem Kapitel sollen nun einige Details der Java Virtual Machine dargestellt werden, die für diese Arbeit relevant sind. Erläutert werden sie anhand der JamVM, einer virtuellen Maschine, die in dieser Arbeit verwendet wird, um eine optimierte Laufzeitunterstützung für ObjectTeams/Java zu realisieren.
Indem wir den Aufruf-Vorgang einer Java-Methode ausführlich in der VM verfolgen, wird es uns möglich sein, fast alle Aspekte der JamVM kennen zu lernen, die wir im Verlauf dieser Arbeit noch benötigen werden. Dazu zählen die Funktionsweise des Interpreters, die Konvertierung von Bytecodebefehlen in ihre _quick-Varianten, der resolve-Vorgang einer Constantpool-Referenz, der das Laden einer Klasse auslösen kann und auch der Ladevorgang einer Klasse selbst. Außerdem gehört der Aufbau eines Stackframes zum Aufruf einer Methode dazu sowie die Konvertierung eines kompletten Codeblocks in ein internes Format, das effizienter zu interpretieren ist als der Bytecode aus den .class-Dateien. Ein weiterer Aspekt, der besprochen werden soll, ist schließlich das Verfahren zur Rückkehr aus einer Methode und zur Rückgabewertübergabe. Anschließend werden wir einen kurzen Blick auf einen Aspekt der VM werfen, der uns nun noch fehlt: die Umsetzung der Speicherverwaltung mit Hilfe eines Garbage-Collector-Algorithmus – und damit verbunden – den Zweck von *weak References*.

3.2 Methoden

Eine Methode wird in der JamVM durch einen `MethodBlock` repräsentiert. Seine wichtigsten Felder sollen nun kurz erläutert werden:

```
typedef struct methodblock
{
   Class*           class;
   char*            name;
   char*            type;
   u2               max_locals;
   u2               args_count;
   int              method_table_index;
   void*            code;
}
MethodBlock;
```

Das Feld `class` verknüpft die Methode mit dem `ClassBlock` ihrer Klasse. Wir werden den `ClassBlock` im Anschluss noch betrachten, denn er gehört zusammen mit dem `MethodBlock` und dem `Frame`, den wir bei der Methodenausführung kennen lernen werden, zu den wichtigsten Datenstrukturen der JamVM. Die im Rahmen dieser Arbeit entwickelten Lösungsansätze verwenden sehr oft Informationen, die in diesen drei Datenstrukturen gespeichert sind.

Die Felder `name` und `type` beinhalten den Namen einer Methode und ihre Signatur. Um eine bestimmte Methode zu finden, reicht es daher aus, ein Array von Methodenblöcken im `ClassBlock` zu durchlaufen und den Namen und die Signatur der gesuchten Methode mit den Namen und Signaturen, die dort gespeichert sind, zu vergleichen. Da die JamVM alle Zeichenketten in einer Hashtabelle ablegt, weisen Zeichenketten-Zeiger, die innerhalb der JamVM gespeichert wurden und die den gleichen Inhalt besitzen, immer auf die gleiche Speicherstelle, dieselbe Zeichenkette. Deshalb ist es möglich, direkt durch einen schnellen Zeigervergleich festzustellen, ob zwei Zeichenketten identisch sind oder nicht. So wird in diesem Fall die Frage nach Gleichheit sehr effizient auf die Frage nach Identität abgebildet.

Die Felder `max_locals` und `args_count` beschreiben den Speicherplatzbedarf der Methode, wenn sie aufgerufen wird. Das Feld `args_count` wird dazu verwendet, um einen Zeiger auf die Parameter, die einer Methode auf dem Stack übergeben werden, zu konstruieren. Wir werden diesen Vorgang noch kennen lernen.

Der `method_table_index` verweist auf den Eintrag der Methode in der `MethodTable` des `ClassBlocks`. Im Rahmen dieser Arbeit wird er dazu verwendet, auf effektive Weise die Vererbung von Callins zu implementieren. Wir werden dies in Kapitel 6 genauer betrachten. Außerdem hilft dieses Feld dabei, feststellen zu können, ob es sich bei einem Supermethoden-Aufruf um einen *Super-Self-Call* handelt, also um den Aufruf der eigenen Super-Methode. Dies wird in Kapitel 9 benötigt werden, um Basecalls für Replace-Callins in der VM unterstützen zu können.

Die Bytecode-Befehle eines Java-Programmes sind in Codeblöcken gespeichert, wobei ein Codeblock genau einer Methode in Java entspricht. Das Feld `code` im `MethodBlock` verweist auf den Codeblock einer einzelnen Methode. Jeder Codeblock besitzt dabei genau einen Einsprungspunkt am Anfang des Codeblocks und eine Reihe von vorgegebenen Austrittspunkten, die durch Return-Befehle und Befehle zur Ausnahmebehandlung festgelegt sind. Der Delegationproxy, der in Kapitel 4, Kapitel 6 und ausführlich in Kapitel 9 vorgestellt werden wird, kann über die Modifizierung dieses Feldes sehr einfach ein- oder ausgeschaltet werden. Dies ist eine Eigenschaft, die sehr hilfreich ist, um den gleichen Codeblock in zwei verschiedenen Methodenblöcken verwenden zu können und wird für die Callin-Vererbung benötigt. Details dazu befinden sich in Kapitel 6.

Ein Codeblock kann aber auch durch den Sprung zu einem neuen Codeblock in Form eines Methodenaufrufs verlassen werden. Ansonsten ist die Navigation innerhalb von Java-Programmen sehr begrenzt und kann nur durch relative Sprünge innerhalb eines Codeblocks erfolgen, nicht aber beliebig aus ihm heraus. Dies ist eine Eigenschaft von Java, die zur Wahrung der Sicherheit dient. Diese Beschränkung wird zum einen durch die für Sprünge vorhandenen Bytecodebefehle und zum anderen durch eine Programmcode-Überprüfung beim Laden einer Klasse durch die virtuelle Maschine realisiert, vor allem aber durch den Verzicht auf direkte Referenzen. Um aber dennoch andere Methoden aufrufen zu können, werden abstrakte Referenzen verwendet. Diese bestehen aus der Klasse, dem Namen und der Signatur einer Methode und werden über einen Verweis auf den Constantpool adressiert. Mit Hilfe dieser Referenzen kann durch den *resolve*-Vorgang der Methodenblock einer anderen Methode ermittelt werden, um diese dann aufrufen zu können. Dazu betrachten wir nun den Bytecode `invokevirtual`, der dazu dient, Methoden aufzurufen.

3.3 Methodenaufruf mit `invokevirtual`

Wenn in einem Java-Programm eine Methode aufgerufen wird, so wandelt der Java-Compiler diesen Aufruf in einen Bytecode aus der Familie der invoke-Bytecodes um. Zu dieser gehören vier Bytecodes: `invokeinterface` für den Aufruf von Interface-Methoden, `invokestatic` für den Aufruf von statischen Methoden, die keinen this-Zeiger besitzen, `invokespecial` für den Aufruf von Konstruktoren, privaten Methoden und Supermethoden – und `invokevirtual` für alle anderen Methoden, die dynamisches Binden verwenden. Dieser ist es, den wir nun exemplarisch etwas genauer betrachten wollen.

3.3.1 Der resolve-Vorgang

Der Bytecode `invokevirtual` besitzt einen 16-Bit Parameter, der statisch auf einen Eintrag im Constantpool derjenigen Klasse verweist, zu der die Methode gehört, in welcher der `invokevirtual`-Befehl vorkommt. Wird `invokevirtual` ausgeführt, so ruft er mit diesem Parameter zunächst die Funktion `MethodBlock *resolveMethod (Class *class, int cp_index)` auf. Diese betrachtet

nun zuerst ein Statusfeld, das unter anderem an der übergebenen Stelle im Constantpool zu finden ist, und wertet es mit Hilfe einer switch-Anweisung aus. Zum einen kann der Fall eintreten, dass die Methode bereits resolved wurde. In diesem Fall wurde von `resolveMethod` an der übergebenen Position im Constantpool bereits ein Methodenblock für die gesuchte Methode gespeichert. Dieser kann dann sofort zurückgegeben werden. Ist dies jedoch nicht der Fall, so befindet sich stattdessen eine abstrakte Methodenreferenz an diesem Ort, bestehend aus Constantpool-Indizes für Strings mit dem Klassennamen, dem Methodennamen und der Signatur. Im nächsten Schritt des resolve-Vorganges wird nun überprüft, ob die Klasse der Methode vielleicht bereits geladen wurde. Dies geschieht analog und so findet der resolve-Vorgang an der Index-Position des Klassennamens entweder eine Referenz auf ein Klassenobjekt inklusive `ClassBlock` oder aber lediglich einen String, der den vollständigen Namen der Klasse enthält. Wurde die Klasse bereits geladen, dann kann der resolve-Vorgang nun die `MethodTable` nach der Methode durchsuchen, die aufgerufen werden soll. Ist sie gefunden und stimmen Namen und auch der Typ, sowie die entsprechenden Zugriffsrechte, so wird der resolve-Vorgang erfolgreich beendet und eine Referenz auf den passenden Methodenblock zurückgegeben. Vorher wird diese Referenz aber noch im Constantpool gespeichert, so dass sie bei zukünftigen resolve-Vorgängen sofort gefunden wird. Wurde dagegen im Constantpool nur ein String mit dem Klassennamen gefunden, so muss die Klasse zunächst in die VM geladen werden.

3.3.2 Class Loading

Das Laden einer Klasse geschieht in drei Phasen: Zunächst wird im eigentlichen Ladevorgang eine Repräsentation der .class-Datei in den Speicher der VM geladen. Dies kann direkt über ein Dateisystem geschehen, aber auch durch einen speziellen Classloader, der die Repräsentation der .class-Datei aus einer anderen Quelle bezieht (z.B. dem Internet) oder synthetisch generiert. Anschließend wird aus dem Binärformat der .class-Datei eine interne Repräsentation der in ihr enthaltenen Klasse erzeugt. Alle zukünftigen Zugriffe des Programms auf diese Klasse erfolgen dann über diese interne Repräsentation. Betrachten wir diese ebenfalls kurz anhand ihrer wichtigsten Felder:

```
typedef struct classblock
{
   char*                  name;
   Class*                 super;
   u2                     fields_count;
   FieldBlock*            fields;
   int                    refs_offsets_size;
   RefsOffsetsEntry*      refs_offsets_table;
   u2                     constant_pool_count;
   ConstantPool           constant_pool;
   u2                     methods_count;
   MethodBlock*           methods;
   int                    method_table_size;
   MethodBlock**          method_table;
   int                    object_size;
} ClassBlock;
```

Neben dem Namen der Klasse und einem Verweis auf ihre Superklasse enthält ein `ClassBlock` vor allem einige sehr wichtige Tabellen. Das Array `fields` enthält eine Beschreibung aller Felder, die in einem Objekt zu finden sind. Benötigt man Zugriff auf ein bestimmtes Feld, so lässt sich dieses Array nach dem Namen des Feldes durchsuchen. Der gefundene Eintrag enthält dann einen Offset, der auf die Startadresse eines Objektzeigers addiert werden kann, um zu dem gewünschten Feld zu gelangen. Die `refs_offsets_table` speichert Verweise auf alle Referenzfelder, die in `fields` enthalten sind. Der Garbage Collector nutzt diese Einträge, um herauszufinden, wo in einem Objekt, welches er auf dem Heap gefunden hat, Referenzen gespeichert sein könnten, die ihn dann zu weiteren Objekten führen. Der Constantpool wurde bereits erwähnt, selbstverständlich befindet sich dieser im `ClassBlock`.

Nicht verwechseln sollte man das Array `methods`, das Methodenblöcke enthält, mit der `method_table`, die für dynamisches Binden verwendet wird und nur Referenzen auf Methodenblöcke beinhaltet. Zudem unterscheiden sich beide darin, dass die Methoden-Tabelle ausschließlich public- und protected-Methoden enthält. Private- und statische Methoden kommen dagegen nur im `methods`-Array vor. Wir werden dieses Wissen verwenden, wenn wir in Kapitel 6 Callins für bestimmte Basismethoden einrichten wollen.
Die `object_size` kann dazu verwendet werden, ein neues Objekt der passenden Größe zu allokieren. Die Implementierung des Liftings, die in Kapitel 7 beschrieben wird, verwendet dieses Feld um Speicherplatz für neue Rolleninstanzen in der richtigen Größe allokieren zu können.

Die auf die Lade-Phase folgende Link-Phase baut viele dieser Datenstrukturen auf, um die Klasse in einen betriebsbereiten Zustand zu versetzen. Dazu gehört unter anderem das Anlegen des Layouts mit Offsets für die Felder einer Instanz dieser Klasse. Damit dieses vollständig erzeugt werden kann, muss jedoch das Layout der Superklasse bekannt sein. Daher löst das Laden einer Klasse automatisch das Laden all ihrer Superklassen aus. Diese werden, der Hierarchie von oben folgend, nun in die VM geladen.

Außerdem wird in dieser Phase die `method_table` angelegt, in der Zeiger auf die Methodenblöcke aller geerbten Methoden gespeichert werden. Diese dient auch zur Realisierung des Overridings. Wird eine Methode über einen Methodentabellen-Index aufgerufen, wie es z.B. invokevirtual tut, so wird der Methodenblock-Zeiger verwendet, der an dieser Stelle steht. Besitzt die Klasse eine eigene Implementierung einer geerbten Methode, so befindet sich ein Zeiger zu ihrem Methodenblock nun genau an dieser Stelle, wo sich andernfalls ein Zeiger zu einem Methodenblock aus der Superklasse befinden würde. Da immer die Methodentabelle derjenigen Klasse verwendet wird, zu der die Instanz gehört, auf der eine Methode aufgerufen wurde, wird je nach Typ der Instanz mal eine Super- und mal eine Submethode aufgerufen.

Abschließend erfolgt die Initialisierungs-Phase. In dieser werden sämtliche static-Felder der Klasse initialisiert. Nun ist die Klasse betriebsbereit.

3.3.3 _quick Instructions

Hat ein invokevirtual erfolgreich den resolve()-Vorgang abgeschlossen, so besteht für diese Anweisung keine Notwendigkeit mehr, diesen zeitaufwändigen Prozess jemals zu wiederholen. Daher ersetzt sich der Bytecode selbst in einem nächsten Schritt durch einen anderen Bytecode, einer so genannten _quick-Variante, die keinen Aufruf des resolve-Vorgangs mehr enthält. Damit aber auch dieser neue Bytecode den passenden Methodenblock finden kann, der aufgerufen werden soll, wird nun der Bytecode-Parameter des alten Bytecodes, der den Constantpool-Index enthielt, ersetzt durch den `MethodTable`-Index. Nun kann der neue Bytecode die Zielmethode sofort mit Hilfe des this-Parameters ermitteln, der jeder mit invokevirtual aufgerufenen Methode übergeben wird. Weil der Methodenaufruf nicht erst geschehen soll, wenn dieser Bytecode das nächste Mal ausgeführt wird, veranlasst er nun in gewandelter Form eine erneute Ausführung seiner selbst. Die neue _quick-Variante ermittelt dann aus dem `MethodTable`-Index und dem this-Zeiger den Methodenblock der aufzurufenden Methode. Anschließend muss der Bytecode noch die zwei lokalen Variablen `arg1` und `new_mb` des Interpreters initialisieren, bevor er zur Aufrufroutine für Methoden springen kann. Um die Bedeutung dieser beiden Variablen zu verstehen, soll zunächst die Interpreter-Funktion der JamVM kurz vorgestellt werden.

3.3.4 Der Interpreter und seine Umgebung

Der Interpreter der JamVM ist eine einfache C-Funktion. Sie besitzt einige lokale Variablen, die den Behandlungsroutinen aller Bytecodes zur Verfügung stehen. Dazu gehören die Variablen `this`, `pc`, `ee`, `frame`, `lvars`, `arg1`, `new_mb`, `mb`, `this`, `ostack` und `cp`. In Analogie zu einem Mikroprozessor könnte man diese Variablen als die Register des Interpreters bezeichnen. Wie ein Prozessor, so besitzt auch der Interpreter mit der Variablen `pc` einen Programmcounter. Dieser zeigt auf den Bytecode im Codeblock, der als nächstes auszuführen ist bzw. gerade ausgeführt wird. Ebenfalls von Mikroprozessoren bekannt ist ein Register, das den Zeiger auf einen Stack enthält. Die JamVM adressiert ihn über die Variable `ostack`. Mit der Variable `ee` erhält der Interpreter Zugriff auf das Environment des Threads, in dem er ausgeführt wird. Dieses enthält unter anderem einen Stackzeiger, der dazu verwendet wird, `ostack` zu initialisieren. Auf diesem Stack findet der Interpreter bei seinem Start einen präparierten Stackframe vor, der ihn auf die erste Methode verweist, die er auszuführen hat. Den Methodenblock der Methode, die gerade ausgeführt wird, speichert der Interpreter in der Variable `mb`. Die Variablen `this` und `cp` dienen dazu, in einzelnen Bytecodes den Zugriff auf den this-Zeiger des Objekts, auf dem eine Methode aufgerufen wurde und den Zugriff auf den Constantpool der aktuellen Klasse, zu der die gerade ausgeführte Methode gehört, zu beschleunigen. Die Variablen `frame`, `arg1`, `new_mb` und `lvars` beschreiben den Kontext einer Methode in der Interpreter-Funktion, spielen eine wichtige Rolle bei der Erzeugung eines *Stackframes* und werden nun im Detail erläutert werden, um den Prozess eines Methodenaufrufs verstehen zu können.

3.3.5 Stackframes

Der Stack ist eine Datenstruktur, mit deren Hilfe die Parameter- und Rückgabewertübergabe einer Methode organisiert wird. Zugleich dient er ihr als lokaler, schnell adressierbarer Speicherbereich, in dem sie ihre lokalen Variablen speichern kann. Da jede Methode einen eigenen Kontext besitzt, findet bei jedem Aufruf einer Methode ein Kontextwechsel statt. Um zu dem alten Kontext des Aufrufers einer Methode zurückkehren zu können, wenn die aufgerufene Methode zurückkehrt, wird auch dieser auf dem Stack gespeichert – in einem *Stackframe*, der in der JamVM durch die Datenstruktur `frame` realisiert wird. Diese Datenstruktur ist folgendermaßen aufgebaut:

```
typedef struct frame
{
   CodePntr               last_pc;
   uintptr_t*             lvars;
   uintptr_t*             ostack;
   MethodBlock*           mb;
   struct frame*          prev;
}
```

Ein `frame` enthält den Methodenblock der aktuell ausgeführten Methode, ihren Stackzeiger zum Zeitpunkt des Aufrufs einer Methode, einen Zeiger auf den Beginn ihrer lokalen Variablen, zu denen auch die Parameter und der this-Zeiger der Methode gehören und schließlich – besonders wichtig – ein Feld, in dem die Adresse des Bytecodes gespeichert ist, der den Methodenaufruf verursacht hat. Dieses wird dazu verwendet, bei der Rückkehr aus einer Methode zum Punkt des Aufrufs zurückzukehren, wobei die weitere Ausführung nach der Rückkehr dann bei dem nachfolgenden Bytecode fortfährt. Der Delegationproxy, den wir in Kapitel 9 kennen lernen werden, manipuliert diese Rücksprungadresse, um zu sich selbst zurückkehren zu können. Dazu speichert er in diesem Feld die Adresse seines Vorgänger-Bytecodes ab.

3.3.6 Ausnahmebehandlung

Alle Frames, die auf dem Stack existieren, sind zu einer Liste verkettet. Diese wird dazu verwendet, *Handler* für einen Ausnahmezustand zu suchen. Wird eine Ausnahme ausgelöst, so sucht die virtuelle Maschine zunächst in der eigenen Methode nach einer Behandlungsroutine. Wird sie nicht fündig, wird die Ausführung der Methode beendet und in der aufrufenden Methode nach einer Ausnahmebehandlungsroutine gesucht. Wird die Ausnahme auch dort nicht behandelt, wird die Liste solange durchlaufen, bis entweder eine Behandlungsroutine gefunden wird oder aber das Ende dieser Liste erreicht wurde. In diesem Fall würde der Interpreter dann die Ausführung dieses Threads mit einer Fehlermeldung abbrechen.

3.3.7 Methodenaufruf

Da jede Methode einen Kontext benötigt, muss ein neuer Kontext vor dem Aufruf einer Methode für sie bereitgestellt werden. Der `invokevirtual`-Bytecode initialisiert dazu zunächst die beiden Interpreter-Variablen `new_mb` und `arg1`. In `new_mb` wird ein Zeiger auf den Methodenblock der aufzurufenden Methode gespeichert. Wir haben bereits besprochen, wie dieser mit Hilfe des `method_table_index` gewonnen werden kann. Die Variable arg1 zeigt, wie in Abbildung 1 zu sehen ist, auf die Stackposition, ab der die Parameter für den Methodenaufruf zu finden sind. Dieser Wert wird von `invokevirtual` mit Hilfe des Feldes `args_count` aus dem neuen Methodenblock und der aktuellen Stackposition berechnet. Im nächsten Schritt wird dieser Wert auch der Variablen `lvars` zugewiesen, über welche die aufzurufende Methode in entsprechenden Bytecodes Zugriffe auf Parameter und lokale Variablen realisiert.

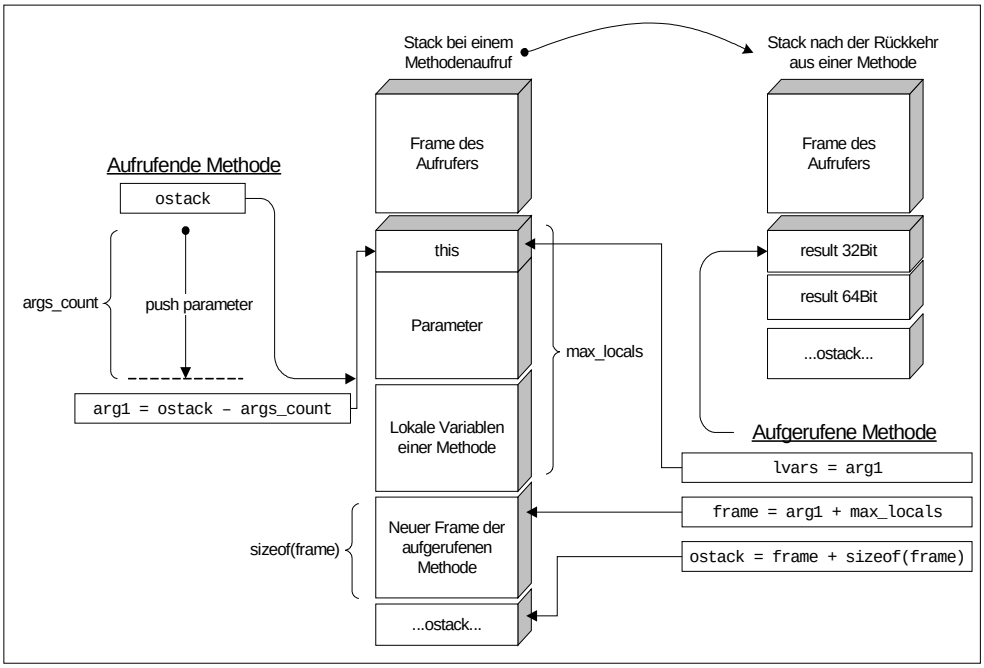


Abbildung 1: ***Stackaufbau zur Ausführung einer Methode***

Anschließend wird ein neuer Frame auf dem Stack allokiert, seine Felder werden initialisiert und der aktuelle Kontext wird im alten Frame gespeichert, um diesen Kontext bei der Rückkehr wieder herstellen zu können. Das Feld `ostack` wird schließlich mit der Position des soeben erzeugten Frames initialisiert, auf welche die Größe des Frames addiert wird. Dadurch beginnt der Stack für die neue Methode direkt im Anschluss an ihren `frame`. Nun kann der eigentliche Aufruf der neuen Methode endlich erfolgen, indem der Programmcounter `pc` mit dem Beginn des Codeblocks der aufzurufenden Methode geladen wird. Diesen kann `invokevirtual` im Methodenblock der aufzurufenden Methode finden, auf den sie über die Variable `new_mb` zugreifen kann.

3.3.8 Interpreter-Varianten und die prepare()-Phase

Doch bevor der Sprung zur neuen Methode erfolgt, wird in einigen Interpreter-Varianten der JamVM zunächst geprüft, ob der Codeblock zu dem gesprungen werden soll, bereits *prepared* ist. Dies wird durch das unterste Bit des Codeblock-Zeigers codiert.

Die JamVM besitzt drei Interpreter-Varianten: indirect, direct und inline. Diese unterscheiden sich sowohl durch die Codierung der Bytecodes eines Codeblocks als auch durch unterschiedliche Mechanismen zur Navigation von einem Bytecode zum jeweils nächsten. Die einfachste und zugleich langsamste Variante ist der indirect-Interpreter. Mit Hilfe einer großen switch-Anweisung arbeitet er Bytecode für Bytecode ab. Dabei entspricht jeder case-Fall dem Byte-Wert eines Bytecodes. Um zum jeweils nächsten Bytecode zu gelangen, addiert der gerade abgearbeitete Bytecode seine Länge auf den `pc` – der dadurch auf den nächsten Bytecode verweist – und springt anschließend vor die switch-Anweisung, um diese erneut auszuführen.

Die direct-Variante verwendet dagegen ein Bytecode-Format mit einheitlicher Länge. Jeder Bytecode eines Codeblocks wird dazu von der prepare()-Methode in dieses neue, 8 Byte große Format umgewandelt. Die ersten 4 Bytes werden dabei von der Adresse einer Sprungmarke belegt, an welcher die Behandlungsroutine des jeweiligen Befehls zu finden ist. Jedem Bytecode stehen nun weitere 4 Bytes für Parameter zur Verfügung, die von jedem Bytecode individuell genutzt werden können. Die Navigation von Bytecode zu Bytecode erfolgt, indem der Interpreter direkt an die Adresse springt, die durch die Sprungmarke des jeweils nächsten Bytecodes gegeben ist. Da es der Branch-Prediction-Einheit eines modernen Mikroprozessors einfacher fällt, diese Art von Sprüngen zu verfolgen, ist diese Interpreter-Variante schneller, als die indirekte Variante[5]. Dies ist auch die Variante des Interpreters, die in dieser Arbeit verwendet wird. In Kapitel 9 wird dazu eine spezifische Eigenschaft der prepare()-Phase ausgenutzt werden. Sie muss für die Umwandlung des Bytecodes in das neue Format einen neuen Codeblock allokieren. Dessen Größe wird nun so modifiziert, dass jeder Codeblock zusätzlich die Bytecodesequenz des Delegationproxy aufnehmen kann.

Die inline-Variante stellt schließlich die fortschrittlichste Interpreter-Variante dar und realisiert einen Ansatz, der einem einfachen JIT-Compiler ähnlich ist. Die inline-Variante kopiert dazu die vom C-Compiler zur nativen Assemblersprache kompilierten Behandlungsroutinen für die einzelnen Bytecodes direkt hintereinander. Dabei muss der Algorithmus jedoch darauf achten, keine Bytecodes miteinander zu verschmelzen, die als Ziel eines Sprungs verwendet werden, da diese nach wie vor über Codeblock-lokale, `pc`-kompatible Adressen erreichbar bleiben müssen[6]. Da die in Kapitel 9 vorgestellten Bytecodes für den Delegationproxy jeweils Sprungziele darstellen, wird auf die Verwendung des inline Interpreters im Rahmen einer Prototyp-Entwicklung vorläufig verzichtet und die Unterstützung dieser Variante für eine spätere Version eingeplant.

[5] Siehe dazu auch [Ertl02] und [Ertl03].
[6] Siehe dazu auch [Pr08] und [Piu98].

Die prepare()-Phase dient auch dazu, eine Bytecode-Verifikation durchzuführen, um unter anderem der VM unbekannte Bytecodes von der Ausführung auszuschließen.

3.3.9 Rückkehr aus einer Methode

In Abbildung 1 ist auch das Layout des Stacks nach einem Methodenaufruf dargestellt. Der `return`-Bytecode einer Methode kopiert seinen auf dem Stack liegenden Rückgabewert nun an die Position lvars[0] bzw. auch lvars[1], wenn ein 64Bit-Wert zurückgegeben wird. Anschließend inkrementiert er lvars um 1 bzw. 2 und weist `ostack` dann den Wert dieser Variablen als neuen Beginn des Stacks zu. Damit wird der gesamte Stackframe der zu verlassenden Methode – bis auf den Rückgabewert – gelöscht. Vor der Rückkehr wird allerdings noch sämtlichen Kontextvariablen wieder der Wert zugewiesen, den sie vor der Ausführung der Methode enthielten. Um zum Ort des Aufrufs zurückkehren zu können, ermittelt der `return`-Bytecode nun die Rückkehradresse. Dazu inkrementiert der `return`–Bytecode das Feld `frame->last_pc` aus dem Stackframe der ursprünglich aufrufenden Methode und weist dem Programmcounter pc diesen Wert zu. Die Rückkehr aus der Methode ist damit abgeschlossen und die aufrufende Methode wird nach dem Aufruf-Bytecode fortgesetzt.

3.4 Garbage Collection

Eine weitere Spracheigenschaft von Java, die von der VM realisiert werden muss, haben wir bisher noch nicht kennen gelernt: Den Garbage-Collector. Die Sprache Java nimmt dem Programmierer die fehleranfällige manuelle Verwaltung von Speicher-Allokationen ab. Ist ein Objekt zu einem bestimmten Zeitpunkt während der Ausführung nicht mehr durch die virtuelle Maschine erreichbar, so wird dieser Speicher zu einem von der VM gewählten Zeitpunkt freigegeben. Der Garbage-Collector trägt dafür die Verantwortung und verwaltet den Heap-Speicher, von dem der Speicher für Objekte durch den `new`-Befehl allokiert wird. In der JamVM kommt dazu ein Stop-the-world-Mark&Sweep-Algorithmus[7] zum Einsatz. Dieser hält von Zeit zu Zeit die Ausführung aller Threads an. Anschließend analysiert er jeden Thread daraufhin, welche Referenzen dieser auf Objekte im Heap enthält. Die Objekte, auf welche die gefundenen Referenzen zeigen, werden markiert. Dies ist die *mark*-Phase. Hat der Algorithmus alle Threads analysiert kann er nun den Heap durchsuchen und alle nichtmarkierten Objekte, die er dabei findet, löschen. Dies ist die *sweep*-Phase. Zusätzlich besitzt die JamVM noch eine optionale *compaction*-Phase, bei der die Fragmentierung des Heaps dadurch reduziert wird, dass alle Objekte reallokiert und direkt hintereinander kopiert werden. Dies bedeutet jedoch auch, dass alle Referenzen, die auf das jeweilige Objekt verweisen, aktualisiert und an die neue Position angepasst werden müssen. Deshalb muss die

[7] Siehe dazu auch [Bac04] und [Jo96].

compaction-Phase die Adressen dieser Referenzen speichern und anschließend, nach dem Compaction-Vorgang, durch die neue Adresse des Objektes ersetzen. Der Compaction-Vorgang wurde für den Prototyp, der in dieser Arbeit entwickelt wurde, deaktiviert, um ihn bei der Entwicklung als potentielle Fehlerursache vorläufig ausschließen zu können.

Die Sprache Java bietet es an, in einer Klasse *finalizer*-Methoden zu definieren. Diese werden aufgerufen, bevor der Garbage Collector beschließt, ein bestimmtes Objekt zu löschen, weil kein Thread mehr eine gültige Referenz darauf enthält. Dies ist hilfreich, um externe System-Ressourcen, über die ein Objekt womöglich verfügt, freigeben zu können. In dieser Arbeit werden finalizer eingesetzt, um die Datenstrukturen wieder freigeben zu können, die für die Speicherung von Rolleninstanzen im Rahmen des Liftings verwendet werden. Details dazu befinden sich in Kapitel 7.

Ein Problem der Garbage Collection tritt bei besonderen Anordnungen zirkulärer Referenzen auf: Da ein Objekt erst dann gelöscht werden kann, wenn keine Referenz mehr auf dieses Objekt verweist, können solche zirkulären Konstruktionen ein Objekt ewig am Leben erhalten. In der Sprache ObjectTeams tritt eine solche Situation dadurch auf, dass Rolleninstanzen sowohl eine Referenz auf ihre Basis als auch auf ihre Teaminstanz enthalten. Umgekehrt enthalten auch Basisinstanzen und Teaminstanzen Referenzen auf Rolleninstanzen. In einer solchen Situation könnte eine Rolleninstanz nicht gelöscht werden, wenn ihre zugehörige Basisinstanz im Programm nicht mehr direkt erreichbar ist, da sie immer noch über eine Teaminstanz erreicht werden kann. Ist sie aber über eine Teaminstanz zu erreichen, so kann über sie auch die Basisinstanz erreicht werden. So kann die Basisinstanz nie freigegeben werden, obwohl das Programm selbst über keine Referenzvariable mehr verfügt, über das sie angesprochen werden könnte. Analog verhält es sich mit Teaminstanzen. Wir werden auf diese Situation in Kapitel 7 noch einmal zurückkommen. Um dieses Problem zu entschärfen, bietet Java die Möglichkeit an, schwache Referenzen (*weak references*) zu verwenden. Diese Referenzen zählt der GarbageCollector nicht mit, wenn er überprüft, ob ein Objekt noch erreichbar ist. Der ot-Compiler und die Laufzeitumgebung OTRE verwenden solche schwachen Referenzen, um das soeben skizzierte Problem der unlöschbaren Instanzen zu vermeiden.

Damit ist unsere kleine Reise durch die JamVM beendet. Wir haben jetzt ihre grundlegende Arbeitsweise kennengelernt. In den folgenden Kapiteln wird nun schrittweise eine neue Laufzeitumgebung für ObjectTeams/Java vorgestellt werden, die direkt in die JamVM integriert wurde, um ObjectTeams/Java mit bestmöglicher Effizienz unterstützen zu können. Im Anschluss an Kapitel 4, das die Aufgabenstellung für diese Arbeit beschreiben und den Lösungsansatz dafür skizzieren wird, werden dann alle Mechanismen, die für dieses Vorhaben in die JamVM integriert wurden, detailliert besprochen.

Kapitel 4

OTRun – Object Teams Runtime

- Delegationsbasierte Aspektausführungs-mechanismen für eine Java Virtual Machine

4.1 Motivation

Die Arbeit begann mit dem abstrakten Wunsch nach der Anpassbarkeit von Computerprogrammen und fokussierte sich dann auf eine mögliche Form, dieser Anpassbarkeit näher zu kommen – mit Hilfe der aspektorientierten Programmiersprache ObjectTeams/Java. Auch die Schwierigkeiten, die bei der Entwicklung eingebetteter Systeme zu berücksichtigen sind – Anpassbarkeit, minimaler Ressourcenverbrauch und Ausführungseffizienz – fanden dabei Erwähnung. Diese Arbeit möchte nun eine Brücke zwischen zwei Welten schlagen: zwischen der aspektorientierten Welt von ObjectTeams/Java und der Welt der eingebetteten Systeme. Sie möchte dies erreichen, indem sie die Laufzeitumgebung von ObjectTeams/Java ersetzt durch eine effiziente, delegationsbasierte Unterstützung der aspektorientierten Sprachmechanismen von ObjectTeams/Java innerhalb einer virtuellen Maschine.

Zur Implementierung des Prototyps „OTRun“ (Object Teams Runtime) wurde die JamVM ausgewählt, eine übersichtliche, kompakte, portable, interpretierende virtuelle Maschine für Java. Die Wahl dieser virtuellen Maschine besitzt zwei Vorteile: Zum einen handelt es sich bei der JamVM um eine virtuelle Maschine, die auch im Bereich der eingebetteten Systeme eingesetzt werden kann, z.B. auf einem Apple IPhone. Zum anderen vermeidet der Verzicht auf eine JIT-basierte, hochoptimierende VM, die mit diesen Verfahren einhergehende hohe Komplexität.

4.2 Delegation

Kapitel 1 erwähnte, dass zwei grundlegende Mechanismen existieren, um eine aspektorientierte Adaption bestehender Programme durchführen zu können: Das *Weben* und die *Delegation*.

In Kapitel 2 wurden dann die vielfältigen aspektorientierten Modularisierungsmöglichkeiten der Programmiersprache ObjectTeams/Java vorgestellt, die bei der Entwicklung anpassungsfähiger Applikationen für eingebettete Systeme sehr hilfreich sein könnten. Bisher verwendet diese Sprache einen *Webemechanismus*, um ihre Spracheigenschaften zu realisieren. Umgesetzt wird dieser durch ein Zusammenspiel aus ot-Compiler und dem ObjectTeamsRuntime-Environment OTRE. Letzteres verwendet einen aufwändigen Ladezeit-Webemechanismus, um Basisklassen aspektorientiert adaptieren zu können, so dass die Ausführung von Callins zur Laufzeit möglich wird. Dieser Vorgang ist nicht sehr flexibel, da er erzwingt, dass alle Teamklassen und auch alle Basisklassen vor der Programmausführung geladen werden müssen. Er ist auch nicht sehr effizient, da er zum einen vollständig in Java implementiert ist und zum anderen komplexe Bytecodetransformationen an Basisklassen vornehmen muss. Dazu müssen die entsprechenden .class-Files aufwändig zur Laufzeit geparst und dann modifiziert werden. Ihr Constantpool wird um neue Einträge erweitert und neue Bytecodefolgen werden generiert und in bestehende und neue Methoden eingefügt. Der eingefügte Bytecode muss dabei individuell an jede Basismethode angepasst werden. Dazu werden die Attribute, die der ot-Compiler in Team- und Rollenklassen gespeichert hat, zur Ladezeit vom Webemechanismus ausgewertet. Auch einige neue Felder müssen in den Basisklassen angelegt werden. Schließlich entstehen durch diesen Webemechanismus auch Abhängigkeiten zu externen Werkzeugen, die den Start des Webeprozesses einleiten und ein Interface für die Bytecodetransformation bereitstellen.

Das gleiche Webeverfahren zur Bytecodetransformation direkt innerhalb einer virtuellen Maschine durchführen zu wollen, wäre sehr aufwendig und würde im Endeffekt zu keiner Verbesserung der Ausgangslage führen. Daher soll das bisherige Ladezeitweben nun durch delegationsbasierte Mechanismen ersetzt werden. Diese sind innerhalb einer virtuellen Maschine einfach zu realisieren und erhöhen zugleich die Flexibilität der auf sie aufsetzenden Umgebung, in diesem Fall die Flexibilität der Sprache ObjectTeams/Java.

4.2.1 Ein delegationsbasiertes Maschinenmodell zur aspektorientierten Programmierung

Ein mögliches Modell für eine delegationsbasierte Unterstützung aspektorientierter Programmiersprachen in einer virtuellen Maschine wird in [HS07] vorgestellt. Es verwendet *Proxies*, um den Programmfluss zur Laufzeit dynamisch verändern zu können. Zunächst erhält jedes Objekt bzw. jede Klasse dazu bei der Allokation ein Proxyobjekt, so dass die Allokation eines neuen Objektes bzw. einer neuen Klasse

nun anstelle einer Referenz auf das eigentliche Objekt nun eine Referenz auf den Proxy zurückgibt. Alle Referenzen, die vom Programm verwendet werden, zeigen nun immer auf den Proxy, der wiederum als einziger einen Zeiger zum originalen Objekt bzw. der originalen Klasse besitzt.

In einem initialen Modell besitzt der Proxy selbst zunächst keine Funktionalität, sondern kann alle Anfragen, die an ihn gerichtet werden, für die er aber selbst keine Handlungsanweisungen besitzt, direkt an das Originalobjekt weiterleiten. Eine mögliche Implementierung eines solchen Verhaltens könnte durch eine „*method not understood*"-Ausnahmebehandlung realisiert werden. In diesem Fall würde der Proxy alle Methoden, die er kennt, bei jedem Aufruf daraufhin untersuchen, ob sich die gewünschte Methode dort finden lässt. Gelangt diese Suche zu keinem Ergebnis, so wird die Nachricht an das Originalobjekt weitergeleitet, in der Hoffnung, dass die Nachricht womöglich dort verstanden wird. Diese Form der Implementierung ist jedoch sehr ineffizient. Alternativ könnte der Proxy auch die Methodentabelle des Originalobjekts kopieren. Dann würde er automatisch zu dem Eintrag verzweigen, der an dieser Position zu finden ist und der aufwändige Suchvorgang könnte entfallen. Diese Lösung wiederum ist zwar effizient, verbraucht aber sehr viel zusätzlichen Speicherplatz.

Nachdem die Nachteile eines delegationsbasierten Mechanismus Erwähnung fanden, sollen nun auch seine Vorteile vorgestellt werden. Mit Hilfe eines Proxies ist es möglich, das Verhalten eines Objektes zur Laufzeit zu verändern. Dazu genügt es, den Proxy umzukonfigurieren oder ein weiteres Objekt zwischen den Proxy und das Originalobjekt einzufügen. Im ersten Fall würde der Proxy selbst nun eine Nachricht verstehen und könnte darauf reagieren, z.B. indem er eine andere Methode aufruft und den Aufruf erst dann an das ursprüngliche Objekt weiterleitet. Im zweiten Fall bleibt die Funktionalität des Proxies selbst unberührt, allerdings verändert sich die Delegationskette. Anstatt die Nachrichten, die er nicht kennt, an das Originalobjekt weiterzuleiten, leitet er sie nun zunächst an das eingefügte Objekt weiter. Dieses kann dann die Nachricht empfangen, bearbeiten und bei Bedarf wiederum eine Nachricht an das Originalobjekt versenden, das als nächstes in der Delegationskette folgt.

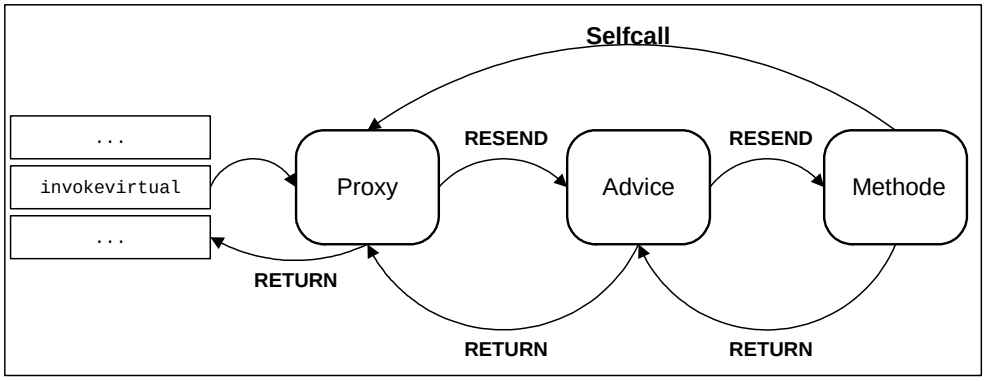


Abbildung 2 : Delegation mit Proxy und RESEND

Im Ansatz von [HS07] wird dazu der neue Befehl RESEND eingeführt. Sobald eine Methode RESEND ausführt, wird versucht, die gleiche Methode auf dem

nachfolgenden Objekt in der Delegationskette aufzurufen. Durch die Platzierung von RESEND am Anfang, in der Mitte oder am Ende einer Methode könnte so ein BEFORE-, REPLACE- oder AFTER-Advice realisiert werden.

4.2.2 Ein delegationsbasiertes Maschinenmodell für ObjectTeams/Java?

Auf die Programmiersprache ObjectTeams/Java lässt sich das delegationsbasierte Modell von [HS07] nicht direkt übertragen. Advices werden in ObjectTeams/Java durch Callin-Methoden in Rollenklassen repräsentiert. Sie können nur auf Rolleninstanzen ausgeführt werden, zu deren Ermittlung ein aufwändiger Lifting-Mechanismus notwendig ist, dessen Ergebnis von der aktuellen Basisinstanz, einer aktiven Teaminstanz und einer gewünschten Zielrollenklasse abhängt. Der Lifting-Mechanismus lässt sich jedoch nicht unmittelbar auf eine Delegationskette abbilden. Zusätzlich besitzt ObjectTeams/Java eine Reihe von Sprachmechanismen, die zwischen den Callin-Aufrufen stattfinden müssen. So muss eine Evaluierung von Guards stattfinden können und die dem Aufruf übergebenen Parameter müssen mit Hilfe eines Mappings auf die Parameter einer Rollenmethode abbildbar sein. Dabei kann es auch notwendig werden, dass Parameter von einem Methodenaufruf zu einem anderen Methodenaufruf, der in der Delegationskette erst viel später erfolgt, „getunnelt“ werden müssen.

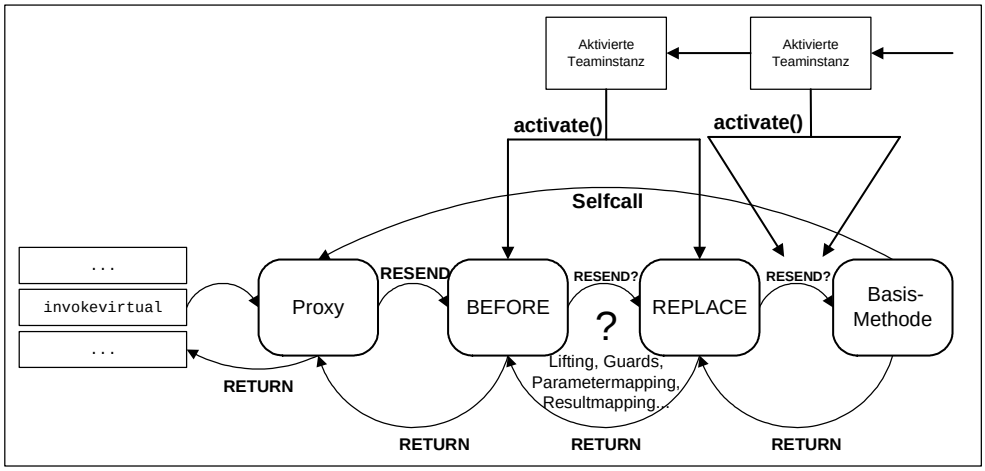


Abbildung 3 : Ein Maschinenmodell für ObjectTeams/Java?

In Abbildung 3 ist zu sehen, dass auch der Aufwand für die Teamaktivierung erheblich anwachsen würde. Jede aktivierte Teaminstanz müsste nun an der richtigen Stelle in der Delegationskette ihre Callins einfügen. Sie müsste diese bei einer Deaktivierung aber auch wieder daraus entfernen können. Dazu wäre ein Verfahren nötig, mit dem sie diese dazu effizient wieder finden kann.

Ein weiteres Problem würde daraus entstehen, dass ObjectTeams/Java keinen RESEND-Befehl kennt. Dieser müsste also zunächst in alle Callin-Methoden gewebt werden. Dies wäre vor allem für AFTER-Callins sehr schwer zu realisieren, da eine Methode an den unterschiedlichsten Stellen eine `return`-

Anweisung enthalten kann. Verändert man aber die `return`-Anweisung selbst, damit sie sich auch wie ein RESEND verhalten kann, erhöht dies die Laufzeit für alle Methoden, die dieses Verhalten nicht benötigen, aber den gleichen `return`-Befehl verwenden.

4.2.3 Proxies als Objekt und als Funktion

Das Maschinenmodell von [HS07] soll nun in ein Modell transformiert werden, dass die Spracheigenschaften von ObjectTeams/Java besser unterstützen kann. Ein erster Schritt dazu ist die Verlagerung des Proxies. Die Verwendung eines Objekts als Proxy kostet viel Speicherplatz. Ein Dispatch-Mechanismus für den Aufruf einer dem Objekt unbekannten Methode ist ebenfalls nur mit zusätzlichem Speicherplatzverbrauch für eine eigene Methodentabelle effizient realisierbar. Daher soll der Proxy nun nicht länger als Objekt aufgefasst werden, sondern als Abbildungsfunktion, die einen Satz von Parametern, der ihr übergeben wird, auf ein spezifisches Laufzeitverhalten abbildet. Diese Funktion kann nun durch einen Bytecode bzw. eine Bytecode-Sequenz repräsentiert werden. Anstatt diese Funktion aber als Methode in einem Proxyobjekt zu realisieren, das jedem allokierten Objekt vorangestellt wird, ist es ausreichend diese Proxyfunktion jeder Methode voranzustellen. Dies hat den Vorteil, dass Speicherplatz gespart werden kann, da die Anzahl der Methoden im Vergleich zu der Anzahl an Objekten zur Laufzeit wesentlich geringer ist.

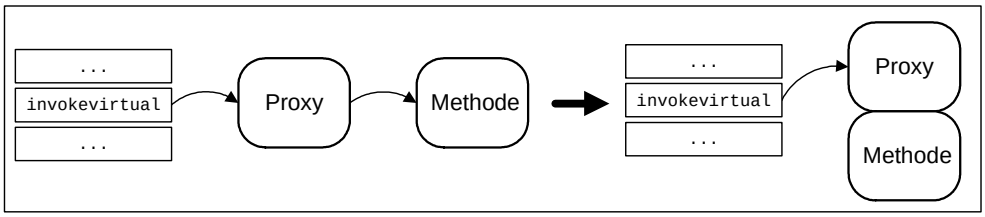


Abbildung 4: Transformation eines objektbasierten Proxy in eine Proxyfunktion

Durch diese Transformation verliert der Proxy die Möglichkeit, einen eigenen Zustand zu besitzen. Dies wird im Modell, das in dieser Arbeit verwendet wird, jedoch durch die Verwendung des Stacks zur Speicherung von temporären Informationen kompensiert. In Kapitel 9 wird dies im Detail beschrieben werden.

4.2.4 Der OTRun Delegationproxy

Im nächsten Schritt soll nun auch die Callin-Ausführung mit Hilfe des funktionsbasierten Proxies modelliert werden. Dies führt uns zu dem in Abbildung 5 zu sehenden Modell.

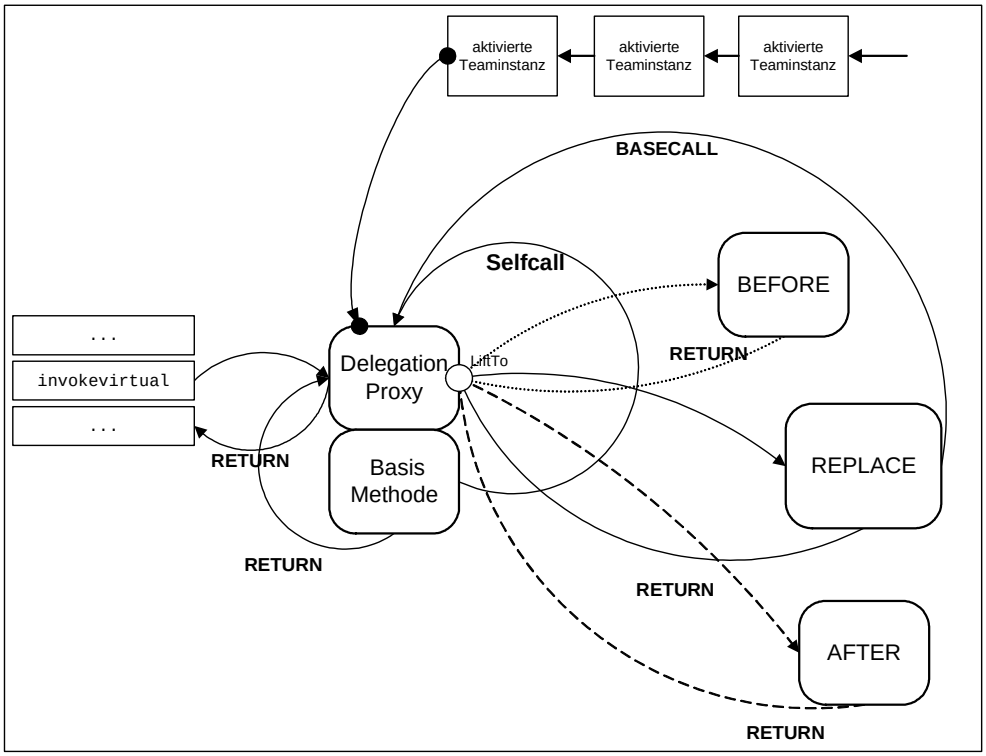


Abbildung 5: Der Delegationproxy - ein Maschinenmodell für ObjectTeams/Java

Der Delegationproxy ist nun eine Funktion die vor jeder Methode ausgeführt werden kann – und vor jeder Basismethode ausgeführt wird. Sie verwendet eine Liste mit aktiven Teaminstanzen um über sie die Callins zu bestimmen, die nun auszuführen sind. Dazu delegiert sie die Ausführung der Callin-Methoden mit Hilfe des Liftings an die entsprechenden Rollenobjekte, die mit der aktuellen Basisinstanz, auf der die Basismethode aufgerufen wurde, assoziiert sind. Mit Hilfe von Informationen, die sie aus den Teaminstanzen gewinnt, kann sie bestimmen, wann die Delegation zu einer bestimmten Rollenmethode stattfinden soll. Dies wird durch den Typ des Callins (BEFORE, AFTER oder REPLACE) und durch ein möglicherweise vorhandenes Precedencemapping bestimmt.

Für die Funktion des Delegationproxy sind daher mehrere aspektorientierte Mechanismen notwendig, die ihn darin unterstützen, herauszufinden, was er wann wohin delegieren soll.

Der Team*aktivierungsmechanismus* beantwortet dazu die Frage, **wann** delegiert werden soll und in **welchem Kontext**.

Der *Liftingmechanismus* beantwortet die Frage, **wohin** delegiert werden soll.

Und der *Delegationproxy* selbst gibt schließlich eine Antwort darauf, **wie** delegiert wird. Diese Frage soll in dieser Arbeit beantwortet werden.

4.3 Durchführung

Das Modell für den Delegationproxy, einem delegationsbasiertem Aspektausführungsmechanismus, soll nun prototypisch in einer Erweiterung für eine virtuellen Maschine implementiert werden. Verwendet wird dazu die JamVM.

4.3.1 Optimierung

Bei der Umsetzung soll auf eine möglichst hohe Effizienz der verwendeten Algorithmen und Verfahren geachtet werden. Daher sollen algorithmische Optimierungen der durch den ot-Compiler und das OTRE vorgegebenen Algorithmen bevorzugt werden. Diese kann im Idealfall beispielsweise zu einer Reduktion des Aufwands von O(n) zu O(1) führen. Zusätzlich wird versucht werden, komplexe Operationen durch selbstadaptierende Mechanismen zu optimieren, wie z.B. durch move-to-front-sortierte Listen oder durch die Verwendung von Caches. Außerdem wird an einigen Stellen versucht werden, den Overhead der durch Synchronisationsmechanismen für den Multithreadingbetrieb entsteht, dort zu reduzieren, wo dies sicher möglich ist. Außerhalb des Fokus dieser Arbeit steht jedoch eine Optimierung des C-Programmcodes direkt auf Sprachebene. Die Optimierungsgewinne, die mit den in dieser Arbeit präsentierten Mechanismen erzielt werden sollen, beruhen dadurch nicht auf Programmiertricks, sondern haben ihren Ursprung in alternativen Algorithmen – und natürlich der Realisierung als C-Funktionen innerhalb einer Java Virtual Machine.

Eine Reduktion des Speicherverbrauchs zur Laufzeit wird zwar ebenfalls angestrebt, bildet aber nicht den Schwerpunkt dieser Arbeit.

4.3.2 Delegationsbasierte Mechanismen

Die in dieser Arbeit vorgestellte Lösung soll ausschließlich delegationsbasierte Mechanismen einsetzen. Dies hat zur Folge, dass für alle Algorithmen von ObjectTeams/Java in denen direkte Beziehungen zu den Strukturen bestehen, die von den Webevorgängen des OTRE erzeugt werden, ein generisches Äquivalent zu finden ist. Dieses wäre dann nicht länger auf die Webevorgänge des OTRE angewiesen. In dieser Arbeit soll dies für das Lifting, die Teamaktivierung und mit dem Delegationproxy auch für die Callin-Ausführung angestrebt werden.

4.3.3 Umsetzung in der JamVM

Die Erweiterung OTRun für die JamVM befindet sich in den drei Quelldateien OTRun_Classloading.c, OTRun_Declarations.h und OTRun_Instructions.c. Bei der Implementierung wurde Wert darauf gelegt, möglichst viele Kommentare zu verwenden, um eine zukünftige Erweiterung und Fehlerkorrektur zu erleichtern.

Außerdem wurde versucht, die Anzahl an Modifikationen, die an dem Programmcode der JamVM vorzunehmen sind, möglichst gering zu halten, sie deutlich hervorzuheben und zu dokumentieren.

Durch die Verwendung eigener Container-Datenstrukturen und einer einheitlichen Namensgebung sollen außerdem der Namespace der JamVM und der Namespace der OTRun-Erweiterung möglichst getrennt voneinander gehalten werden.

Zur Ausführung von ObjectTeams/Java-Programmen wird zusätzlich eine kleine Bibliothek von Java-Classen benötigt. Diese enthält aus Kompatibilitätsgründen im wesentlichen alle Klassen des OTRE, fügt ihnen allerdings noch die Klassen RoleInstanceTable.class und TeamInstanceManager.class hinzu. Diese sind in der Datei otrun.jar zusammengefasst. Die Klasse Team.class wurde für die Zusammenarbeit mit OTRun modifiziert.

4.3.4 Zusammenfassung

In diesem Kapitel wurde das Konzept für einen „Delegationproxy“ vorgestellt, einem Proxy der als Funktion realisiert wird. Dieser soll die aspektorientierten Mechanismen von ObjectTeams/Java zur Ausführung von Callin-Methoden effizient unterstützen und zu einer Flexibilisierung von Callin-Bindungen zur Laufzeit führen. Außerdem soll er die Ausführung von ObjectTeams/Java-Programmen ohne die Verwendung von Weaving-Mechanismen erlauben und ausschließlich delegationsbasiert arbeiten.

Um an dieses Ziel zu gelangen, sind mehrere Schritte zu vollziehen:

Zunächst muss eine Kommunikationsmöglichkeit zwischen ObjectTeams/Java und der Welt des Delegationproxy in der VM etabliert werden. Dazu wird in Kapitel 5 das „Infrastructure Weaving“ vorgestellt werden, mit dessen Hilfe der Delegationproxy völlig transparent von ObjectTeams/Java-Programmen verwendet werden kann. Die JamVM wird in die Lage versetzt, ObjectTeams/Java-Programme direkt ausführen zu können.

Im nächsten Schritt wird eine Umgebung für den Betrieb des Delegationproxies und der von ihm verwendeten Mechanismen bereitgestellt. Außerdem müssen die Informationen, die in den Attributen von Team- und Rollenklassen gespeichert sind, ausgewertet werden. Dies wird in Kapitel 6 dargestellt, in dem der Classloading-Prozess in OTRun beschrieben wird.

Die darauf folgenden zwei Kapitel stellen jeweils einen Delegationsmechanismus vor, den der Delegationproxy benötigt. Dies ist in Kapitel 7 der Liftingmechanismus und in Kapitel 8 der Teamaktivierungsmechanismus.

In Kapitel 9 soll schließlich die Frage beantwortet werden, wie der Delegationproxy funktioniert.

Kapitel 5

Infrastructure Weaving

- Kommunikation zwischen OTRun und Java

5.1 Die Kommunikation zwischen virtueller Maschine und Java Programmen

Zur direkten Kommunikation zwischen einer Virtuellen Maschine und einem Java-Programm existieren zunächst zwei nahe liegende Möglichkeiten. Zum einen können native Methoden verwenden werden, die im Rahmen des Programmcodes der virtuellen Maschine implementiert sind. Zum anderen können neue Bytecodes definiert werden, die dann direkt von der virtuellen Maschine ausgeführt werden können. Beide Möglichkeiten besitzen jedoch schwerwiegende Nachteile, wenn sie zur direkten Kommunikation mit der VM verwendet werden. So kompromittiert die Verwendung von nativen Methoden in der Regel die Sicherheit des Systems, da native Methoden den Systemstack verwenden, den die virtuelle Maschine selbst für ihre eigene Ausführung benutzt. Es findet damit ein Wechsel zwischen der geschützten Java-Welt und der lokalen Systemwelt statt, der einen potentiellen Angriffspunkt für schädlichen Programmcode darstellt. Zusätzlich ist der Aufruf einer nativen Methode im Vergleich zur zweiten Möglichkeit sehr ineffizient (Aufruf eines Java-Wrappers (invokevirtual), Aufbau eines Java-Stackframes, Aufbau eines nativen Stackframes, Aufruf der eigentlichen nativen Methode, Rückkehr aus der nativen Methode, Rückkehr aus dem Java-Wrapper). Die zweite Möglichkeit, die Einführung neuer Bytecodes, führt zu einer nicht-standardisierten virtuellen Maschine. Außerdem wird nun ein modifizierter Compiler benötigt, der die neuen Bytecodes an der richtigen Stelle verwendet. Der Implementierungsaufwand ist also erheblich (Einen Bytecode in die virtuelle

Maschine einfügen, den Compiler modifizieren), dafür wird jedoch die bestmögliche Geschwindigkeit bezüglich der Kommunikation zwischen virtueller Maschine und Java-Programm erreicht, da im Gegensatz zur ersten Möglichkeit keine Methodenaufrufe stattfinden müssen. Die Parameter für den neuen Bytecode werden auf dem Java-Stack abgelegt und dann von der Implementierung des Bytecodes ausgewertet und wieder vom Java-Stack entfernt. Anschließend kann dieser einen Rückgabewert zurück auf den Stack legen, der von dem Java-Programm dann weiterverwendet werden kann.

5.2 Infrastructure Weaving - Eine alternative Kommunikationsmöglichkeit

Diese Arbeit stellt nun eine dritte Möglichkeit vor, welche die Nachteile dieser beiden Möglichkeiten vermeidet sowie die Vorteile beider Möglichkeiten kombiniert. Diese neue Möglichkeit, das 'Infrastructure Weaving', verwendet zur Implementierung der Funktionalität neue Bytecodes, bildet diese allerdings auf nicht-native Java-Methoden ab, indem der Aufruf einer bestimmten Java-Methode von der virtuellen Maschine in den Aufruf eines einzelnen, neuen Bytecodes direkt umgewandelt wird. Der Entwickler, der einen bestimmten Kommunikations-vorgang umsetzen möchte, bestimmt beim Infrastructure Weaving zunächst die gewünschte Methode einer bestimmten Klasse, die auf einen neuen Bytecode abgebildet werden soll, oder stellt eine eigene Klasse mit der entsprechenden Methode zur Verfügung. Die Parameter dieser Methode beschreiben dabei exakt die Parameter, die anschließend an die virtuelle Maschine übergeben werden, der Rückgabewert analog den Rückgabewert, den die virtuelle Maschine zurückliefern wird. Im Gegensatz zu einer nativen Methode werden diese Parameter und der Rückgabewert nun jedoch ausschließlich über den Java-Stack übergeben und befinden sich daher zu keiner Zeit auf dem System-Stack. Diese Methode, die der Entwickler für das Infrastructure Weaving bestimmt hat, kann nun von einem Java-Programm wie eine normale Java-Methode verwendet werden. Dadurch wird die gleiche Eleganz in der Verwendung erreicht, wie sie auch eine native Methode bereitstellen würde. Eine Anpassung des Compilers zur Verwendung eines neuen Bytecodes ist daher auch nicht erforderlich. Der Java-Compiler wird diesen Methodenaufruf in einen Bytecode aus der Familie der invoke-Bytecodes, z.B. invokevirtual, übersetzen. Damit nun jedoch nicht die Methode ausgeführt wird, sondern stattdessen der neue Bytecode, wird der invoke-Befehl bei seiner ersten Ausführung von der virtuellen Maschine durch den neuen Bytecode ersetzt. Dies geschieht analog zu der Bytecode-Transformation, wie sie für _quick-Befehle verwendet wird, jedoch mit dem Unterschied, dass anstelle des invoke_quick-Bytecodes der neue Bytecode an diese Stelle geschrieben und anschließend ausgeführt wird. Damit es tatsächlich zu dieser Transformation kommt, wird der `MethodBlock` dieser Methode vorher mit dem gewünschten Bytecode und einem optionalen Bytecode-Parameter annotiert und diese Annotation vom invoke-Bytecode ausgewertet. Die Annotation selbst erfolgte beim Ladevorgang der Klasse, welche diese Methode enthält. Die Lade-Routine der virtuellen Maschine muss daher für die Verwendung des Infrastructure Weavings um ein

entsprechendes Programmfragment erweitert werden. Dieses testet, ob die gerade geladene Klasse identisch mit der Klasse ist, welche die für die Kommunikation zu verwendende Methode enthält. Ist dies der Fall, wird nach der gewünschten Methode gesucht und ihr Methodenblock anschließend passend annotiert.

Der Name ‚Infrastructure Weaving' wurde gewählt, um den primären Einsatzzweck dieser Methode zu betonen. Bei ihr handelt es sich um ein Weaving-Verfahren, da ein Bytecode durch einen anderen ausgetauscht wird, der nun eine andere Funktionalität besitzen kann, als der ursprüngliche Bytecode. Außerdem besteht die Möglichkeit, dass sich die Funktionalität des neuen Bytecodes wie ein BEFORE-, REPLACE- oder AFTER-Callin verhält. Im Gegensatz zu einem allgemeinen Webeverfahren ist es dem Infrastructure Weaving jedoch nur möglich, eine konkrete Klasse einmalig zum Ladezeitpunkt so zu präparieren, dass ein Infrastructure Weaving zur Laufzeit stattfinden kann. Wurde eine Methode bereits einmal ausgeführt, so verliert ihr Bytecode die Fähigkeit, vom Infrastructure Weaving beeinflußt zu werden, da nach der Ausführung sämtliche Instruktionen, die ein Infrastructure Weaving unterstützen, bereits in einen neuen Bytecode transformiert wurden. Ihre Ursache hat dies darin, dass die Funktionalität zum Austausch des Bytecodes genauso wie die Funktionalität eines resolve-Vorgangs nur ein einziges Mal ausgeführt werden soll. Daher findet auch die Überprüfung, ob nun ein Infrastructure Weaving stattfinden soll nur ein einziges Mal in der initialen Variante des Bytecodes statt, bevor er sich selbst austauscht.

5.3 Ein Infrastructure-Weaving-Beispiel

Ein Beispiel soll nun die Funktionsweise des Infrastructure-Weavings erläutern: Die Klasse 'Team' enthält die Methode 'void activate()'. Diese soll in Zukunft direkt von der virtuellen Maschine ausgeführt werden. Daher wird die Klassen-Laderoutine der virtuellen Maschine[8] um folgendes Programmfragment erweitert:

```
if(cb->name == SYMBOL(otj_team))[9]
{
       otjReplaceOpcode(newUtf8("activate"),"()",class,
                        OTJ_OP_ACTIVATELOCAL,0, OPC_OTJAPI);
}
```

Wird die Klasse ‚Team' geladen, dann sucht die Funktion otjReplaceOpcode in dieser Klasse, die hier mit class übergeben wurde, nach einer Methode 'activate' mit der Signatur '()'. Wird diese gefunden, so wird in ihrem `MethodBlock` vermerkt, dass diese durch den neuen Bytecode OPC_OTJAPI[10] ersetzt werden soll. Dieser Bytecode wird anschließend mit zwei 16Bit Parametern versehen, die durch diese

[8] Der Beispielcode befindet sich tatsächlich in einer Unterfunktion von void otjCheckAttributes(Class class*) (Datei OTRun_Classloading.c), die am Ende der JamVM-Funktion void linkClass(Class* class) (Datei class.c) aufgerufen wird.

[9] cb ist ein Zeiger auf den ClassBlock der Klasse, die gerade geladen wird. Siehe dazu auch Kapitel 3.

[10] Die im Rahmen dieser Arbeit entstandene VM-Erweiterung OTRun verwendet die zwei neuen Bytecodes `OPC_OTJAPI` (Bytecode 248) und `OPC_OTJPROXY` (Bytecode 249).

Funktion ebenfalls im `MethodBlock` gespeichert werden. Der Parameter OTJ_API_ACTIVATELOCAL wird innerhalb einer switch-Anweisung im OPC_OTJAPI-Bytecode verwendet, um zu der Implementierung der Verhaltensweise für void activate() innerhalb des Bytecodes zu springen. Der zweite Parameter ist 0. Für die Abbildung einer _OT$LiftTo-Methode auf einen Bytecode würde der zweite Parameter dann z.B. einen Code enthalten, mit dem die von ihm verwendete Rollenklasse identifiziert werden kann.

Der für das Infrastructure Weaving angepasste invoke-Befehl überprüft, ob das bytecode-Feld innerhalb des MethodBlocks, der ihm vorliegt, nicht Null ist. Ist dies gegeben, so wird der Invoke-Bytecode durch den Bytecode, der im Bytecode-Feld codiert ist, ersetzt. Anschließend wird der neue Bytecode mit den Parametern aus dem `MethodBlock` vervollständigt und im Anschluss daran ausgeführt. Da diese Bytecode-Transformation erst nach dem prepare()-Vorgang stattfindet, der den Bytecode jeder Methode verifiziert und in ein internes Bytecodeformat umwandelt, kann der neue Bytecode innerhalb der virtuellen Maschine beliebig verwendet werden. Außerhalb der virtuellen Maschine, also z.B. in einem .class-File, kann er jedoch nicht verwendet werden, da ihn die Bytecode-Verifizierung des prepare()-Vorgangs als unbekannten Bytecode erkennen würde. Der neue Bytecode bleibt nach außen hin also unbekannt und kann nur durch den Aufruf der dafür vorgesehenen Methode verwendet werden.

5.4 Vergleich der Kommunikationsmethoden

Das im Beispiel beschriebene Programmfragment ist zusammen mit der Bereitstellung einer entsprechenden Methode in einer Klasse sowie der Implementierung der neuen Verhaltensweise in einem neuen Bytecode alles, was benötigt wird, um das Infrastructure Weaving einsetzen zu können. Es ist also nicht erforderlich, einen Compiler so zu erweitern, dass er einen neuen Bytecode verwendet, weil die gewünschte neue Funktionalität wie bei der Verwendung von nativen Methoden direkt in einem Java-Programm aufgerufen werden kann. Dennoch wird anstelle eines Methodenaufrufs der neue Bytecode verwendet. Damit vermeidet man den Geschwindigkeitsverlust und die Sicherheitsrisiken, die durch die Verwendung von nativen Methoden entstehen. Da der neue Bytecode erst nach dem prepare()-Vorgang verwendet wird, dieser den neuen Bytecode aber gewollter Weise nicht kennt, ist es zusätzlich unmöglich, den neuen Bytecode direkt im Bytecode einer Methode zu verwenden. Dies würde durch die Bytecode-Verifikation der virtuellen Maschine erkannt und verhindert. Dadurch bleibt die Java Virtual Machine trotz des intern verwendeten neuen Bytecodes standard-konform und kennt nach außen hin nach wie vor nur die Bytecodes, die im Standard definiert sind. So vereint das Infrastructure Weaving also alle Vorteile von nativen Methoden und neuen Bytecodes, nämlich Sicherheit, einfache Anwendbarkeit und hohe Geschwindigkeit, und vermeidet deren Nachteile, bestehend aus dem erhöhten Aufwand zur Verwendung (Compiler-Anpassung bei neuen Bytecodes), Sicherheitsrisiken (durch die Verwendung des Systemstacks bei nativen Methoden) und einer Verletzung des Standards für virtuelle Maschinen (bei der öffentlichen Einführung eines neuen Bytecodes). Der einzige Nachteil ergibt sich aus einer geringfügig höheren Laufzeit des ersten Aufrufs einer Methode

(bestehend aus einer if-Anweisung in der Programmiersprache der virtuellen Maschine (für die JamVM wäre das die Sprache C), die überprüft, ob der invoke-Bytecode durch den _quick-Bytecode - oder aber durch einen anderen Bytecode zu ersetzen ist). Diese erhöhte Laufzeit ist jedoch kaum messbar, da bei einem ersten Methodenaufruf auch der resolve-Vorgang für Methoden und der prepare()-Vorgang durchgeführt werden, so dass ein einzelnes `,if'` nicht nennenswert zur Gesamtlaufzeit des Befehls beiträgt. Zusätzlich kommt es auch zu einer geringfügig längeren Ladezeit für Klassen, bedingt durch die Überprüfung, ob für die gerade geladene Klasse ein Infrastructure Weaving durchzuführen ist oder nicht.

Da der Austausch von bestimmten Bytecodes durch _quick Varianten auch in Virtuellen Maschinen zum Einsatz kommt, die Just-in-time Kompilierungs-techniken verwenden, kann Infrastructure Weaving prinzipiell auch dort implementiert werden. Infrastructure Weaving kann bei allen Befehlen eingesetzt werden, die über eine _quick-Variante verfügen, da dies in der Regel auch alle Bytecodes sind, die den Constantpool oder eine vergleichbare Datenstruktur adressieren. Dazu gehören unter anderem getfield, setfield und new. Diese Bytecodes bieten sich für das Infrastructure Weaving an, da ihnen entweder ein `MethodBlock` oder ein Feld im Constantpool individuell zur Verfügung steht, in dem gespeichert werden kann, ob eine Infrastructure-Weaving-Transformation für das spezielle Ziel dieses Bytecodes stattfinden soll oder nicht.

5.5 Einsatzmöglichkeiten für Infrastructure Weaving

Infrastructure Weaving wird im weiteren Verlauf dieser Arbeit verwendet, um die wesentliche Kernfunktionalität von ObjectTeams (Lifting, Aktivierung, Callin-Ausführung) in eine virtuelle Maschine übertragen zu können, ohne dabei den ot-Compiler verändern zu müssen oder die alte Laufzeitumgebung OTRE zu verwenden. Damit wird anschaulich die Nützlichkeit dieser Methode zur Kommunikation zwischen Java Programm und virtueller Maschine demonstriert.

Aber auch andere Einsatzgebiete für Infrastructure Weaving sind denkbar. So könnte mit Hilfe von Infrastructure Weaving ein beliebiges API direkt in eine virtuelle Maschine integriert werden (z.B. für Grafik oder zur zeitnahen Abfrage von Sensoren oder Systeminformationen). Auch Kryptographie-Algorithmen oder besondere Transaktionsverfahren ließen sich so in der VM selbst verstecken (und wären damit z.B. gut vor Manipulation geschützt, da die virtuelle Maschine mit ihren Sicherheitsmechanismen selbst als Brandmauer wirken würde). Prozessor-Befehlserweiterungen wie z.B. SSE (Intel x86) könnten ebenfalls auf Java Methoden abgebildet werden, die dann direkt in einem Programm verwendet werden können, um bestimmte Berechnungen zu beschleunigen. In virtuellen Maschinen ohne diese Erweiterung könnten sie durch mitgelieferten Java Code, der das API nach außen hin bereitstellt, emuliert werden. Schließlich können sich selbstgeschriebene Befehle, die für Infrastructure Weaving konzipiert wurden, ähnlich verhalten wie ein REPLACE-Callin in ObjectTeams/Java. Sie könnten einzelne Methoden komplett ersetzen oder sie während ihrer Abarbeitung aufrufen.

Infrastructure Weaving wäre damit eine Technik, die es erlauben würde, spezialisierte virtuelle Maschinen für bestimmte Anwendungs- oder Einsatzgebiete mit relativ wenig Aufwand maßgeschneidert entwickeln zu können, um diese dann z.B. serverseitig in einem komplexen Transaktionssystem einsetzen zu können. Außerdem könnte sie in einem kleinen, eingebetteten, mobilen Gerät die bestmögliche Ausnutzung aller verfügbaren Ressourcen und spezieller Peripherie ermöglichen.

Ein weiteres interessantes Einsatzgebiet wäre die Kombination von Infrastructure Weaving mit Meta-Objekt-Protokollen, die es direkt erlauben sollen, die Semantik einer Programmiersprache zu verändern. Eine Programmiersprache, die ihre Semantik ähnlich wie ObjectTeams/Java partiell über Java-Methoden erhält (wie z.B. lifTo(...) und activate()) oder die komplett durch das Zusammenspiel von Java-Methoden ausgedrückt wird, ließe sich mit Infrastructure Weaving einfach und effizient auf eine virtuelle Maschine abbilden. Die Methoden eines Meta-Protokolles könnten dann die Arbeitsweise der Maschine selbst beeinflussen. Eine Variante eines solchen Meta-Protokolls für Aspektorientierte Programmierung wird in [Din09] vorgestellt.

Kapitel 6

Object Teams Class Data

- Classloading mit OTRun

6.1 Classloading

Die OTRun-Implementierung der drei Mechanismen Lifting, Teamaktivierung und Callin-Ausführung ist jeweils auf einen Satz von Datenstrukturen angewiesen, der zum Ladezeitpunkt einer Klasse erzeugt und initialisiert werden muss. Dieser Prozess ist in der Funktion `void otjCheckAttributes(Class* class)` realisiert, die am Ende der JamVM-Funktion `void linkClass(Class *class)` aufgerufen wird. Zum Zeitpunkt des Aufrufs von `otjCheckAttributes` ist der `ClassBlock` einer Klasse nahezu vollständig initialisiert. Nur die Initialisierung statischer Felder wurde noch nicht durchgeführt. Da `otjCheckAttributes` nicht auf statische Felder einer Klasse zugreift, spielt dies jedoch keine Rolle.

Bevor otjCheckAttributes aufgerufen wird, wurde die JamVM jedoch noch um einige weitere Funktionen zur Initialisierung ergänzt. Zunächst wurden der `ClassBlock` und der `MethodBlock` jeweils um einen Zeiger auf eine neue Datenstruktur erweitert, in dem alle Felder zu finden sind, um die der `ClassBlock` bzw. der `MethodBlock` für die Funktionalität von OTRun zu erweitern wären. Dies soll die Felder, die für OTRun verwendet werden, von den Feldern der JamVM trennen, so dass die Übersichtlichkeit darüber erhalten bleibt, welches Feld seinen Ursprung in der JamVM besitzt und welches Feld in OTRun. Bei diesen neuen Datenstrukturen handelt es sich um die Typen `ObjectTeamsClassData` (otjc) und `ObjectTeamsMethodData` (otjm). Sie werden allokiert und initialisiert sobald die

JamVM einen `ClassBlock` bzw. einen `MethodBlock` anlegt. Dafür wurden entsprechende Funktionen in den Programmcode der JamVM eingefügt.

Dieses Kapitel verwendet die Abkürzungen otjc bzw. otjm wenn die entsprechenden Datenstrukturen derjenigen Klasse gemeint sind, die gerade geladen wird. Die Abkürzungen otjcT, otjcB und otjcR verweisen auf die ObjectTeamsClassData-Datenstruktur einer Teamklasse, Basisklasse oder Rollenklasse. Im weiteren Verlauf der Arbeit werden die Felder dieser Datenstrukturen direkt dem ClassBlock bzw. dem MethodBlock zugeordnet. Zwischen ClassBlock und ObjectTeamsClassData-Struktur wird dann nicht mehr unterschieden.

Die Funktion `void defineClass(...)` der JamVM wurde ebenfalls erweitert. Diese ist dafür zuständig eine .class-Datei zu parsen und dabei ihren Inhalt in passenden Datenstrukturen des ClassBlocks zu speichern. Sie liest nun auch alle ihr unbekannten Attribute aus einer .class-Datei und speichert sie in otjc. Darunter fallen auch alle Attribute, die der ot-Compiler angelegt hat, damit das OTRE alle Informationen erhält, die es für ihren Weaving-Prozess benötigt. Auch OTRun benötigt diese Attribute. Außerdem wurde `defineClass` (...) um eine Funktion erweitert, die einen Verweis auf die gerade geladene Klasse in einer Subklassenliste im otjc ihrer Superklasse speichert. Mit Hilfe dieser Liste wird zur Laufzeit der vollständige Vererbungsbaum erfasst und kann sowohl Richtung Wurzel als auch in Richtung der Blätter traversiert werden. OTRun benötigt diese Möglichkeit unter anderem zur *Smartlifting*-Analyse, zur *RootRole*-Analyse und zum Setzen von Callins an verschiedenen Punkten einer bereits bestehenden Basisklassenhierarchie.

Die Funktion `linkClass(...)` ist unter anderem dafür zuständig, die `MethodTable` aufzubauen, die für das dynamische Binden bei Methodenaufrufen benötigt wird. An dieser Stelle wurde die Funktion so erweitert, dass sie nun ein Flag in jedem `MethodBlock` setzt, der für eine Methode steht, die eine geerbte Methode durch Overriding ersetzt. Dies erleichtert OTRun das Erkennen dieses Spezialfalles, der beim Setzen von Callins zu beachten ist.

6.2 Aufgaben des OTRun-Classloading-Prozesses

Für die Unterstützung von Lifting, Teamaktivierung und Callin-Ausführung werden verschiedene Informationen benötigt. In diesem Abschnitt sollen diese kurz vorgestellt werden. Ihre Verwendung wird dann in den Kapiteln 7, 8 und 9 im Detail beschrieben. Außerdem soll in diesem Abschnitt auf Shadowproxies eingegangen werden, die notwendig sind um die Unabhängigkeit der Ladereihenfolge von Team- und Basisklassen zu gewährleisten. Separate Erwähnung findet außerdem das Callin-Deployment.

6.2.1 Shadowproxies

Bei einem Shadowproxy handelt es sich um einen otjcB, der für eine Basisklasse allokiert wurde, die noch gar nicht geladen wurde. Beim Laden einer Teamklasse wird überprüft, ob alle Basisklassen an die sie über ihre Rollenklassen bindet, bereits geladen wurden. Für alle Basisklassen, bei denen dies nicht der Fall ist, wird ein Shadowproxy verwendet, der dann stellvertretend alle Informationen zu dieser Basisklasse speichert. Shadowproxies werden global in einer Liste gespeichert. Jeder Shadowproxy enthält den Namen der Basisklasse, für die er als Stellvertreter auftritt. Wird eine Klasse geladen, so wird diese Liste nach einem Shadowproxy durchsucht, der den selben Namen wie diese Klasse enthält. Wird er gefunden, so wird dieser Shadowproxy aus der Shadowproxy-Liste entfernt und von der Basisklasse zukünftig als ihre otjc-Datenstruktur verwendet. Durch das Auffinden eines Shadowproxies erkennt eine Klasse, dass sie selbst tatsächlich eine Basisklasse ist.

6.2.2 Callin-Deployment

Damit für eine Basismethode Callins ausgeführt werden können, muss für diese Basismethode ein *Callin-Deployment* stattfinden. Ein Callin-Deployment speichert im otjm der Methode eine CallinID und einen Zeiger auf eine CallinSignature.
Diese werden durch entsprechende Algorithmen von den Funktionen

```
       int                 otjGetCallinID(otjm, otjm->CallinID)
und    CallinSignature*    otjGetCallinSignature(otjm)
```

ermittelt, die berücksichtigen, welche Teamklassen jeweils Callins auf diese Basismethode setzen. Außerdem wird der Codezeiger auf den Start des Bytecodes der Methode so verändert, dass er nun auf den Beginn des Delegationproxies zeigt. Zur Unterstützung der Rückkehr aus Basismethoden wird außerdem die Größe des Rückgabewertes berechnet. Diese kann 0, 4 oder 8 Byte betragen und wird vom Delegationproxy benötigt.

In jedem otjcB ist eine `deployList` gespeichert, die Einträge für jede Methode enthält, für die ein Callin-Deployment durchzuführen ist.

6.2.2.1 Ermittlung der CallinSignature

Zur Ermittlung einer passenden CallinSignature, die unter anderem bei der Teamaktivierung verwendet wird, wird eine globale Liste aller bisher vorhandenen CallinSignatures verwendet. Eine CallinSignature enthält eine Liste aller Teamklassen, die alle gemeinsam an eine Basismethode binden. Es kann mehrere Basismethoden geben, welche die gleiche CallinSignature verwenden. Binden beispielsweise Teamklasse A und Teamklasse B an die Methoden x() und y(), so enthält der `MethodBlock` von x und der `MethodBlock` von y jeweils einen Verweis auf die gleiche CallinSignature, die auf die Klassen A und B verweist. Bestimmt werden die CallinSignatures, indem die deployList eines Methodenblocks

untersucht wird. Dazu wird zunächst die globale CallinSignature-Liste `otjCallinSignatures` nach dem ersten Vorkommen der Teamklasse durchsucht, die zu dem ersten Element in der deployList gehört. Wurde eine CallinSignature in der globalen Liste gefunden, so wird anschließend überprüft, ob in dieser CallinSignature auch alle weiteren Teamklassen aus der deployList vorkommen. Ist dies nicht der Fall, so wird nach einer weiteren passenden CallinSignature gesucht. Wenn keine gefunden werden konnte, wird eine neue CallinSignature erzeugt und ein Zeiger auf sie zurückgegeben. Außerdem wird `otjCallinSignatures` um diese CallinSignature erweitert.

6.2.2.2 Ermittlung der CallinID

Jeder Basismethode wird eine einzigartige `CallinID` zugewiesen. Diese wird als Index in die `CallinTable` einer Teamklasse verwendet, um herauszufinden, welche Callins für eine bestimmte Teaminstanz durch den Delegationproxy auszuführen sind. Der Eintrag der `CallinTable` verweist auf einen Eintrag des Arrays `bmBaked`, der diese Informationen enthält. Jeder Callin Deskriptor enthält diese Position ebenfalls im Feld `bmidx`, damit sie nun an die Stelle in der `CallinTable` kopiert werden kann, die durch die neu ermittelte `CallinID` bestimmt wird.

Zur Bestimmung einer neuen `CallinID` werden nun die CallinTables aller Teamklassen aus der CallinSignature nach einem freien Eintrag untersucht, der für alle Teamklassen an der gleichen Position liegen muss. Konnte eine solche Position nicht ermittelt werden, so werden alle CallinTables so weit vergrößert, bis dies der Fall ist. Die so ermittelte Position wird als `CallinID` zurückgegeben. Anschließend werden alle CallinTables der beteiligten Teamklassen aktualisiert, indem an die Position, auf die `CallinID` nun verweist, der Eintrag des Arrays `bmBaked` gespeichert wird, auf den der `bmidx` aus dem Callin Deskriptor für die jeweilige Teamklasse verwies[11].

6.2.2.3 Callin-Deployment-Sonderfälle

Beim Callin-Deployment können eine Reihe von Sonderfällen auftreten. So muss das Callin-Deployment statische Basismethoden bzw. statische Rollenmethoden unterstützen[12]. Bei Callins auf implizit geerbte Methoden einer Rollenklasse ist möglicherweise die modifizierte Signatur zu berücksichtigen, um die passende Methode in der Basisklasse finden zu können[13]. Der häufigste Sonderfall dürfte jedoch das Callin-Deployment für eine Methode sein, die von der Basisklasse lediglich aus einer ihrer Superklassen geerbt wurde. Für diese Methoden existiert in der Basisklasse kein eigener Methodenblock sondern nur ein Eintrag in der `MethodTable`, der auf den Methodenblock in der Superklasse verweist. OTRun

[11] Siehe auch Kapitel 9 Abschnitt 9.6.1.
[12] Der OTRun Prototyp unterstützt noch keine Callins für statische Methoden.
[13] Implizite Vererbung wird vom OTRun Prototyp bisher nicht berücksichtigt. Siehe auch Kapitel 2, Abschnitt 2.2.5 bzgl. impliziter Vererbung und [OTJLD §1.3].

kopiert deshalb diesen Methodenblock und speichert ihn im otjcB. Anschließend überschreibt er den Zeiger, der in der `MethodTable` gespeichert ist, mit dem Zeiger, der auf den neuen Methodenblock zeigt. Da alle invoke-Befehle, die dynamisches Binden verwenden, den passenden `MethodBlock` über einen Index in die `MethodTable` adressieren, verwenden sie zukünftig den durch das Callin-Deployment modifizierten `MethodBlock`. Dieser enthält dann entsprechend die CallinID, den CallinSignature-Zeiger und einen modifizierten Codezeiger, der auf den Delegationproxy verweist.

6.2.2.4 Callin-Vererbung

Da Callin-Bindungen vererbt werden, müssen nach der Modifizierung des Methodenblocks der Basismethode nun auch alle Subklassen entsprechend angepasst werden. Dazu müssen alle Subklassen gefunden werden, welche diese Basismethode durch ein Override ersetzen. Für die entsprechenden Methodenblöcke der Override-Methoden muss dann ebenfalls ein Callin-Deployment stattfinden. Subklassen, welche die Basismethode nicht durch Overriding ersetzen, verwenden über die `MethodTable` automatisch den Methodenblock derjenigen Klasse, die diese Methode definiert. Daher wirkt sich ein Callin-Deployment automatisch auf diese Subklassen aus, solange sie die Basismethode nicht durch Overriding ersetzen.

6.2.3 Lifting

Das Lifting benötigt eine ganze Reihe von Datenstrukturen. In der `rcTable` werden alle Rollenklassen einer Teamklasse aufgelistet. Die `bcTable` enthält Verweise auf alle durch diese Rollenklassen gebundenen Basisklassen. Eine `SmartTable` enthält Informationen, die für das Smartlifting benötigt werden. In Basisklassen muss für das Lifting eine `tcTable` angelegt werden, die auf alle Teamklassen verweist, die Rollenklassen mit Bindungen zu dieser Basisklasse enthalten.

Von großer Bedeutung für das Lifting ist die *RootRole*-Analyse. Sie wird in Kapitel 7 beschrieben. In der tcTable ist eine Liste aller RootRoles für jede enthaltene Teamklasse gespeichert. Für jeden Eintrag in der RootRole-Liste erfolgt beim Laden einer Basisklasse eine *Smartlifting-Analyse*, die ebenfalls in Kapitel 7 beschrieben wird.

6.2.4 Callin-Ausführung

Die Callin-Ausführung ist von der `CallinTable` und dem Array `bmBaked` abhängig, die in jedem otjcT angelegt werden. Bei `bmBaked` handelt es sich um eine optimierte Variante der `bmTable`. Für jede Basismethode, an die aus der Teamklasse des otjcT heraus ein Callin gebunden wird, enthalten sowohl `bmBaked` als auch die `bmTable` einen Eintrag. Dieser Eintrag enthält in der `bmTable` jeweils eine Listenstruktur mit Knoten für alle Rollenklassen, die einen Callin auf diese

Basismethode setzen. Jeder dieser Knoten enthält wiederum je eine Liste aller BEFORE-, REPLACE- und AFTER-Callins. Mit Hilfe von Precedence[14]-Werten können diese Listen einfach sortiert werden, damit der Delegationproxy alle Callins in der korrekten Reihenfolge ausführen kann. Der Aufbau von `bmBaked` wird in Kapitel 9 erläutert. Jeder Basismethode wird, wie in Abschnitt 6.2.2.2 erwähnt, eine `CallinID` zugeordnet, die als Index für die `CallinTable` einer Teamklasse verwendet wird. In der `CallinTable` befindet sich an dieser Position ein Verweis auf den Eintrag des Arrays `bmBaked`, der zu der Basismethode gehört, aus der die `CallinID` stammt.

6.2.5 Teamaktivierung

Die Teamaktivierung verwendet die in Abschnitt 6.2.2.1 beschriebenen CallinSignatures. Zur Cache-Invalidierung, die in Kapitel 8 beschrieben wird, benötigt sie außerdem die callinSignatureTable, die in jedem otjcT gespeichert ist. Diese enthält Verweise auf alle CallinSignatures, in welcher die Teamklasse des otjcT vorkommt.

6.3 Die Funktion `otjCheckAttributes`

Die Funktion otjCheckAttributes fasst alle Classloading-Phasen zusammen, welche die Datenstrukturen aufbauen, die von OTRun benötigt werden. Es existieren fünf Phasen, die jeweils in eine eigene Funktion ausgelagert wurden.

6.3.1 Phase 1: Laden einer Basisklasse

Phase 1 umfasst alle Schritte, die beim Laden einer neuen Basisklasse anfallen. Dazu gehört das Übertragen von Informationen aus einem *Shadowproxy*, das Anlegen einer `tcTable` im otjc, eine optionale *Smartlifting-Analyse* und das Setzen von Callins auf Methoden dieser Basisklasse (*Callin Deployment*).

Beim Anlegen einer tcTable wird zunächst die `tcTable` der Superklasse kopiert. Die `tcTable` enthält einen Eintrag für jede Teamklasse, die an diese Klasse bindet. Da Bindungen vererbt werden, erbt eine Subklasse auch alle Einträge der `tcTable` ihrer Superklasse. Da jedoch auch der Shadowproxy bereits eine `tcTable` enthielt, müssen diese nun zusammengeführt werden. Dies hat seine Ursache darin, dass die Vererbungshierarchie der Basisklasse bisher noch nicht bekannt war, denn diese Information wird erst durch den Ladevorgang der Basisklasse verfügbar. Immer dann, wenn eine Rollenklasse aus einer *RoleHierarchy*[15] eine Bindung an eine Superbasisklasse enthält und eine Rollenklasse aus einer anderen RoleHierarchy an

[14] Precedence-Werte können über das Precedence-Attribut gewonnen werden. Der OTRun-Prototyp unterstützt noch keine Precedences. An dieser Stelle könnte die Unterstützung jedoch mit wenig Aufwand implementiert werden.
[15] Siehe dazu Kapitel 7.

die aktuelle Basisklasse bindet, entsteht ein doppelter `tcTable`-Eintrag – ein Eintrag in der `tcTable` der Superklasse der Basis und ein Eintrag in der Shadowproxy-`tcTable`. Es gibt weitere Fälle, in denen dies auftritt. Der Eintrag in die Shadowproxy-`tcTable` muss getätigt werden, da beim Laden der Teamklasse die Information gespeichert werden muss, dass es diese Teamklasse gibt und sie an diese Basisklasse bindet. Da die Superklassen einer nichtgeladenen Basisklasse jedoch unbekannt sind, kann nicht ermittelt werden, ob die Teamklasse möglicherweise durch andere Rollenklassen an eine Superklasse der Basisklasse bindet. Das Kopieren und Zusammenfügen wird von der Funktion `otjMergeTCTables(class* Class)` durchgeführt, die zu Beginn der Phase 1 für alle Klassen, die geladen werden, ausgeführt wird.

Jede Basisklasse, für die Callins existieren, besitzt eine deployList in ihrem otjcB. Diese enthält Deskriptoren zum Deployment von Callins. In jedem dieser Deskriptoren ist der Name und die Signatur der Basismethode gespeichert, an die das Callin bindet. Diese Deskriptoren können von unterschiedlichen Teamklassen stammen, daher ist es möglich, dass mehrere Deskriptoren existieren, die an die gleiche Methode binden. Deshalb enthält jeder Deskriptor neben dem Namen und der Signatur der Basismethode auch einen Verweis auf die Teamklasse, die diesen Deskriptor erzeugt hat, als sie geladen wurde. In Phase 1 wird nun die deployListe abgearbeitet und die deployDeskriptoren auf die jeweiligen Basismethoden verteilt. Dazu besitzt auch jeder otjm eine deployList, die diese aufnimmt. Bei diesem Vorgang wird in jedem otjm ein Flag gesetzt, welches aussagt, das mindestens ein Callin für diese Basismethode existiert. Nachdem alle Deskriptoren verteilt wurden, wird das MethodBlock-Array der Basisklasse durchlaufen und nach Methodenblöcken gesucht, die dieses Flag enthalten. Ist dies der Fall, dann wird für sie ein CallinDeployment durchgeführt, wie es in Abschnitt 6.2.2 bereits erläutert wurde. Sowohl bei der Verteilung der deployList auf einzelne Methodenblöcke als auch beim anschließenden CallinDeployment werden die Spezialfälle aus Abschnitt 6.2.2.3 berücksichtigt.

6.3.2 Phase 2: Attribut-Analyse für Team- und Rollenklassen

Phase 2 analysiert die ObjectTeams-Attribute, die in den .class-Dateien von Team- und Rollenklassen gespeichert sind. Wird gerade eine Teamklasse geladen, so lädt sie zusätzlich alle Rollenklassen dieser Teamklasse, analysiert die Vererbungsbeziehungen zwischen den Rollenklassen, ermittelt die *RootRoles* und baut mit diesen Informationen die wichtigen Tabellen `rcTable` und `bcTable` im otjc auf. Dazu muss eine Vererbungshierarchie-Analyse aller Rollenklassen einer Teamklasse durchgeführt werden. Diese ist für das Lifting notwendig und ihr Zweck wird in Kapitel 7 erläutert.

Der OTRun-Prototyp analysiert bisher die ObjectTeams-Attribute `OTClassFlags`, `CallinPrecedence`, `CallinRoleBaseBindings` und `CallinMethodMappings`.

6.3.3 Phase 3: Callin-Analyse beim Laden einer Teamklasse

In der Phase 3 werden die Callin-Attribute aus allen Rollenklassen einer Teamklasse analysiert, wenn gerade eine Teamklasse geladen wird. Die Ergebnisse dieser Analyse werden dann dazu verwendet, um die `bmTable` aufzubauen und deployListen in Basisklassen anzulegen. Diese enthalten Deskriptoren mit Informationen zu den Callins für einzelne Basismethoden, mit deren Hilfe das Callin Deployment realisiert wird. Außerdem wird in dieser Phase Infrastructure Weaving[16] eingesetzt, um Basecalls in REPLACE-Methoden durch den Basecall-Opcode des Bytecodes `OPC_OTJPROXY` zu ersetzen.

Am Ende von Phase 3 wird der Inhalt der bmTable in die bmBaked-Tabelle übertragen und dabei optimiert, indem die verketteten Listen der bmTable zu einem Array mit statischer Länge zusammengefasst werden. Dies reduziert die Anzahl der Cachelines, auf die wärend der Ausführung des Delegationproxies zugegriffen werden muss. Die Einträge in der bmBaked-Tabelle steuern die Ausführung des Delegationproxies und werden in Kapitel 9 beschrieben.

6.3.4 Phase 4: Infrastructure Weaving

Die Phase 4 ist ausschließlich für das Infrastructure-Weaving von OTRun zuständig. Sie ersetzt eine Reihe von Methoden, die zur Laufzeit eingesetzt werden, um in ObjectTeams/Java das Lifting und die Teamaktivierung zu realisieren. Dabei handelt es sich um Methoden, die der ot-Compiler erzeugt, bzw. die durch die Klasse Team vorgegeben sind. Der OTRun-Prototyp ersetzt bisher die folgenden Methoden: activate(), deactivate(), _OT$implicitlyActivate, _OT$implicitlyDeactivate und alle _OT$LiftTo$-Methoden. Außerdem führt er einen Opcode für die neue statische Methode `_OT$set_ALL_THREADS(...)` ein, die einmalig beim Laden der Klasse Team aufgerufen wird. Sie wird dazu verwendet, OTRun die Referenz auf das Threadobjekt ALL_THREADS zu übergeben, die für die globale Teamaktivierung benötigt wird, sowie Referenzen auf Instanzen aller Exception-Klassen, die von ObjectTeams/Java verwendet werden. Schließlich werden auch noch die Methoden `_OTRun$createTIM`, `_OTRun$finalizeTIM`, `_OTRun$createRIT` und `_OTRun$finalizeRIT` durch Opcodes ersetzt. Diese werden in den Konstruktoren und den Finalizern von TeamInstanceManager-Instanzen und RoleInstanceTable[17]-Instanzen verwendet.

6.3.5 Phase 5: Laden einer Teamklasse

Phase 5 führt analog zu Phase 1 ein Callin Deployment durch, behandelt aber den Fall, dass gerade eine Teamklasse geladen wurde. Für alle bereits geladenen Basisklassen einer Teamklasse wird in dieser Phase nun ein Callin Deployment durchgeführt. Vorher müssen allerdings alle tcTables derjenigen Klassen und

[16] Siehe dazu Kapitel 5.
[17] Siehe dazu Kapitel 7

Subklassen erweitert werden, die von Bindungen der Teamklasse betroffen sind. Der neu erzeugte tcTable-Eintrag ist dann mit dieser Teamklasse zu initialisieren. Dabei ist jedoch zu berücksichtigen, dass die Position der Teamklasse in der tcTable für alle Klassen identisch sein muss. Dies ist nicht automatisch gewährleistet, da die tcTables in unterschiedlichen Zweigen der Vererbungshierarchie unterschiedlich groß sein können. Daher wird zunächst eine freie Position gesucht, die in allen Zweigen verwendet werden kann. Anschliessend wird die Größe von tcTables, die zu klein für den errechneten Index sind, angepasst. Im nächsten Schritt sind für alle Basisklassen an die eine Bindung besteht, b_{Dyn}-Werte für das Lifting zu vergeben. Dazu muss die in Kapitel 7 beschriebene Invariante eingehalten werden, die besagt, dass der `bDyn`-Wert von Subklassen immer größer sein muss, als die `bDyn`-Werte der Superklassen. Deshalb muss zuerst die Rollenklasse gefunden werden, die an die höchste Position in der Basisklassenhierarchie bindet. Von dort aus muss die `bDyn`-Vergabe erfolgen. Dies ist immer auch zugleich eine der RootCasses. Schließlich ist auch noch für alle neu vergebenen b_{Dyn}-Werte eine Smartlifting-Analyse durchzuführen.

6.4 Zusammenfassung

Der OTRun-Classloading-Prozess teilt sich auf in fünf Phasen. Er analysiert die Attribute, die der ot-Compiler in Team- und Rollenklassen anlegt und generiert daraus eine Reihe von Datenstrukturen, die in Class- und Methodenblöcken von Team-, Rollen- und Basisklassen gespeichert werden. Die Lifting-, Teamaktivierungs- und Callin-Ausführungsalgorithmen verwenden diese Strukturen zur Laufzeit um ihre jeweilige Aufgabe erfüllen zu können.

Die Verwendung von Shadowproxies erlaubt es, dass Basisklassen vor- oder nach den Teamklassen geladen werden können, die an sie binden. Werden Basisklassen vor ihren Teamklassen geladen, sind allerdings zusätzliche Besonderheiten zu beachten, die in Absatz 6.2.2.4 und Absatz 6.3.5 beschrieben wurden. Im Gegensatz zum OTRE kann durch diese Maßnahmen eine Unabhängigkeit von der Ladereihenfolge gewonnen werden, so dass JamVM/OTRun lazy loading für Team- und Basisklassen unterstützen kann. Nur Rollenklassen müssen zusammen mit ihren Teamklassen geladen werden, da nur so ein Zugriff auf die Callin-Attribute und die Hierarchie der Rollenklassen möglich ist.

In Kapitel 6.2.2 wurde der Prozess des Callin Deployments beschrieben. Bei diesem handelt es sich um das delegationsbasierte Äquivalent zum Webevorgang des *ChainingWrappers*[18] im OTRE. In Abschnitt 6.2.2.3 wurden zusätzlich einige Sonderfälle erwähnt, die beim Callin Deployment zu berücksichtigen sind.

In den folgenden drei Kapiteln wird erläutert werden, wie die beschriebenen Datenstrukturen eingesetzt werden, um Lifting, Teamaktivierung und Callin-Ausführung delegationsbasiert umsetzen zu können.

[18] siehe dazu auch Kapitel 9.

Kapitel 7

Role Instance Tables

- Lifting mit OTRun

7.1 Translation Polymorphism

Die Verwaltung von Rolleninstanzen gehört zu den wichtigsten Konzepten von ObjectTeams/Java. In ObjectTeams/Java vollzieht sie sich weitestgehend hinter den Kulissen. Ein Programmierer bemerkt mitunter nur, wie Basisobjekte auf wundersame Weise neue Rollenattribute dazu gewinnen, wenn diese über eine Methode einer Team- oder Rollenklasse übergeben werden. Und er staunt dann womöglich noch viel mehr darüber, dass sie zwischen zwei Besuchen in einer Team- oder Rollenklasse auch ihren Zustand bewahren konnte. Dieses kleine Wunder nennt sich *Translation Polymorphism* und ist ein Konzept, das es erlaubt, Rolleninstanzen mit einzelnen Basisinstanzen typkonform zu assoziieren, so dass sie eine unsichtbare Verbindung eingehen, die immer dann wirksam wird, wenn eine Basisinstanz einen Kontextwechsel zu einem anderen Team vollzieht. Durch dieses Konzept wird es möglich eine einzelne Klasse schichtweise zu modellieren, wobei die Attribute und Methoden der einen Schicht gänzlich für eine andere Schicht unzugänglich bleiben, wenn der Programmierer dies wünscht. In ObjectTeams/Java werden solche Schichten durch Rollenklassen modelliert. Jede Rollenklasse, die an eine Basisklasse bindet, fügt den Objekten dieser Basisklasse eine weitere Schicht an Methoden und Attributen hinzu. Innerhalb einer Rollenklasse können diese dann verwendet werden. Aber es können auch Schnittstellen zu anderen Methoden und Attributen der Basisklasse definiert werden – so dass sowohl *encapsulation* als auch *decapsulation* möglich ist.

7.2 Realisierung des Liftings durch den ot-Compiler

Rolleninstanzen werden zur Laufzeit eines ObjectTeams/Java-Programmes bisher in Hashtabellen gespeichert. Diese werden in jeder Teaminstanz angelegt und nehmen eine Menge von Rollen-Basis-Paaren auf. Die Anzahl der Hashtabellen, die in einer Teaminstanz angelegt werden, ist identisch mit der Anzahl der *Rollenhierarchien*, die in einer Teamklasse existieren.

Für jede Basisinstanz, die den Kontext dieser Teaminstanz betritt und zu einer Rolleninstanz *geliftet* wird, wird geprüft, ob sich bereits eine Rolleninstanz in der entsprechenden Hashtabelle befindet. Ist dies der Fall, so wird die Rolleninstanz zurückgegeben, ist dies aber nicht der Fall, so wird eine neue Rolleninstanz erzeugt. Dabei sorgt das *Smartlifting* dafür, dass eine zur Basisinstanz möglichst kompatible Rolleninstanz erzeugt wird.

7.2.1 Rollenhierarchien

Der ot-Compiler analysiert die Vererbungsbeziehungen aller an Basisklassen gebundenen[19] Rollenklassen. Ein wichtiges Ergebnis dieser Analyse ist eine Menge von *RootClasses*. Jede RootClass spannt einen Vererbungsbaum von gebundenen Rollenklassen auf, eine *RootHierarchy*, die zu den Vererbungsbäumen der anderen Rootklassen disjunkt ist. Eine RootClass ist damit zugleich die oberste gebundene Rollensuperklasse einer Menge von Rollenklassen in der Teamklasse. Sie ist die Superklasse einer einzelnen RootHierarchy. Die Sprachdefinition von ObjectTeams/Java verbietet es, dass die Subrollenklassen einer Rollenklasse an eine Basisklasse binden, die weder identisch mit der von der Rollenklasse gebundenen Basisklasse ist noch eine Subklasse dieser Basisklasse ist. Alle ungebundenen Subrollenklassen erben zudem die Bindung ihrer nächsthöheren gebundenen Superrollenklasse.

Für jede RootClass enthält eine Teaminstanz eine eigene Hashtabelle. Der Zugriff auf eine Hashtabelle erfolgt über einen Schlüssel. Das Ergebnis eines solchen Zugriffs ist ein Wert, der mit dem Schlüssel assoziiert ist und vorher in der Hashtabelle gespeichert wurde. Wenn für einen Schlüssel noch kein Wert gespeichert wurde, so würde ein Zugriff auf die Hashtabelle den Wert NULL zurückliefern. Der Lifting-Algorithmus des ot-Compilers verwendet Basisinstanzen als Schlüssel und speichert Rolleninstanzen als Wert. Daraus folgt, dass eine Basisinstanz mit genau einer Rolleninstanz für jede *RootHierarchy* assoziiert werden kann. Bei der Erzeugung einer Rolleninstanz wird mit Hilfe des *Smartliftings* entschieden, welche Rollenklasse einer RootHierarchy instanziiert wird. Diese Instanz wird anschließend in der Hashtabelle, die zu der RootHierarchy gehört, gespeichert.

[19] In einem ObjectTeams/Java-Programm wird eine solche Bindung mit dem Schlüsselwort **playedBy** erzeugt. Siehe dazu auch [OTJLD § 2.1].

7.2.2 Smartlifting

Smartlifting versucht eine bestmöglich typkonforme Abbildung zwischen einer Basisklassenhierarchie und einer Rollenklassenhierarchie zu realisieren. Der Smartlifting-Analyse-Algorithmus ist detailliert in [Herr04] beschrieben, wir werden ihn daher hier nicht exakt definieren, sondern seine Funktion in Abschnitt 7.3.3 lediglich anhand einiger Beispiele darstellen und dabei beschreiben, wie der in OTRun implementierte Smartlifting-Algorithmus arbeitet.

Auf ein Detail sei jedoch hingewiesen. Jeder gebundenen Basisklasse wird beim Ladezeitweben vom OTRE ein Integerwert b_{Dyn} zugewiesen, durch den die Basisklasse beim Smartlifting identifiziert werden kann. Der Wert b_{Dyn} wird von einer Klasse geerbt. Er verändert sich nur, wenn diese Klasse selbst eine Basisklasse ist, an die eine Rollenklasse direkt bindet. Auch OTRun verwendet die Zuweisung eines b_{Dyn}-Wertes an Basisklassen, verfährt jedoch bei der Vergabe eines bDyn-Wertes strikter und so, dass eine spezielle Invariante eingehalten wird. Auch dies wird ausführlich in Abschnitt 7.3.3 behandelt werden. OTRun vergibt b_{Dyn}-Werte separat für jede Teamklasse.

7.2.3 Lifting zur Laufzeit

Lifting erfolgt zur Laufzeit über einen vom ot-Compiler eingefügten Aufruf einer LiftTo-Methode mit dem Namen _OT$LiftTo$<Rollenklassenname>. Diese werden vom ot-Compiler in jeder Teamklasse generiert, die gebundene Rollenklassen enthält. <Rollenklassenname> entspricht dabei dem Namen einer gebundenen Rollenklasse der Teamklasse. Sie werden vor allem in den *Callin Wrapper*-Methoden dazu verwendet, eine mit einer Basisinstanz assoziierte Rolleninstanz zu ermitteln, auf der dann eine Callin-Methode aufgerufen werden kann. In Kapitel 9 wird auf den Aufruf von Callins detailliert eingegangen werden.

Der Lifting-Algorithmus des ot-Compilers, der sich in jeder LiftTo-Methode befindet, ist sehr übersichtlich aufgebaut. Er gliedert sich im Wesentlichen in drei Schritte:

1) Den Versuch, eine passende Rolleninstanz aus einer Hashtabelle in der Teaminstanz zu ermitteln.

2) Die Erzeugung einer neuen Rolleninstanz, falls 1) fehlschlägt, weil eine solche Rolleninstanz noch nicht existiert. An dieser Stelle findet das *Smartlifting* statt, dessen Umsetzung wir im Anschluss betrachten werden.

3) Einen Test, ob die Rolleninstanz, die wir gefunden haben, auch zu dem Typ passt, den wir in der jeweiligen LiftTo-Methode erwarten.

Der erste Schritt ist trivial. Er besteht aus dem Aufruf einer entsprechenden Zugriffsmethode für Hashtabellen.

Der zweite Schritt ist trickreich. Er besteht aus einer switch-Anweisung, die für einige *bDyn*-Werte von Basisklassen einen Fall enthält, der den Konstruktoraufruf für eine bestimmte Rollenklasse enthält. Der Rollenkonstruktor, der einem bestimmten b_{Dyn}-Wert zugeordnet ist, wurde dabei vom Smartlifting-Algorithmus festgelegt. Die Liftingmethode wurde dann vom ot-Compiler entsprechend generiert. Alle b_{Dyn}-Werte, die in der switch-Anweisung nicht explizit behandelt werden, lösen den default-Fall aus, der eine LiftingFailedException wirft.

Der dritte Schritt wird durch den Versuch implementiert, eine Typumwandlung der in Schritt 1 ermittelten Rolleninstanz durchzuführen. Der Zieltyp ist identisch mit dem Rollenklassentyp, der sich auch in dem Namen der jeweiligen LiftTo-Methode finden lässt. Auf Bytecodeebene wird diese Überprüfung durch einen `checkcast` Bytecode realisiert, der von der ihm übergebenen Instanz aus alle Superklassen dieser Instanz aufsucht, um sie daraufhin zu überprüfen, ob sie mit der Zielklasse übereinstimmen. Erreicht er dabei das obere Ende der Vererbungshierarchie ohne die Zielklasse zu finden, so schlägt der Test fehl, ansonsten ist er erfolgreich. Bei einem Erfolg dieses Tests kann die Lifting-Methode mit der ermittelten Rolleninstanz als Rückgabewert verlassen werden. Schlägt der Test jedoch fehl, so wird eine *WrongRoleException* ausgelöst.

7.2.4 WrongRoleException

Eine WrongRoleException kann auftreten, wenn sich der Vererbungsbaum in einer Rollenklassenhierarchie verzweigt, jeder Zweig aber nach wie vor an die gleiche Basisklasse bindet. Dann entscheidet der erste Aufruf einer Lifting-Methode darüber, ob eine Instanz von einer Rollenklasse des einen Zweiges oder des anderen Zweiges erzeugt wird. Erfolgt nun ein weiterer Aufruf einer Lifting-Methode, der diesmal jedoch zu einer Klasse des anderen Zweiges gehört, so kommt es zu der Situation, dass die in der Hashtabelle gefundene Rolleninstanz nicht mehr typkompatibel zu der Klasse ist, zu der geliftet werden soll, da verschiedene Zweige in einer Vererbungshierarchie niemals typkompatibel zueinander sind. Damit das Programm nun nicht mit einer Instanz weiterarbeitet, die nicht typkonform ist, wird das Lifting mit der WrongRoleException in diesem Fall abgebrochen. Kommt es dagegen in dem Programm nie dazu, dass auf den Aufruf einer Lifting-Methode für den einen Zweig eine Lifting-Methode des anderen Zweigs erfolgt, kann das Programm fehlerfrei ausgeführt werden, obwohl die Vererbungshierarchie in diesem Fall das Risiko für eine WrongRoleException enthält.

7.3 Lifting in OTRun

Das Lifting in OTRun muss die gleichen drei Schritte realisieren können, die schon in Abschnitt 7.2.3 für den ot-Compiler beschrieben wurden. Für den ersten Schritt ist zunächst eine Datenstruktur notwendig, mit der Rolleninstanzen, die zu einer spezifischen Rollenklassenhierarchie und einer individuellen Basisinstanz gehören, gespeichert, verwaltet und wieder gefunden werden können. In OTRun wird diese

Aufgabe von der Role Instance Table (RIT) übernommen. Sie wird in Abschnitt 7.3.1 beschrieben.

Im Anschluss daran werden wir einen Blick auf die Beziehung zwischen dem Java-Garbage Collector und Rolleninstanzen werfen und beschreiben, welche Änderungen am Garbage Collector diese Beziehung in der Java Virtual Machine notwendig werden lässt.

Der zweite Schritt bedarf zunächst einer Smartlifting-Analyse. Ihre Umsetzung in OTRun wird in Abschnitt 7.3.3 erläutert. Im Anschluss daran wird erklärt, wie OTRun das Smartlifting mit Hilfe von smartTables zur Laufzeit realisiert und damit ohne ein statisch kompiliertes switch-case-Konstrukt für jede einzelne LiftTo-Methode auskommen kann.

Für den dritten Schritt wird in Abschnitt 7.3.4 zunächst eine statische Rollenklassenhierarchie-Analyse vorgestellt, die einmalig beim Laden einer Teamklasse erfolgt. Die dabei generierten Informationen können dann zur Laufzeit vom liftTo Opcode genutzt werden, um auf die iterative Superklassen-Analyse verzichten zu können, die der ot-Compiler in seiner Implementierung durch den `checkcast` Bytecode zur Laufzeit erzwingt.

7.3.1 Role Instance Tables

OTRun verwendet eine dreistufige Tabelle, um Rolleninstanzen über ihre Basisinstanz direkt mit Hilfe eines Lookup-Mechanismus erreichen zu können. Als Eingabe in den Algorithmus dient eine Basisinstanz bi, eine Teaminstanz ti und eine Rollenklasse rc, zu der geliftet werden soll. Die Rollenklasse rc ist dabei als Integer gegeben, der zugleich den Index in die `rcTable` einer Teamklasse darstellt. An dieser Position lassen sich alle für das Lifting relevanten Informationen finden, die dieser Rollenklasse zugeordnet sind. Mit der rcTable beginnt auch der Lifting-*Tablewalk* der in Abbildung 6 dargestellt ist. Zunächst werden aus den beiden Instanzen die beteiligten Klassen bc (BaseClass) und tc (TeamClass) bestimmt. Nun wird der durch den Lifting-Opcode gegebene Index `rc` dazu verwendet, in der `rcTable` von tc den Index bcidx zu bestimmen. Mit diesem kann nun wiederum in der `bcTable` von tc der Index tcidx bestimmt werden, unter dem schließlich der Offset zu finden ist, an dem im *tcFrame* der RoleInstanceTable der Basisinstanz bi ein Verweis auf einen *tiFrame* zu finden ist. Dieser enthält Einträge für jede Teaminstanz der Teamklasse tc, wobei diese Einträge wiederum jeweils auf ein riFrame verweisen, dass dann schließlich für jede RoleHierarchy der Teamklasse tc genau eine Rolleninstanz speichern kann. Der riFrame, welcher der aktuellen Teaminstanz ti zugeordnet ist, lässt sich durch den *`tic`* bestimmen, einem Indexwert für tiFrames, der im *TeamInstanceManager* von jeder Teaminstanz gespeichert wird. Mit Hilfe des RoleHierarchySlot-Index RHs aus der `rcTable` kann nun schließlich die noch fehlende Position im riFrame bestimmt werden, unter der die Rolleninstanz zu finden ist, zu der geliftet werden soll. Ist dieser Eintrag leer, so muss nun ein Smartlifting folgen, um eine neue Rolleninstanz mit einem zur Basisinstanz passenden Typ zu erzeugen. Wenn dieses Feld aber bereits mit einer Rolleninstanz belegt ist, dann wird diese – sollte sie den

WrongRoleException-Test bestehen – schließlich als die geliftete Rolleninstanz zurückgegeben. Der gleiche RHs-Wert ist allen Rollenklassen einer RoleHierarchy zugeordnet. Daher teilen sich die Rollenklassen einer RoleHierarchy jeweils einen Slot im riFrame. Dies hat den gleichen Effekt wie die Allokation einer eigenen Hashtable für jede RootRole (was der Vorgehensweise des ot-Compilers entspricht).

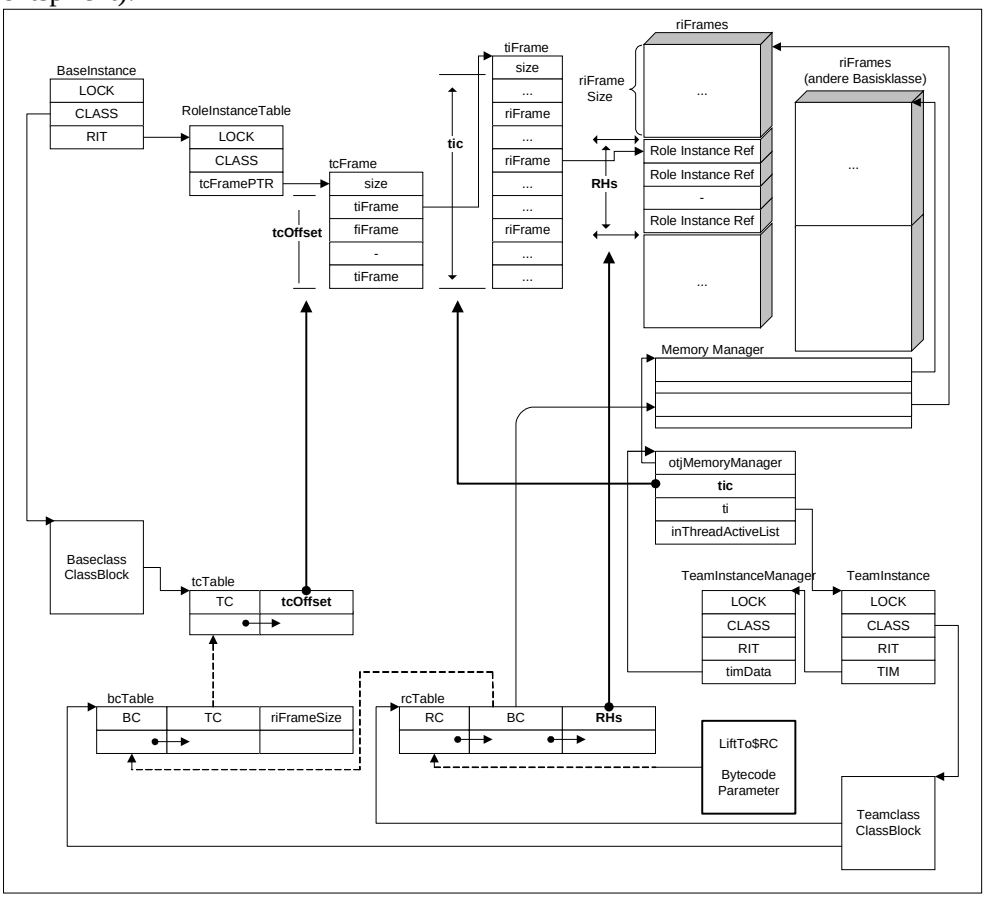


Abbildung 6: Lifting-Vorgang unter Verwendung einer RoleInstanceTable

Zusammenfassend lässt sich festhalten, dass mit Hilfe der Adresse **tcOffset.tic.RHs** eine eindeutig bestimmte Rolleninstanz bezüglich einer Basisinstanz für jede an sie gebundene Rollen– und Teamklasse und eine als Parameter vorgegebene Teaminstanz und Rollenklasse gefunden werden kann.

Die Arrays `tcFrame` und `tiFrame` der Basisinstanz bi wachsen mit jeder Teaminstanz, in deren Kontext ein Lifting von bi stattfindet. Die `riFrames` werden aus Arrays allokiert, die vom TeamInstanceManager der jeweiligen Teaminstanzt ti mit Hilfe von MemoryManagern verwaltet werden. So ist es möglich, dass die Teaminstanz zu einer gegebenen Basisklasse unmittelbar alle Rolleninstanzen ermitteln kann, die für die jeweilige Basisklasse im Kontext von ti erzeugt wurden. Mit Hilfe der `rcTable` kann auch für eine vorgegebene Rollenklasse ermittelt werden, welche und wie viele Rolleninstanzen für sie im aktuellen Kontext von ti

existieren. Damit ist es auch in OTRun prinzipiell möglich, die Funktionalität der Reflection-API-Methode getAllRoles() zu implementieren[20].

Der `tic`-Indexwert wird jedem TeamInstanceManager von seinem Konstruktor zugeordnet. Jede Teaminstanz einer Teamklasse besitzt innerhalb der Teaminstanzen dieser Teamklasse einen jeweils einzigartigen `tic`-Wert. Die Vergabe wird mit Hilfe eines in der Teamklasse gespeicherten Counters und einer Liste mit freigewordenen `tic`-Werten organisiert. Enthält die Liste einen ungenutzten Wert so wird dieser an eine neue Teaminstanz vergeben, wenn nicht, so wird der Counter erhöht und der Wert dieses Counters als neuer `tic` für die Teaminstanz zurückgegeben.

7.3.2 Role Instance Tables und Garbage Collection

Die Struktur der RoleInstanceTable soll eine effiziente Speicherverwaltung erlauben. Der Garbage Collector der JamVM kann dazu so angepasst werden, dass er Rolleninstanzen nur über das Traversieren der RoleInstanceTables von Basisinstanzen in der Mark-Phase erreicht, aber nicht über Teaminstanzen. Dadurch können Rolleninstanzen automatisch vom Garbage Collector freigegeben werden, sobald die Basisinstanz nicht mehr im *Scope* des ausführenden Programms erreichbar ist. Der Finalizer der RoleInstanceTable-Instanz sorgt dann dafür, dass auch alle riFrames, tiFrames und tcFrames freigegeben werden. Die riFrames werden von den MemoryManagern einer Teaminstanz mit Hilfe von Freispeicherlisten verwaltet. Die Zeiger dieser Freispeicherlisten sind jeweils im ersten Eintrag eines riFrames gespeichert. Da in den Zeigern der Freispeicherliste zur Erkennung das niedrigste Bit gesetzt wird, kann der RIT-Finalizer beim Durchsuchen der RIT-Tabellenstruktur feststellen, wann ein Eintrag im tiFrame auf einen ungenutzten riFrame zeigt und wann auf einen riFrame, der für die Basisinstanz der RoleInstanceTable allokiert wurde.
Auch die Compaction-Phase des Garbage Collectors kann so erweitert werden, dass sie die Referenzen auf Rolleninstanzen in den riFrames findet[21].

Mit diesem Modell sind jedoch Teaminstanzen nicht mehr löschbar, da sie über das `$this0`-Feld in Rolleninstanzen immer erreichbar bleiben. Daher soll dieses Feld zukünftig von der mark-Phase „übersehen“ werden. Dann entscheidet allein die Anwesenheit im Scope des aktuellen Programms darüber, ob eine Teaminstanz gelöscht werden kann oder nicht. Tritt eine Löschung ein, kann der Finalizer des TeamInstanceManagers in den Arrays seiner MemoryManager alle Rolleninstanzen finden, die für den Kontext der Teaminstanz erzeugt wurden. Diese enthalten im Feld `$base` einen Verweis auf die ihnen zugeordnete Basisinstanz. Damit kann der Finalizer nun die RoleInstanceTables aller Basisinstanzen aktualisieren, die eine Rolleninstanz im Kontext dieser Teaminstanz besitzen.

[20] Der OTRun-Prototyp unterstützt dies jedoch noch nicht.
[21] Für den OTRun-Prototyp wurde Compaction abgeschaltet. Die Unterstützung der Compaction-Phase ist eine Option für zukünftige Versionen.

Ein Problemfall ist in diesem Modell jedoch für externalisierte Rollen gegeben, da es nun möglich ist, dass diese ihre Teaminstanz verlieren, wenn diese nicht mehr direkt im Scope erreichbar ist, sondern nur noch über die externalisierte Rolleninstanz erreichbar wäre. Ein separater Konstruktor für externalisierte Rolleninstanzen könnte dieses Problem jedoch lösen, indem er die Teaminstanz in einem Feld speichert, das wieder von der Mark-Phase des Garbage Collectors gefunden wird.

Mit der beschriebenen Technik ist es möglich, auf die Verwendung von Weak References verzichten zu können. Dies ist von Vorteil, da Weak References relativ viel Speicherplatz verbrauchen, der so eingespart werden kann.

7.3.3 Smartlifting in OTRun

Smartlifting wird in OTRun mit Hilfe einer zweidimensionalen `SmartTable` realisiert. Diese befindet sich in jeder Teamklasse und wird jedes Mal aktualisiert, wenn eine gebundene Basisklasse der Teamklasse geladen wird. Beim Laden erhält die Basisklasse einen b_{Dyn}-Wert, der ihr eine eigene Spalte in der `SmartTable` zuweist. Die Zeilen in der SmartTable sind jeweils einem rc-Indexwert der `rcTable` in der Teamklasse zugeordnet. Dies ist auch zugleich der rc-Wert, mit dem der LiftTo-Opcode die Zielrollenklasse beschreibt, zu der er liften will. Soll also eine neue Rolleninstanz erzeugt werden, so genügt es, in der `SmartTable` an der Position (rc, b_{Dyn}) nachzuschlagen, welcher rc_{smart}-Wert dort gespeichert ist. Dieser steht dann für die Rollenklasse, zu der eine neue Rolleninstanz erzeugt werden soll. In der `rcTable` kann an der Position rc_{smart} dann ein Zeiger auf den `MethodBlock` der `<init>` Methode für die neu zu allokierende Rolleninstanz gefunden werden. Außerdem ist dort eine Referenz auf die Rollenklasse selbst gespeichert, die zur Allokation einer neuen Rolleninstanz auf dem Heap von der JamVM-Funktion `Object* allocObject(Class* c)` benötigt wird.

Nun wäre noch zu klären, wie die `SmartTable` initialisiert wird. Dies geschieht mit Hilfe eines Smartlifting-Analyse-Algorithmus. Dieser nutzt eine besondere Invariante aus, die bei der Vergabe von `bDyn`-Werten aufrechterhalten wird.

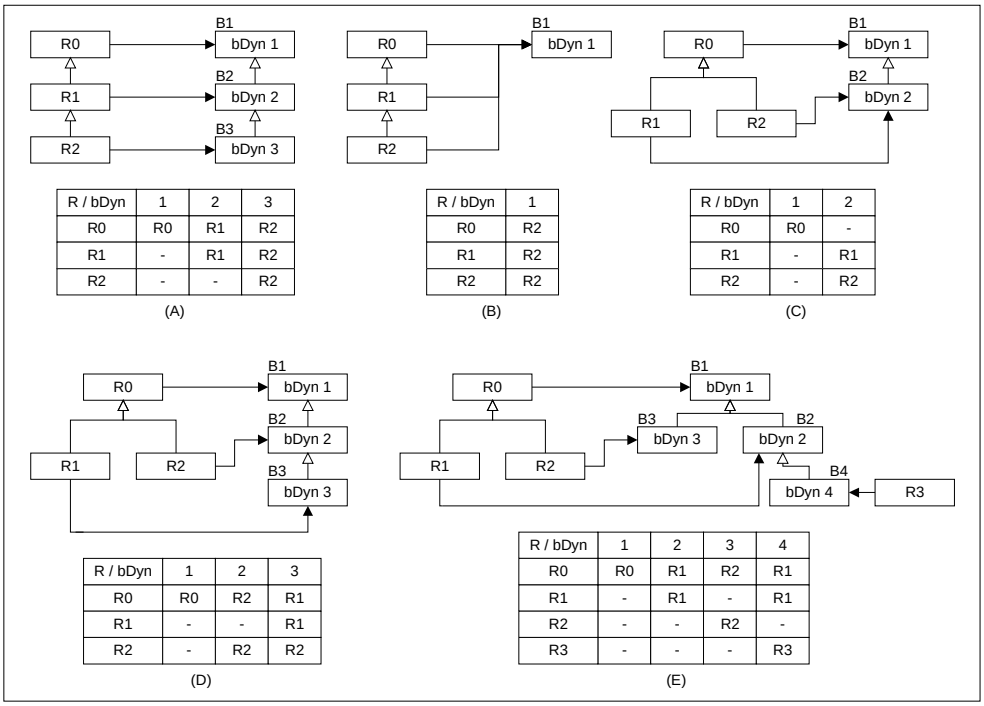
Alle `bDyn`-Werte werden in OTRun für jede Teamklasse separat vergeben. Jede Basisklasse besitzt eine `tcTable` und in dieser Tabelle eine Zeile für jede Teamklasse, die an diese Basisklasse bindet. In dieser Zeile wird auch ihr b_{Dyn}-Wert gespeichert. Die `bDyn`-Werte werden in aufsteigender Reihenfolge bezüglich einer absteigenden Vererbungshierarchie vergeben. Daraus folgt für den Smartlifting-Analyse-Algorithmus folgende einfache Invariante:

$$\texttt{bDyn}(\text{Superbaseclass}) < \texttt{bDyn}(\text{Subbaseclass})$$

Diese Invariante verwendet der Smartlifting-Analyse-Algorithmus, um den Punkt zu bestimmen, an dem er die Rollenklasse gefunden hat, die in einem bestimmten Zweig der Rollenvererbungshierachie am nächsten an die Zielbasisklasse bindet. Erreicht der Algorithmus beim Traversieren der Subrollenklassen eine Rollenklasse, die an eine Basisklasse mit einem höheren `bDyn`-Wert bindet, als der

bDyn-Wert der Zielbasisklasse, so kann die Traversierung abgebrochen werden, da ab diesem Zeitpunkt nur noch Rollenklassen gefunden werden, die eine Basisklasse mit mindestens diesem höheren bDyn-Wert voraussetzen (d.h. mindestens den zugehörigen Basisklassensubtyp benötigen, da sie z.B. möglicherweise an eine Methode binden, die erst von diesem Subklassenbasistyp eingeführt wird).

Diese Invariante allein reicht jedoch nicht in jedem Fall aus. In Abbildung 7 wäre der Fall (E) ein solcher Sonderfall, in dem eine zusätzliche Basisklassenhierachie-Analyse notwendig wird, um die Smarttable korrekt aufzubauen.



R / bDyn	1	2	3
R0	R0	R1	R2
R1	-	R1	R2
R2	-	-	R2

R / bDyn	1
R0	R2
R1	R2
R2	R2

R / bDyn	1	2
R0	R0	-
R1	-	R1
R2	-	R2

R / bDyn	1	2	3
R0	R0	R2	R1
R1	-	-	R1
R2	-	R2	R2

R / bDyn	1	2	3	4
R0	R0	R1	R2	R1
R1	-	R1	-	R1
R2	-	-	R2	-
R3	-	-	-	R3

Abbildung 7 : Smartlifting Beispiele

Betrachten wir aber zunächst Beispiel (A) und nehmen wir an, dass die Teamklasse zu der R0, R1 und R2 gehören, bereits geladen wurde. Wird nun die Basisklasse geladen, die von R0 mit playedBy gebunden ist, so hat sie in diesem Moment und in diesem Beispiel den b_{Dyn}-Wert 1 erhalten, weil eine direkte Bindung zu ihr existiert. Wäre dies nicht der Fall, weil R0 nicht an diese Klasse, sondern ihre Superklasse binden würde, so würde die gerade geladene Klasse nur den b_{Dyn}-Wert aus ihrer Superklasse kopieren. Die Vergabe eines neuen b_{Dyn}-Wertes löst immer auch eine Smartlifting-Analyse aus, um die Spalte zu füllen, die mit dem neu vergebenen b_{Dyn}-Wert in der `SmartTable` adressiert wird. Der Analyse-Algorithmus versucht nun, die Rollenklasse zu finden, die bestmöglich zu dieser Basisklasse passt. Dazu muss er die Rollenklassenhierarchie untersuchen. Als Startpunkt wählt er dazu die RootRoles von jeder RoleHierarchy der Teamklasse. In unseren Beispielen existiert bis auf Fall (E) nur eine Rollenhierarchie. Die Wurzel ist jeweils die Klasse R0.

Im Beispiel (A) sucht der Smartlifting-Algorithmus nun nach einer Subklasse R0, die möglichst direkt an die Basisklasse bindet, aber dabei nicht „über das Ziel hinausschießt“. Nähe wird grob durch den b_{Dyn}-Wert gemessen. Je numerisch dichter der b_{Dyn}-Wert der gebundenen Basisklasse einer Rollenklasse an dem b_{Dyn}-Wert liegt, für den momentan die `SmartTable`-Spalte ausgefüllt werden soll, um so näher, d.h. um so passender ist die Rollenklasse. Ist der bDyn-Wert der Basisklasse, an den die betrachtete Rollenklasse bindet, sogar identisch mit dem bDyn-Wert für den die SmartTable gerade aufgebaut wird, dann ist eine ideal passende Rollenklasse gefunden worden. Möglicherweise gibt es aber noch eine andere Subrollenklasse, die ebenfalls an die gleiche Basisklasse bindet. Dann wäre diese Rollenklasse zu bevorzugen, da sie die höher spezialisierte passende Variante ist. Dies ist in Beispiel (B) zu beobachten. Dort binden R0, R1 und R2 an B1. Der Algorithmus liefert entsprechend für ein LiftTo$R0, LiftTo$R1 und LiftTo$R2 jeweils R2 zurück, die am höchsten spezialisierte Variante, die er finden konnte. In Beispiel (A) findet der SmartLifting-Algorithmus dagegen heraus, dass R0 zwar an B1 bindet, R1 aber bereits eine Basisklasse bindet, die einen höheren b_{Dyn}-Wert besitzt, als der Wert $b_{Dyn} = 1$, für den wir gerade die Spalte in der `SmartTable` füllen. Er liefert für LiftTo$R0 also R0 zurück. Ein LiftTo$R1 oder LiftTo$R2 einer Basisinstanz von B1 würde allerdings scheitern und NULL[22] zurückliefern. Ein Ereignis, auf das ein Smartlifting-Algorithmus zur Laufzeit eine LiftingFailed-Exception folgen lässt.

Beispiel (C) ist bereits etwas ausgefallener. Ein LiftTo$R0 für eine Instanz von B2 könnte nach den bereits intuitiv vorgestellten Regeln entweder R1 oder R2 für B2 zurückliefern. Da der Algorithmus jedoch keine Münze wirft, bleibt dieser Fall undefiniert und das Feld in der SmartTable entsprechend leer.

Beispiel (D) besitzt für den Fall LiftTo$R2 für eine Instanz von B3 ein interessantes Verhalten. Zwar existiert über R1 eine Bindung an B3, deshalb wäre der ideale `bDyn`-Wert = 3. R2 bindet jedoch nur an B2 und da B3 eine Subklasse von B2 ist, erbt B3 diese Bindung. Daher wird in diesem Fall die Rollenklasse R2 zurückgegeben.

Im letzten Beispiel (E) verdient der Fall besondere Beachtung. Ein LiftTo$R0 würde für eine Instanz von B4 zunächst die Rollenklasse R2 ermitteln, weil sie mit $b_{Dyn}(B3) = 3$ näher an B4 mit $b_{Dyn}(B4) = 4$ bindet als R1 mit $b_{Dyn}(B2) = 2$. Würde der Algorithmus nun mit R2 zurückkehren, so wäre ihm ein Fehler unterlaufen, da R2 an eine Basisklasse bindet, die gar nicht im Vererbungspfad von B4 liegt. Der Smartlifting-Algorithmus wurde deshalb um eine zusätzliche Hierarchieanalyse erweitert, die immer dann verwendet wird, wenn der ermittelte nächstpassende b_{Dyn}-Wert nicht identisch ist mit dem gesuchten b_{Dyn}-Wert, für den gerade die SmartTable-Spalte aufgebaut wird. Sie untersucht dann, ob die Basisklasse die mit der gefundenen Rollenklasse assoziiert ist, auch tatsächlich eine Superklasse dieser Basisklasse ist. Wenn nicht, sucht die Smartlifting-Analyse nach dem nächstbesten Treffer. Damit soll die Betrachtung des Smartliftings abgeschlossen sein. Der

[22] Tatsächlich wird in OTRun die Konstante OTJ_SMARTMAX verwendet, die den Wert 2^{16} besitzt.

Algorithmus, der in OTRun implementiert wurde, liefert für diese 5 Beispiele das gleiche Ergebnis, das auch der ot-Compiler liefern würde.

7.3.4 Optimierung des WrongRoleException-Check

Für die Beschleunigung des WrongRole-Exception-Test wird jeder Rollenklasse ein R_{Dyn}-Wert und ein R_{Max}-Wert zugewiesen. Diese werden von einem rekursiven Algorithmus so vergeben, dass für alle Subklassen einer Rollenklasse R gilt, dass $R_{Dyn}(R_{sub}) <= R_{Max}(R)$ ist. Dies ist in Abbildung 8 dargestellt.

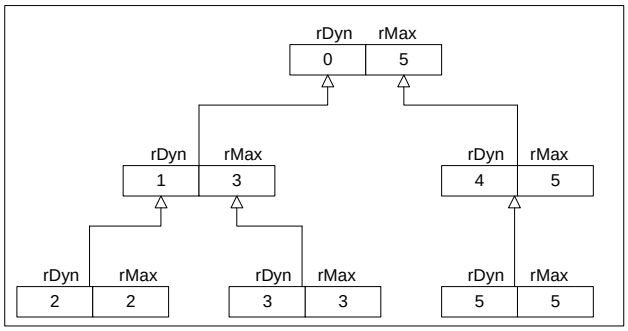


Abbildung 8 : R_{Dyn} und R_{Max}-Vergabe

Wenn nun mit R_{target} die Rollenklasse gegeben ist, zu der geliftet werden soll, und mit R_{lifted} das Ergebnis des Liftings, so kann mit der einfachen Überprüfung

$$R_{Dyn}(R_{lifted}) <= R_{Max}(R_{target}) \wedge R_{Dyn}(R_{lifted}) >= R_{Dyn}(R_{target})$$

ermittelt werden, ob eine WrongRole-Exception vorliegt. Evaluiert der Ausdruck zu TRUE, so liegt keine WrongRole-Exception vor, evaluiert er zu FALSE, so muss eine Ausnahme ausgelöst werden. Die Laufzeit für diesen Test ist bedeutend niedriger als die Laufzeit für einen `checkcast`-Bytecode.

7.3.5 Zusammenfassung

In diesem Kapitel wurde der Lifting-Algorithmus vorgestellt, der aus drei Teilen besteht. OTRun optimiert jeden dieser drei Teile auf unterschiedliche Art und Weise. Die Speicherverwaltung für Rolleninstanzen basiert auf RoleInstanceTables. Dadurch kann der Overhead von Hashtabellen eingespart werden, aber dennoch eine konstante Zugriffszeit auf Rolleninstanzen gewährleistet werden. Die Organisation der RoleInstanceTable erlaubt die Option, durch eine enge Verzahnung mit dem Garbage Collector auf Weak References verzichten zu können. So kann auch auf eine Hashtabelle in Basisklassen verzichtet werden, die nur dazu dient, Rolleninstanzen über einen Verweis am Leben zu erhalten.

Smartlifting konnte durch SmartTables ersetzt werden, so dass die separate Erzeugung von liftTo$-Methoden nicht länger notwendig ist, weil ein generischer Algorithmus diese Aufgabe nun ebenfalls übernehmen kann.

Schließlich konnte auch der Zeitaufwand der WrongRole-Überprüfung mit Hilfe einer speziellen Datenstruktur optimiert werden.

Kapitel 8

Callin Signatures, Timestamps und Team Activation Caches

- Teamaktivierung in OTRun

8.1 Teamaktivierung in ObjectTeams/Java

Teaminstanzen können entweder für alle Threads, für einen bestimmten Thread oder für den lokalen, gerade aktiven Thread aktiviert und deaktiviert werden, der den Aktivierungsbefehl selbst ausführt. Die nichtlokale Aktivierung kann ausschließlich über den Aufruf der Methode activate(Thread t) erfolgen. Der Parameter t darf dabei auch den speziellen Wert ALL_THREADS besitzen, dessen Übergabe die Aktivierung der Teaminstanz für alle Threads veranlasst.

Eine lokale Aktivierung erfolgt entweder über die Methode activate() oder indirekt durch Methoden, die der ot-Compiler in bestimmten Situationen einfügt. Das Sprachkonstrukt `within(Teaminstance){...}` erlaubt z.B. die temporäre Aktivierung einer bestimmten Teaminstanz in einem Codeblock, sollte sie noch nicht aktiviert sein. Diese Blockanweisung stellt sicher, dass sich die Teaminstanz an ihrem Ende wieder in dem Aktivierungszustand befindet, in dem sie zu Beginn des Blocks war. Dabei gilt jedoch die Ausnahme, dass eine manuelle Aktivierung, die durch den Aufruf von activate() oder deactivate() innerhalb des Blocks erfolgt, dadurch nicht rückgängig gemacht wird. Umgesetzt wird dies durch die Methoden _OT$saveActivationState und _OT$restoreActivationState der Klasse Team und durch die Verwendung von activate() bzw. deactivate().

Außerdem kann der Compiler die Methoden `_OT$implicitlyActivate` und `_OT$implicitlyDeactivate` unter anderem in öffentlichen Teammethoden einfügen. Diese sorgen in diesem Fall dafür, dass der Aufruf einer öffentlichen Teammethode die Teaminstanz für die Dauer des Methodenaufrufs aktiviert und bei Verlassen der Methode diese entsprechend wieder deaktiviert, wenn sie nicht explizit aktiviert wurde. Dies wird als implizite Aktivierung bezeichnet.

8.2 Umsetzung der Teamaktivierung zur Laufzeit

Zur Teamaktivierung verwendet der ot-Compiler bzw. das OTRE verschiedene Datenstrukturen. Jede Teaminstanz enthält eine Hashtabelle, in der alle Threads gespeichert werden, für die diese Teaminstanz aktiviert wurde. Zusätzlich enthält jede Basisklasse zwei Arrays, in der alle aktiven Teaminstanzen gespeichert werden, die jeweils Rollenklassen besitzen, die an diese Basisklasse binden.

Erfolgt eine Teamaktivierung für eine Teaminstanz, so wird eine Referenz auf den Thread, für den sie aktiviert werden soll, in ihrer Hashtabelle gespeichert. Anschließend wird die Methode `_OT$registerAtBases` aufgerufen, die der Compiler für jede Teamklasse generiert. Diese ruft für jede der gebundenen Basisklassen die statische Methode `_OT$addTeam` auf, der eine vom Compiler für alle Teamklassen jeweils einzigartig vergebene ID übergeben wird. Zusätzlich erhält sie als Parameter die aktuelle Teaminstanz. Die Methode `_OT$addTeam` muss vom OTRE zur Ladezeit in jede Basisklasse gewebt werden. Dieser Webevorgang fügt der Basisklasse auch zwei statische Arrays hinzu. In einem wird die `_OT$addTeam(...)` übergebene Teaminstanz gespeichert, in dem anderen die TeamID. Die Indexposition, an der das Einfügen erfolgt, ist dabei für beide Arrays identisch. Die Verwendung dieser Arrays im Rahmen der Ausführung von Callins wird im nächsten Kapitel beschrieben.

Bei der impliziten Aktivierung wird noch ein zusätzlicher Schritt ausgeführt. Sie führt Buch darüber, wie oft die implizite Aktivierung für den jeweils aktuellen Thread erfolgte. Dazu benutzt sie ein threadlokales Feld, in dem sie einen Zähler speichert. Jede implizite Aktivierung für den passenden Thread und die jeweilige Teaminstanz erhöht diesen Zähler, jede implizite Deaktivierung reduziert ihn. Nur wenn dieser Zähler 0 erreicht, wird eine implizite Deaktivierung auch tatsächlich durchgeführt. Der Zweck dieses Zählers ist es, eine vorzeitige Deaktivierung zu verhindern, wenn ein Pfad der Ausführung mehrere implizite Aktivierungen in Folge auslöst. Würde die Rückkehr aus der letzten Methode mit impliziter Aktivierung in dieser Folge bereits zur Deaktivierung führen, so wäre die Teaminstanz für die Operationen, die auf die Rückkehr aus dieser Methode folgen, bereits deaktiviert. Die Deaktivierung darf jedoch erst erfolgen, wenn der Kontext der Teaminstanz endgültig verlassen wird. Dazu dient der beschriebene Zähler.

Da eine Teaminstanz nur aktiviert werden kann, wenn auf jeder ihrer gebundenen Basisklassen die Methode `_OT$addTeam(...)` aufgerufen wurde, ist das Laden und Weben vor der ersten Teamaktivierung für das OTRE obligatorisch.

8.3 Umsetzung der Teamaktivierung in OTRun

Für OTRun stellt sich zunächst die Aufgabe, einen Teamaktivierungsmechanismus zu finden, dessen Aktivierungsmethoden nicht davon abhängig sind, ob eine Basisklasse geladen wurde oder nicht. Da OTRun einer Basisklasse keine neuen Methoden hinzufügen kann, ist eine Implementierung mit Hilfe einer `_OT$addTeam(...)`-Methode außerdem ausgeschlossen. Da mit OTRun zusätzlich das Ziel verfolgt wird, die Spracheigenschaften von ObjectTeams/Java effizienter umzusetzen als dies bisher der Fall war, wird nun ein Ansatz vorgestellt, der sich grundlegend von dem bisherigen Verfahren zur Teamaktivierung unterscheidet.

In diesem neuen Ansatz lässt sich die Teamaktivierung zunächst in zwei Phasen unterteilen. In der ersten Phase wird der Aktivierungszustand einzelner Teaminstanzen verwaltet. Sie ähnelt dem bisherigen Verfahren zur Teamaktivierung, verwendet jedoch andere Datenstrukturen. So werden aktive Teaminstanzen nun nicht länger in den Basisklassen, sondern stattdessen in den einzelnen Teamklassen gespeichert.

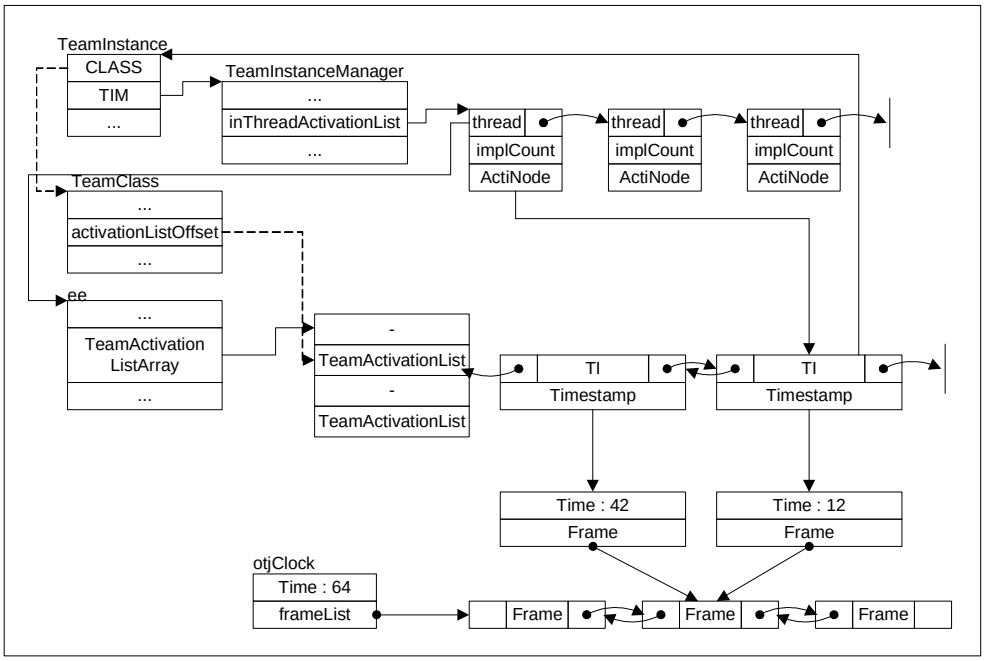


Abbildung 9 : Teamaktivierung in OTRun

In der Environmentvariable ee, die jeder Thread besitzt, existiert dazu das `TeamActivationListArray`, in dem jede Teamklasse einen eigenen Eintrag besitzt. Der Index dieses Eintrags ist dabei über alle Threads hinweg für die jeweilige Teamklasse gleich. Ist also der Thread gegeben, für den eine Aktivierung erfolgen soll, so kann auch auf dessen Environment ee zugegriffen werden. Kann auf das Environment zugegriffen werden, kann auch der Eintrag im `TeamActivationListArray` mit Hilfe des passenden Index erreicht werden. Im

Feld `activationListOffset` ist dieser im `ClassBlock` der Teamklasse gespeichert, zu der die Teaminstanz, die gerade aktiviert werden soll, gehört. Der Eintrag enthält nun die doppeltverkettete Liste `TeamActivationList` mit Knoten, die neben den Teaminstanzen auch eine *Timestamp* enthalten, die zum Zeitpunkt ihrer Aktivierung vergeben wurde. Bei einer Teamaktivierung wird ein neuer Knoten mit einer aktuellen Timestamp eingefügt. Bei einer Deaktivierung wird der Knoten aus der doppeltverketteten Liste entfernt. Der Aufwand für beide Operationen beträgt O(1).

Zwei Fragen sind nun bezüglich der ersten Phase noch zu beantworten. Woher weiß eine Teaminstanz für welche Threads sie aktiviert wurde? Und wozu dienen die Timestamps und wie funktionieren sie? Beantworten wir zunächst die erste Frage, bevor die zweite dann im nächsten Unterabschnitt dieses Kapitels ausführlich behandelt wird.

Um festzustellen, ob eine Teaminstanz bereits aktiviert ist, enthält jede Teaminstanz in ihrem TeamInstanceManager-Objekt die einfach verkettete Liste `inThreadActivationList`. Jeder Eintrag in dieser Liste enthält eine Referenz auf einen Thread, für den diese Teaminstanz aktiviert wurde. Außerdem enthält jeder Eintrag einen Verweis auf den Knoten, der für die aktuelle Teaminstanz in der zum Thread passenden `TeamActivationList` existiert. Und schließlich besitzt jeder Eintrag auch noch einen Aktivierungszähler, der für die implizite Aktivierung benötigt wird und analog zum OTRE verwendet wird.

Um herauszufinden, ob eine Teaminstanz für einen bestimmten Thread aktiviert ist, muss `inThreadActivationList` in der Teaminstanz nach dem passenden Eintrag zu diesem Thread durchsucht werden. Konnte kein Eintrag gefunden werden, so wurde die Teaminstanz für diesen Thread bisher noch nicht aktiviert. Daher wird nun ein neuer Eintrag erzeugt und am Beginn der Liste eingefügt. Ob eine Teaminstanz implizit oder explizit aktiviert wurde, wird im untersten Bit des Zeigers auf den `TeamActivationList`-Knoten gespeichert.

Um bei der Aktivierung einer bereits aktivierten Teaminstanz schneller feststellen zu können, dass diese Teaminstanz bereits aktiviert wurde, wird die `inThreadActivationList` *move-to-front-sortiert*. Das bedeutet, dass jeder Aktivierungszugriff, der für einen bestimmten Thread erfolgt, den Knoten in der Liste, der für diesen Thread steht, an den Anfang der Liste verschiebt. Folgen nun mehrere Aktivierungsoperationen für eine Teaminstanz und denselben Thread mehrmals aufeinander, z.B. in einer Programmschleife, so reduziert sich der Suchaufwand dadurch bereits nach der ersten Aktivierungsoperation auf O(1).

8.3.1 Callin Signatures, Timestamps und der Team Activation Cache

Werden aktivierte Teaminstanzen nur in einer Liste innerhalb einer Teamklasse verwaltet, so kann allein damit nur eine lokale Ordnung der Teamaktivierungsabfolge innerhalb dieser Teamklasse erfasst werden. Deshalb

existiert ein globaler Zeitgeber, der jeder Aktivierungsoperation eine einzigartige *Timestamp* zuweist. Mit Hilfe dieser Timestamp können die Teamaktivierungslisten mehrerer Teamklassen gemischt werden, um eine neue Liste zu erhalten, in der alle Aktivierungseinträge nach dem Zeitpunkt ihrer Aktivierung sortiert sind. Anstelle einer Liste wird für das Ergebnis jedoch ein Array verwendet, der *TeamActivationCache*. Jeder *CallinSignature* ist dabei in jedem Environment ee eines Threads genau ein TeamActivationCache zugeordnet. Diese befinden sich analog zu den Teamaktivierungslisten ebenfalls in einem Array und werden über einen Index, der in der CallinSignature gespeichert ist, adressiert. Jeder Basismethode für die Callins existieren, ist eine CallinSignature zugeordnet, die Verweise auf alle Teamklassen enthält, deren Rollenklassen ein Callin für diese Basismethode deklarieren. Wird der Delegationproxy – den wir im nächsten Kapitel kennen lernen werden – aufgerufen, um diese Callins auszuführen, so schaut dieser zuerst nach, ob bereits ein TeamActivationCache für den aktuellen Thread und die jeweilige CallinSignature existiert. Ist das der Fall, so prüft er, ob der Cache *gültig* ist. Gültigkeit wird im nächsten Abschnitt erläutert werden. Trifft beides nicht zu, so beginnt der Delegationproxy damit, einen neuen Cache anzulegen. Dazu mischt er die Teamaktivierungslisten aller Teamklassen, die von der CallinSignature vorgegeben sind und speichert das nach dem Aktivierungs-zeitpunkt sortierte Ergebnis im TeamActivationCache. Dieser enthält nun ein Array mit Teaminstanzen, die entsprechend ihrer Aktivierungsreihenfolge angeordnet sind. Im Gegensatz zur OTRE-Implementierung enthält dieses Array allerdings nur die Teaminstanzen, die auch tatsächlich für die gegebene Basisinstanz mit den auf ihr liegenden Callins benötigt werden. Es sind weder leere Felder von deaktivierten Teaminstanzen enthalten noch Teaminstanzen, die für einen anderen Thread aktiviert sind. Auch Teaminstanzen von Teamklassen, deren Rollenklassen zwar an diese Basisklasse binden, aber nur Callins auf andere Basismethoden setzen, sind nicht enthalten. Dies optimiert die Abarbeitung eines TeamActivationCaches, da er nur Einträge enthält, die für die Callin-Ausführung auch relevant sind.

8.3.2 otjClock und Timestamps

Die globale Variable otjClock wird als Uhr zur Vergabe neuer Timestamps verwendet. Ein Timestamp besteht aus einem 32Bit-Zähler und einem Zeiger auf einen *Timeframe*. Auch otjClock besitzt einen 32Bit-Zähler und eine doppeltverkettete Liste von Timeframes. Bei der Vergabe einer neuen Timestamp wird der aktuelle Wert des Zählers und ein Zeiger auf den ersten Timeframe in der Liste von otjClock in einer neu allokierten Timestamp gespeichert, die dann zurückgegeben wird. Außerdem wird der Zähler nach der Vergabe um eins hochgezählt. Kommt es nun zu einem Überlauf des Zählers, so wird ein neuer Timeframe allokiert. Dieser wird an den Beginn der Timeframe-Liste angehängt. Durch diese Liste von Timeframes kann eine Timestamp-Vergabe realisiert werden, die nicht überlaufen kann. Jede Timestamp ist durch ihren Verweis auf einen bestimmten Timeframe von anderen Timestamps mit gleichem Zählwert eindeutig zu unterscheiden. Da Timeframes doppelt verkettet sind, können zwei Timestamps mit identischem numerischem Wert in ihrer zeitlichen Reihenfolge bestimmt werden, indem herausgefunden wird, welcher der beiden Timeframes der jüngere ist. Dazu braucht nur die Liste der Timeframes solange durchlaufen zu werden, bis entweder der jeweils andere Timeframe gefunden wird oder das Ende

der Liste erreicht ist. Diese Methode wird bei dem Zeitvergleich zwischen Timestamps verwendet, der notwendig ist, um die TeamActivationLists verschiedener Teamklassen geordnet zu einem TeamActivationCache mischen zu können.

Timeframes verfügen außerdem über einen Referenzzähler. Für jede Timestamp, die auf den Timeframe verweist, wird dieser Referenzzähler erhöht. Entsprechend wird er für jede Timestamp, die auf ihn verweist, aber freigegeben wird, um eins reduziert. Erreicht er 0, so existieren keine Timestamps mehr, die ihn verwenden. Daher wird der Timeframe freigegeben und aus der doppeltverketteten Liste von otjClock gelöscht. So kann der Speicherverbrauch der Aktivierungszeitverwaltung mit Timestamps im Verlauf einer Programmausführung nicht ins Unermessliche wachsen.

8.3.3 Invalidierung des Team Activation Cache

Die bereits erwähnte zweite Phase der Teamaktivierungsmethoden ist dafür zuständig, alle TeamActivationCaches zu invalidieren, die von der Teamklasse, zu der die aktivierte bzw. deaktivierte Teaminstanz gehört, abhängig sind. Abhängig ist ein TeamActivationCache von allen Teamklassen, die zu seinem Aufbau verwendet werden, also von allen Teamklassen der entsprechenden CallinSignature. In jeder Teamklasse ist nun ein Array gespeichert, das einen Verweis auf alle CallinSignatures enthält, in denen die aktuelle Teamklasse vorkommt.

Jede nichtlokale Aktivierungsoperation invalidiert alle von ihr abhängigen TeamActivationCaches. Für jede lokale Deaktivierungsoperation gilt dies immer dann, wenn die zu deaktivierende Teaminstanz nicht die Teaminstanz ist, mit welcher der jeweilige Cache beginnt. Wird ein Cache von einem Delegationproxy verwendet, so speichert er den Startwert mit dem er beginnt in dem Feld `lastPos` des Caches, wenn sein Startwert numerisch größer ist, als der dort gespeicherte Wert. Findet eine lokale Deaktivierungsoperation nun die zu deaktivierende Teaminstanz an der Startposition des Caches und ist `lastPos` kleiner als die aktuelle Startposition des Caches, so darf die Teaminstanz aus dem Cache gelöscht werden, ohne dass dies den Cache invalidieren würde, weil diese Teaminstanz keinen Einfluss auf gerade ausgeführte Callins haben kann. Dieser Sonderfall wurde implementiert, damit ein Cache nicht jedes Mal bei der häufig auftretenden Sequenz von impliziter Aktivierung mit unmittelbar darauf folgender Deaktivierung invalidiert wird.

Bei einer lokalen Teamaktivierung wird der Cache nur dann invalidiert, wenn die Größe des Caches nicht ausreicht, um die neu zu aktivierende Teaminstanz aufzunehmen.

8.4 Zusammenfassung

Mit den vorgenommenen Veränderungen an den Algorithmen zur Teamaktivierung konnten verschiedene Vorteile erreicht werden. Die Abhängigkeit bezüglich der Ladereihenfolge von Team- und Basisklassen besteht nicht länger. Die verwendeten Algorithmen sind außerdem generisch und benötigen daher kein spezifisches Weaving mehr. Durch die Verwendung eines Team Activation Caches kann der Zeitaufwand, der für das Management der Teamaktivierung bei der Callin-Ausführung besteht, erheblich reduziert werden. So muss das OTRE dazu beispielsweise jeweils zwei neue Arrays allokieren und die Methode isActivate für alle Teaminstanzen aufrufen, die für die jeweilige Basisklasse registriert wurden. Dieser Aufwand wird komplett vermieden. Zusätzlich kann durch das Timestamp-Mischverfahren zum Aufbau eines TeamActivationCaches ein Cache erzeugt werden, der ausschließlich Teaminstanzen enthält, die ein Callin auf die Basismethode des jeweiligen Delegationproxy setzen. Schließlich konnten durch die separate Implementierung des activate() und deactivate()-Befehls zeitintensive Lock()-Operationen eingespart werden. Alle Operationen, die einen TeamActivationCache invalidieren oder modifizieren können, wurden ebenfalls so implementiert, dass für sie diesbezüglich keine Lock-Operationen benötigt werden. Das OTRE nutzt diesen Vorteil nicht aus und bildet activate() stattdessen auf activate(Thread self) ab.

Bei der Entwicklung der neuen Algorithmen für die Teamaktivierung wurde die Entscheidung getroffen, dass eine optimierte Ausführungszeit des Delegationproxy Priorität besitzt. Diese Entscheidung ist dadurch motiviert, dass die Ausführung von Callins in der Regel häufiger vorkommt, als die Ausführung von Aktivierungs-operationen. Daher findet die Cache Invalidierung in den Teamaktivierungs-operationen statt, obwohl sie dort auch für Caches geschehen muss, die möglicherweise im Moment gar nicht verwendet werden. Für den Delegationproxy wäre es jedoch ebenfalls recht aufwändig, festzustellen, ob der Cache invalidiert werden müsste. Dazu müsste er alle Teamklassen der CallinSignature einzeln daraufhin überprüfen, ob für diese Teamklasse seit seiner letzten Überprüfung eine Teamaktivierung oder Deaktivierung stattfand. Befindet sich der Delegationproxy in einer Schleife, die keinerlei Aktivierungsoperationen enthält, würde diese Kontrolle dennoch immer wieder ausgeführt werden und damit kostbare Rechenzeit kosten. Die gewählte Implementierung der Cache Invalidierung in den Teamaktivierungsoperationen vermeidet dies.

Kapitel 9

Der Delegationproxy

- Callinausführung in OTRun

9.1 Callins

Eine Callin-Deklaration verknüpft in ObjectTeams/Java die Methode einer Rollenklasse mit der Methode einer Basisklasse. Wann immer diese Basismethode aufgerufen wird, erfolgt auch ein Aufruf pro Callin-Methode, wenn für den aktuellen Thread zugleich auch mindestens eine Teaminstanz der Teamklasse aktiviert ist, zu der die Rollenklasse gehört, die wiederum die Callin-Methode enthält.

Es existieren drei Arten von Callins in ObjectTeams/Java. BEFORE, AFTER und REPLACE-Callins. BEFORE-Callins werden vor einer Basismethode ausgeführt, AFTER-Callins danach. REPLACE-Callins werden anstelle der Basismethode ausgeführt, können diese aber selbstständig mit einem *Basecall* aufrufen. Alle drei Arten von Callins bieten die Möglichkeit des Parameter-Mappings. Dieses erlaubt es, Parameter, die ursprünglich für die Basismethode gedacht waren, in beliebiger Reihenfolge auf die Parameter einer Callin-Rollenmethode zu verteilen oder auch wegzulassen. AFTER-Callins dürfen zusätzlich auch das Ergebnis des Basismethodenaufrufs (welches auch das Ergebnis eines REPLACE-Callins sein könnte) als Parameter verwenden. Der Rückgabewert von BEFORE- und AFTER-Callins wird ignoriert. REPLACE-Callins liefern entweder den Rückgabewert eines von ihnen durchgeführten Basecalls zurück oder besitzen einen eigenen Rückgabewert.

Die Reihenfolge der Abarbeitung von Callins folgt einer ganzen Reihe von Regeln. Zunächst werden alle BEFORE- und REPLACE-Callins aller Rollenklassen durchgeführt, die zur zuletzt aktivierten Teaminstanz gehören. Anschließend folgen alle weiteren BEFORE- und REPLACE-Callins aller aktiven Teaminstanzen. Wird die letzte aktive Teaminstanz erreicht, so wird zunächst die Basismethode aufgerufen. Anschließend erfolgt der Aufruf aller AFTER-Callins. Hier wird in umgekehrter Teamreihenfolge verfahren – die AFTER-Callins der zuletzt aktivierten Teaminstanz werden also auch zuletzt ausgeführt (Im Gegensatz zu den BEFORE- und REPLACE-Callins, die für diese zuerst ausgeführt werden).
Die Abarbeitung aller aktiven Teaminstanzen kann dabei von einem REPLACE-Callin unterbrochen werden, wenn es auf den Basecall verzichtet.
Durch diese Regeln wird die äußere Abfolge der Callin-Ausführung festgelegt, die durch das Aktivieren und Deaktivieren von Teaminstanzen zur Laufzeit gesteuert werden kann.

Eine innere Abfolge von Callins wird durch *precedence*-Deklarationen bestimmt. Diese definieren die Ausführungsreihenfolge von Callins innerhalb einer Rollenklasse oder zwischen den Callins mehrerer Rollenklassen, wenn diese an dieselbe Basismethode binden. Eine precedence-Deklaration weist den einzelnen Callin-Methoden dann innerhalb einer Rollenklasse eine Priorität zu. So können die Callins innerhalb einer Rollenklasse in die gewünschte Reihenfolge gebracht werden, aber auch verschiedenen Rollenklassen kann so eine Priorität zugewiesen werden. Diese Form der Reihenfolgenfestlegung ist statisch und lässt sich zur Laufzeit nicht mehr verändern.

Von der Sprache selbst fest vorgegeben ist, dass BEFORE-Callins grundsätzlich vor allen anderen Callin-Typen ausgeführt werden. Es ist also keine Mischung aus BEFORE- und REPLACE-Callins möglich. Besitzt eine gegebene Rollenklasse Priorität, so würden zunächst all ihre BEFORE-Callins und anschließend erst alle REPLACE-Callins ausgeführt werden. Innerhalb der Gruppe der BEFORE-Callins kann dann durch eine Precedence-Deklaration eine bestimmte Reihenfolge vorgegeben werden, ebenso innerhalb der Gruppe aller REPLACE-Callins und der Gruppe aller AFTER-Callins.

Callin-Methoden sind Methoden einer Rollenklasse. Sie benötigen also ein Objekt der jeweiligen Rollenklasse, um aufgerufen werden zu können. Daher muss für jeden Aufruf eines Callins auch ein Lifting[23] durchgeführt werden, um die Rolleninstanz zu ermitteln, die dazu der jeweiligen Basisinstanz zugeordnet ist, auf der die ursprüngliche Basismethode aufgerufen wurde.

Nicht bei jedem Aufruf einer Basismethode darf der Callin-Mechanismus aktiviert werden. Ruft eine Basismethode ihre eigene Supervariante auf, so darf dies nicht zur erneuten Ausführung aller Callins führen. Da Callins vererbt werden, besitzt die Basismethode immer auch alle Callins ihrer Supermethode, die deswegen bereits beim ursprünglichen Aufruf der Basismethode zur Ausführung gelangen.

[23] Siehe dazu Kapitel 7

Für REPLACE-Callins existiert schließlich der Sonderfall, dass ein Basecall nicht in der REPLACE-Methode selbst ausgeführt zu werden braucht, sondern dass dies stattdessen in seiner super- oder tsuper-Methode erfolgen kann.

ObjectTeams/Java erlaubt es nicht, REPLACE-Methoden außerhalb eines Callin-Vorgangs, also explizit, aufzurufen.
Zusammenfassend lässt sich festhalten, dass die Implementierung der Callin-Ausführung die Regeln für die äußere und die innere Reihenfolge der Ausführung zu beachten hat, ein Lifting zu Rolleninstanzen durchführen muss, Mechanismen zum Parametermapping benötigt und schließlich auch in besonderen Situationen ein vernünftiges, sprachkonformes Verhalten zeigen soll.

9.2 Der Callin-Ausführungsmechanismus des OTRE

Nachdem nun die grundlegenden Regeln bekannt sind, nach denen die Callin-Ausführung vollzogen werden muss, werden wir in diesem Abschnitt kurz die Umsetzung dieser Regeln im OTRE betrachten um diese schließlich am Ende dieses Kapitels mit der Umsetzung in OTRun mit Hilfe des Delegationproxys vergleichen zu können.

Damit das OTRE die Callin-Ausführung unterstützen kann, verwendet es eine Loadtime-Weaving-Strategie, die den Programmcode in Basisklassen stark verändert. Zunächst verschiebt es den Programmcode von Basismethoden in neu generierte Methoden und ersetzt diesen jeweils durch den Programmcode des so genannten *Initial Wrappers*. Anschließend wird eine neue Methode generiert, der so genannte *Chaining Wrapper*, die dann vom Initial Wrapper aufgerufen werden kann. Dieser wiederum ist nun unter Einhaltung aller Ausführungsregeln für Callins in der Lage, entweder die ursprüngliche Basismethode aufzurufen oder aber die einzelnen Callin-Methoden. Diese werden jedoch nicht direkt aufgerufen – stattdessen erfolgt ihr Aufruf über *Callin Wrapper*, die für jede Callin-Methode vom ot-Compiler in den einzelnen Teamklassen erzeugt werden. Erst diese Callin Wrapper rufen dann die eigentliche Callin-Methode auf. Mit dieser Aufteilung des Callin-Mechanismus in einzelne Methoden geht auch eine deutliche Aufgabenteilung einher.

Der Initial Wrapper erfüllt die Aufgabe, den Chaining Wrapper mit zwei von ihm benötigten Arrays zu versorgen. Das eine enthält alle Teaminstanzen, die für den aktuellen Thread aktiviert wurden und die zugleich Rollenklassen besitzen, die an die Basisklasse binden, zu der die Basismethode gehört, die der Callin Wrapper nun ersetzt hat. Das zweite Array enthält IDs, mit denen die Teamklasse der Teaminstanz eindeutig identifiziert werden kann. Die IDs werden im Chaining Wrapper von switch-Anweisungen verwendet[24], die zu Programmfragmenten für die jeweilige Teamklasse verzweigen.

[24] Diese switch-Anweisung wird nicht zu einem schnellen tableswitch, sondern zu einer langsamen lookupswitch-Anweisung kompiliert, die bei jeder Ausführung mit einer Schleife alle case-Fälle durchgeht, um so nach Übereinstimmung mit dem switch-Parameter zu suchen.

Der Chaining Wrapper enthält dazu für jede Teamklasse die jeweiligen Aufrufe ihrer für diese Basismethode zu verwendenden Callin Wrapper, angeordnet in der inneren Abfolge, die von den precedence-Attributen aus den .class-Files der beteiligten Teamklassen vorgegeben ist. Der Chaining Wrapper arbeitet also nicht generisch, sondern ist nur für den Einsatz für eine ganz bestimmte Basismethode geeignet (und muss daher auch für jede Basismethode separat erstellt werden).

Um durch das Array der aktiven Teaminstanzen zu navigieren, verwendet der Chaining Wrapper Rekursion. Er ruft sich für jede Iteration mit einer inkrementierten Position als Parameter auf, die auf den Eintrag verweist, der für den jeweiligen Durchlauf zu verwenden ist.

Dieses Array kann auch Teaminstanzen enthalten, die für die Ausführung der aktuellen Basismethode irrelevant sind, weil die zugehörige Teamklasse keine Callins auf die aktuelle Basismethode setzt, sondern auf eine andere Methode der Basisklasse. Dieser Fall kostet zusätzliche Laufzeit in Form eines für ihn erfolgten rekursiven Aufrufs des Chaining Wrappers und durch die Switch-Anweisungen, die, da sie durch Schleifen realisiert sind, alle case-Fälle durchgehen müssen, bevor sie den Default-Fall erreichen, der diese Situation behandelt. Der Delegationproxy-Ansatz soll diesen Aufwand vermeiden.

Das Lifting einer Basisinstanz zur Rolleninstanz erfolgt zusammen mit dem Parametermapping in den Callin Wrappern. Für den Fall von AFTER-Callins übernehmen sie bei Bedarf daher auch das Parametermapping des Rückgabewertes auf einen Parameter der AFTER-Callin-Methode.

Der grundlegende Programmablauf des OTRE bei der Callin-Ausführung ist damit beschrieben. Eine detaillierte Beschreibung des Chaining Wrappers und aller weiteren beteiligten Methoden lässt sich in [Hu03] finden.

Die Erzeugung eines Chaining Wrappers, der für jede einzelne Basismethode angepassten Code enthält, führt zu einem erheblichen Webeaufwand zur Ladezeit. Außerdem lässt sich dieser Vorgang zur Laufzeit nicht wieder rückgängig machen. Ein weiterer Nachteil besteht darin, dass zum Webezeitpunkt alle Team- bzw. Rollenbindungen an die Basisklasse bekannt sein müssen und zur Laufzeit keine neuen Bindungen dazukommen können. So ist es beispielsweise bisher nicht möglich, dass der Benutzer einer Applikation zur Laufzeit des Programms eine Teamklasse frei auswählt, um sie dazu zu laden.
Da der Chaining Wrapper Code von einem Code Generator Programm erzeugt wird, entsteht außerdem zusätzlicher Wartungsaufwand, da zunächst der Code Generator geschrieben werden muss, also ein Programm, das ein anderes Programm schreibt. Fehler können nun in beiden Teilen auftreten, im Generator selbst und in dem vom Generator generierten Code. Änderungen am zu erzeugenden Code sind schwieriger durchzuführen, da zunächst die Funktionsweise des Generator-Programms verstanden sein will. Schließlich erzeugt die verwendete Lösung auch einen erheblichen Laufzeit-Overhead. Aus dem einzelnen Aufruf einer Basismethode wird eine ganze Kette von Wrapper- und Hilfsmethoden-Aufrufen, bevor die tatsächlichen Nutzmethoden, d.h. die Callin-

Methoden oder die Basismethode selbst, ausgeführt werden können. Für den rekursiven Aufruf des Chaining Wrappers sind jedes Mal fünf Parameter zur internen Steuerung zu übergeben und zudem wird er wie bereits erwähnt mitunter auch für Teaminstanzen aufgerufen, die zu der aktuellen Basismethode kein Callin beisteuern. Der Chaining Wrapper Code verwendet außerdem die langsame Form der switch-Anweisung (den lookupswitch Bytecode), so dass es sich bei ihm tatsächlich um eine Schleife mit einem Aufwand von O(k)[25] anstelle von den vermuteten O(1) handelt. Der Initial Wrapper allokiert bei jedem Aufruf zwei Arrays um darin die Teaminstanzen und die TeamIDs zu speichern –ebenfalls ein rechenzeitintensiver Vorgang. Auch die Methode, die verwendet wird, um dieses Array aufzubauen, kostet sehr viel Rechenzeit. Da in den Basisklassen nur gespeichert wird, welche aktiven Teaminstanzen existieren, nicht aber, für welchen Thread sie aktiviert sind, muss der Füllvorgang für das Array zunächst auf jeder Teaminstanz die Methode isActive() aufrufen, die wiederum in einer Hashtabelle der Teaminstanz nachschlägt, ob der aktuelle Thread dort zu finden ist. Schlägt dieser Test fehl, so wird dennoch ein Eintrag im Array für den Chaining Wrapper erzeugt (Eine Null-Referenz bzw. –1 als TeamID), was letztlich zu einem weiteren Chaining Wrapper Aufruf führt, dessen einziger Zweck das inkrementieren des Positionszeigers für das Array ist.[26]

All diese Nachteile sollen nun mit dem Konzept des Delegationproxys überwunden werden.

9.3 Der Delegationproxy

Beim Delegationproxy handelt es sich um eine Sequenz aus 3 Bytecodes, die jeder Methode durch die virtuelle Maschine vorangestellt wird[27]. Dies geschieht bei der ersten Ausführung einer Methode, während der prepare-Phase[28]. Außerdem wird in der Prepare-Phase überprüft, ob im Methodenblock der Methode ein spezielles Flag[29] gesetzt ist. So kann entschieden werden, ob der Codezeiger des

[25] k = die Anzahl der Teamklassen, die Rollenklassen mit playedBy-Bindungen zu der Basisklasse der Basismethode enthalten.

[26] In [Hu09] und [Hu10] wird allerdings ein Lösungsansatz beschrieben, der die Laufzeit des Aktivierungsmechanismus verbessert.

[27] Die virtuelle Maschine wurde so angepasst, dass dies keine Auswirkung auf die Ausnahmetabellen hat, die von der virtuellen Maschine für try-catch-Blöcke verwendet werden. Diese benutzen Start- und Endpositionen, um damit den Umfang des try-Blocks und die Position des Exception Handlers zu codieren und würden daher ohne Anpassung falsche Positionen beschreiben. Die Anpassung selbst ist einfach. Der Beginn des ursprünglichen Codeblocks wird zusätzlich im Methodenblock gespeichert und wird nun anstelle des Zeigers auf den Start der Methode von der entsprechenden Funktion der virtuellen Maschine verwendet. Unabhängig davon, ob der Codezeiger der Methode nun also auf den Delegationproxy zeigt oder nicht, wird immer der ursprüngliche Methodenanfang verwendet, weshalb der Inhalt des Codezeigers im Methodenblock auf die Positionsbestimmung bei der Ausnahmebehandlung keinen Einfluss mehr hat.

[28] Siehe dazu Kapitel 3.

[29] Es handelt sich um das Flag `OTJ_STATE_DEPLOYPROXY` , welches im `state` –Feld der ObjectTeamsMethodData-Struktur zu finden ist. Dieses wird durch den Callin Deploy-

Methodenblocks auf den neuen Beginn der Methode verweisen soll, der den Delegationproxy einschließt, oder auf den ursprünglichen Beginn direkt hinter den Bytecodes des Delegationproxy.

Eine Methode, deren Codezeiger auf den Delegationproxy zeigt, wird diesen bei jedem Aufruf ausführen und damit den Aufruf aller Callin-Methoden veranlassen, die für diese Basismethode aufgrund der geladenen Teamklassen und aktiven Teaminstanzen vorgesehen sind. Eine Basismethode, die sich rekursiv aufruft, würde so bei jedem Aufruf ebenfalls den Aufruf aller Callin-Methoden auslösen, weil sie jedes Mal zuerst den Delegationproxy ausführen wird. Ruft sie dagegen ihre eigene Supermethode auf, so wird der Aufruf des Delegationproxy unterdrückt und stattdessen an die Stelle direkt hinter den Delegationproxy der Supermethode gesprungen. Dies wurde durch eine Anpassung des invokespecial Bytecodes realisiert, der für Super-Aufrufe verwendet wird[30].

Der Delegationproxy vereint in Form einer effizienten Superinstruction die drei zentralen Delegationsmechanismen von ObjectTeams: Lifting, Teamaktivierung und Callin-Ausführung. Bevor wir dies jedoch im Detail betrachten, untersuchen wir zunächst den Vorteil der sich durch den Delegationsmechanismus für das dynamische Laden von Teamklassen ergibt.

Zunächst ist festzuhalten, dass es sich beim Delegationproxy um einen generischen Algorithmus handelt. Er besitzt also im Gegensatz zum Chaining Wrapper keine abstrakte Struktur, die dann für jeden Einsatzort individuell angepasst und gewebt wird, sondern wird allein durch Datensätze dynamisch gesteuert. Daher genügt es, den Programmcode des Delegationproxy-Bytecodes einmalig in der virtuellen Maschine vorliegen zu haben.[31] Der Webevorgang für ObjectTeams/Java wird so erheblich vereinfacht und Generator-Programme für den Chaining Wrapper werden nicht mehr benötigt. Vor allem aber kann die Verwendung des Delegationproxys jederzeit für eine Methode aktiviert oder deaktiviert werden, indem der Codeblock-Zeiger so verändert wird, dass er entweder auf den Delegationproxy zeigt, oder auf den Originalcode. Dies wird so auch vom Klassenladealgorithmus eingesetzt, um Callins von Teamklassen wirksam werden zu lassen, die erst nach ihren Basisklassen geladen wurden, d.h. zu einem Zeitpunkt, an dem die Basismethoden ihre prepare()-Phasen bereits abgeschlossen haben, weil sie bereits mindestens einmal ausgeführt wurden. Zusätzlich ist es irrelevant, ob diese Teamklasse zum Startpunkt des Programms verfügbar war oder nicht, so dass Teamklassen nun auch während ein Programm ausgeführt wird, theoretisch – abhängig von den

---

Vorgang für Basismethoden gesetzt, wenn eine Teamklasse geladen wird. Siehe dazu auch Kapitel 6.

[30] Für tsuperself Aufrufe existiert eine solche Anpassung noch nicht, da für tsuper()-Aufrufe invokevirtual verwendet wird – und nicht invokespecial. Für eine zukünftige Erweiterung von OTRun ist aber geplant, den tsuperself-Aufruf durch einen eigenen Bytecode zu ersetzen, der dann analog zu invokespecial arbeitet.

[31] Ein Nebeneffekt ist die dadurch auftretende Speicherplatz-Ersparnis, da Codeblöcke für Initial- und Chaining Wrapper nicht mehr benötigt werden. Allerdings kostet das Einfügen des Delegationproxys 24 Byte pro Methode. Für eine zukünftige Version von OTRun ist es daher vorgesehen, Speicherplatz für den Delegationproxy kontextsensitiv nur dann zu allokieren, wenn er auch tatsächlich benötigt wird.

Entscheidungen des Benutzers – bei einer entsprechenden Applikation über ein Netzwerk nachgeladen werden können. Dies eröffnet Anwendungsszenarien für ObjectTeams/Java, die mit den Möglichkeiten des OTRE bisher nicht in dieser Form realisiert werden konnten, da das Ladezeitweben des OTRE voraussetzt, das zum Startpunkt des Programms alle Teamklassen verfügbar sein müssen.

9.4 Aufbau des Delegationproxy

Der Delegationproxy besteht aus dem für OTRun neu eingeführten Bytecode `OPC_OTJPROXY` (Bytecode Nr. 249). Dieser wird ausschließlich innerhalb der virtuellen Maschine verwendet und wird direkt im Anschluss an die prepare()-Phase eingefügt. Er kann daher nicht extern in .class-Dateien verwendet werden, weil die Bytecode-Verifikation der virtuellen Maschine ihn als unbekannten Bytecode erkennen würde, was zu einem Abbruch der prepare()-Phase führt. Dies ist ein gewünschtes Verhalten, da der Standard für die Java Virtual Machine im Rahmen dieser Arbeit nicht erweitert werden soll. Die exakte Funktion dieses Bytecodes wird erst durch den ersten 16-Bit Parameter des Bytecodes festgelegt. Dieser Parameter sorgt in einer switch-Anweisung dafür, dass der eigentliche Opcode, für den der Bytecode jeweils steht, ausgeführt wird. `OPC_OTJPROXY` kennt bisher drei Opcodes:

`OTJ_OP_DELEGATIONINTRO`	leitet den Delegationproxy ein und ist funktionell äquivalent zum Initial Wrapper des OTRE.
`OTJ_OP_DELEGATIONPROXY`	enthält die Logik, welche die Aufrufreihenfolge von Callins steuert und ist mit dem Chaining Wrapper des OTRE vergleichbar.
`OTJ_OP_BASECALL`	Basecalls werden durch Infrastructure Weaving zu diesem Bytecode transformiert. Dieser realisiert bei seinem Aufruf entsprechend einen Basecall.

Jede Methode, deren Codezeiger auf einen Delegationproxy zeigt, beginnt mit der folgenden Sequenz:

```
PC[32]
[0]    249, 0, 0      OTJ_OP_DELEGATIONINTRO
[1]    249, 1, 1      OTJ_OP_DELEGATIONPROXY
[2]    249, 1, 2      OTJ_OP_DELEGATIONPROXY
[3]    …der Programmcode der Methode…
```

Der Delegationproxy Opcode ist zweimal vorhanden. Über den 2. Parameter kann der Delegationproxy Opcode zur Laufzeit feststellen, ob es sich um den

[32] Der Programmcounter relativ zum Start der Methode. Die JamVM verwendet für den PC einen direkten Zeiger, der auf die Speicheradresse des jeweils gerade ausgeführten Bytecodes zeigt.

Delegationproxy Opcode an Position 1 oder 2 handelt. Dies ist notwendig, weil jeder der einzelnen Bytecodes des Delegationproxy einen eigenen Einsprungspunkt in die Logik des Delegationproxy darstellt. Aus dem Delegationproxy heraus werden Aufrufe von BEFORE-, AFTER- und REPLACE-Methoden getätigt. Auch die Basismethode wird von dort aufgerufen. Da jedoch REPLACE-Methoden und auch die Basismethode einen Rückgabewert liefern können, muss der Delegationproxy die Rückkehr aus diesen Methoden erkennen, damit er die Rückgabewerte entsprechend entgegen nehmen kann und diese nicht verloren gehen. Daher kehren Aufrufe von REPLACE-Methoden und der Basismethode zu dem ersten Delegationproxy Opcode zurück, BEFORE- und AFTER-Methoden dagegen zum zweiten Delegationproxy Opcode. Dieser ist außerdem der Einsprungspunkt für den Aufruf eines Basecalls, weil der Aufruf des Delegationproxy Intro Opcodes durch einen Basecall eine Endlosschleife auslösen würde.

Der Delegationproxy verwaltet seinen aktuellen Zustand mit Hilfe von Feldern im Stackframe, der bei jedem Methodenaufruf für eine Methode angelegt wird. Dazu wurde der Aufbau des Stackframes der JamVM um 9 Felder erweitert[33]:

```
typedef struct frame[34]
{
   CodePntr                        last_pc;
   uintptr_t*                      lvars;
   uintptr_t*                      ostack;
   MethodBlock*                    mb;
   struct frame*                   prev;
   ------------------------------------ < ab hier neu Hinzugefügt
   struct frame*                   basecallFrame;
   int                             basecallbmPos;
   int                             curpos;
   int                             startpos;
   struct teamactivationcache*     curCache;
   Object*                         ri;
   int                             bmPos;
   uintptr_t                       result1;
   uintptr_t                       result2;
} Frame;
```

Die Bedeutung der einzelnen Felder wird im Verlauf dieses Kapitels erläutert werden.

Der Codezeiger einer Methode, für die Callins existieren, zeigt, wie bereits erwähnt, immer auf einen Delegationproxy. Daher nimmt dieser seine Arbeit auf, sobald die ursprüngliche Basismethode von einer anderen Methode aufgerufen wird. Die Ausführung des Delegationproxy beginnt daher mit dem Stackframe, der

[33] Zurzeit werden diese Felder bei jedem Aufruf einer Methode zusammen mit den Feldern des JamVM-Stackframes allokiert. Dies wirkt sich nicht auf die Laufzeit aus, allerdings auf den Speicherverbrauch rekursiver Methoden, der sich dadurch verdreifacht. Deshalb soll die Framegröße in zukünftigen Versionen dynamisch gesteuert werden, so dass nur noch Aufrufe, die zu einem Delegationproxy springen, einen entsprechend großen Frame auf dem Stack allokieren.

[34] Für die Funktionsweise von Stackframes und von Methodenaufrufen siehe Kapitel 3.

für die ursprüngliche Basismethode angelegt wurde. In `frame->mb` findet der Delegationproxy den Methodenblock der Basismethode. Über die lokale Variable `lvars` des JamVM-Interpreters kann der Delegationproxy auf alle Parameter zugreifen[35], die dem Basismethodenaufruf mitgegeben wurden, einschließlich des this-Zeigers auf die Basisinstanz.

9.5 Die Arbeitsweise von `OTJ_OP_DELEGATIONINTRO`

Der Delegationproxyintro Opcode dient dazu, dem Delegationproxy einen gültigen Aktivierungscache zur Verfügung zu stellen. Ein Aktivierungscache enthält in einem Array alle Teaminstanzen, die für den aktuellen Thread aktiv sind und die außerdem über ihre Rollenklassen mindestens ein Callin besitzen, dessen Ziel die aktuell ausgeführte Basismethode ist. Aktivierungscaches werden in jedem Thread jeweils für jede Callin Signature angelegt, wenn ein Delegationproxy ausgeführt wird, der die entsprechende Callin Signature verwendet. Dieser speichert den von ihm erzeugten Cache jeweils in einem Array der Environment-Variable *ee* des aktuellen Threads. Solange dieser Cache nicht durch eine Aktivierungsoperation invalidiert wird, kann er bei jedem Aufruf des Delegationproxys erneut verwendet werden. Dies beschleunigt insbesondere die Ausführung von Schleifen, wenn innerhalb der Schleifen der Aktivierungszustand der Teaminstanzen nicht verändert wird, weil der Cache nun nur noch aus dem Environment-Array geladen werden braucht. Die aufwendige Erstellung eines Caches entfällt dann.

Außerdem initialisiert `DELEGATIONPROXYINTRO` alle weiteren Felder des Stackframes, die der Delegationproxy benötigt.

Der `DELEGATIONPROXYINTRO` Opcode soll nun im Detail betrachten werden:

```
case (OTJ_OP_DELEGATIONINTRO):
{

(1)   CallinSignature* cis = frame->mb->otjm->signature;

(2)   <...BoundCheck für TeamActivationCacheArray...>

(3)  curCache  = ee->TeamActivationCacheArray[cis->cacheOffset];

(4)  if(!curCache)
     {
               curCache = otjBuildCache(cis, curCache, ee);
(5)            if(!curCache){return(OTJ_PROXY_CONTROL_FALLTHROUGH);}
     }
// post : curCache enthält nun einen gültigen Zeiger auf eine Cache-Struktur
(6)   if(COMPARE_AND_SWAP(&curCache->isInvalidated, OTJ_CACHE_INVALIDATE,
                                                    OTJ_CACHE_OK))
     {
       oldCache = curCache;
       curCache = otjBuildCache(cis, curCache, ee);
       if(!curCache){return(OTJ_PROXY_CONTROL_FALLTHROUGH);}
       ee->TeamActivationCacheArray[cis->cacheOffset] = curCache;
       COMPARE_AND_SWAP(&curCache->isInvalidated, OTJ_CACHE_OK,
```

[35] Dabei enthält lvars[0] die this-Referenz. Von lvars[1]...lvars[`mb->args_count-1`] sind alle Parameter, die der Methode übergeben wurden, zu finden.

```
                                          oldCache->isInvalidated);

        if(oldCache->refCount <= 1){otjSysFree(oldCache);}
        else{oldCache->refCount--;}
     }
// post : curCache ist nun verwendbar.
(7)  frame->curCache                = curCache;
     frame->curpos                  = curCache->start;
     frame->startpos                = curCache->start;
     frame->curCache->lastPos       = curCache->start;
     frame->bmPos                   = 1;
     frame->basecallbmPos           = 0;
     frame->curCache->refCount++;

       // return(ostack);
(8)  return(OTJ_PROXY_CONTROL_GOTOPROXY2);
}
```

In (1) wird die CallinSignature ermittelt, die zu der aktuellen Basismethode gehört. Da diese im Methodenblock der Basismethode gespeichert ist, können wir sie aus diesem unmittelbar laden. Im Moment interessiert uns jedoch nur ein Feld aus der CallinSignature, welches wir für (3) benötigen, das `cacheOffset` Feld. Dieses enthält den Index, den wir in (3) verwenden können, um über `ee->Team-ActivationCacheArray` auf den ActivationCache für diese CallinSignature und damit auch für den aktuellen Delegationproxy Zugriff zu erhalten. Bevor dies jedoch möglich ist, muss in (2) zunächst geprüft werden, ob das ActivationCacheArray im aktuellen Thread überhaupt groß genug ist. Es kann passieren, dass dies nicht der Fall ist, nämlich dann, wenn zwischenzeitlich neue Teamklassen geladen wurden, die aufgrund der in ihnen enthaltenen Callins die Erzeugung neuer CallinSignaturen anstoßen. Jedes Mal wenn eine CallinSignature neu angelegt wird, wächst auch die Größe der ActivationCacheArrays, da jede CallinSignature bei ihrer Erzeugung einen einzigartigen Index in das ActivationCacheArray erhält. Da dies jedoch nicht die tatsächliche Vergrößerung der Arrays in allen Threads auslöst, bleibt es dem Delegationproxy überlassen, dieses Array bei Bedarf zu vergrößern.[36]

In (4) wird dann getestet, ob bereits ein ActivationCache in diesem Thread und für diese CallinSignature existiert. Ist dies nicht der Fall, so wird er neu erzeugt[37]. Die Funktion zur Erzeugung eines neuen Caches `TeamActivationCache* otjBuildCache(…)` kann jedoch auch NULL zurückliefern. Dies ist immer dann der Fall, wenn keine passenden Teaminstanzen aktiviert sind. In diesem Fall wird die Ausführung des Delegationproxy durch (5) sofort abgebrochen und der OTJProxy Bytecode veranlasst, über die restlichen beiden Anweisungen des Delegationproxy hinweg direkt zum Beginn der Basismethode zu springen. Da der Stackframe des Delegationproxys zugleich der Stackframe der Basismethode ist, kann diese sofort ausgeführt werden. Im Fall, dass keine aktiven Teaminstanzen

[36] Dies soll verhindern, dass ActivationCacheArrays für Threads allokiert werden, in denen nie ein Delegationproxy ausgeführt wird. Ein Delegationproxy testet nur, ob das Array groß genug ist, um mit dem aktuellen Index darauf zugreifen zu können. Auch dies bewahrt das Array davor vergrößert zu werden, wenn dies gar nicht notwendig ist, weil womöglich nie ein Delegationproxy in diesem Thread ausgeführt wird, dessen CallinSignature einen höheren Indexwert besitzt.

[37] Dieser Vorgang wird in Kapitel 8 beschrieben.

vorliegen, wird also keine andere Java-Methode ausgeführt als allein die Basismethode. Am Ende von (4) enthält die Variable `curCache` garantiert einen gültigen Zeiger auf einen ActivationCache, allerdings wissen wir noch nicht, ob der Cache auch gültig ist. Dies wird nun in (6) getestet.

Der Punkt (6) ist etwas schwieriger nachzuvollziehen – er löst das Multithreading-Problem, auf den ActivationCache ohne Verwendung einer zeitaufweändigen lock()-Operation[38] zugreifen zu können. Dazu wird die atomare[39] Operation `COMPARE_AND_SWAP(variableToTest, TestValue, NewAssignmentValue)` verwendet. Diese testet in (6) zunächst, ob der Cache ungültig ist. Ist das der Fall, so ist `COMPARE_AND_SWAP` wahr und schreibt zugleich den Wert `OTJ_CACHE_OK` zurück in das `isInvalidated` -Feld des Caches. Dies soll anzeigen, dass der Cache von nun an verwendbar sein wird, da wir ihn nun neu erzeugen werden. Schlägt `compare_and_test` dagegen fehl, so wissen wir, dass der Cache gültig sein muss, so dass wir ihn sofort verwenden und zu (7) übergehen können.

Bleiben wir aber noch kurz bei (6). Nachdem wir festgestellt haben, dass der Cache ungültig ist, sichern wir uns zunächst den Zeiger auf den alten Cache in einer lokalen Variable und erzeugen anschließend einen neuen Cache. Dies kann wie in (4) dazu führen, dass kein Cache erzeugt wird, weil keine aktivierte Teaminstanz vorhanden ist, weshalb wir in diesem Fall wie bei (5) vorgehen und die Ausführung des Delegationproxys abbrechen. Da wir nun einen neuen Cache besitzen, wollen wir ihn auch in das threadlokale TeamActivationCacheArray zurückschreiben, um ihn zukünftig wieder verwenden zu können. Da dieses Array nicht durch eine Lock-Operation geschützt ist, kann es sein, dass das alte Array inzwischen invalidiert wurde. Würden wir nun einfach den alten Cache durch den neuen Cache ersetzen, würde uns diese Information verloren gehen. Aber wir haben ja noch den Zeiger auf den alten Cache lokal gespeichert, weswegen wir nun ohne Bedenken unseren neuen Cache im TeamActivationCacheArray speichern können.

Nun liegt folgende Situation vor: Der neue Cache ist höchstwahrscheinlich gültig, der alte Cache kann es immer noch sein, da wir ihn bei unserem ersten Test auf gültig gesetzt hatten. Von keinem der beiden wissen wir dies aber mit Sicherheit, da auch der neue Cache gleich nach dem Moment invalidiert werden könnte, in dem wir ihn in den Cache geschrieben haben. Deshalb sorgt die folgende `COMPARE_AND_SWAP` Operation dafür, dass wir auf keinen Fall eine Invalidierung verlieren. Sie testet zunächst, ob der aktuelle Cache gültig ist. Wenn das der Fall ist, dann erhält der neue Cache den Zustandswert des alten Caches zugewiesen.

---

[38] Wir werden in Kapitel 10 sehen, dass die Lock-Operationen der LiftTo-Funktion im Delegationproxy mehr als 50% der Laufzeit des gesamten Delegationproxy ausmachen. Optimierungen zur Vermeidung von Locking sind daher sehr effektiv.

[39] Eine atomare Operation ist unteilbar, kann also in einem Taktzyklus ausgeführt werden. Mit atomaren Operationen wird vor allem bei Zuweisungen verhindert, dass während ein Wert aus einer Speicherstelle in den Prozessor geladen wird, um diesen zu verändern, zeitgleich ein anderer Thread das gleiche versucht. Ist dieser nun bereits dabei, seinen veränderten Wert zurückzuschreiben, so landet der neue Wert zwar in der Speicherstelle, wird aber kurz darauf von dem veränderten Wert des ersten Threads überschrieben, wenn dieser wieder an der Reihe ist. Eine atomare Operation führt den laden-modifizieren-speichern Zyklus nicht teilbar aus und verhindert damit das beschriebene Problem.

Wurde der alte Cache also seit unserem ersten `COMPARE_AND_SWAP` invalidiert, so wurde diese Information nun auf den neuen Cache übertragen. War dagegen der neue Cache ungültig, so ändert sich an seinem Zustandsfeld nichts, er bleibt also ungültig. Das wir den neuen Cache nun trotzdem verwenden dürfen, liegt an der Semantik von ObjectTeams, die den Aktivierungszustand während des Verlaufs eines Delegationproxys für diesen Delegationproxy für nicht veränderbar erklärt. Zu dem Zeitpunkt, als wir ihn erstellt haben, war er gültig und ist deshalb der Cache, den wir nun auch verwenden werden. Der nächste Aufruf des Delegationproxys würde dann womöglich feststellen, dass der Cache inzwischen invalidiert wurde und wird dann einen neuen anlegen. Nachdem wir nun also einen Zeiger auf einen gültigen TeamActivationCache besitzen, können wir mit (7) fortfahren und den Stackframe für den anschließenden Delegationproxy Opcode initialisieren. Vorher reduzieren wir jedoch noch den Referenzzähler des alten Cache und testen anschließend, ob er nun noch jemandem gehört. Ist das nicht der Fall, so können wir ihn freigeben. Wenn doch, wird er von einem Delegationproxy verwendet, der in der Aufrufhierarchie über dem aktuellen Delegationproxy angesiedelt ist, also irgendwo in der Kette der Methodenaufrufe zu finden ist, die letztendlich zum Aufruf des aktuellen Delegationproxy geführt hat. Dieser wird dann nach seiner Beendigung feststellen, dass niemand mehr eine Referenz auf diesen Cache hält und selbstständig den Cache freigeben. Da wir den aktuellen Cache natürlich verwenden wollen, erhöhen wir dessen Referenzzähler um eins.

Im Stackframe wird nun der in den vorherigen Schritten ermittelte Zeiger auf einen gültigen Cache gespeichert. Außerdem kopieren wir aus dem Cache das Startpositionsfeld in das gleichnamige Feld im Stackframe. Dieses Feld enthält den Index der zuletzt aktivierten Teaminstanz für das Teaminstanz-Array in unserem TeamActivationCache. Da wir mit der Ausführung des Delegationproxy erst noch beginnen wollen, wird auch der aktuellen Position `frame->curpos` der Startwert zugewiesen. Indem wir diesen Wert auch in den TeamActivationCache schreiben, sichern wir den Cache dagegen ab, durch eine UpdateOperation[40] in einen ungültigen Zustand zu geraten.

Schließlich initialisieren wir noch die beiden Felder `frame->bmPos = 1` und `frame->basecallbmPos = 0`, die im folgenden Abschnitt erläutert werden. Mit der Rückkehroperation in (8) springen wir nun zum 2. Delegationproxy-Opcode und überspringen damit den 1. Delegationproxy-Opcode, der wie bereits erwähnt nur für die Rückkehr aus REPLACE Methoden und der Rückkehr aus dem Basismethodenaufruf zuständig ist.

9.6 Die Arbeitsweise von `OTJ_OP_DELEGATIONPROXY`

Um die Funktionsweise des Delegationproxy Opcodes zu verstehen, müssen wir zunächst den Aufbau des Arrays `otjcT->bmArray` untersuchen und den Zweck der `CallinID` verstehen, die genauso wie die CallinSignature im Methodenblock der Basismethode gespeichert ist.

[40] Siehe Kapitel 8.

9.6.1 bmArray und CallinID

Zum bmArray ist zunächst zu sagen, dass jede Teamklasse ein Array von bmArrays[41] besitzt. Jedes dieser bmArrays beschreibt die Menge aller Callins der Rollenklassen dieser Teamklasse, die an eine gemeinsame Basismethode binden. Zusätzlich besitzt jede Teamklasse eine `callinTable`. Diese enthält Referenzen auf einzelne bmArrays. Mit der `CallinID` kann nun genau der Eintrag aus der `callinTable` ausgelesen werden, der auf das bmArray zeigt, welches die Callins für die aktuelle Basismethode speichert. Das Besondere an der `CallinID` ist nun, dass sie so vergeben wird, dass sich alle Teamklassen, die zu einer bestimmten CallinSignatur gehören, jeweils eine `CallinID` für eine bestimmte Basismethode teilen. Auf Abbildung 10 ist dies dargestellt. Dies erlaubt es nun, die Liste der aktivierten Teaminstanzen zu durchlaufen und über die `CallinID` aus dem Methodenblock der Basismethode sofort das jeweils richtige bmArray für diese Teaminstanz zu ermitteln, das zu der aktuellen Basismethode passt – und das, obwohl die Liste der aktivierten Teaminstanzen die Instanzen verschiedener Teamklassen enthalten kann. Dies ist das OTRun-Äquivalent zum TeamID-Switch des OTRE Chaining Wrappers.

Nachdem wir nun durch die Verwendung der `callinID` aus dem Basismethodenblock für jede Teaminstanz aus dem ActivationCache ein passendes bmArray mit einem Aufwand von O(1) ermitteln können, soll nun beschrieben werden, wie das bmArray aufgebaut ist, wozu es nützt und wie es im Delegationproxy verwendet wird.

Ein bmArray enthält Einträge vom Typ BakedMethodEntry:

```
typedef struct bakedmethodentry
{
	int			state;
	union
	{
		int			rc;
		MethodBlock*		mb;
		int			afterPos;
	}data;
	// paramapping*		paramap;[42]
}BakedMethodEntry;
```

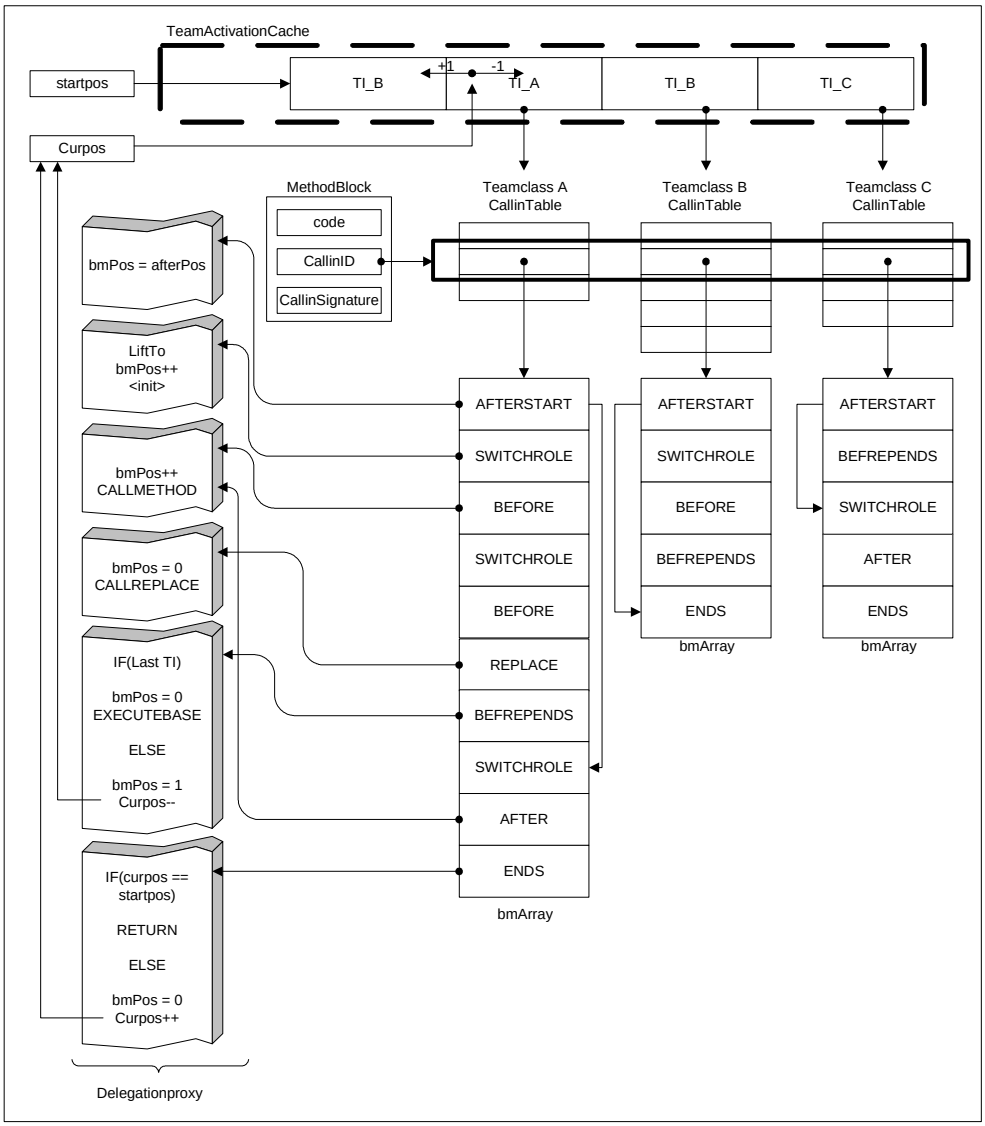
Der Inhalt des union-Felds `data` richtet sich nach dem Typ des Eintrags, der in dem Feld `state` codiert ist. Das Feld `state` kann einen der folgenden Werte annehmen:

```
#define OTJ_PROXY_STATE_AFTERSTART	0
#define OTJ_PROXY_STATE_SWITCHROLE	1
#define OTJ_PROXY_STATE_BEFORE		2
#define OTJ_PROXY_STATE_REPLACE		3
#define OTJ_PROXY_STATE_AFTER		4
```

[41] bmArray steht für bakedMethodArray.
[42] Parametermapping wird in der aktuellen Version von OTRun noch nicht unterstützt. Eine zukünftige Version würde an dieser Stelle jedoch die entsprechenden Informationen dafür speichern.

```
#define OTJ_PROXY_STATE_BEFREPENDS      5
#define OTJ_PROXY_STATE_END             6
```

Das bmArray codiert für jede Teamklasse und jede Basismethode eine feste Abfolge von BEFORE, REPLACE und AFTER-Callins, die durch precedence-Deklarationen vorgegeben ist[43]. Da die einzelnen Callins zu verschiedenen Rollenklassen gehören können, existieren im bmArray zusätzliche Einträge, welche die jeweilige Rollenklasse spezifizieren. Der Delegationproxy-Opcode verwendet das bmArray um durch diese Menge von Callins zu navigieren und sie in der richtigen Reihenfolge auszuführen. Dazu enthält er eine switch-Anweisung, die für jeden Eintragstyp eines bmArrays eine Behandlungsroutine besitzt.



[43] Der OTRun-Prototyp unterstützt noch kein Precedencemapping.

Abbildung 10: CallinTable, bmArray und Delegationproxy

Jedes bmArray beginnt mit einem Eintrag vom Typ `AFTERSTART` an der Array-Position 0. Dieser Typ verwendet das union-Feld `afterPos`, in dem die Array-Position gespeichert ist, ab der im Array alle Einträge vom Typ `AFTER` zu finden sind.

Existieren BEFORE- oder REPLACE-Callins, folgt nun ein Eintrag vom Typ `SWITCHROLE` an Position 1, in dessen union-Feld `rc` die zugehörige Rollenklasse codiert ist. Bei dem Wert in `rc` handelt es sich um einen Index in die `rcTable` der Teamklasse. Über den Eintrag in der `rcTable` kann der Lifting-Algorithmus alle Informationen erhalten, die er zu einer spezifischen Rollenklasse benötigt[44].

Auf den `SWITCHROLE`–Eintrag folgen zunächst alle `BEFORE`–, anschließend alle `REPLACE`–Einträge, die dieser Rollenklasse zugeordnet sind. Jeder dieser Einträge enthält in dem union-Feld `mb` einen Zeiger auf den Methodenblock der Rollenmethode des Callins. Existieren für die Teamklasse weitere Rollenklassen mit BEFORE- oder REPLACE-Callins, so folgen diese nun analog, jeweils beginnend mit einem `SWITCHROLE`-Eintrag.

Folgen keine weiteren BEFORE- oder REPLACE-Callins mehr oder existieren für diese Teamklasse weder BEFORE- noch REPLACE-Callins, so enthält das bmArray an dieser Stelle ein Element vom Typ `BEFREPENDS`.

Nun folgen – wiederum beginnend mit einem `SWITCHROLE`-Eintrag – alle Einträge vom Typ `AFTER`. Auch diese enthalten in ihrem union-Feld `mb` einen Zeiger auf den Methodenblock der zugehörigen Rollenmethode. Existieren mehrere Rollenklassen mit AFTER-Callins, so werden die `AFTER`–Einträge für jede Rollenklasse ebenfalls mit einem `SWITCHROLE`–Eintrag eingeleitet.

Der letzte Eintrag des bmArrays besitzt den Typ `END`.

9.6.2 Aufbau des Delegationproxy Opcode

Der Programmcode des Delegationproxy Opcodes besteht im Wesentlichen aus zwei Teilen. Der erste Teil prüft, ob es sich um den ersten Delegationproxy-Opcode des Delegationproxy handelt. Ist dies der Fall, so lässt dies auf die Rückkehr aus einer REPLACE-Methode oder aus einem Basismethodenaufruf schließen. In diesem Fall wird der Rückgabewert in den aktuellen Stackframe kopiert. AFTER-Methoden können diesen Rückgabewert dann als Parameter übergeben bekommen[45]. Außerdem steht der Rückgabewert so zur Verfügung, wenn der Delegationproxy selbst zu seinem Aufrufer zurückkehrt. Das Kopieren des Rückgabewertes in den Stackframe ist erforderlich, da der Stack nach jedem

[44] Siehe dazu Kapitel 7.

[45] Dies ist in OTRun bisher noch nicht implementiert, da kein Attribut existiert, welches dieses Mapping beschreibt. Der ot-Compiler erzeugt zur Implementierung des Rückgabewert-Mappings ein Programmfragment in den jeweiligen Callin Wrappern.

Methodenaufruf, den der Delegationproxy veranlasst hat, wieder in den Zustand versetzt wird, den der Delegationproxy bei seinem Start vorgefunden hat. Dadurch werden auch die Rückgabewerte von BEFORE- und AFTER-Methoden gelöscht, da sie in der Sprache ObjectTeams/Java nicht verwendet werden.

Der zweite Teil kontrolliert den Kontrollfluss des Delegationproxy Opcode und implementiert zugleich die eigentliche Funktionalität des Delegationproxy, deshalb werden wir ihn nun im Detail betrachten. Er besteht aus einer switch-Anweisung, die für jeden Elementtyp des bmArrays eine eigene Behandlungsroutine besitzt. Unterstützt wird diese Komponente durch Programmcode der hier nicht sichtbar ist, weil er im Delegationproxy Bytecode implementiert ist. Zu diesem kehrt dieses Programmfragment zurück. Je nach Rückgabewert des Opcodes baut der Delegationproxy Bytecode dann einen passenden Stackframe auf, um eine bestimmte Methode aufzurufen oder veranlasst die Rückkehr aus dem Delegationproxy zu seinem Aufrufer. Die Details die dabei jeweils zu beachten sind, werden jeweils erwähnt werden.

```
otj_restartProxy:

(0.1)   ti      = frame->curCache->teamInstanceArray[frame->curpos];
(0.2)   otjcT   = CLASS_CB(ti->class)->otjc;
(0.3)   bmArray = otjcT->callinTable[frame->mb->otjm->CallinID].bmArray;

otj_restartSwitch:

(0.4) switch(bmArray[frame->bmPos].state)
{

(1.0)   case(OTJ_PROXY_STATE_SWITCHROLE):
        {
(1.1)     frame->bmPos++;
(1.2)     ri = (Object*) otjLiftTo(bi, ti, bmArray[frame->bmPos].data.rc,
                                                          otjcT, otjcB);
(1.3)     frame->ri = ri;
(1.4)     if(((uintptr_t) ri) & 1) { <call ri.<init>(ti, bi)> }
(1.5)     goto otj_restartSwitch;
        }

(2.0)   case(OTJ_PROXY_STATE_BEFORE):
        {
(2.1)     frame->bmPos++;
(2.2)     <prepare stack>, <prepare call>
(2.3)     return(OTJ_PROXY_CONTROL_CALLMETHOD);
        }

(3.0)   case(OTJ_PROXY_STATE_REPLACE):
        {
(3.1)     *ostack++ = (uintptr_t) frame->ri;
(3.2)     new    = CLASS_CB(frame->ri->class)->method_table[ \
          bmArray[frame->bmPos].data.mb->method_table_index];
(3.3)     int p; ostack += 7;
(3.4)     for(p = 7; p < new->args_count; p++){*ostack++ = frame->lvars[p-6];}
(3.5)     frame->basecallbmPos  = frame->bmPos + 1;
(3.6)     frame->bmPos  = 0;
          <prepare call>
(3.7)     return(OTJ_PROXY_CONTROL_CALLREPLACE);
        }

(4.0)   case(OTJ_PROXY_STATE_AFTERSTART):
        {
(4.1)           frame->bmPos = bmArray[0].data.afterPos;
(4.2)           goto otj_restartSwitch;
```

```
        }

(5.0)   case(OTJ_PROXY_STATE_AFTER):
        {
          frame->bmPos++;
          <prepare stack>, <prepare call>
          return(OTJ_PROXY_CONTROL_CALLMETHOD);
        }

(6.0)   case(OTJ_PROXY_STATE_BEFREPENDS):
        {
(6.1)     if(frame->curpos == 0)
          {
(6.2)       frame->bmPos = 0;
            <prepare stack>
(6.3)       return(OTJ_PROXY_CONTROL_EXECUTEBASE);
          }
(6.4)     frame->curpos--;
(6.5)     frame->bmPos = 1;
(6.6)     goto otj_restartProxy;
         }

(7.0)   case(OTJ_PROXY_STATE_END):
        {
(7.1)     if(frame->curpos == frame->startpos)
          {
(7.2)       frame->curCache->refCount--;
(7.3)       if(!frame->curCache->refCount){<free cache>}
(7.4)       return(OTJ_PROXY_CONTROL_RETURN);
          }
(7.5)     frame->curpos++;
(7.6)     frame->bmPos = 0;
(7.7)     goto otj_restartProxy;
        }
};
```

Jeder Aufruf des Delegationproxy Opcode beginnt damit, den aktuellen Kontext für seine Ausführung zu ermitteln. Dazu wird in (0.1) zunächst die aktuelle Teaminstanz aus dem ActivationCache geladen. Diese befindet sich immer an der Position `frame->curpos`. Eine Veränderung des Inhaltes dieses Feldes hat daher zur Folge, dass der Delegationproxy Opcode seinen Kontext wechselt und entweder zur vorherigen Teaminstanz im ActivationCache zurückkehrt oder zur nächsten Teaminstanz voranschreitet. Mit diesem Feld wird also über alle relevanten aktiven Teaminstanzen iteriert, womit die äußere Reihenfolge der Callin-Methoden-Ausführung gesteuert wird. Die innere Reihenfolge wird analog dazu über das Feld `frame->bmPos` gesteuert und enthält die Position des jeweils aktuell verwendeten Eintrags im `bmArray`. Vom Delegationproxy Intro Opcode wurde es mit dem Wert 1 initialisiert, da der Eintrag an Position 1 im bmArray den Einstieg in die innere Reihenfolge der Callin-Methoden für die jeweilige Teamklasse der aktuellen Teaminstanz markiert. Durch (0.2) und (0.3) wird das zur aktuellen Teaminstanz passende `bmArray` ermittelt, das über die `callinID` und den `ClassBlock` der Teaminstanz zu finden ist.

An der Stelle (0.4) wertet die switch-Anweisung den Eintragstyp des aktuellen `bmArray`-Feldes aus, auf das `frame->bmPos` gerade zeigt.

9.6.2.1 OTJ_PROXY_STATE_SWITCHROLE

Typischerweise startet ein Durchlauf des Delegationproxy Opcodes mit einem Eintrag des Typs `SWITCHROLE`. Das Programmfragment für diesen Typ führt in (1.2) ein Lifting[46] der aktuellen Basisinstanz zu einer Rolleninstanz der Rollenklasse durch, die über den `rc`-Wert im `bmArray`-Eintrag bestimmt ist. Die Funktion `Object* otjLiftTo(...)`, die dazu verwendet wird, liefert dann die geliftete Rolleninstanz zurück. Diese wird im Feld `frame->ri` gespeichert und steht damit allen weiteren BEFORE- und REPLACE-Methoden, die nun folgen werden, zur Verfügung[47]. Wurde die Rolleninstanz neu erzeugt, so muss zunächst noch der Konstruktor <init> für diese Instanz aufgerufen werden. Codiert wird dies durch das unterste Bit des zurückgegebenen Objekt-Zeigers, das in (1.2) ausgewertet wird. Ist ein Konstruktor-Aufruf notwendig, so würde der Delegationproxy Opcode dafür kurz verlassen werden, um diesen Aufruf zu tätigen. Die Rückkehradresse, die normalerweise bei einem Methodenaufruf die aktuelle Adresse des Bytecode-Befehls im Codeblock ist, wird für diesen Aufruf um 1 reduziert, um auf den Befehl vor dem aktuellen Bytecode zu zeigen. Dies hat zur Folge, dass der RETURN-Bytecode, mit dem jede Methode abschließt, nicht zum nächsten Bytecode des aktuellen Codeblocks zurückkehrt, sondern stattdessen der Bytecode des Delegationproxy von dem aus wir gestartet sind, erneut ausgeführt wird. Da wir in (1.1) den Wert von `frame->bmPos` erhöht haben, wertet die switch-Anweisung nun allerdings das nächste Element des bmArrays aus. Dies würde durch die Sprunganweisung in (1.5) auch geschehen, wenn der Rollenkonstruktor nicht aufgerufen worden wäre.

9.6.2.2 OTJ_PROXY_STATE_BEFORE und OTJ_PROXY_STATE_AFTER

Diese beiden Zustände werden auf die gleiche Weise behandelt. Ihre zentrale Aufgabe ist die Vorbereitung eines Stackframes zum Aufruf der BEFORE- bzw. AFTER-Methode. Dazu werden alle Parameter, die von einem Parametermapping für diese Methode vorgegeben sind, aus dem Array `lvars[]` welches die Parameter der Basismethode enthält, auf den Stack kopiert. Liegt kein Parametermapping vor, so werden so viele Parameter kopiert, wie der Methodenblock der Callin-Methode im Feld `args_count` vorgibt[48]. Anschließend wird die Methode vom Delegationproxy Bytecode aufgerufen. Dies erfolgt analog zum Aufruf der Rollenkonstruktormethode, der in 9.6.2.2 beschrieben wurde. Da auch diese

[46] Siehe dazu Kapitel 7.

[47] Das Feld `frame->ri` verhält sich daher wie ein Rollencache. Dies ist sehr effizient, da so weniger lifting-Operationen durchgeführt werden müssen. Das OTJ Reflection API [OTJLD §6.1.(a)] für Rolleninstanzen erlaubt jedoch auch das Löschen einer Rolleninstanz. Durch den Rollencache würde dieser Effekt erst mit Verzögerung – beim Aufruf der ersten AFTER-Methode - Wirkung entfalten. Daher kann es sein, dass der Rollencache in zukünftigen Versionen von OTRun entfernt oder modifiziert werden wird.

[48] `args_count` zählt auch den `this`-Parameter mit. Dieser wird jedoch nicht aus der Basismethode kopiert. Stattdessen wird die geliftete Rolleninstanz auf den Stack gelegt. Statische Callin-Methoden besitzen keinen this-Parameter. Dies ist ein Sonderfall, der berücksichtigt werden sollte. Da dieser jedoch noch nicht implementiert wurde, werden statische Callin-Methoden von OTRun bisher noch nicht unterstützt.

Zustände vor diesem Aufruf `frame->bmPos` erhöhen, wird bei einer Rückkehr aus der Callin-Methode zum Delegationproxy Opcode der nächste Eintrag des bmArrays ausgewertet.

9.6.2.3 OTJ_PROXY_STATE_REPLACE

Der Aufruf einer REPLACE-Methode gestaltet sich etwas aufwändiger als der Aufruf einer BEFORE- oder AFTER-Methode. Die Parameterübergabe muss zunächst einmal berücksichtigen, dass die Signatur von REPLACE-Methoden vom ot-Compiler verändert wurde und nun der Signatur eines ChainingWrappers entspricht. Die eigentlichen Parameter der REPLACE-Methode wurden an das Ende dieser Signatur gehängt. OTRun berücksichtigt dies, indem es in (3.4) den festen Offset 7 verwendet, um nur die eigentlich benötigten Parameter zu übergeben. Dennoch wäre es zu bevorzugen, wenn der ot-Compiler die Signatur der REPLACE-Methoden nicht verändern würde bzw. eine Kopie der REPLACE-Methode mit ihrer Originalsignatur erstellt[49].
Bevor der Aufruf der REPLACE-Methode erfolgt, wird `frame->basecallFrame` mit einem Zeiger auf den aktuellen Stackframe initialisiert. Dieses Feld wird von Basecalls verwendet, um das Ziel des Basecalls bestimmen zu können.

Eine REPLACE-Methode kann über zwei Wege zum Delegationproxy Opcode zurückkehren. Zum einen kann ein Basecall innerhalb der REPLACE-Methode dazu führen, dass der Delegationproxy Opcode erneut aufgerufen wird. Zum anderen wird die Methode nach ihrer Abarbeitung wie jede Methode mit RETURN zu ihrem Aufrufer zurückkehren.

Die Form der Rückkehr entscheidet auch darüber, wie sich der weitere Programmverlauf im Delegationproxy Opcode entwickelt. Die Rückkehr aus einem Basecall wird dazu führen, dass die Abarbeitung des bmArrays an der nächsten Position fortgesetzt wird, so dass weitere BEFORE- und REPLACE-Methoden folgen können oder tatsächlich die Basismethode aufgerufen wird. Dazu wird diese Position im Feld `frame->basecallbmPos` des Stackframes gespeichert. Kehrt die REPLACE-Methode jedoch mit einem RETURN zurück, so geht der Delegationproxy davon aus, dass sie entweder alle noch ausstehenden BEFORE- und REPLACE-Methoden durch den Aufruf eines Basecalls abgearbeitet hat, ebenso wie alle AFTER-Methoden der Teaminstanzen die bis zum Ende des ActivationCache noch erreicht werden würden – oder aber auf deren Abarbeitung gezielt verzichten wollte, indem sie keinen Basecall-Aufruf durchführt. Deshalb wird `frame->bmPos` für diesen Fall auf 0 gesetzt, bevor die REPLACE-Methode aufgerufen wird, so dass der Delegationproxy Opcode nach der Rückkehr aus der REPLACE-Methode mit der Abarbeitung der AFTER-Callins für die aktuelle

[49] Die veränderte Signatur erschwert auch das Callin Deployment, da in den Attributen, die Callins beschreiben, ein Verweis auf die originale Signatur zu finden ist. Da jedoch keine Methode mit dieser Signatur in der entsprechenden Rollenklasse existiert, scheitert die Suche nach ihr. Im Moment wird die Signatur daher bei der Suche nach REPLACE-Methoden ignoriert. Dies hat zur Folge, dass Overloading für REPLACE-Methoden von OTRun im Moment nicht unterstützt werden kann.

Teaminstanz beginnt. Die Durchführung eines Basecalls würde dagegen über den ersten Delegationproxy Opcode zum Delegationproxy zurückkehren, was dazu führt, das `frame->bmpos` den Inhalt von `frame->basecallbmPos` erhält. Zugleich dient dieses Feld auch dazu, zu erkennen, ob die Rückkehr aus einem Basismethodenaufruf oder ein Basecall vorliegt. Es enthält 0 nach der Rückkehr aus dem Basismethodenaufruf, ansonsten die letzte Position für das bmArray.

Neben der Signaturveränderung bezüglich der Parameter einer REPLACE-Methode verändert der ot-Compiler auch deren Rückgabewert-Übergabe, indem er für diese *Boxing* einführt. Der Rückgabewert wird so zunächst in ein Objekt verpackt, um dann als Objektreferenz zurückgegeben werden zu können. Im OTRE würde der Callin Wrapper nun nach dem Aufruf der REPLACE-Methode ein *Unboxing* durchführen, den Wert also aus dem übergebenen Objekt auslesen und so zurückgeben, wie er vom Aufrufer erwartet wird. Auf der Ebene einer virtuellen Maschine ist dieser *Unboxing*-Prozess nur sehr aufwändig zu implementieren. Zusätzlich besitzt er keinen praktischen Nutzen, kostet aber Rechenzeit. Daher wurde darauf verzichtet, ein *Unboxing* zu implementieren. Stattdessen wird eine Anpassung des ot-Compilers befürwortet, die REPLACE-Methoden zusätzlich zu der für das OTRE modifizierten Form auch in ihrer ursprünglichen Form bewahrt. Aus diesem Grund funktionieren REPLACE-Methoden in der momentanen Version von OTRun nur dann korrekt, wenn auch die Basismethode *Boxing* für den Rückgabewert verwendet (also z.B. die Signatur *Integer myBase()* besitzt, anstelle von *int myBase()*).

9.6.2.4 OTJ_PROXY_STATE_BEFREPENDS

Wird dieser Zustand erreicht, so wurden für die aktuelle Teaminstanz alle BEFORE- und alle REPLACE-Methoden ausgeführt. Der äußeren Ordnung folgend, müssten nun die BEFORE- und REPLACE-Methoden der nächsten Teaminstanz aus dem ActivationCache ausgeführt werden. Es kann allerdings auch sein, dass bereits das Ende des ActivationCaches erreicht wurde. In diesem Fall müsste nun die Basismethode aufgerufen werden, damit im Anschluss daran die Abarbeitung aller AFTER-Methoden erfolgen kann. Genau dies wird von `BEFREPENDS` in (6.1) überprüft. Das Ende des ActivationCaches ist erreicht, wenn `frame->curpos` den Wert 0 erreicht hat. In diesem Fall wird der Aufruf der Basismethode vorbereitet, indem für ihren Aufruf ein neuer Stackframe angelegt wird und alle Parameter aus dem Array kopiert werden, das über `lvars` zu erreichen ist. Dies können auch die Parameter sein, die eine REPLACE-Methode ihrem Basecall übergeben hat, da `lvars` immer auf die Parameter zeigt, die zuletzt auf dem Stack abgelegt wurden. Nach der Rückkehr aus dem Basisaufruf soll mit der Ausführung aller AFTER-Methoden fortgefahren werden, daher wird nun `frame->bmPos` auf 0 gesetzt (6.2) und anschließend der Basisaufruf durchgeführt (6.3). Dies führt dazu, dass der Delegationproxy Opcode beim nächsten Aufruf einen Eintrag vom Typ `afterstart` im bmArray vorfindet. Dieser löst dann die Ausführung aller AFTER-Methoden aus.

Wurde das Ende des Aktivierungsarrays jedoch noch nicht erreicht, so wird das Feld `frame->bmPos` stattdessen auf 1 gesetzt und das Feld `frame->curpos`

dekrementiert. Anschließend wird zurück zum Start des Delegationproxy Opcodes gesprungen. So kann eine neue Teaminstanz mit Hilfe des inkrementierten Feldes geladen und das bmArray durch das bmArray der neuen Teaminstanz-Klasse ausgetauscht werden. An der bmArray-Position 1 startet nun für diese Teaminstanz die Ausführung ihrer BEFORE- und REPLACE-Methoden.

9.6.2.5 `OTJ_PROXY_STATE_AFTERSTART`

Dieser Zustand speichert die bmArray-Position ab der die AFTER-Methoden beginnen. Führt der Delegationproxy das dazugehörige Programmfragment aus, so wird diese Position lediglich `frame->bmPos` zugewiesen und die switch-Anweisung des Delegationproxy Opcodes erneut ausgeführt.

9.6.2.6 `OTJ_PROXY_STATE_END`

Mit `OTJ_PROXY_STATE_END` ist schließlich das Ende des aktuellen bmArrays erreicht. Für die aktuelle Teaminstanz wurden alle BEFORE-, REPLACE- und AFTER-Methoden und der Basecall ausgeführt, sofern einige davon nicht durch einen fehlenden Basecall in einer REPLACE-Methode unterdrückt wurden. In (7.1) wird daher nun überprüft, ob wir wieder die Startposition im ActivationCache erreicht haben, mit der wir die Ausführung des Delegationproxys begonnen hatten. Ist das der Fall, so dekrementieren wir den Referenzzähler des ActivationCache und überprüfen, ob dieser dadurch nun 0 erreicht hat. Ist dies der Fall, so geben wir ihn frei und kehren anschließend zum Aufrufer des Delegationproxys zurück. Vorher kopieren wir allerdings noch den Rückgabewert auf den Stack, damit der Aufrufer diesen dort vorfinden kann.
Hatten wir in (7.1) dagegen festgestellt, das wir die Startposition des ActivationArrays noch nicht erreicht haben, so inkrementieren wir die aktuelle Position in `frame->curpos`, setzen `frame->bmPos` auf 0, um bei dem nächsten Durchlauf des Delegationproxy Opcodes mit der Ausführung der AFTER-Methoden der nächsten Teaminstanz fortzufahren und springen anschließend wieder an den Anfang des Delegationproxy Opcodes um diese zu beginnen.

Um das detaillierte Bild, dass wir nun von der Funktionsweise des Delegationproxy gewinnen konnten, abzurunden, soll nun noch ergänzend die Umsetzung von Basecalls beschrieben werden, bevor wir den Delegationproxy-Ansatz mit dem Ansatz des OTRE abschließend vergleichen.

9.7 Die Arbeitsweise von `OTJ_OP_BASECALL`

Basecalls werden beim Laden von Rollenklassen mit Hilfe des Infrastructure Weavings auf den `OTJ_OP_BASECALL` Opcode abgebildet. Bei ihrer Ausführung sollen sie die jeweils passende Basismethode ausführen. Welche das ist, erfahren sie aus dem Feld `frame->basecallframe`, das auf den Stackframe desjenigen zeigt, der die REPLACE-Methode aufgerufen hat. Dies war entweder der

Delegationproxy der Basismethode, oder aber ein Delegationproxy, der wiederum über einen Basecall aufgerufen wurde. Da der Basecall Opcode selbst nicht zwangsweise in der REPLACE-Methode vorkommt, sondern auch in einer super- oder tsuper-Methode der REPLACE-Methode vorkommen kann, lässt sich aus dem aktuellen Stackframe nicht unmittelbar schließen, dass der vorangegangene Stackframe auch der Stackframe des Aufrufers der REPLACE-Methode gewesen sein muss. Deshalb ist ein Basecall Opcode auf das Feld `frame->basecallframe` dafür angewiesen. Jedes Mal wenn die REPLACE-Methode ihre eigene Superversion aufruft – und auch bei allen nachfolgenden Superaufrufen dieser Art, wird der Inhalt von `frame->basecallframe` aus dem alten Stackframe des Aufrufenden in den Stackframe des Aufgerufenen kopiert. Dazu wurde der Bytecode `invokespecial` entsprechend angepasst. Zumindest für Supermethoden, die einen Basecall enthalten, ist nun sichergestellt, das sie auf den Stackframe der Basismethode zugreifen können, indem sich auch der Methodenblock befindet, der auf den Programmcode der Basismethode verweist.
Für tsuper wurde dieses Verhalten bisher jedoch noch nicht implementiert, da tsuper-Aufrufe nicht den Bytecode `invokespecial`, sondern den Bytecode `invokevirtual` verwenden. In einer zukünftigen Version von OTRun sollen diese tsuper-Aufrufe mit InfrastructureWeaving daher auf einen eigenen Opcode abgebildet werden, der analog zum angepassten `invokespecial` arbeitet.

9.8 Zusammenfassung

In diesem Kapitel wurde der Delegationproxy vorgestellt. Er besteht aus einer Sequenz aus drei Bytecodes, die sich am Beginn von jeder Methode befindet, für die Callins existieren. Er übernimmt als Proxy die Aufgabe, stellvertretend für die aufgerufene Basismethode zu agieren und den Programmfluss je nach aktivierten Teaminstanzen zu verschiedenen Callin-Methoden zu delegieren und dabei auch die Basismethode bei Bedarf aufzurufen. Die innere Ausführungsreihenfolge wird durch eine Datenstruktur gesteuert und nicht statisch in den Delegationproxy hineingewebt. Daher ist er flexibel und benötigt im Gegensatz zum ChainingWrapper des OTRE kein Weaving. Durch die Verwendung eines TeamActivationCache braucht der Delegationproxy nur in seltenen Fällen ein neues Array bei einem Aufruf zu allokieren. Der InitialWrapper des OTRE muss dies dagegen bei jedem Aufruf tun. Zusätzlich ist das verwendete Array mit Teaminstanzen, welches vom OTRE verwendet wird, nicht optimal besetzt, enthält Lücken und Teaminstanzen, die für die aktuelle Basismethode kein Callin enthalten. Der TeamActivationCache des Delegationproxy ist dagegen optimal besetzt und enthält keine nicht benötigten bzw. inaktiven Teaminstanzen. Die Ausführung eines Callins mit dem OTRE führt zu einer langen Kette von teuren Java-Methodenaufrufen. Der Delegationproxy vereint dagegen die Funktionalität von Lifting, Teamaktivierung und Callin-Ausführung in einer einzelnen Superinstruction. So kann er den Overhead einsparen, der dem OTRE aus der Kommunikation zwischen diesen einzelnen Komponenten entsteht. Zugleich wurde jede dieser Komponenten im Delegationproxy optimiert. So verwendet das Lifting nun einen effizienten Lookup-Mechanismus, der keine Methodenaufrufe erfordert. Außerdem setzt es einen optimierten Vergleich ein, um auf das Vorliegen einer WrongRoleException zu testen. Sind keine Teaminstanzen aktiviert, so führt der

Delegationproxy keine weiteren Methodenaufrufe aus, sondern kann direkt zum Code der Basismethode springen. Für die Ausführung von REPLACE-Callins und Basecalls benötigt der Delegationproxy im Gegensatz zum OTRE keine Rollenmethoden mit einer speziell angepassten Signatur. Damit kann abschließend festgehalten werden, dass mit dem Delegationproxy nun ein delegationsbasierter Mechanismus implementiert wurde, der gegenüber dem Callin-Ausführungs-mechanismus des OTRE an vielen Stellen optimiert wurde. Wie sich diese Optimierungen auf die Laufzeit der Callin-Ausführung auswirken, soll im nächsten Kapitel mit Hilfe von Benchmarks gezeigt werden.

Kapitel 10

OTRun outruns...

- Zusammenfassung und Benchmarks

10.1 Zusammenfassung

Fassen wir die Ergebnisse der letzten fünf Kapitel noch einmal kurz zusammen.

Es wurde ein Verfahren mit dem Namen Infrastructure Weaving entwickelt und vorgestellt, welches es erlaubt, Java Methoden auf Bytecodes abbilden zu können, ohne dabei die virtuelle Maschine nach außen hin um diese neuen Bytecodes erweitern zu müssen. Dieses Verfahren wurde dazu verwendet, Methoden die vom ot-Compiler und vom OTRE verwendet werden, auf neue Bytecodes abzubilden. Dadurch konnte die Funktionalität des OTRE transparent durch die neue Laufzeitumgebung OTRun ersetzt werden, ohne dass der ot-Compiler dazu verändert zu werden brauchte.

Mit OTRun wurde eine neue Laufzeitumgebung für ObjectTeams/Java geschaffen, die direkt in eine Java Virtual Machine integriert wurde. Damit können ObjectTeams/Java Programme direkt von einer virtuellen Maschine ausgeführt werden.

Die Algorithmen für das Lifting, die Teamaktivierung und die Callin-Ausführung wurden durch neue delegationsbasierte Algorithmen ersetzt. Dadurch ist kein explizites Weben durch Bytecode-Instrumentierung mehr notwendig. Sämtliche _OT$liftTo$... Methoden können auf einen einzelnen neuen generischen Bytecode liftTo abgebildet werden.

Durch die Verwendung von *Shadowproxies* ist die Ladereihenfolge von Team- und Basisklassen nun beliebig. Dies erhöht die Flexibilität von ObjectTeams/Java dahingehend, dass der Benutzer nun zur Laufzeit eines Programms eine von ihm gewählte Teamklasse dazu laden kann, um das Verhalten eines Programms zur Laufzeit zu verändern.

Die Einführung einer Liste in jeder Klasse, die all ihre Subklassen speichert, ermöglicht zusammen mit der Laufzeit-Smartliftinganalyse und den Algorithmen zur Gewährleistung der Ladereihenfolgeunabhängigkeit die Option auf zur Laufzeit quantifizierbare playedBy- und Callinbindungen mit Hilfe von Querry-Methoden.

Für den neuen Lifting-Algorithmus wurde ein alternativer Speicher-verwaltungsansatz geschaffen. Auf große, viel Speicherplatz verbrauchende Hashtabellen und auf die Verwendung von *weak References* kann dadurch verzichtet werden. Auch die Hashtabelle, die bisher für die Teamaktivierung in jeder Teaminstanz angelegt wurde, ist nun durch eine weniger Speicherplatz verbrauchende einfach verkettete Liste ersetzt worden.

Die Algorithmen für das Lifting, die Teamaktivierung und die Callinausführung wurden algorithmisch optimiert und in Form von Superinstructions implementiert. Aus beidem sollte der neuen Laufzeitumgebung ein signifikanter Geschwindigkeitsvorteil erwachsen. Wie hoch dieser ist, soll im folgenden Abschnitt festgestellt werden.

10.2 Benchmarks

Es soll nun gemessen werden, ob die Mechanismen, die in dieser Arbeit vorgestellt wurden, einen vorteilhaften Effekt auf die Laufzeit von ObjectTeams/Java Programmen haben können. Es wurde die Zeit für 11.000.000 Methodenaufrufe gemessen, die auf unterschiedliche Weise generiert wurden. Zunächst wurde die Zeit gemessen, die der Aufruf unveränderter Java-Methoden benötigt, um später den Overhead bestimmen zu können, den die aspektorientierten Mechanismen von ObjectTeams/Java im Vergleich zu Java verursachen. Anschließend wurden 1.000.000 Aufrufe auf einer Basismethode gemessen, auf die 10 aktivierte Teaminstanzen ein BEFORE-Callin setzen. Daraus ergeben sich 1.000.000 Aufrufe der Basismethode und 10.000.000 Aufrufe der BEFORE-Methode, insgesamt also ebenfalls 11.000.000 Aufrufe. OTRun kann bisher nur mit einer JamVM betrieben werden, die den „directthreaded“ Interpreter verwendet. Das OTRE konnte mit dieser Interpreter-Variante jedoch leider nicht fehlerfrei ausgeführt werden, daher wurde das OTRE auf einer JamVM mit „inlined“ Interpreter und aktiviertem Stackcaching betrieben. Mit diesen beiden Optimierungen ist die JamVM bedeutend schneller als in einer directthreaded-Variante ohne StackCaching. Daher sind die Wert in Abbildung 11 nicht direkt miteinander vergleichbar. Dennoch konnte mit den Optimierungen, die im Rahmen dieser Arbeit vorgenommen wurden, eine erhebliche Verbesserung der Laufzeit bei Callin-Aufrufen erzielt werden. So brauchte die optimierte JamVM mit dem OTRE für die 11.000.000 Aufrufe auf einem Intel Pentium D mit 3.4Ghz 47430 ms. Die neue Erweiterung

OTRun benötigte dagegen mit einer weniger optimierten JamVM nur 2258 ms. Damit wurde die Laufzeit im Vergleich zum OTRE um 95% reduziert, so dass die Kombination JamVM/OTRun nun in diesem Benchmark 21mal schneller ist, als die Kombination JamVM/OTRE. Eine weitere Messung wurde mit einer OTRun-Variante durchgeführt, bei der eine Lock/Unlock-Kombination entfernt wurde, die im Lifting-Opcode verwendet wird. Damit konnte die Laufzeit von OTRun experimentell um weitere 60% verbessert werden. Die experimentelle Implementierung eines Spinlocks anstelle dieser Lock-Funktionen konnte diesen Geschwindigkeitsgewinn weitestgehend bewahren[50]. In dieser Variante wäre OTRun um den Faktor 33 schneller als das OTRE.

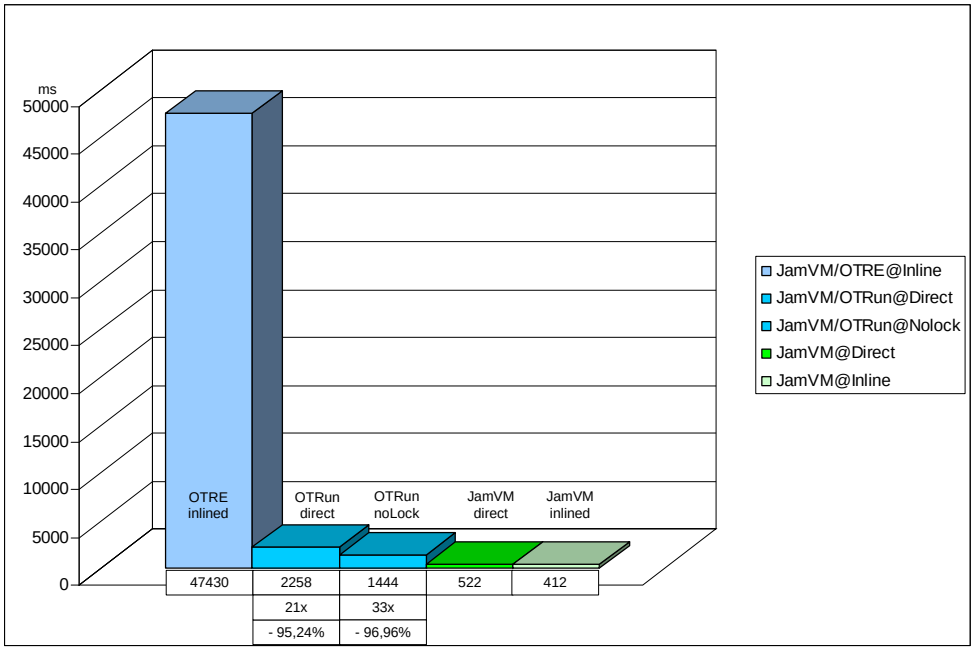


Abbildung 11: before&base Benchmark OTRE / OTRun / JamVM

Vergleicht man die Laufzeit von OTRun mit der Laufzeit für Java-Methoden, ist festzustellen, dass der Overhead des Delegationproxy im Vergleich zu einem normalen Methoden-Aufruf nur noch um den Faktor 5 geringer ist. Die experimentelle Version ohne die Lock-Funktion war sogar nur noch 3,5mal langsamer als die Ausführung von Java-Methoden mit einer optimierten JamVM.

Die Teamaktivierung, deren Optimierung nicht im Fokus dieser Arbeit stand, konnte in ihrer Laufzeit nur geringfügig verbessert werden. Da sie mehrere Lock-Aufrufe enthält, kann jedoch womöglich auch an dieser Stelle noch optimiert werden. So benötigten je 100.000 Aktivierungs- und Deaktivierungsaufrufe mit OTRun 135 ms gegenüber 175 ms des OTRE, so dass hier nur eine Optimierung um 22% erzielt wurde.

Schließlich wurde auch noch eine Vergleichsmessung mit der hochoptimierenden VM Hotspot Client und dem OTRE durchgeführt. Es wurde die gleiche

[50] Die Implementierung des Spinlocks arbeitete jedoch noch nicht fehlerfrei und wurde daher vorläufig wieder entfernt.

Messfunktion mit 11.000.000 Aufrufen verwendet[51]. Wurde Hotspot lediglich mit Interpreter betrieben (Option –Xint), so benötigte sie auf einem 2.4Ghz Intel Core Duo dafür 49547 ms. Bei aktiviertem JIT-Compiler (ohne Optionen) verbesserte sich das Ergebnis von Hotspot auf 6156 ms. JamVM/OTRun benötigte 4409 ms und konnte damit in diesem Benchmark erstaunlicherweise sogar eine Virtuelle Maschine mit JIT-Compiler übertreffen, wenn auch nur um 28%. Im Vergleich zur interpretierenden Variante von Hotspot erreichte sie eine Verbesserung um sehr gute 91%.

[51] D.h. 10 aktivierte Teaminstanzen mit je einem BEFORE-Callin.

Kapitel 11

OTRun, quo vadis?

- Ausblick und Schluss

11.1 offengebliebene Aufgaben

Im Rahmen dieser Arbeit konnte nur ein Bruchteil dessen umgesetzt werden, was theoretisch umsetzbar ist, um die Kompatibilität zwischen OTRun und ObjectTeams/Java zu verbessern. Allerdings handelt es sich bei vielen dieser offengebliebenen Punkten um triviale Aufgaben, die in wenigen Stunden oder Tagen implementiert werden können und für die es nicht erforderlich ist umfangreiche neue Datenstrukturen und Algorithmen zu entwickeln. Der Fokus dieser Arbeit lag auch nicht darauf, eine vollständige Abdeckung der ObjectTeams/Java-Sprachspezifikation in einer virtuellen Maschine zu erreichen. Mit den grundlegenden Mechanismen Lifting, Teamaktivierung und Callin-Ausführung, sowie umfangreichen, nützlichen Datenstrukturen, die beim Laden von Klassen angelegt werden, wurde aber ein Rahmen geschaffen, in dem vielerlei Erweiterungen zukünftig ihren Platz finden können.

So fehlt z.B. die Implementierung von _OT$saveActivationState und _OT$restoreActivationState, ohne die kein within()-Block verwendet werden kann. Um sie zu implementieren würde es allerdings ausreichen, zu testen, ob die jeweilige Teaminstanz implizit oder explizit aktiviert ist, bzw. ob sie überhaupt aktiviert ist und diesen Zustand dann als Integer zurückzugeben. Ein restore-

Opcode würde diesen Integer als Parameter erhalten und kann dann den passenden Zustand wieder herstellen. Trivial.

Auch eine Reihe weiterer Methoden wird bisher noch nicht unterstüzt, wie z.B. die _OT$create$-Methoden, die für externalisierte Rolleninstanzen benötigt werden - und für Rolleninstanz-Felder und Variablen. IsActivate() ist eine weitere Methode, die einfach zu implementieren wäre. Auch die Methoden der Rollen-Reflection-API werden bisher noch nicht unterstüzt, obwohl das Lifting bereits so konzipiert wurde, dass es diese Methoden einfach unterstützen kann. So werden z.B. alle `riFrames`[52], welche Rolleninstanzen enthalten, in Arrays gespeichert, die einer Teaminstanz über die Liste ihrer MemoryManager, die diese riFrames verwalten, zugänglich sind. Soll also herausgefunden werden, welche Rolleninstanzen bereits existieren, so reicht es, diese Arrays nach Referenzen zu durchsuchen, die nicht NULL sind.

Etwas schwieriger wäre die Implementierung der globalen Teamaktivierung. Für sie wurde ein Konzept entwickelt, das jedoch noch nicht umgesetzt wurde. Dies sieht es vor eine separate globale Teamaktivierungsliste zu verwalten, die alle global aktivierten Teaminstanzen enthält. Sie dient dazu, neu erzeugte Threads mit allen global aktivierten Teaminstanzen zu versehen, wenn ihre Environment-Variable initialisiert wird. Threads, die bereits bestehen, können über die verkettete Liste der Environment-Blöcke schnell gefunden werden. Für jeden so gefundenen Thread würde die globale Teamaktivierung dann eine externe Aktivierung mit activate(Thread t) durchführen.

In Kapitel 7 wurden Anpassungen an den Garbage Collector erwähnt. Auch sie wurden bisher noch nicht implementiert.

Umfangreicher sind die Änderungen, die für die volle Unterstützung von impliziter Vererbung getätigt werden müssen. Es wäre ein Konzept zu entwickeln wie die tcTables[53] die nun aus zwei verschiedenen Quellen geerbt werden könnten, miteinander zu kombinieren sind. Das Hauptproblem besteht hier darin, dass die Offsets, die für Einträge in der tcTable vergeben werden, für alle Subklassen identisch sein müssen. Das Erben aus zwei Quellen kann nun dazu führen, dass in jeder der beiden tcTables der gleiche Index von einer jeweils anderen Teamklasse belegt ist. Dieses Problem könnte jedoch dadurch gelöst werden, dass bei dem Zugriff auf die tcTable immer auch geprüft wird, ob die Klasse, die man dort gefunden hat, auch die Klasse ist, die man erwartet hat. Ist das nicht der Fall könnte man die tcTable linear nach einem passenden Eintrag durchsuchen. Da eine solche Suche nur in seltenen Fällen ausgelöst werden dürfte, wäre der Geschwindigkeitsverlust für die IF-Abfrage innerhalb des liftTo-Opcodes vernachlässigbar. Auch für die Unterstützung von tsuper-self-Aufrufen in REPLACE-Methoden müsste eine eigene Lösung entwickelt werden, da tsuper-self nicht durch `invokespecial` sodern durch `invokevirtual` realisiert ist.

[52] Siehe Kapitel 7
[53] Siehe Kapitel 6 und 7

Bisher wird auch das CopySrc-Attribut noch nicht unterstützt. Mit dessen Hilfe könnten alle Methodenblöcke gefunden werden, die implizit geerbt wurden. Innerhalb der virtuellen Maschine wäre es trivial deren Codeblock freizugeben und stattdessen einen Zeiger auf den Codeblock der Originalmethode zu speichern. Damit könnte etwas Speicherplatz eingespart werden.

Callins können bisher nur auf eine sehr eingegrenzte Menge von Methodenvarianten gesetzt werden. Statische Methoden werden z.B. noch nicht unterstützt. Ebenso wenig Methoden die implizit geerbt wurden. Diese besitzen einen neuen Methodenblock, der aufgrund der Signaturveränderung, die der ot-Compiler bei impliziter Vererbung vornimmt, auch nicht einfach durch den Methodenblock ihrer impliziten Superklasse ersetzt werden können. Wäre das jedoch möglich, so wäre die Unterstützung von Callins in diesem Fall trivial. Sobald der Methodenblock aus der impliziten Superkasse verändert werden würde, würden sich diese Auswirkungen auch auf die implizit erbende Klasse auswirken.

Für die Unterstützung von Confinement müssten die beim Classloading verwendeten Vererbungshierarchie-Analysealgorithmen daraufhin geprüft werden, ob nun Sonderfälle auftreten können, die zu fehlerhaften Ergebnissen führen.

Etwas aufwändiger wäre auch die Unterstützung von Callout-get- und set-Methoden. Diese werden bisher vom OTRE in die Basisklassen gewebt.

11.2 Kompatibilität

Um die Kompatibilität von OTRun zu ObjectTeams/Java zu erhöhen, wäre es hilfreich, wenn einige Veränderungen am ot-Compiler vorgenommen werden würden.

So wäre ein neues, separates Attribut für Rollenprecedences hilfreich, um Eindeutigkeit zu bewahren. Bisher wird für Rollen- und Callinprecedence das gleiche Attribut verwendet. Da bei einem doppelten Vorkommen des Attributs, beide mitunter widersprüchlichen Inhalt besitzen können, wurde die Entscheidung getroffen Callinprecedence vorläufig nicht zu unterstützen.

Für das Parametermapping wäre es hilfreich, wenn der ot-Compiler auch die Größe des Parameters auf dem Stack im Attribut ablegt. Dies wäre notwendig, um auch 8 Byte große Parameter unterstützen zu können. Weil dies noch nicht der Fall ist, wird Parametermapping bisher noch nicht unterstüzt. Ausserdem könnte das Parametermapping-Attribut um die Behandlung des Rückgabewert-Parametermappings für AFTER-Callins erweitert werden.

Problematisch ist die Signatur-Modifizierung von REPLACE-Methoden und von Basecalls. Diese mag für das OTRE hilfreich sein, erschwert die Implementierung von REPLACE-Callins und Basecalls in einer virtuellen Maschine erheblich. Das gleiche gilt für die Entscheidung Boxing zur Parameterübergabe zu verwenden. Für OTRun wäre es von Vorteil, wenn keine Signaturen oder Rückgabewerte vom ot-Compiler verändert werden würden.

Ein weiteres Problem stellen Rollenkonstruktoren dar. Diese sind leider mit Funktionalität überladen, die zur Initialisierung im OTRE benötigt werden. Besonders problematisch ist hierbei der Aufruf von _OT$addRole. Dies ist eine Methode die vom OTRE in Basisklassen gewebt wurde. Bei der Ausführung mit OTRun existiert diese Methode jedoch nicht. OTRun überprüft nun jede „MethodNotUnderstood“-Ausnahme, ob ursächlich addRole() aufgerufen werden sollte. In diesem Fall wird der Aufruf ignoriert. Um dieses Problem zu lösen, wäre es hilfreich, wenn der ot-Compiler solche Infrastrukturaufrufe in eine eigene Methode verlagern würde. Dann könnte OTRun sie mit Hilfe von Infrastructure Weaving deaktivieren. Werden dagegen Programmfragmente vom Entwickler einer Applikation mit Prgrammfragmenten des ot-Compilers so vermischt, wie es bei den Rollenkonstruktoren der Fall ist, dann ist OTRun dazu gezwungen, dieses ganzen Block ausführen zu müssen.

Zur Unterstützung von Guards wäre es hilfreich, wenn diese eine schematisch relativ konstant aufgebaute Signatur besitzen würden. Alternativ dazu wäre auch ein Parametermapping für Guards eine gute Lösung. Auch ein Attribut, welches den Typ eines Guards spezifiziert, sowie das Ziel, welches überwacht werden soll, wäre notwendig, um Guards mit OTRun unterstützen zu können.

11.3 Verwandte Arbeiten

Der Ausblick auf andere Arbeiten fällt übersichtlich aus, direkt vergleichbare Ansätze sind kaum zu finden. Es existieren jedoch einige Arbeiten, die dieser vorausgehen bzw. sich um eine ähnliche Aufgabenstellung bemüht haben.

In [Hä06] wurde ebenfalls ein Versuch unternommen, ObjectTeams/Java zu optimieren. Dies geschah jedoch auf der Ebene des in Java geschriebenen OTRE und erbrachte nur einen sehr geringen Optimierungsgewinn, der im Mittel bei ca. 7% lag. Vergleicht man diesen Gewinn mit den Resultaten, die in dieser Arbeit bezüglich der Optimierung erzielt wurden, so liegt der Schluss nahe, dass es vielversprechender ist, die Unzulänglichkeiten einer individuell angepassten virtuellen Maschine in Kauf zu nehmen und an deren Überwindung zu arbeiten, als darauf zu hoffen, dass sich die Laufzeitumgebung auf Java-Ebene signifikant optimieren lässt. Andererseits ist es natürlich möglich, das eine zu tun, ohne das andere zu lassen. Möglicherweise wäre es daher auch hilfreich, einige der Algorithmen, die im Rahmen dieser Arbeit entstanden sind, in die Java-Welt zurück zu übertragen, damit sie auch dort einen Beitrag zur Optimierung von ObjectTeams/Java leisten können.

Die Arbeiten von [Flüh06] und von [Fr09] versuchen beide, das OTRE mit einem Laufzeit-Webemechanismus auszustatten, um die Ladereihenfolgeabhängigkeit zwischen Basis- und Teamklassen, die bisher besteht, beseitigen zu können. Auch diese Lösungsansätze bewegen sich ausschließlich auf der Java-Ebene. Der ausgereiftere Ansatz in der Arbeit von [Fr09] erkauft diesen Zugewinn an Flexibilität aber vermutlich mit zusätzlichem Laufzeit-Overhead, da die

vorgeschlagene Lösung es vorsieht, alle Parameter, die für ein Callin benötigt werden, mit Hilfe eines *Boxing*-Mechanismus zu übergeben. Für jeden Methodenaufruf müssten so zusätzlich eine Vielzahl von Objekten – einschließlich eines Arrays zur Speicherung der Referenzen auf diese Objekte – allokiert werden. Die Arbeit enthält jedoch keine Benchmarks, die diese Vermutung bestätigen oder widerlegen würden.

In [Hu10] wird ebenfalls ein Ansatz beschrieben, ObjectTeams/Java mit Hilfe einer angepassten virtuellen Maschine zu optimieren. Dabei wird allerdings eine andere Philosophie verfolgt, als in dieser Arbeit. Anstatt das gesamte OTRE auszutauschen und durch eine neue Laufzeitumgebung wie OTRun zu ersetzen, wird dort das bestehende OTRE behutsam um optimierte, VM-gestützte Mechanismen erweitert. Dies erlaubt es, die volle Sprachkompatibilität zu bewahren, kann dadurch aber natürlich auch nur eine geringere Optimierungsleistung erzielen, als eine vollständig in eine virtuelle Maschine integrierte Lösung. Zudem gewinnt ObjectTeams/Java dadurch nicht an zusätzlicher Flexibilität, beispielsweise um das Laden von Teamklassen zur Laufzeit eines Programms zu unterstützen. Schließlich ist die Kompatibilität des OTRE zur JamVM auch sehr eingeschränkt, so dass die Lösung in [Hu10] voraussetzt, dass die JamVM über ein Plugin für ObjectTeams/Equinox angesprochen wird. Im Gegensatz dazu besitzt der in dieser Arbeit vorgestellte Ansatz keinerlei Abhängigkeiten mehr zu externen Werkzeugen und kann ObjectTeams/Java Programme direkt ausführen. Kann die Abhängigkeit zwischen OTRE, JamVM und ObjectTeams/Equinox nicht überwunden werden, so wäre die in dieser Arbeit vorgestellte Lösung im Bereich von Embedded Systemen nicht nur aufgrund der höheren Performance zu bevorzugen. Auch dass die Laufzeitumgebung in diesem Fall nur aus JamVM/OTRun besteht und nicht zusätzlich das umfangreiche ObjectTeams/Equinox-Framework geladen werden braucht, stellt einen entscheidenden Vorteil dar. In [Hu10] wird als eine mögliche Optimierung ein Cache für Rolleninstanzen vorgestellt. Da wir in Kapitel 10 gesehen haben, dass mehr als 50% der Laufzeit des DelegationProxy für den Lock-Mechanismus beim Lifting aufgebraucht wird, wäre ein solcher Mechanismus auch für OTRun von Vorteil. Denn OTRun verwendet bisher nur einen minimalen Rollencache, der es lediglich direkt aufeinander folgenden Callin-Methoden einer Rollenklasse erlaubt, die entsprechende Rolleninstanz ohne erneutes Lifting zu nutzen. Ein DelegationProxy, der geliftete Rolleninstanzen auch über den Aufruf eines DelegationProxy hinweg speichern und erneut nutzen könnte, würde OTRun erheblich beschleunigen.

Der delegationsbasierte Ansatz von [HS07] wurde bereits in Kapitel 4 diskutiert. Er war die Inspiration für diese Arbeit, auch wenn letztendlich ein gänzlich anderes Konzept als in der Arbeit von [HS07] gewählt worden ist, um den Delegationproxy zu realisieren, da dieses den spezifischen Rahmenbedingungen von ObjectTeams/Java näher kommt als die Delegationsmechanismen in [HS07].

Als Verfahren zum effizienten Laufzeitweben wird in [Bo06] das Envelope-Weben vorgestellt. Dieses Verfahren sieht es vor, alle Codeblöcke von Methoden durch einen Wrapper zu ersetzen, der dann zu dem eigentlichen Codeblock springt. Durch den Austausch oder die Modifikation dieses Wrappers braucht dann nur

noch an der Stelle des Aufgerufenen gewebt zu werden, aber nicht mehr an all den Stellen, von denen aus die Methode aufgerufen wird. Der Effekt ist daher mit der Funktion des Delegationproxy vergleichbar: Das Ziel eines Methodenaufrufs wird verändert, nicht die Quelle des Aufrufs. Allerdings bietet der Delegationproxy den Vorteil, dass er keinerlei Laufzeitminderung für Methoden bewirkt, für die keine Callin-Bindung existiert.

11.4 Schluss

Anwendungen der Informationstechnologie durchdringen unser Leben in immer größerem Umfang und durch die kontinuierliche Integration der Informationstechnologie in unseren Alltag gelangen immer mehr Menschen auch ganz unmittelbar in den Kontakt mit ihr. So lassen sich eingebettete Systeme nicht mehr nur in großen Industrieanlagen finden, sondern inzwischen bald in jedem technischen Alltagsgerät, jedem Fahrzeug, einer Vielzahl mobiler Gerätschaften und bald womöglich sogar in unseren Kleidungsstücken. Dadurch steigt die Vielfalt der Systeme, die unterstützt werden wollen und die Anzahl der Systeme, die miteinander interagieren, drastisch an. Aber auch die Ansprüche des Benutzers an Zuverlässigkeit, Benutzerfreundlichkeit, individueller Anpassbarkeit und Sicherheit werden größer. Der Bedarf an Methoden und Werkzeugen der Softwaretechnik nimmt so auch im Bereich der eingebetteten Systeme kontinuierlich zu. Doch dank der zunehmenden Leistungsfähigkeit eingebetteter Systeme wird auch der Einsatz dieser Methoden und Werkzeuge zunehmend möglich. Dennoch sind eingebettete Systeme nach wie vor oftmals in Leistung und Speicherkapazität anderen Systemen der Informationstechnologie unterlegen, weshalb die vorhandenen Ressourcen bestmöglich genutzt werden wollen, um die Applikation mit ihrem Bedarf an Rechenzeit und Speicherplatz möglichst effizient und damit zugleich auch Strom sparend ausführen zu können. Zugleich müssen diese Applikationen aber auch in vertretbarer Zeit, mit vertretbarem Aufwand und unter den gegebenen Sicherheits- und Zuverlässigkeitskriterien in einer hochvernetzten Welt entwickelbar sein. Dieses Anforderungsprofil muss ein Werkzeug zur Softwareentwicklung, z.B. eine Programmiersprache, erfüllen können. Es muss also zum einen die Bereiche Komplexitätsreduzierung, Portabilität, Wiederverwendbarkeit und Erweiterbarkeit abdecken können, zum anderen aber auch den hohen Anforderungen an Effizienz genügen, damit es im Bereich der eingebetteten Systeme eingesetzt werden kann. Die Programmiersprache ObjectTeams/Java ist ein solches Werkzeug der Softwareentwicklung, welches auch die Entwicklung von Software im Bereich eingebetteter Systeme erleichtern kann. Sie erweitert die sehr portable Programmiersprache Java um ein ausdrucksstarkes Modularisierungskonzept und erreicht zugleich über aspektorientierte Mechanismen eine Verbesserung bezüglich der Erweiterbarkeit, Wiederverwendbarkeit und Fehlerkorrektur.

Um diese Vorteile bisher nutzen zu können, wurden in der Laufzeitumgebung für die Programmiersprache ObjectTeams/Java relativ rechenintensive und speicherplatzverbrauchende Mechanismen eingesetzt. Im Rahmen dieser Arbeit wurde deshalb die Erweiterung OTRun für die virtuelle Maschine JamVM entwickelt, die es mit hoher Effizienz erlaubt, ObjectTeams/Java Programme direkt

auszuführen. Gleichzeitig wurde mit dieser Erweiterung auch die Flexibilität von ObjectTeams/Java gesteigert, so dass Teamklassen nun zu einem beliebigen Zeitpunkt einer bereits laufenden Applikation dazu geladen werden können, um von diesem Zeitpunkt an den weiteren Programmverlauf beeinflussen zu können. Damit leistet OTRun zumindest einen kleinen Beitrag zur Erfüllung des Wunsches nach vom Benutzer anpassbarer Software, mit dem diese Arbeit begann.

Zum Erreichen dieses Ziels wurde zunächst eine neue Kommunikationsform zwischen der Java-Welt und der virtuellen Maschine vorgestellt: Das *Infrastructure Weaving*. Dieses erlaubt es, Java Methoden direkt auf neu eingeführte Bytecodes innerhalb der virtuellen Maschine abzubilden. Da diese Abbildung jedoch hinter den Kulissen der virtuellen Maschine stattfindet, gelingt es mit diesem Verfahren, konform zum Standard für die Java Virtual Machine zu bleiben, aber dennoch sicher und hocheffizient mit der VM kommunizieren zu können. Die neuen Bytecodes sind nach außen hin nicht sichtbar und können von keinem Java-Programm direkt verwendet werden. OTRun fügt sich vollkommen transparent in die Ausführung eines ObjectTeams/Java Programms ein und übernimmt die Aufgaben des OTRE, ohne dass der ot-Compiler dafür verändert zu werden brauchte.

Mit Hilfe des Infrastructure Weavings gelang es, die wesentlichen Methoden der ObjectTeams-Laufzeitumgebung auf Bytecodes innerhalb der virtuellen Maschine abzubilden und diese in Form von Superinstructions zu optimieren. Auch die Algorithmen, die ObjectTeams/Java für diese Methoden zur Laufzeit verwendet, wurden durch neue, optimierte und delegationsbasierte Algorithmen ersetzt. Im Zentrum stand dabei die Entwicklung eines neuen Konzeptes zur Callin-Ausführung: Der *Delegationproxy*, der in dieser Arbeit detailliert beschrieben wurde. Bei Benchmarks ergab die Kombination all dieser Optimierungen eine Steigerung der Ausführungsgeschwindigkeit bei Callins um den Faktor 21. Dies entspricht einer Reduktion der Laufzeit von 95% im Vergleich zum OTRE. Durch einige einfache weitere Optimierungen zur Vermeidung des Lock/Unlock()-Overheads beim Lifting konnte gezeigt werden, dass sich die Geschwindigkeit relativ einfach sogar bis um den Faktor 33 erhöhen ließe. In diesem Testfall nährte sich OTRun bereits zunehmend dem Optimierungsgrenzwert, der durch die Ausführungszeit von unveränderten Java-Methoden durch die VM gegeben ist. So war der Overhead des Delegationproxy in diesem günstigsten Fall bereits nur noch dreieinhalb mal so hoch wie der Overhead eines normalen Methodenaufrufs. Dies wäre ein Overhead für ObjectTeams/Java, der bereits durch den Geschwindigkeitszuwachs einer einzigen neuen Prozessorgeneration kompensiert werden könnte, wenn man berücksichtigt, dass eine reale Applikation vermutlich nur in ca. 10% aller Methodenaufrufe ein Callin ausführt. So konnte mit dieser Arbeit gezeigt werden, dass der Overhead von ObjectTeams/Java vernachlässigbar sein kann, wenn die Laufzeitumgebung angemessen optimiert wurde.

Zusätzlich wurden im Rahmen dieser Arbeit die speicherplatzverbrauchenden Hashtabellen durch geringfügig effizientere LookupTables ersetzt.

Zusammenfassend kann daher festgehalten werden, dass das Ziel dieser Arbeit erreicht wurde. Lifting, Teamaktivierung und Callin-Ausführung wurden

signifikant optimiert und auch der Speicherverbrauch konnte reduziert werden, so dass sich ObjectTeams/Java nun bedeutend besser positionieren kann, wenn sein Einsatz im Bereich der eingebetteten Systeme erwogen wird. Zusätzlich konnte die Flexibilität von ObjectTeams/Java durch die Unterstützung des Lazy Loadings für Team- und Basisklassen gesteigert werden. Dies eröffnet neue Möglichkeiten für den Einsatz von ObjectTeams/Java.

OTRun ist allerdings bei weitem noch nicht vollständig äquivalent zum OTRE, der momentan verwendeten Laufzeitumgebung von ObjectTeams/Java. Bei der derzeitigen Implementierung von OTRun handelt es sich lediglich um einen Prototyp, mit dem gezeigt werden sollte, dass die Konzepte von ObjectTeams/Java nicht zwangsläufig das Stigma der Ineffizienz tragen müssen und von einer Implementierung innerhalb einer Java Virtual Machine erheblich profitieren können. Dies wurde erreicht. Doch OTRun erfüllt bisher nur einen Bruchteil der ObjectTeams/Java Sprachspezifikation und ist auch noch nicht fehlerfrei. In diesem Kapitel wurde deshalb ein Ausblick darauf gegeben, wie viele dieser noch fehlenden Sprachkonzepte in OTRun umgesetzt und implementiert werden können. Die Umsetzung einiger dieser Konzepte setzt jedoch eine Anpassung des ot-Compilers voraus.

Ob dies für den weiteren Verlauf der Entwicklung von ObjectTeams/Java strategisch sinnvoll ist, bleibt eine offene Frage. Ich würde mich jedoch dafür aussprechen, zumal die Erfahrungen, die so gesammelt werden können, es irgendwann vielleicht einmal ermöglichen, auch eine der großen virtuellen Maschinen wie z.B. Oracle Hotspot so anzupassen, dass sie ObjectTeams/Java Programme nativ und hocheffizient ausführen können. Dies würde der weiteren Verbreitung von ObjectTeams/Java – und damit der breiten Anwendung seiner vortrefflichen Mechanismen zur aspektorientierten Modularisierung – sicherlich sehr zuträglich sein.

Die Programmiersprache Java, auf der ObjectTeams/Java aufbaut, ist eine sehr portable Programmiersprache, ein wichtiges Kriterium für die Entwicklung eingebetteter Software. Auch C ist eine Programmiersprache, die auf vielen Plattformen weit verbreitet ist, sie eignet sich jedoch aufgrund ihrer kaum vorhandenen Modularisierungskonzepte nicht gut für die Entwicklung von großen und komplexen Applikationen, sondern dient eher dazu hocheffiziente, hardwarenahe, plattformspezifische Systemwerkzeuge zu entwickeln. Die JamVM ist ein solches Werkzeug, ein kompakter Interpreter für die Programmiersprache Java, der sich gut für die Verwendung in eingebetteten Systemen eignet. In ihr treffen so C und Java, zwei wichtige Werkzeuge für die Entwicklung von Software für eingebettete Systeme, direkt aufeinander. Und mit ObjectTeams gesellt sich nun noch ein weiterer wertvoller Mitspieler dazu, der uns in die adaptive – und doch typsichere – Softwarewelt der aspektorientierten Modularisierung trägt. Mit OTRun, einer Erweiterung für die JamVM zur direkten Unterstützung von ObjectTeams/Java, die im Rahmen dieser Arbeit entwickelt wurde, ist schließlich ein Werkzeug gegeben, mit dem gezeigt werden konnte, dass die Frage nach Effizienz, wie sie insbesondere bei eingebetteten Systemen aufkommt, kein Grund mehr zu sein braucht, auf die mächtigen aspektorientierten Modularisierungs-mechanismen der Sprache ObjectTeams/Java zu verzichten.

Literatur

[JAMVM] JamVM homepage: http://jamvm.sourceforge.net.

[OTJLD] Stephan Herrmann, Christine Hundt, and Marco Mosconi. ObjectTeams/Java Language Denition | version 1.3., Fak. IV, Technical University Berlin, 2010.

[Br95] Frederick P. Brooks : The Mythical Man-Month - Essays on Software Engineering, Addison-Wesley 1975/1995 20th Anniversary, Chapter 16, No Silver Bullet

[Dij72] Edsger W. Dijkstra, Communications of the ACM archiv, Volume 15 , Issue 10 (October 1972) Pages: 859 - 866, 1972, The humble programmer

[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.

[Hä06] Paul Häder. Benchmarking und Optimierung der Aspektwebestrategie von ObjectTeams/Java. Diplomarbeit. Technische Universität Berlin, 2006

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

[JVMSpec] Tim Lindholm, Frank Yellin, The Java Virtual Machine Specification, 2nd Edition. Prentice Hall, 2000

[Ertl02] M. Anton Ertl. Threaded Code Variations and Optimizations. TU Wien, Forth-Tagung 2002

[Ertl03] M. Anton Ertl. David Gregg. The Structure and Performance of Efficient Interpreters. Journal of Instruction-Level Parallelism, 2003

[Pr08] Gregory B. Propkopski, Clark Verbrugge. Analyzing the Performance of Code-copying Virtual Machines. Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications . 2008

[Piu98] Ian Piumarta. Fabio Riccardi. Inria Rocquencourt. Optimizing Direct Threaded Code By Selective Inlining. SIGPLAN ’98 Conference on Programming Language Design and Implementation, 1998.

[Bac04] David F. Bacon, Perry Cheng. David Grove. Garbage collection for embedded systems. Proceedings of the 4th ACM international conference on Embedded software, 2004

[Jo96] Richard Jones. Rafel D. Lins. Garbage Collection: Algorithms for Automatic Dynamic Memory Management, Wiley 1996.

[HS07] Michael Haupt and Hans Schippers. A machine model for aspect-oriented programming. In ECOOP, pages 501{524. Springer, 2007.

[Din09] Tom Dinkelaker. Mira Mezini. Christoph Bockisch. The Art of the Meta-Aspect Protocol. Proceedings of the 8th ACM International Conference on Aspect-Oriented Software Development. New York, NY, USA , pp. 51-62, 2009

[Herr04] Stephan Herrmann, Christine Hundt, Katharina Mehner. Translation Polymorphism in Object Teams. Bericht Nr. 2004/05 .Technische Universität Berlin.

[Hu03] Christine Hundt: Bytecode-Transformationen zur Laufzeitunterstützung von aspekt-orientierter Modularisierung mit ObjectTeams/java, Technische Universität Berlin, Diplomarbeit, 2003

[Hu09] Christine Hundt and Sabine Glesner. Optimizing aspectual execution mechanisms for embedded applications. In Proceedings of the First Workshop on Generative Technologies (WGT) 2008, volume 238

[Hu10] Optimizing Aspect-oriented Mechanisms for Embedded Applications, Christine Hundt, Daniel Stoehr, Sabine Glesner, Technische Universität Berlin

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

[Fr09] Oliver Frank. Praxistaugliches dynamisches Aspektweben für ObjectTeams/Java im Kontext des OSGi Komponentenframeworks. Diplomarbeit. Technische Universität Berlin, 2009

[Bo06] Christoph Bockisch. Matthew Arnold. Tom Dinkelaker. Mira Mezini. Adapting Virtual Machine Techniques for Seamless Aspect Support. Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2006