Lava

Delegation in einer streng typisierten Programmiersprache — Sprachdesign und Compiler

Diplomarbeit von

Pascal Costanza
Gotenstraße 19
D-53844 Troisdorf

Email: costanza@cs.uni-bonn.de

Institut für Informatik III

Rheinische Friedrich-Wilhelms-Universität Bonn

Professor Dr. A. B. Cremers

14. Januar 1998

Zusammenfassung

Im Gegensatz zu objektorientierten Programmiersprachen, die Vererbung auf Klassenebene realisieren, gibt es Sprachen, die dieses Konzept ausschließlich auf Objektebene verwirklichen. Dabei können „Unterobjekte“ nicht nur Methoden ihrer „Oberobjekte“ lokal „überschreiben“ (*overriding*), sondern die Oberobjekte lassen sich auch zur Laufzeit gegen andere Objekte austauschen.

Prinzipiell lassen sich die Vorteile beider Konzepte – klassenbasierte und objektbasierte Veerbung – in einer Sprache integrieren. Anhand einer Erweiterung der Sprache Java haben wir (Matthias Schickel und ich) nun gezeigt, daß sich das Konzept der objektbasierten Vererbung in bestehenden klassenbasierten, streng typisierten Sprachen effizient implementieren läßt. Unser gemeinsames Ziel war es, die Programmiersprache Java um zwei Varianten dynamischer objektbasierter Vererbung (Delegation und Konsultation), passend zu den anderen Sprachkonstrukten von Java, zu ergänzen.

Der Name Java bezeichnet sowohl die bekannte Programmiersprache als auch das zugehörige Laufzeitsystem. Analog dazu haben wir dem Ergebnis unserer Arbeit den Namen Lava gegeben.

In meiner Diplomarbeit werde ich

- die beim Sprachdesign auftretenden konzeptuellen Probleme, sowie die durch die speziellen Eigenschaften der Zielsprache Java bedingten zusätzlichen Komplikationen diskutieren,
- unsere Lösungen kritisch beleuchten,
- die resultierende Architektur des Gesamtsystems Lava vorstellen,
- auf die durchgeführten Erweiterungen des Java-Compilers eingehen und
- die Ergebnisse der Implementierungsarbeiten an Beispielen demonstrieren.

Erklärung: Hiermit erkläre ich, diese Diplomarbeit selbständig durchgeführt zu haben. Alle Quellen und Hilfsmittel, die ich verwendet habe, sind angegeben. Zitate habe ich als solche kenntlich gemacht.

Troisdorf, 14. Januar 1998, Pascal Costanza

Inhaltsverzeichnis

1 Einleitung **7**

1.1 Ziel der Diplomarbeit 7

1.2 Gliederung 8

1.3 Rechtliche Hinweise 9

1.4 Danksagungen 9

I Grundlagen **11**

2 Objektorientierte Konzepte **13**

2.1 Objekte und Nachrichten 13

2.2 Klassenbasierte Vererbung 14

2.3 Objektbasierte Vererbung 15

2.4 Typisierung 16

2.5 Design Patterns 18

3 Java Konzepte **21**

3.1 Die Programmiersprache Java 22

3.2 Die Java Virtual Machine 26

II Lava: Java += Delegation **33**

4 Aufgaben **35**

5 Sprachdesign **37**

5.1 Delegation und Konsultation 37
5.2 Deklaration von Elternverweisen 38
5.2.1 delegatee Felder 38
5.2.2 consultee Felder 43
5.3 Erweiterung des Typsystems 44
5.3.1 mandatory Felder 45
5.3.2 optional Felder 47
5.3.3 Typsichere Delegation 47
5.3.4 Typunsichere Delegation 48
5.3.5 Zusammenfassung 48
5.4 Behandlung von this 49
5.4.1 Zugriffe auf Felder von this 49
5.4.2 Nachrichten an this 49
5.4.3 Zusammenfassung 52
5.5 Auflösung von Namenskonflikten 53
5.5.1 Umbenennungen 55
5.5.2 Umbenennung von Feldern 55
5.5.3 Umbenennung von Methoden 59
5.5.4 Namenskonflikte zwischen Klassen- und Instanzelementen 62
5.5.5 Diskussion 63
5.5.6 Zusammenfassung 65
5.6 Kapselung bei objektbasierter Vererbung 65
5.6.1 delegatee Klassen 65
5.7 Zusammenfassung 66

6 Das Strategy Pattern **69**

6.1 Implementation in Java 69
6.2 Implementation in Lava 74
6.3 Zusammenfassung 75

7 Design der Lava Virtual Machine **77**

7.1 Feldannotationen 77

7.2 Feldzugriffe 78

7.3 Übersetzung von Methoden 78

7.4 Klassenannotationen 79

7.5 Methoden aus Elterntypen 79

7.5.1 Unterscheidung zwischen this-Objekt und Träger der aktuellen Methode 79

7.5.2 Explizite Delegation 81

7.5.3 Nachrichten an this 81

7.5.4 Suche nach Delegatees in der Elternhierarchie 82

7.6 Umbenennungen und Selektionen 85

7.7 Die Annotation mandatory 86

7.8 Zusammenfassung 87

8 Der Lava-Compiler **89**

8.1 Der Java-Compiler von Sun Microsystems 89

8.1.1 Die Pakete sun.tools.java und sun.tools.javac 90

8.1.2 Das Paket sun.tools.tree 92

8.1.3 Das Paket sun.tools.asm 94

8.1.4 Der Übersetzungsvorgang 95

8.2 Erweiterungen des Lava-Compilers 95

8.2.1 Änderungen in sun.tools.asm 95

8.2.2 Änderungen in sun.tools.tree 96

8.2.3 Änderungen in sun.tools.java und sun.tools.javac 97

8.3 Zusammenfassung 99

9 Evaluation **101**

9.1 Sprachdesign 101

9.2 Lava Virtual Machine 102

9.3 Lava-Compiler 103

9.4 Ausblick 104

10 Resümee **105**

A Spezifikation der Sprache Lava **109**

A.1 Einleitung 109

A.2 Notation 109

A.3 Ergänzungen 110

A.3.1 Schlüsselwörter 110

A.3.2 Trennzeichen 110

A.3.3 Konversionen 110

A.3.4 Zugriffskontrolle 111

A.3.5 Standardklassen 111

A.3.6 Klassenannotationen 111

A.3.7 Felddeklarationen 112

A.3.8 Methodendeklarationen 113

A.3.9 Umbenennungen 114

A.3.10 Methodenaufrufe 114

B Spezifikation der Lava Virtual Machine **115**

B.1 Einleitung 115

B.2 Ergänzungen zum Class File Format 115

B.2.1 Klassenbezogene Ergänzungen 115

B.2.2 Feldbezogene Ergänzungen 115

B.2.3 Methodenbezogene Ergänzungen 116

B.2.4 Neue Attribute 116

B.3 Ergänzungen zum JVM Befehlssatz 118

Literaturliste **123**

Kapitel 1

Einleitung

1.1 Ziel der Diplomarbeit

Objektorientierte Programmiersprachen sind im Laufe des letzte Jahrzehnts zu etablierten Werkzeugen für die Implementation von Softwarelösungen geworden. Gerade die weite Verbreitung von C++ und in letzter Zeit Java, aber auch anderer Programmiersprachen, für die mit der Unterstützung objektorientierter Konzepte als einer der herausragenden Eigenschaften geworben wird, belegen dies.[1]

Am weitesten verbreitet sind jene Sprachen, die *Vererbung auf Klassenebene* verwirklichen. In Verbindung mit einem *strengen Typsystem* eignen sie sich hervorragend für die Implementation flexibler, aber dennoch effizienter und sicherer Systeme. Die oben genannten Sprachen sind Vertreter solcher klassenbasierter Sprachen.

Auf der anderen Seite gibt es Sprachen, die *Vererbung auf Objektebene* verwirklichen; am bekanntesten sind SELF, NewtonScript und Cecil. Objekte können in diesen Sprachen Referenzen auf sogenannte Elternobjekte besitzen. Über solche Referenzen werden Eigenschaften der Elternobjekte geerbt. Sie sind, wie andere Referenzen auch, zur Laufzeit eines Programms veränderbar. Die Vererbungsbeziehungen innerhalb eines Programms sind demnach nicht wie in klassenbasierten Sprachen statisch zur Übersetzungszeit festgelegt, sondern können dynamisch zur Laufzeit verändert werden.

Das macht solche objektbasierten Sprachen besonders geeignet für *dynamische Verhaltensänderungen* von Objekten, *Modellierung von Rollen*, Versioning, etc. Solche Konzepte lassen sich nur sehr umständlich in klassenbasierten Sprachen realisiseren.

Ein wesentlicher Nachteil, der bisher in objektbasierten Sprachen in Kauf genommen werden muß, ist ein *fehlendes statisches Typkonzept*, das es ermöglichen

[1] Für Literaturverweise zu den in der Einleitung erwähnten Programmiersprachen siehe die erste Fußnote in Kapitel 2.

würde, zur Übersetzungszeit eines Programms festzustellen, welche Objekte welche Nachrichten verstehen. Daher können bestimmte Programmierfehler nicht bereits vom Compiler für eine solche Sprache abgefangen werden.

In „Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems“ von Günter Kniesel (s. [Kniesel 95]) wird ein Vorschlag gemacht, wie *klassen- und objektbasierte Vererbung in einer statisch typisierten Sprache* vereint werden können. Nach unserem Kenntnisstand ist dies der erste Vorschlag dieser Art.

Das Ziel dieser Diplomarbeit war es, diesen Vorschlag in einer konkreten Sprache zu verwirklichen, und in Zusammenhang mit der Diplomarbeit von Matthias Schickel (s. [Schickel 97]) ein lauffähiges System aus Compiler und Laufzeitumgebung für diese Sprache zu implementieren. Wir haben *Java als Ausgangspunkt für unsere Arbeiten* gewählt. Das Ergebnis unserer Arbeit haben wir Lava genannt, gemäß dem Motto *Lava Is Hotter Than Java*.

1.2 Gliederung

Im ersten Teil meiner Diplomarbeit erläutere ich die Grundlagen, auf denen Lava aufgebaut wurde. Kapitel 2 gibt eine kurze Einführung in objektorientierte Konzepte. Insb. werden die *Unterschiede zwischen klassenbasierter und objektbasierter Vererbung* herausgearbeitet. Außerdem wird der Begriff des Typsystems erläutert, sowie das neuartige Konzept der sog. *Design Patterns* vorgestellt.

In Kapitel 3 wird die Programmiersprache Java und das damit verbundene Ausführungsmodell beschrieben. Es wird eine *vollständige Charakterisierung der Sprache Java* gegeben. Die *Java Virtual Machine* wird soweit vorgestellt, wie es für das Verständnis meiner Diplomarbeit erforderlich ist.

Der zweite Teil enthält die eigentliche Beschreibung der Ergebnisse meiner Arbeit. Kapitel 4 beschreibt in groben Zügen, welche *Aufgaben bei der Verwirklichung unserer Arbeiten* gelöst werden mußten. Aus dem Ausführungsmodell von Java ergab sich eine naheliegende Aufgabenteilung in zwei Bereiche, die von Matthias Schickel und mir weitgehend unabhängig bearbeitet werden konnten.

Kapitel 5 behandelt das *Design der Sprache Lava*: Alle neuen Sprachkonstrukte in Lava werden hier aufgeführt und diskutiert.

In Kapitel 6 demonstriere ich die neuen Sprachkonstrukte anhand des *Strategy Patterns* aus [Gamma 95], einem Design Pattern, das häufig für die Implementation dynamischer Verhaltensänderungen von Objekten in klassenbasierten Sprachen verwendet wird.

Kapitel 7 diskutiert, wie die neuen Sprachkonstrukte übersetzt werden und stellt das *Design der Lava Virtual Machine* vor.

In Kapitel 8 werden die *Erweiterungen des Java-Compilers* von Sun Microsystems beschrieben, die ich vorgenommen habe, damit Lava-Programme korrekt übersetzt werden.

In Kapitel 9 versuche ich, eine *Bewertung* des Designs der Sprache Lava und der Lava Virtual Machine, sowie des Lava-Compilers vorzunehmen.

In Kapitel 10 wird schließlich knapp zusammengefaßt, welche *Ergebnisse* ich mit meiner Arbeit erzielt habe.

Meine Diplomarbeit enthält darüberhinaus zwei Anhänge, die die Spezifikationen von Lava und der Lava Virtual Machine enthalten.

1.3 Rechtliche Hinweise

Sun, Sun Microsystems, Java und JavaSoft sind Warenzeichen oder eingetragene Warenzeichen der Firma Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Kalifornien, USA. Alle anderen Produktnamen, die in dieser Diplomarbeit erwähnt werden, sind Warenzeichen der jeweiligen Markeninhaber.

Der in Kapitel 8 beschriebene Java-Compiler unterliegt dem Copyright der Firma Sun Microsystems, Inc. Die Beschreibung des Java-Compilers in dieser Diplomarbeit darf für nicht-kommerzielle Zwecke verwendet werden. Eine kommerzielle Auswertung ist nicht erlaubt.

1.4 Danksagungen

Ich danke Herrn Professor Dr. A. B. Cremers für die Betreuung meiner Diplomarbeit. Ein großer Dank geht an Günter Kniesel; ohne seine theoretische Grundlagenarbeit wäre diese Diplomarbeit nicht möglich gewesen.

Ebenfalls bedanken möchte ich mich bei Matthias Schickel für die reibungslose Zusammenarbeit und bei Dirk Theisen und Thomas Kolbe für sehr anregende Diskussionen. Die Grafiken in dieser Diplomarbeit wurden von Ingo Breuer nach meinen Vorlagen angefertigt, dem hierfür mein besonderer Dank gilt.

Teil I

Grundlagen

Kapitel 2

Objektorientierte Konzepte

Dieses Kapitel gibt eine kurze Einführung in Begriffe und Zusammenhänge, die im Umfeld der Objektorientierten Programmierung Verwendung finden und wichtig für das Verständnis dieser Arbeit sind.[1] Da unsere Arbeit eine Erweiterung von Java darstellt, stützen wir uns im Wesentlichen auf die Terminologie, die in Zusammenhang mit Java üblicherweise verwendet wird.

Im folgenden werde ich einige Begriffe erläutern, zwei Verfahren zur Realisierung von Vererbung darstellen, Fragen der Typisierung behandeln und zum Schluß einen Überblick über *Design Patterns* geben.

2.1 Objekte und Nachrichten

Objekte sind Träger von Informationen (Daten) und *Methoden*. Methoden enthalten Algorhitmen, die die Daten eines Objekts manipulieren.[2] Um eine Methode auszuführen, muß man an ein Objekt eine *Nachricht senden*, in der Hoffnung, daß es eine zur Nachricht passende Methode bereitstellt.[3] Der Mechanismus, der zu einer Nachricht eine passende Methode findet, wird *dynamische Nachrichtenbindung* oder *dynamische Bindung* genannt, da er in der Regel erst zur Laufzeit eines Programms greift.[4]

Das Versenden einer Nachricht `m` an ein Objekt, das an die Variable `obj` gebunden ist, wird z.B. in der Programmiersprache Java wie folgt notiert:

```
obj.m(par1, ..., parn)
```

[1] Es wird dabei auf eine Reihe von objektorientierten Programmiersprachen Bezug genommen. Es folgt eine Liste aller genannten Programmiersprachen mitsamt Literaturverweisen: BETA [Madsen 93], C++ [Ellis 95], Cecil [Chambers 92], CLOS [Moon 89], Delegation [Lieberman 86], Eiffel [Meyer 92], Java [Gosling 96], NewtonScript [Smith 95], Oberon-2 [Mössenböck 93], SELF [Ungar 87], Simula [Dahl 68], Smalltalk [Goldberg 89]

[2] „Objects are collections of operations that share a state.“ [Wegner 90], S. 8.

[3] In einigen Programmiersprachen kann nicht nur ein einzelnes Objekt, sondern auch eine Gruppe von Objekten Adressat einer Nachricht sein. Beispiele für solche Programmiersprachen sind CLOS und Cecil. Dieser Fall stellt jedoch eine Ausnahme dar, auf die ich in meiner Arbeit nicht weiter eingehen werde.

[4] Weitere Bezeichnungen sind auch *message binding*, *late binding* oder *dynamic binding*.

Dabei sind par1,...,parn die Parameter der Nachricht.

Eine Variable kann zu verschiedenen Zeitpunkten unterschiedliche Objekte enthalten, die für dieselbe Nachrichten unterschiedliche Methoden bereitstellen. Daher wird ein Ausdruck obj.m(par1, ..., parn) als *polymorph* bezeichnet, da durch ihn nicht festgelegt wird, welche konkrete Methode ausgeführt wird.

Das Objekt, das eine Nachricht empfängt, spielt in der Methode, die ausgeführt wird, eine besondere Rolle und wird daher üblicherweise an einen festgelegten Bezeichner gebunden, der für den Programmierer wie eine nicht modifizierbare Variable aussieht. In Java heißt diese Pseudovariable this.[5]

Vererbung kann als ein Mechanismus verstanden werden, Eigenschaften von Objekten in abgeleiteten Objekten wiederzuverwenden.[6] Was dies im Einzelnen bedeutet, hängt von der jeweils verwendeten Programmiersprache ab. Es lassen sich aber zwei verschiedene Vererbungskonzepte in den bisher bekannten Programmiersprachen unterscheiden: Zum Einen *klassenbasierte* und zum Anderen *objektbasierte* Vererbung.

2.2 Klassenbasierte Vererbung

In den sogenannten *klassenbasierten Sprachen*[7] wird Vererbung auf Klassenebene realisiert. Eine *Klasse* kann als eine Schablone aufgefaßt werden, die beschreibt, welche Eigenschaften ein Objekt besitzt. Objekte werden in solchen Sprachen als *Instanzen* solcher Klassen gebildet, d.h. es muß bei der Erzeugung eines Objekts angegebenen werden, welcher Klasse es angehören soll.

Bei der *Deklaration* einer Klasse wird festgelegt, welche Daten ein Objekt der Klasse enthält. Diese Daten werden als *Instanzvariablen* oder *Felder* bezeichnet. Desweiteren wird festgelegt, welche Nachrichten ein Objekt der Klasse versteht und welche Methoden diesen Nachrichten zugeordnet werden. Diese Methoden werden als *Instanzmethoden* oder einfach nur als *Methoden* bezeichnet. Analog zu den Instanzvariablen und Instanzmethoden gibt es auch *Klassenvariablen* und *Klassenmethoden*, die nicht für jedes Objekt, sondern nur einmal für die gesamte Klasse existieren. Instanzvariablen und Instanzmethoden bilden zusammen die *Instanzelemente* oder *Elemente*, Klassenvariablen und Klassenmethoden die *Klassenelemente*.

In der Regel kann das Objekt / die Klasse Nachrichten von allen anderen Objekten / Klassen erhalten. In den meisten Programmiersprachen existieren jedoch Mechanismen, die die Verwendung von Nachrichten für bestimmte Klassen ausschließen oder zulassen; somit lassen sich verschiedene *Schnittstellen* zu einem Objekt / einer Klasse für verschiedene Kontexte definieren. Elemente, die zur

[5] Dieser Name wird auch in BETA, C++ und Simula verwendet. Die Arbeiten zu Darwin ([Kniesel 95] und [Kniesel 96a]), die die Grundlagen unserer Arbeit darstellen, benutzen in Anlehnung an SELF und Smalltalk den Namen self. In Oberon-2 wird kein einheitlicher Name festgelegt, sondern kann vom Programmierer für jede Methode einzeln vergeben werden.

[6] „Depending on the way it is used, inheritance can be considered as a means of sharing code, of sharing properties or of representing a type hierarchy.“ [Masini 91]

[7] Dazu gehören die meisten und bekanntesten Vertreter der objektorientierten Programmiersprachen, wie C++, Eiffel, Java, Simula und Smalltalk.

öffentlichen Schnittstelle eines Objekts oder einer Klasse gehören, können von allen anderen Objekten / Klassen verwendet werden. Je nach Programmiersprache existieren weitere nicht-öffentliche Schnittstellen. Wenn von *der* Schnittstelle eines Objekts / einer Klasse die Rede, so ist damit im Allgemeinen die öffentliche Schnittstelle gemeint.

Klassen können von anderen Klassen abgeleitet werden, indem bei der Deklaration einer Klasse C eine andere Klasse D als *direkte Oberklasse* angegeben wird. Umgekehrt ist C dann eine *direkte Unterklasse* von D. In der Unterklasse sind alle Elemente der Oberklasse enthalten. Zusätzlich können Methoden *redefiniert* und neue Felder und Methoden hinzugefügt werden, was auch als *Spezialisierung* bezeichnet wird.[8]

Die direkte Ober- oder Unterklassenbeziehung kann als Relation aufgefaßt werden. In den o.a. Sprachen bildet diese Relation eine strikte Ordnung. Die Beziehungen zwischen den Klassen sind statisch und können während der Ausführung eines Programms nicht mehr abgeändert werden. Die transitive Hülle der direkten Ober- bzw. Unterklassenbeziehung wird einfach als Ober- bzw. Unterklassenbeziehung bezeichnet.

Das Bilden von Unterklassen wird in solchen klassenbasierten Sprachen als Vererbung bezeichnet. Es gibt Sprachen, die es erlauben, mehr als eine direkte Oberklasse bei der Deklaration einer Klasse anzugeben, was als *Mehrfachvererbung* oder *multiple Vererbung* bezeichnet wird.

Wie bereits erwähnt, stehen den Instanzen einer Klasse C alle Elemente der Oberklassen von C zur Verfügung. Wird eine Nachricht an ein Objekt der Klasse C gesendet, so wird in der Regel die jeweils *spezifischste* Methode ausgewählt: Das ist diejenige Methode, die zu der Nachricht paßt und die in einer Klasse definiert ist, die im Klassengraphen, der aus der Oberklassenbeziehung resultiert, den geringsten Abstand zur Klasse C hat. Ob eine Methode zu einer Nachricht paßt, hängt von der Semantik einer Programmiersprache ab.[9] Da im Allgemeinen nur zur Laufzeit eines Programms feststellbar ist, welches die spezifischste Methode für eine Nachricht ist, hat man hier ein Beispiel für dynamische Nachrichtenbindung vorliegen. Viele Programmiersprachen erlauben es, Ausnahmen von dieser Regel bei der Definition einer Methode festzulegen, um statische Nachrichtenbindung zu ermöglichen und dadurch Optimierungen für Speicherbedarf und Laufzeit eines Programms zu erreichen.

2.3 Objektbasierte Vererbung

Bei objektbasierter Vererbung, wie sie mit der Sprache Delegation eingeführt wurde und die z.B. in Cecil, NewtonScript und SELF verwendet wird, wird auf die Deklaration von Klassen verzichtet. Vererbung wird hier direkt über die Objektstruktur verwirklicht.

Objekte sind in solchen Sprachen klassenlos: Bei ihrer Erzeugung können sie nicht als Instanz einer Klasse angelegt werden, sie können aber die Eigenschaften

[8] Eine Redefinition von Feldern ist in den meisten klassenbasierten Sprachen nicht möglich.

[9] Eine genaue Definition dessen, was es z.B. in Java bedeutet, daß eine Methode zu einer Nachricht paßt, ist in [Gosling 96], §15.11 enthalten.

anderer Objekte kopieren, um gleichartiges Verhalten zu erzielen. Solche Objekte werden daher auch als *Prototypen* bezeichnet. Neben den bekannten Feldern und Methoden können sie auch Verweise auf andere, sog. *Elternobjekte* enthalten, an die sie Nachrichten *delegieren*, zu denen sie selbst keine passende Methode bereitstellen. Diese Verweise können im Gegensatz zu den statischen Klassenbeziehungen bei klassenbasierter Vererbung zur Laufzeit dynamisch verändert werden. Dadurch wird eine dynamische Zuordnung von Eigenschaften erreicht. Analog zu den Ober- und Unterklassen bei klassenbasierter Vererbung spricht man bei objektbasierter Vererbung von Eltern- und Kindobjekten. Die Bemerkungen zu direkter und transitiver Oberklassen- und Unterklassenbeziehung lassen sich ebenfalls auf Elternobjekt- und Kindobjektbeziehung übertragen.

Entscheidend dabei ist, daß bei Delegation einer Nachricht m an ein Elternobjekt der Bezeichner this (oder der in der jeweiligen Programmiersprache dazu äquivalente Bezeichner) immer an das Objekt gebunden bleibt, an welches die Nachricht ursprünglich gerichtet war. Sendet die an m gebundene Methode eine weitere Nachricht n an this, so wird diese Nachricht daher wieder an den ursprünglichen Empfänger zurückgesendet. Stellt dieses Objekt eine zu n passende Methode zur Verfügung, so wird diese ausgeführt. Aufgrund dieser Tatsache können also auch bei objektbasierter Vererbung Methoden in Kindklassen redefiniert werden, ähnlich zur Redefinition von Methoden in Unterklassen bei klassenbasierter Vererbung.

Aus diesem Grund wird zwischen dem this-Objekt und dem *Träger* einer Methode unterschieden: Der Träger einer Methode ist das Objekt, das die Methode bereitstellt, die aktuell ausgeführt wird. Aus den obigen Ausführungen ist ersichtlich, daß es sich dabei nicht um das this-Objekt handeln muß, im Gegensatz zu Programmiersprachen mit klassenbasierter Vererbung, wo das this-Objekt immer auch der Träger der aktuellen Methode ist.

2.4 Typisierung

Über Aufbau und Notwendigkeit von *Typsystemen* gibt es kontroverse Standpunkte, die z.B. in [Wegner 90] zusammengefaßt und in [Palsberg 94] ausführlich behandelt werden. Wir verwenden in unserer Arbeit die Begriffe Typ und Typsystem, wie sie im Zusammenhang mit den gängigen klassenbasierten Programmiersprachen gebraucht werden.[10]

Ein Typsystem definiert Konsistenzkriterien, die es *Typcheckern* erlauben, Programme vor ihrer Ausführung zu prüfen. In der Regel sind solche Typchecker Bestandteil eines Compilers. Sie können sicherstellen, daß nur Ausdrücke zugelassen werden, die den Schnittstellen der beteiligten Klassen bzw. Objekte genügen, um so zum Einen fehlerhafte Verwendung von Nachrichten zu vermeiden, zu denen keine passende Methoden gefunden werden können. Zum Anderen wird es einem Compiler durch diese Einschränkung zulässiger Ausdrücke ermöglicht, effizienten Code zu erzeugen. Die Korrektheit eines Programm kann mit Hilfe eines Typcheckers jedoch nicht garantiert werden.

[10] Dies sind z.B. C++, Eiffel, Java, Oberon-2 und Simula.

Um Typchecks zu ermöglichen, definiert ein Typsystem eine Äquivalenz zwischen Schnittstellen und Typen. Somit kann zu jedem Ausdruck eine Schnittstelle, der *statische Typ*, bestimmt werden, die erfüllt werden muß. Dieser statische Typ stellt eine Mindestanforderung dar, die von allen Objekten erfüllt wird, deren Schnittstelle den statischen Typ als Teilmenge enthalten. Eine Teilmenge eines Typs wird als *Supertyp* des Typs bezeichnet; ein Objekt eines Typs kann auch dort verwendet werden, wo „nur" ein Objekt eines Supertyps erwartet wird. Die Umkehrrelation zur Supertypbeziehung ist die *Subtypbeziehung*.

Die Konvertierung eines Typs zu einem Supertyp wird *Typerweiterung (widening)* genannt, und die eines Typs zu einem Subtyp *Typverengung (narrowing)*. Eine Typverengung ist in der Regel mit einem Laufzeittest verbunden, der feststellt, ob ein Objekt tatsächlich der größeren Schnittstelle genügt – andernfalls kommt es zu einem Laufzeitfehler. Bei einer Typerweiterung ist ein solcher Laufzeittest nicht notwendig, da ein Objekt, das einem Typ genügt, immer auch einer kleineren Schnittstelle genügt.

Zwischen Klassen und Schnittstellen besteht ebenfalls eine Äquivalenzbeziehung, da Schnittstellen von Klassen implementiert werden. Darum liegt es nahe, aus einer Klassenhierarchie, die sich aus der Deklaration von Unterklassen ergibt, eine Typhierarchie aus Subtypen abzuleiten.[11]

Im Gegensatz zur Subtypbeziehung impliziert eine Unterklassenbeziehung keine Schnittstellenerhaltung: Manche Sprachen erlauben zur Wiederverwendung von Code die Deklaration von Unterklassen, bei der die Schnittstelle der Oberklasse eingeschränkt wird.[12] Um trotzdem eine weitgehende Gleichsetzung von Klassen und Typen zu ermöglichen, können Schnittstelleneinschränkungen bei der Deklaration von Unterklassen in den o.a. klassenbasierten Sprachen nicht oder nur begrenzt vorgenommen werden. Daher ergibt sich „normalerweise" aus einer Unterklassenbeziehung auch eine Subtypbeziehung.[13]

Neben den Klassen existieren in typisierten objektorientierten Sprachen auch andere Formen der Schnittstellenbeschreibung, so z.B. primitive Datentypen, `typedef` in C++ oder die Interfaces in Java.

Eine Sprache ist dann *streng typisiert*, wenn ihre Ausdrücke typkonsistent sind.[14] Durch die Zuordnung von Objekten zu Klassen und von Klassen zu Typen kann in einer streng typisierten Programmiersprache sichergestellt werden, daß ein Objekt nur dort verwendet wird, wo es dem verlangten Typ genügt.

Sprachen ohne Typsystem sind einfacher in ihrer syntaktischen Struktur, da sie keine Konstrukte zur Typisierung benötigen. Mit ihnen lassen sich im Vergleich zu streng typisierten Sprachen relativ schnell Problemlösungen umsetzen, da ihre Compiler deutlich weniger Einschränkungen machen bei der Frage,

[11] Das hat zur Folge, daß die meisten klassenbasierten Programmiersprachen auch über ein Typsystem verfügen. Dafür, daß diese Möglichkeit jedoch nicht wahrgenommen werden muß, ist Smalltalk das wohl bekannteste und erfolgreichste Beispiel.

Die Zusammenfassung von Klassen- und Typdefinition ist wohl auch der Grund dafür, daß die Konzepte Klasse und Typ leider häufig verwechselt werden.

[12] Als Beispiel hierfür sind kovariante Unterklassen [Palsberg 94] zu nennen.

[13] Ein Gegenbeispiel stellen die `private`-Unterklassen in C++ dar, die keine Subtypen darstellen.

[14] „Languages in which all expressions are type-consistent are strongly typed languages." [Wegner 90], S. 43.

welche Ausdrücke zulässig sind. Sie werden aus diesem Grund auch als *Rapid-Prototyping Languages* bezeichnet. Da jedoch erst zur Laufzeit eines Programms sichergestellt werden kann, daß Nachrichten nur an Objekte versendet werden, die entsprechende Methoden bereitstellen, reduzieren sich Zuverlässigkeit und Ausführungsgeschwindigkeit solcher Programme. Gerade der Aspekt der Zuverlässigkeit fällt bei großen Programmen besonders ins Gewicht. Zusätzlich wird ihre Optimierung erschwert, da erst zur Laufzeit Informationen über Struktur und Verhalten von Objekten und Methoden gewonnen werden können und nicht mit Hilfe des statischen Klassen- und Typsystems, dessen Informationen bereits vor Ausführung eines Programms zur Verfügung stehen.

2.5 Design Patterns

Objektorientierte Programmiersprachen sind ein wichtiger Bestandteil objektorientierter Methodologien zur Lösung von Problemen in der Informatik. Sie reichen jedoch nicht aus, sondern müssen immer im Zusammenhang von objektorientierter Analyse eines Problemfelds und dem objektorientierten Design einer Lösung betrachtet werden. Sie sind demnach die Hilfswerkzeuge für das letzte Glied in der Dreierkette Analyse – Design – Kodierung.[15]

Design Patterns sind ein neuartiges Teilgebiet der Forschung im Rahmen objektorientierter Methodologien: Sie sollen helfen, Designerfahrung durch systematische Dokumentation praxiserprobter Lösungen zu verallgemeinern und mitteilbar zu machen, die ein Softwareentwickler in der Regel nur im Laufe der Jahre gewinnen kann. Über eine Beschreibung eines Problems und seiner Lösung hinaus benötigt ein Softwareentwickler ein tieferes Verständnis einer Lösung, um sie auf ein neues gegebenes Problem anwenden zu können. Daher erläutert ein Design Pattern auch die Anwendbarkeit, Kosten und Konsequenzen einer Lösung. Ein Design Pattern illustriert darüberhinaus Beispielimplementationen der Lösung in gängigen objektorientierten Programmiersprachen.

Im Laufe der letzten Jahre ist das Konzept der Design Patterns, oder kurz *Patterns*, immer bekannter geworden. Eines der am häufigsten zitierten Bücher ist [Gamma 95], in dem ein Katalog mit 23 Design Patterns präsentiert wird, die alle aus tatsächlich existierenden Softwarelösungen entnommen sind. Diese werden alle nach dem gleichen Schema vorgestellt, was das Erlernen, den Vergleich und den Gebrauch von Design Patterns erleichtern soll. Das Schema hat folgende Bestandteile:

Name Ein prägnanter Name, der möglichst den Kern des Design Patterns beschreibt.

Aufgabe Eine kurze Beschreibung dessen, welches Problem das Pattern wie löst.

Andere Namen Andere Namen, unter denen das Pattern bekannt ist.

[15] Ein Standardwerk, das sich eingehend mit den hier angerissenen Begriffe beschäftigt, ist [Booch 94].

Motivation Hier wird ein Beispielproblem vorgestellt und gezeigt, wie die Klassen- und / oder Objektstruktur im Pattern das Problem löst. Das Beispiel hilft, die folgende abstrakte Beschreibung des Patterns besser zu verstehen.

Anwendbarkeit In welchen Situationen kann das Pattern angewendet werden? Was sind Beispiele schlechten Designs, die durch dieses Pattern besser gelöst werden können? Wie erkennt man solche Situationen?

Struktur Eine graphische Repräsentation der Klassen / Objekte im Pattern.

Teilnehmer Teilnehmer sind Klassen oder Objekte, die an dem Pattern beteiligt sind. Deren Zuständigkeiten werden aufgezeigt.

Zusammenarbeit Hier wird beschrieben, wie die Teilnehmer zusammenarbeiten, um ihre Zuständigkeiten wahrzunehmen.

Konsequenzen Welche Auswirkungen hat das Pattern auf die Klassen / Objekte, auf die es angewendet wird? Was sind die Vorteile und Nachteile des Patterns? Welche Bestandteile des Patterns können variiert werden?

Implementierung Welche Fallstricke, Hinweise und Techniken müssen bei der Implementierung berücksichtigt werden? Gibt es spezifische Eigenarten in Bezug auf bestimmte Programmiersprachen?

Beispielquelltexte Ausschnitte aus Beispielquelltexten sind enthalten, die zeigen, wie das Pattern in C++ oder Smalltalk implementiert werden kann.

Bekannte Anwendungen Hier werden Beispiele aus tatsächlich existierenden Systemen aufgezeigt. Für jedes Design Pattern gibt es mindestens zwei Beispiele aus verschiedenen Anwendungsgebieten.

Bezug zu anderen Design Patterns Zu welchen anderen Patterns gibt es eine enge Beziehung? Wo liegen die Gemeinsamkeiten, wo die Unterschiede? Mit welchen anderen Design Patterns sollte das Pattern verwendet werden?

In [Gamma 95] wird Delegation als eine Programmiertechnik beschrieben, die es ermöglicht, das Verhalten von Objekten zur Laufzeit eines Programms (scheinbar) zu ändern.[16] Diese Technik kommt in verschiedenen Patterns in dem Buch zum Tragen. Da die verwendeten Programmiersprachen auf klassenbasierter Vererbung beruhen, die kein Sprachkonstrukt für Delegation bieten, muß diese Technik jedoch immer „von Hand“ programmiert werden.

Ein promimentes Beispiel ist das *Strategy Pattern*[17], das es ermöglicht, eine Familie von Algorithmen zu definieren, und die einzelnen Algorithmen zur Laufzeit eines Programms beliebig auszutauschen. Somit kann ein Algorithmus unabhängig von den Programmteilen variieren, in denen er verwendet wird. Ich verwende dieses Pattern beispielhaft in Kapitel 6, um zu zeigen, wie es in der von uns entwickelten Programmiersprache Lava implementiert werden kann.

[16] [Gamma 95], S. 20f
[17] [Gamma 95], S. 315ff

Kapitel 3

Java Konzepte

Die Programmiersprache Java ist eine kommerzielle Entwicklung der Firma Sun Microsystems und wird von der Tochterfirma JavaSoft vertrieben. Sie wurde erstmals 1995 der Öffentlichkeit präsentiert, die erste Version, die dem Alpha- und Betastadium entwachsen war, wurde 1996 fertiggestellt. Sie hat in der kurzen Zeit ihrer Existenz eine große Verbreitung gefunden und wurde von allen bedeutenden Softwareherstellern lizensiert.

Java ist aber nicht nur eine Programmiersprache, sondern steht auch für ein Ausführungsmodell, das es ermöglicht, plattformunabhängige und sichere Programme zu entwickeln. Plattformunabhängigkeit bedeutet in diesem Zusammenhang, daß das Kompilat einer Software unverändert unter verschiedenen Betriebssystemen und Hardwarearchitekturen lauffähig ist – daher können keine Annahmen über spezifische Eigenarten einer Plattform, wie z.B. die Größe eines Maschinenworts, vorausgesetzt werden und das Kompilat kann nicht in einer konkreten, sondern muß in einer verallgemeinerten Maschinensprache vorliegen.

Der Aspekt der Sicherheit nimmt im Zusammenhang mit Java eine wichtige Stellung ein, da Java u.a. für die verteilte Ausführung in Netzwerken, z.B. in Intranets oder dem Internet entwickelt wurde. Daher muß sichergestellt sein, daß die Ausführung von Java-Programmen nicht zu gewollten oder ungewollten Schäden führt. Bestimmte Sprachkonstrukte, wie sie in anderen Programmiersprachen üblich sind, wie z.B. Pointerarithmetik in C++, müssen daher von vorneherein ausgeschlossen werden.

Aus den genannten Gründen unterscheidet sich das Ausführungsmodell von klassischen Ansätzen. Der Übersetzer erzeugt einen plattformunabhängigen Zwischencode, den *Bytecode*, der in *Class Files* gespeichert wird. Eine Implementation der *Java Virtual Machine* (die Spezifikation des Laufzeitsystems von Java) ist in der Lage, solche Class Files zu laden, ggfs. über ein Netzwerk. Dabei werden auch die Aufgaben wahrgenommen, die traditionell von einem Linker gelöst werden. Zusätzlich kommt ein *Bytecode Verifier* zum Einsatz, der sicherstellt, daß der geladene Bytecode den Spezifikationen der Java Virtual Machine entspricht, um gewollte oder ungewollte Schäden auf der Plattform zu vermeiden, auf der ein Programm ausgeführt werden soll.

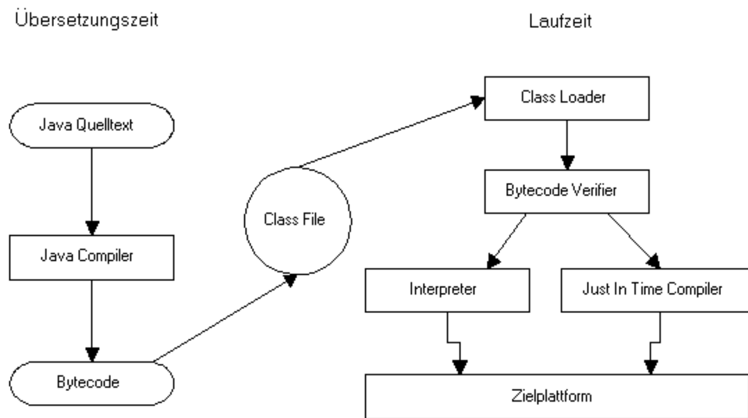

Abbildung 3.1: Das Ausführungsmodell von Java

In der Originalimplementation der Java Virtual Machine von Sun Microsystems kommt anschließend ein Interpreter zum Einsatz, der den Bytecode versteht. Alternativ kann aber auch ein *Just In Time Compiler* (*JIT*) gestartet werden, der maschinenabhängigen, direkt ausführbaren Code für die jeweilige Plattform erzeugt und diesen ausführt. Abbilung 3.1 veranschaulicht das soeben beschriebene Ausführungsmodell.

Im folgenden werde ich die Programmiersprache Java vollständig charakterisieren und auf die Besonderheiten des Ausführungsmodells und der *Java Virtual Machine*, oder kurz *JVM*, eingehen. Eine ausführlichere Einführung in die Konzepte findet sich in [Gosling 95] oder auf den WWW-Seiten von JavaSoft (http://java.sun.com). Eine Einführung in die Programmiersprache Java findet sich in [Arnold 96], die offizielle Spezifikation der Programmiersprache Java in [Gosling 96] und die offizielle Spezifikation der Java Virtual Machine in [Lindholm 96].[1]

3.1 Die Programmiersprache Java

Java ist eine allgemeine, klassenbasierte, objektorientierte Programmiersprache, die darüberhinaus Konstrukte für nebenläufige Prozesse und Ausnahmebehandlung bietet. Java hat viele Ähnlichkeiten zu C++, aber einiges wurde ausgelassen und einige Ideen aus anderen Programmiersprachen übernommen.

Java ist streng getypt im Sinne von Kapitel 2.4. Zu den Aktionen die zur Übersetzungszeit eines Programms ausgeführt werden, gehört die Übersetzung eines

[1] Der Java-Compiler und der Java-Interpreter werden zusammen mit einigen weiteren Werkzeugen kostenlos von JavaSoft zur Verfügung gestellt und kann z.B. über die o.a. WWW-Seiten bezogen werden. Das entsprechende Softwarepaket lautet *Java Development Kit* oder kurz *JDK*. Wir beziehen uns in unserer Arbeit auf die Version 1.0.2 des JDK; alle Verweise und Literaturangaben beziehen sich dementsprechend auch auf dieselbe Version.

Java-Programms in eine maschinenunabhängige Repräsentation, dem *Bytecode*. Aktionen, die zur Laufzeit stattfinden, sind u.a. das Laden und Linken von Klassen, ggfs. Erzeugung von weiterem Code, Optimierungen eines Programms und die eigentliche Ausführung eines Programms.

Details einer bestimmten Maschine, wie z.B. die Repräsentation von Datentypen, sind nicht in Java sichtbar. Eine automatische Speicherverwaltung, typischerweise in Form eines *Garbage Collectors*, wird vorausgesetzt, um die Probleme bei der expliziten Freigabe von benutztem Speicher zu vermeiden. Unsichere Sprachkonstrukte sind in Java nicht enthalten, wie z.B. Arrayzugriffe ohne Bereichsüberprüfung. Nebenläufige Prozesse mit gemeinsam benutztem Speicher werden unterstützt.

Typsystem: Java unterscheidet zwischen *primitiven Typen* und *Referenztypen*. Die primitiven Typen werden immer gleich definiert, unabhängig von Implementation und darunterliegender Plattform. Sie bestehen aus vorzeichenbehafteten ganzzahligen Typen in verschiedenen Größen, zwei Fließkommatypen mit unterschiedlicher Genauigkeit, ein boolscher Typ für Wahrheitswerte und ein Zeichentyp für den Unicode Zeichensatz.

Die Referenztypen von Java sind die *Klassentypen*, die *Interfacetypen* und die *Arraytypen*. Die Referenztypen werden von Objekten implementiert, die zur Laufzeit erzeugt werden, die entweder *Klasseninstanzen* oder *Arrayinstanzen* sind. Zu jedem einzelnen Objekt können mehrere Referenzen existieren. Alle Objekte (inklusive Arrays) stellen die Methoden der Standardklasse Object bereit. Diese Klasse ist die (einzige) Wurzel der Klassenhierarchie, alle anderen Klassen sind davon direkt oder indirekt abgeleitet. Eine vordefinierte Klasse String unterstützt Unicode Zeichenketten. Es existieren Standardklassen, um Werte primitiver Typen innerhalb von Objekten bereitzustellen.

Variablen sind typisierte Speicherbereiche. Eine Variable eines primitiven Typs enthält einen Wert exakt dieses primitiven Typs. Eine Variable eines Klassentyps kann eine null-Referenz (eine Referenz auf kein Objekt), oder eine Referenz auf eine Instanz dieses Klassentyps oder jeder Unterklasse dieses Klassentyps enthalten. Eine Variable eines Interfacetyps kann eine null-Referenz oder eine Referenz auf eine Instanz jeder Klasse enthalten, die dieses Interface explizit implementiert. Eine Variable eines Arraytyps kann eine null-Referenz oder eine Referenz auf ein Array enthalten. Eine Variable des Klassentyps Object kann eine null-Referenz oder eine Referenz auf eine beliebige Klasseninstanz oder ein Array enhalten.

Da Java eine streng typisierte Programmiersprache ist, wird zu jedem Ausdruck zur Übersetzungzeit der statische Typ des Ausdrucks bestimmt. Es besteht jedoch die Möglichkeit, den Typ eines Ausdrucks mit Hilfe einer *Konversion* explizit zu ändern, was ggfs. auch eine Änderung des Werts des Ausdrucks zur Folge hat. Ggfs. wird in einem numerischen Ausdruck implizit eine Konversion eines Typs vorgenommen, um den Wert des Ausdrucks bestimmen zu können. Konversionen von Referenztypen werden ggfs. zur Laufzeit überprüft, um Typsicherheit zu gewährleisten.

Deklarationen: Der Einfachheit halber wird in Java-Quelltexten kein Unterschied zwischen der Deklaration und der Implementation eines Typs gemacht. Es muß also kein separates Headerfile o.ä. zur Verfügung gestellt werden.

Java verlangt nicht, daß ein Typ oder dessen Elemente deklariert werden müssen, bevor sie verwendet werden. Die Reihenfolge von Deklarationen ist lediglich für lokale Variablen und für die Reihenfolge von statischen Initialisierungscodeblöcken[2] relevant.

Pakete: Ein Java-Programm ist in *Pakete* (*Packages*) gegliedert, die *Übersetzungseinheiten* und weitere Unterpakete enthalten können. Übersetzungseinheiten beinhalten Typdeklarationen und können Typen von anderen Paketen importieren. Pakete haben hierarchisch geordnete Namen, und die Namenskonventionen für Domains im Internet können verwendet werden, um weltweit eindeutige Namen zu erhalten.

Sichtbarkeit: Java ermöglicht es, den Gültigkeitsbereich von Namen mit Hilfe der Schlüsselwörter public, protected und private zu definieren und den externen Zugriff auf Elemente von Paketen, Klassen und Interfaces einzuschränken. Dabei definiert public einen Namen als gültig in allen anderen Klassen / Interfaces, protected nur in Unterklassen und private nur in der eigenen Klasse. Wird keines der drei Schlüsselwörter public, protected oder private angegeben, so wird damit standardmäßig eine paketweite Sichtbarkeit eines Namens definiert.

Klassen: Die *Elemente* einer Klasse sind *Felder* und *Methoden*. *Klassenvariablen* existieren einmal je Klasse. *Klassenmethoden* sind statisch gebunden und werden ohne Referenz auf ein spezifisches this-Objekt ausgeführt. *Instanzvariablen* werden zur Laufzeit bei der Erzeugung von Objekten erstellt. *Instanzmethoden* werden auf Instanzen von Klassen ausgeführt, die dann das jeweilge this-Objekt innerhalb der Instanzmethode darstellen.

Klassen erben immer von genau einer Oberklasse, entweder von einer explizit angegebenen Oberklasse oder implizit von der Klasse Object. Variablen eines Klassentyps können auf eine Instanz dieser Klasse oder einer beliebigen Unterklasse verweisen. In Unterklassen können Methoden der Oberklasse redefiniert werden, um so Polymorphie zu ermöglichen. Für jedes Objekt kann eine finalize-Methode bereitgestellt werden, die aufgerufen wird, bevor der Garbage Collector das Objekt freigibt, damit es entsprechende Aufräumarbeiten vornehmen kann.

Konstruktoren sind besondere Methoden, die nur bei der Erzeugung eines Objekts ausgeführt werden. Sie dienen dazu, die Felder eines Objekts zu initialisieren. Die Erzeugung eines Objekts ohne die Ausführung eines Konstruktors ist nicht möglich. Wird in einer Klasse kein Konstruktor deklariert, so erzeugt der Java-Compiler automatisch einen *Standardkonstruktor*, der keine Parameter erwartet und jede Instanzvariable mit einem Standardwert initialisiert. Jeder Konstruktor muß als erste Anweisung einen Aufruf eines weiteren Konstruktors enthalten. Dies kann entweder ein Konstruktor der gleichen Klasse sein, oder aber ein Konstruktor der direkten Oberklasse.[3] Somit ist gewährleistet, daß bei der Erzeugung eines Objekts als Instanz eines Klassentyps mindestens ein Konstruktor dieser Klasse und mindestens ein Konstruktor für jede Oberklasse dieser Klasse aufgerufen wird.

[2]Statischer Initialisierungscode wird einmal für jede Klasse / jedes Interface unmittelbar nach dem Laden des dazugehörigen Kompilats ausgeführt.

[3]Die einzige Ausnahme bildet der Konstruktor der Klasse Object – dieser enthält keinen weiteren Aufruf eines Konstruktors.

Soll die Instanziierung einer Klasse außerhalb der Klasse selbst verhindert werden, so muß mindestens ein Konstruktor deklariert werden, um die automatische Erzeugung des Standardkonstruktors zu verhindern, und die externe Verwendung aller Konstruktoren mit Hilfe des Schlüsselworts private verhindert werden.

Klassen unterstützen nebenläufige Programmierung mit Hilfe von synchronized Methoden, die sicherstellen, daß während ihrer Ausführung keine andere Methode auf dem gleichen this-Objekt ausgeführt wird.

Es gibt in Java keine parametrisierten Klassen, jedoch ist die Semantik von Arrays die einer parametrisierten Klassen. Bei schreibenden Arrayzugriffen werden zur Laufzeit Typchecks durchgeführt, um vollständige Typsicherheit zu gewährleisten. Darüberhinaus werden bei allen Arrayzugriffen Bereichsüberprüfungen vorgenommen. Arrays sind Objekte, die zur Laufzeit erzeugt werden, und die Variablen des Typs Object zugewiesen werden können. Es werden keine mehrdimensionalen Arrays untersützt, statt dessen können Arrays von Arrays gebildet werden.

Interfaces: Die Interfacetypen von Java deklarieren eine Menge *abstrakter Methoden* und *konstanter Werte*. Klassen, die sonst möglicherweise in keiner Beziehung zueinander stehen, können den gleichen Interfacetyp implementieren. Variablen eines Interfacetyps können auf Instanzen von Klassen verweisen, die diesen Interfacetyp explizit implementieren. Interfacetypen können voneinander erben, multiple Vererbung auf Interfaceebene ist möglich.

Ausnahmebehandlungen: *Ausnahmebehandlungen* sind vollständig in die Semantik von Java und die Mechanismen miteinbezogen, die Nebenläufigkeit ermöglichen. Es gibt drei Arten von Ausnahmen: *überprüfte Ausnahmen*, *Laufzeitausnahmen* und *Fehler*. Der Compiler stellt sicher, daß überprüfte Ausnahmen nur in Methoden ausgelöst werden können, die diese als mögliche Ausnahmen deklarieren. Daher kann zur Übersetzungszeit sichergestellt werden, daß Ausnahmesituationen tatsächlich behandelt werden. Unzulässige Operationen, die von der Java Virtual Machine entdeckt werden, lösen Laufzeitausnahmen aus, wie z.B. NullPointerException. Fehler werden durch die Nichtdurchführbarkeit prinzipiell möglicher Operationen ausgelöst, wie z.B. OutOfMemoryError.

Anweisungen und Ausdrücke: Die Anweisungskonstrukte in Java entsprechen weitgehend denen von C++. Es gibt kein goto, aber es existieren break- und continue-Anweisungen. Anweisungen zur Programmflußkontrolle (if, while, etc.) erwarten boolsche Ausdrücke in ihren Bedingungen; Ausdrücke anderer Typen werden nicht implizit zum boolschen Typ konvertiert. Die synchronized-Anweisung stellt eine Kontrolle über die Ausführung nebenläufiger Prozesse auf Objektebene bereit. Eine try-Anweisung kann catch- und finally-Klauseln enthalten, um die Ausführung von wichtigem Code auch in Ausnahmesituationen zu gewährleisten.

In Java ist vollständig festgelegt, in welcher Reihenfolge Ausdrücke ausgewertet werden. Die Verwendung überladener Methoden und Konstruktoren wird zur Übersetzungszeit aufgelöst, indem jeweils die spezifischste Methode / der spezifischste Konstruktor von allen möglichen ausgewählt wird. Lokale Variablen werden im Gegensatz zu allen anderen Variablen nicht zur Laufzeit mit einem

Standardwert initialisiert. Aus diesem Grund wird zur Übersetzungszeit sichergestellt, daß vor einem lesenden Zugriff auf eine lokale Variable ein schreibender Zugriff stattgefunden hat.

Übersetzung und Ausführung: Eine Java-Programm wird normalerweise in Binärdateien gespeichert, die übersetzte Klassen und Interfaces repräsentieren. Diese Binärdateien können in eine Java Virtual Machine geladen, zu anderen Klassen und Interfaces gelinkt und initialisiert werden.

Nach der Initialisierung können Klassenmethoden und Klassenvariablen verwendet werden. Manche Klassen können instanziiert werden, um neue Objekte des jeweiligen Klassentyps zu erzeugen. Klasseninstanzen enthalten alle Instanzvariablen, die in der jeweiligen Klasse und in allen Oberklassen dieser Klasse definiert sind.

Objekte, auf die nicht mehr verwiesen wird, können vom Garbage Collector freigegeben werden. Wenn eine finalize-Methode von dem Objekt bereitgestellt wird, wird diese vorher ausgeführt. Wenn eine Klasse nicht mehr in Gebrauch ist, kann sie aus der Java Virtual Machine entfernt werden. Wenn sie die Klassenmethode classFinalize bereitstellt, wird diese vorher ausgeführt.

Damit ist die Programmiersprache Java vollständig charakterisiert. Auf Details, die für das Sprachdesign von Lava wichtig sind, werde ich in Kapitel 5 gesondert eingehen. Die Originalimplementation des Java-Compilers von Sun Microsystems werde ich in Kapitel 8 beschreiben.

3.2 Die Java Virtual Machine

Die Java Virtual Machine (JVM) ist abstrakt in dem Sinne, daß es sich lediglich um die Spezifikation handelt, die nicht vorschreibt, wie sie implementiert werden muß. Zum Beispiel handelt es sich bei den Maschinenbefehlen, die von der JVM spezifiziert werden, um abstrakte Befehle, es wird aber nicht festgelegt, ob diese Befehle von einer Implementation der JVM interpretiert, in Maschinencode der jeweiligen Zielplattform übersetzt, oder aber direkt von einer spezialisierten Hardware ausgeführt werden.

Eine Implementation der JVM muß in der Lage sein, das *Class File Format* einzulesen und die darin festgelegten Operationen korrekt auszuführen. Das Class File Format spezifiziert, wie eine übersetzte Java-Klasse oder ein übersetztes Java-Interface aussieht. Darin ist der bereits erwähnte Bytecode enthalten, der aus der Deklaration einer Klasse / eines Interfaces und den übersetzten Methoden einer Klasse besteht.

Typsystem: Wie in der Programmiersprache Java, existieren auch in der JVM zwei Arten von Typen, *primitive Typen* und *Referenztypen*. Die JVM erwartet, daß nahezu alle Typchecks zur Übersetzungszeit eines Java-Programms durchgeführt wurden. Insbesondere enthalten Daten keine Typinformationen in Form von Tags oder dergleichen. Stattdessen unterscheidet der Befehlssatz der JVM zwischen Befehlen, die auf unterschiedlichen Typen arbeiten. Zum Beispiel existieren vier verschiedene Befehle, um Zahlen zu addieren, die nach dem Typ der Operanden unterschieden sind.

An primitiven Typen werden byte, short, int, long, char, float und double zur Verfügung gestellt, die den entsprechenden primitiven Datentypen in Java entsprechen. Darüberhinaus gibt es den Typ returnAddress, dessen Werte Verweise auf Befehle einer Java-Klasse darstellen. Dieser Typ hat keine Entsprechung in der Programmiersprache Java. Es existiert kein boolscher Typ in der JVM, stattdessen wird der Typ int verwendet, um Wahrheitswerte zu repräsentieren.

Die JVM bietet explizite Unterstützung für Objekte, wobei ein Objekt entweder eine Klasseninstanz oder ein Array ist, das zur Laufzeit eines Programms erzeugt wird. Ein Verweis auf ein Objekt hat in der JVM den Typ reference. Einen solchen Verweis kann man sich als Zeiger auf ein Objekt vorstellen, zu jedem Objekt können mehrere Verweise existieren. Obwohl die JVM Befehle bereitstellt, die auf Objekten arbeiten, werden Objekte niemals direkt adressiert, sondern immer nur über Werte vom Typ reference. Wie Verweise auf Objekte in einer konkreten Implementation der JVM realisiert werden, ist nicht festgelegt und für den Code eines Java-Programms unsichtbar.

Der Typ reference wird in der JVM nochmal unterschieden in Klassen-, Array- und Interfacetypen, dessen Werte auf Klasseninstanzen oder Arrays verweisen, oder aber auf Klasseninstanzen oder Arrays, die Interfaces implementieren. Es gibt auch den speziellen Wert null, der für einen Verweis auf kein Objekt steht und zu keinem der drei Referenztypen gehört, aber zu jedem dieser drei Referenztypen konvertiert werden kann.

Constant Pool: Der *Constant Pool* existiert für jede Klasse / Interface einmal und enthält verschiedene konstante Werte, von zur Übersetzungszeit bekannten numerischen Werten bis hin zu Methoden- und Feldreferenzen, die zur Laufzeit aufgelöst werden müssen. Der Constant Pool entspricht in etwa der Symboltabelle, die in konventionellen Programmiersprachen zum Einsatz kommen, es werden hier aber wesentlich mehr Informationen bereitgestellt.

Ausführung von Methoden: Eine JVM kann die Ausführung nebenläufiger Prozesse unterstützen. Jeder Prozeß hat einen eigenen Programmzähler, das sogenannte pc-Register. Zu jedem Zeitpunkt führt jeder Prozeß den Code einer einzelnen Methode aus, namentlich die *aktuelle Methode*. Das pc-Register enthält jeweils die Adresse des Befehls der JVM, der aktuell ausgeführt wird, wobei es sich um Werte vom Typ returnAddress handeln kann.[4]

Jeder Prozeß besitzt einen eigenen *Java-Stack*, der sogenannte *Frames* speichert. Die Java-Stacks entsprechen den Stacks konventioneller Programmiersprachen, wie z.B. C++. Ein Java-Stack kann zusammenhängend oder nicht-zusammenhängend sein, und seine Größe kann statisch festgelegt sein oder dynamisch zur Laufzeit angepaßt werden. Bei statisch festgelegter Größe kann ggfs. ein StackOverflowError, bei dynamisch anpaßbarer Größe ein OutOfMemoryError ausgelöst werden.

Ein *Frame* speichert Daten, Teilergebnisse und ggfs. Debug-Informationen einer aktuellen Methode. Ein neuer Frame wird jedesmal erzeugt, wenn eine Methode aufgerufen wird, und freigegeben, wenn die zugehörige Methode beendet wird.

[4] Es werden ggfs. auch *native* Methoden unterstützt, die nicht die Befehle der JVM verwenden, sondern in einer plattformabhängigen Maschinensprache vorliegen. Für diesen Fall stimmen die hier genannten Ausführung nicht unbedingt. Dieser Fall ist jedoch für meine Arbeit nicht von Bedeutung und wird daher nicht weiter beschrieben.

Frames werden im Java-Stack des Prozesses gespeichert, der den Frame erzeugt. Ein Frame kann immer vollständig erzeugt werden, da die Größen aller lokalen Variablen und des Operandenstacks zur Übersetzungszeit einer Methode bekannt sind. Der Frame der aktuellen Methoden wird *aktueller Frame* genannt, die Klasse, in der die aktuelle Methode definiert ist, wird *aktuelle Klasse* genannt. Ein Frame ist nicht mehr aktuell, wenn die aktuelle Methode eine andere aufruft oder aber die aktuelle Methode beendet wird.

Ein Frame enthält einen Verweis auf den Constant Pool der aktuellen Klasse, um *dynamisches Linken* zu ermöglichen. Der Code einer Methode verwendet symbolische Referenzen, um auf weitere Methoden und Variablen zuzugreifen. Diese symbolischen Referenzen werden erst zur Laufzeit eines Programms in konkrete Methodenadressen oder in konkrete Offsets in den jeweiligen Speicherstrukturen umgesetzt. Für ein Java-Programm sind die zu Grunde liegenden Methodentabellen oder Speicherstrukturen unsichtbar, und die enstprechenden Adressen, Indizes oder Offsets können nicht im Code eines Java-Programms ermittelt oder verwendet werden.

Wenn eine Methode regulär beendet wird, so kann sie durch eine *return*-Anweisung einen Wert auf dem Operandenstack der aufrufenden Methoden ablegen. Wird eine Methode nicht regulär beendet, so wird kein Wert auf dem Operandenstack der aufrufenden Methode abgelegt. Dieser Fall tritt auf, wenn die Ausführung eines Befehls innerhalb der Methode eine Ausnahme auslöst, die nicht in der gleichen Methode behandelt wird. Eine Ausnahme kann auch explizit durch eine throw-Anweisung ausgelöst werden, was dieselbe Wirkung hat.

Konstruktoren: Auf der Ebene der JVM entspricht jeder Konstruktor einer Klasse einer Methode mit dem speziellen Namen <init>, der innerhalb eines Java-Programms nicht bekannt ist.[5] Diese speziellen Methoden werden im Code nach der Erzeugung eines jeden Objekts explizit aufgerufen. Der Initialisierungscode für eine Klasse befindet sich in einer Methode mit dem speziellen Namen <clinit>, für den ähnliches wie für <init> gilt, jedoch wird diese Methode niemals explizit aufgerufen, sondern implizit nach dem Laden einer Klasse.

Speicherorganisation: Neben dem bereits erwähnten Java-Stack besitzt die JVM einen *Heap*, der alle Objekte und Arrays eines Programms speichert und von allen Prozessen gemeinsam genutzt wird. Objekte oder Arrays werden niemals explizit von einem Programm freigegeben, sondern die JVM kümmert sich selbst um die Freigabe von nicht mehr benötigten Objekten. Dabei kommt typischerweise ein *Garbage Collector* zum Einsatz. Die Größe des Heaps kann statisch festgelegt sein oder dynamisch zur Laufzeit angepaßt werden. Ggfs. kann ein OutOfMemoryError ausgelöst werden.

Die JVM besitzt einen *Methodenbereich*, der von allen Prozessen gemeinsam genutzt wird. Er entspricht z.B. dem Textsegment eines UNIX-Prozesses. Er enthält Daten, die einmal für jede Klasse / jedes Interface existieren, wie z.B. den Constant Pool, Felder und Methoden, sowie den Code für Methoden und Konstruktoren inklusive speziellen Methoden zur Initialisierung von Klassen, Interfaces und Instanzen. Die Größe des Methodenbereichs kann statisch festgelegt sein oder dynamisch zur Laufzeit angepaßt werden. Ggfs. kann ein OutOfMemoryError ausgelöst werden.

[5] Wohl aber im Bytecode einer Java-Klasse!

Befehle: Ein Befehl der JVM besteht immer aus einem Byte, dem *Opcode*, der die Operation spezifiziert, die ausgeführt werden soll, gefolgt von keinem oder mehreren *Operanden*. Der Befehlssatz umfaßt Befehle für verschieden Aufgaben:

Lade- und Speicherbefehle um lokale Variablen auf den Operandenstack zu laden; um Werte des Operandenstacks in lokale Variablen zu speichern; um Konstanten auf den Operandenstack zu laden.

Arithmetische Befehle um Werte vom Operandenstack zu holen, einen Wert zu berechnen und diesen Wert zurück auf den Operandenstack zu laden. Es existieren Befehle für Addition, Subtraktion, Multiplikation, Division, Modulo-Berechnung, Negation, Bildung einer Zweier-Potenz (Shift), bitweises Oder, bitweises Und, bitweises Exklusiv-Oder, sowie Inkrementierung lokaler Variablen.

Typkonversions-Befehle für die Konversion zwischen verschiedenen primitiven Typen.

Objekt-Befehle um neue Klasseninstanzen oder Arrayinstanzen zu erzeugen; um auf Klassen- und Instanzvariablen oder Arraykomponenten zuzugreifen (Laden auf den Operandenstack / Speichern vom Operandenstack); um die Länge eines Arrays zu ermitteln; um die Zugehörigkeit eines Objekts oder eines Arrays zu einer bestimmen Klassen zu überprüfen.

Operandenstack-Befehle um Werte auf dem Operandenstack zu löschen, zu duplizieren oder zu vertauschen.

Sprung-Befehle für bedingte oder unbedingte Sprünge an Adressen von Befehlen innerhalb einer aktuellen Methode.

Methodenaufruf- und Return-Befehle um Instanzmethoden einer Klasse oder eines Interfaces aufzurufen; um Klassenmethoden aufzurufen; um die aktuelle Methode zu beenden und zur Aufrufstelle in der aufrufenden Methode zurückzukehren und dabei ggfs. einen Wert auf den Operandenstack der aufrufenden Methoden zu laden.

Befehle zur Synchronisation um gleichzeitige Zugriffe nebenläufiger Prozesse auf bestimmte Objekte zu kontrollieren.

Class Files: Übersetzter Code, der von einer Java Virtual Machine ausgeführt werden soll, wird in einer Binärdatei mit einem plattformunabhängigen Format, dem *Class File Format*, gespeichert. Dieses Format spezifiziert eindeutig den Inhalt einer solchen Datei, inklusive Details wie z.B. die Reihenfolge, in der Bytes gespeichert werden, was in einem plattformabhängigen Format selbstverständlich wäre und nicht weiter spezifiziert werden müßte.

Für jede Klasse und für jedes Interface existiert genau ein Class File. Das Class File enthält folgende Informationen:

Constant Pool String-Konstanten, Klassennamen, Elementnamen und andere Konstanten, die im Class File verwendet werden.[6]

[6] Wenn im weiteren Verlauf des Texts die Rede davon ist, daß eine Struktur des Class Files einen Namen enthält, so enthält die Struktur tatsächlich nur einen Index in den Constant Pool des Class Files.

Access Flags Handelt es sich um eine Klasse oder ein Interface? Kann die Klasse / das Interface außerhalb ihres Pakets verwendet werden?[7] Dürfen Unterklassen von der Klasse abgeleitet werden?[8] Dürfen Instanzen vom Typ der Klasse / des Interfaces erzeugt werden?[9]

Klassenname

Name der Oberklasse

Namen der implementierten Interfaces

Felddefinitionen Es werden nur Felder (Klassen- und Instanzvariablen) aufgeführt werden, die genau in dieser Klasse / diesem Interface deklariert sind. Felder, die von der Oberklasse oder den implementierten Interfaces geerbt werden, werden nicht aufgeführt.

Methodendefinitionen Es werden nur Methoden (Klassen- und Instanzmethoden) aufgeführt, die genau in dieser Klasse / diesem Interface deklariert sind. Methoden, die von der Oberklasse oder den implementierten Interfaces geerbt werden, werden nicht aufgeführt, es sei denn, es handelt sich um Redefinitionen.

Attribute Diese benannten Attribute können weitere Informationen über eine Klasse enthalten, wobei eine Implementation einer JVM eigene Attribute mit eigenen Namen festlegen kann.[10] Die Semantik eines Class Files darf von solchen Attributen jedoch nicht geändert werden, es darf sich lediglich um beschreibende Informationen (z.B. für Debugger) handeln.

Felddefinitionen: Jede Felddefinition enthält folgende Informationen:

Access Flags In welchen anderen Klassen kann das Feld verwendet werden?[11] Handelt es sich um eine Klassen- oder um eine Instanzvariable? Darf dem Feld nach seiner Initialisierung ein neuer Wert zugewiesen werden?[12] Darf das Feld in einem Cache gehalten werden?[13] Soll es von einem System berücksichtigt werden, das persistente Objekte ermöglicht?[14]

Name des Felds

Typ des Felds Typen werden in Form von *Deskriptoren* gespeichert – das sind Strings, die einer im Class File Format spezifizierten Grammatik entsprechen.

---

[7] Entsprechend der Annotation public.
[8] Entsprechend der Annotation final.
[9] Entsprechend der Annotation abstract.
[10] Die Namenskonventionen für Domains im Internet können verwendet werden, um weltweit eindeutige Namen zu erhalten.
[11] Entsprechend den Annotationen public, protected oder private.
[12] Entsprechend der Annotation final.
[13] Entsprechend der Annotation volatile.
[14] Entsprechend der Angabe transient.

Attribute Diese benannten Attribute können weitere Informationen über ein Feld enthalten, wobei eine Implementation einer JVM eigene Attribute mit eigenen Namen festlegen kann.[15] Die Spezifikation der JVM legt das Attribut ConstantValue fest, das den Initialisierungswert einer Klassenvariable enthält.[16] Die Semantik eines Felds darf von allen übrigen Attributen nicht geändert werden, es darf sich lediglich um beschreibende Informationen handeln.

Methodendefinitionen: Jede Methodendefinition enthält folgende Informationen:

Access Flags In welchen anderen Klassen kann die Methode verwendet werden?[17] Handelt es sich um eine Klassen- oder um eine Instanzmethode? Darf die Methode in einer Unterklasse redefiniert werden?[18] Soll das this-Objekt während der Ausführung der Methode für andere Prozesse gesperrt werden?[19] Verwendet der Code der Methode die Befehle der JVM, oder liegt er in einer plattformabhängigen Maschinensprache vor?[20] Liegt eine Implementation der Methode vor?[21]

Name der Methode

Typ der Methode Typen werden in Form von *Deskriptoren* gespeichert – das sind Strings, die einer im Class File Format spezifizierten Grammatik entsprechen.

Attribute Diese benannten Attribute können weitere Informationen über ein Feld enthalten, wobei eine Implementation einer JVM eigene Attribute mit eigenen Namen festlegen kann.[22] Die Spezifikation der JVM legt die Attribute Code und Exceptions fest. Das Attribut Code enthält eine Folge von Maschinenbefehlen, die bei Aufruf der Methode ausgeführt werden, sowie weitere Angaben, die für die Ausführung einer Methode wichtig sind, wie z.B. die maximale Größe des Operandenstacks. Das Attribut Exceptions enthält die Namen der geprüften Ausnahmen, die von der Methode ausgelöst werden können. Die Semantik einer Methode darf von allen übrigen Attributen nicht geändert werden, es darf sich lediglich um beschreibende Informationen handeln.

Sicherheitsaspekte: Die Spezifikation der JVM legt Einschränkungen fest, die von einem Class File strikt erfüllt werden müssen. Es handelt sich dabei um Einschränkungen, die sicherstellen sollen, daß ausschließlich korrekte Class Files von einer JVM ausgeführt werden. Beispielsweise wird festgelegt, daß das Ziel eines Sprungbefehls innerhalb der aktuellen Methode liegt und den Beginn eines

[15] Die Namenskonventionen für Domains im Internet können verwendet werden, um weltweit eindeutige Namen zu erhalten.

[16] Bei Instanzvariablen kann dieses Attribut demnach nicht verwendet werden.

[17] Entsprechend den Annotationen public, protected oder private.

[18] Entsprechend der Annotation final.

[19] Entsprechend der Annotation synchronized.

[20] Entsprechend der Annotation native.

[21] Entsprechend der Annotation abstract.

[22] Die Namenskonventionen für Domains im Internet können verwendet werden, um weltweit eindeutige Namen zu erhalten.

weiteren Befehls darstellt, oder daß die Zahl und Typen der Operanden eines Befehls den Anforderungen eines Befehls entsprechen, etc. pp.

Nachdem eine JVM ein Class File geladen hat, aber noch bevor dieses Class File gelinkt und benutzt wird, kann es von einem Bestandteil der JVM, dem *Bytecode Verifier*, auf die Einhaltung der Einschränkungen überprüft werden. Wird einer der Einschränkungen von einem Class File nicht erfüllt, so wird es vom Bytecode Verifier verworfen und aus dem Speicher entfernt. Somit ist sichergestellt, daß fehlerhafte Class Files nicht benutzt und deren Methoden nicht ausgeführt werden.

Damit sind die für das Sprachdesign von Lava und den Lava-Compiler wesentlichen Aspekte der Java Virtual Machine charakterisiert. Auf Details werde ich an den entsprechenden Stellen in den Kapiteln 7 und 8 eingehen. Weitere Details sowie eine Beschreibung der Originalimplementation der JVM von Sun Microsystems finden sich in [Schickel 97].

Teil II

Lava: Java += Delegation

Kapitel 4

Aufgaben

In diesem Kapitel gebe ich einen Überblick über die Aufgaben, die gelöst werden müssen, um objektbasierte Vererbung in Java zu ermöglichen.[1]

Ursprünglich hatten wir geplant, eine eigene Sprache namens Erasmus zu entwerfen und für diese Sprache ein vollständiges System aus Compiler und Laufzeitumgebung zu implementieren. Im Laufe der Planungen waren wir auf die zum damaligen Zeitpunkt noch im Betastadium befindliche Programmiersprache Java aufmerksam geworden. Nach eingehenderem Studium von Java waren wir zu dem Schluß gekommen, daß sie ideale Voraussetzungen für unsere Arbeit bietet und hatten daher beschlossen, statt einer eigenen Sprache Erweiterungen für Java zu entwerfen.

Wesentliche Entwurfsentscheidungen, die wir für Erasmus gefällt hatten, konnten wir relativ leicht auf Java übertragen. Da wir darüberhinaus die Quelltexte von Java zur Verfügung haben, konnten wir uns bei der Implementation unserer Konzepte auf die für objektbasierte Vererbung wesentlichen Erweiterungen konzentrieren.

<u>Aufgabenteilung:</u> Das Ausführungsmodell von Java läßt sich in die beiden Bereiche Compiler und Laufzeitsystem unterteilen, die mit der Übersetzungszeit und der Laufzeit eines Java-Programms korrespondieren. Die einzige Schnittstelle zwischen diesen Bereichen ist der Bytecode, dessen Struktur im Class File Format spezifiziert ist.

Damit ergab sich eine natürliche Aufgabenteilung: Auf der einen Seite mußte die Syntax und Semantik der Programmiersprache Java erweitert, sowie der Java-Compiler geändert werden, um für die neue Syntax und Semantik korrekten Code zu erzeugen. Auf der anderen Seite haben wir uns entschieden, Erweiterungen an der Spezifikation der Java Virtual Machine vorzunehmen, so daß sie

[1] Die Ideen für die Implementation von Lava basieren auf [Kniesel 95]. Der darin enthaltene Vorschlag, wie klassen- und objektbasierte Vererbung in einer typisierten Programmiersprache vereint werden kann, orientiert sich an den Techniken zur Implementation klassenbasierter Programmiersprachen mit traditionellem Ausführungsmodell, einige wesentliche Punkte ließen sich aber dennoch auf Java übertragen.

objektbasierte Vererbung explizit unterstützt. Daher mußte auch die uns vorliegende Implementation der Java Virtual Machine geändert werden, um die neue Spezifikation korrekt umzusetzen.

Das Sprachdesign und die Änderung des Compilers wurde von mir übernommen, während Matthias Schickel sich um die Änderung und Erweiterung der Java Virtual Machine gekümmert hat. Für das Resultat unserer Arbeit wählten wir den Namen *Lava*, der analog zu Java sowohl das Gesamtsystem, als auch die Programmiersprache Lava bezeichnet. Die Erweiterung der Java Virtual Machine heißt dementsprechend *Lava Virtual Machine*, oder kurz *LVM*.

Kompatibiliät: Ein wesentliches Ziel unserer Arbeit war es, daß Lava möglichst kompatibel zu Java ist, um bestehende Klassenbibliotheken nutzen zu können und eine größere Akzeptanz von Lava bei Programmierern zu erzielen. Das bedeutet insbesondere, daß Java-Quelltexte möglichst nicht verändert werden müssen, um vom Lava-Compiler übersetzt zu werden, und daß Java Class Files unverändert unter der LVM, und damit auch in Lava-Programmen verwendbar sind; außerdem sollten Lava Class Files, die die neuen Eigenschaften von Lava nicht verwenden, ebenfalls unverändert unter Implementierungen der JVM lauffähig sein.

Übersetzungszeit: Die Erweiterungen in der Sprache Lava umfassen zunächst ein Konstrukt, daß es Programmierern ermöglicht, für gegebene Objekte Elternobjekte festlegen zu können. Das reicht jedoch nicht aus, um objektbasierte Vererbung sinnvoll verwenden zu können, sondern es müssen weitere Sprachkonstrukte definiert werden, mit denen die resultierenden Aufgaben und Probleme bewältigt werden können. Für alle Sprachkonstrukte muß die Syntax und Semantik genau bestimmt werden, ohne die Semantik der bereits existierenden Sprachkonstrukte von Java zu ändern.

Die Erweiterungen in der Lava Virtual Machine, insb. dem Class File Format, korrespondieren zu den neuen Sprachkonstrukten von Lava. Das Format und die Semantik der Erweiterungen muß ebenfalls genau bestimmt werden, ebenfalls ohne die Semantik des existierenden Java Class File Formats zu ändern, und insb. ohne die Sicherheitsaspekte von Java zu verletzen.

Bei der Implementation des Lava-Compilers konnte auf die bestehende Implementation des Java-Compilers von Sun Microsystems zurückgegriffen werden: Die neuen Sprachkonstrukte mußten in den bestehenden Compiler eingeflochten werden, und anhand der definierten Semantik von Lava mußte sichergestellt werden, daß korrekter Code gemäß der LVM Spezifikation erzeugt wird.

Laufzeit: Die Konzeptionierung und eine Implementierung der Lava Virtual Maschine, sowie Optimierungsmöglichkeiten werden in [Schickel 97] vorgestellt.

Kapitel 5

Sprachdesign

Ich gehe jetzt ausführlich auf alle Erweiterungen der Sprache Java ein, die beim Design von Lava vorgenommen wurden. Für eine knappe Darstellung der Syntax von Lava verweise ich auf Anhang A.

5.1 Delegation und Konsultation

In Kapitel 2.3 habe ich den Begriff Delegation eingeführt, der beschreibt, was bei der Weiterleitung einer Nachricht von einem Objekt, das diese Nachricht nicht versteht, zu einem Elternobjekt passiert. Wesentlich dabei ist, daß die Bindung des this-Bezeichners an den ursprünglichen Empfänger der Nachricht erhalten bleibt, so daß eine weitere Nachricht an this innerhalb der ausgeführten Methode im Elternobjekt wieder zurück an diesen gesendet wird. (s. Abb. 5.1)

Es gibt das alternative Verfahren *Konsultation*, Elternobjekte zur Behandlung von Nachrichten zu Rate zu ziehen: Das Schema ähnelt dem der Delegation, allerdings wird der this-Bezeichner bei der Weiterleitung einer Nachricht an ein Elternobjekt an dieses gebunden. Eine weitere Nachricht an this wird daher unmittelbar an das Elternobjekt gesendet, und nicht an den ursprünglichen Empfänger der ersten Nachricht. (s. Abb. 5.2)

M.a.W. handelt es sich also bei Konsultation schlicht um ein neues Versenden einer Nachricht an ein Elternobjekt, was in Programmiersprachen mit klassen-

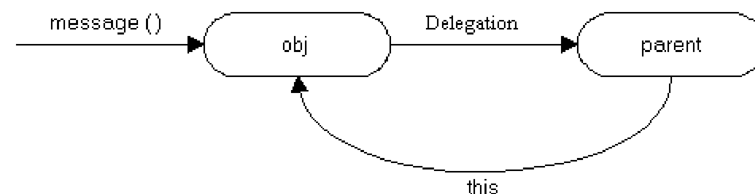

Abbildung 5.1: Delegation: Die Bindung von this bleibt erhalten.

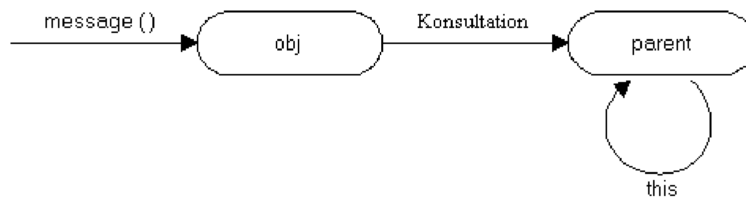

Abbildung 5.2: Konsultation: this wird an das Elternobjekt gebunden.

basierter Vererbung ohne Probleme von Hand programmiert werden kann.[1] Im nächsten Abschnitt wird jedoch ein Sprachkonstrukt in Lava eingeführt, das die Programmierung von Konsultationsmechanismen vereinfacht; insb. muß damit das Neuversenden von Nachrichten an Elternobjekte nicht explizit programmiert werden.

Im folgenden werde ich den Begriff Delegation in der Regel als Oberbegriff sowohl für Delegation als auch für Konsultation verwenden, da diese Mechanismen ähnlich eingesetzt werden können. Wenn Unterschiede vorliegen, werde ich gesondert darauf hinweisen.

5.2 Deklaration von Elternverweisen

Verweise auf Elternobjekte unterscheiden sich zunächst einmal nicht von gängigen Referenzen auf Objekte in der Sprache Java. Insb. das Ändern eines Elternobjekts in einem Kindobjekt kann als einfache Zuweisung an ein Feld des Kindobjekts aufgefaßt werden, das ein Verweis auf ein Elternobjekt darstellt. Da das Typsystem von Java statisch ist, und dies in Lava beibehalten werden soll, darf sich die Eigenschaft, welches Feld in einem Objekt ein Verweis auf ein Elternobjekt darstellt, nicht zur Laufzeit eines Programms ändern. Daher muß die Festlegung, welche Felder Verweise auf Elternobjekte sind, bei der Deklaration einer Klasse vorgenommen werden.

Zu diesem Zweck werden den in Java bekannten Annotationen für Felder die Schlüsselwörter delegatee und consultee hinzugefügt.

5.2.1 delegatee Felder

Ein Feld kann wie folgt als delegatee deklariert werden:

```
class childClass {
  delegatee parentClass parent; // "parent" verweist auf
                                // ein Elternobjekt vom Typ
                                // "parentClass" in Instanzen
                                // von "childClass"
}
```

[1] Aus diesem Grund habe ich den Begriff Konsultation bisher noch nicht eingeführt.

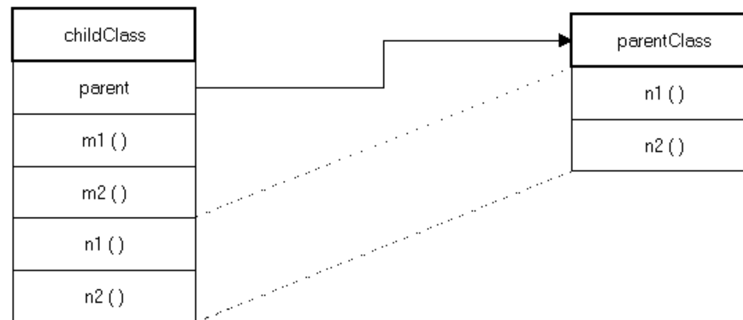

Abbildung 5.3: delegatee Felder: childClass wird um die Elemente von parentClass erweitert.

Die Annotation delegatee läßt sich nur bei Feldern verwenden, die von einem Klassentyp sind. An Felder von Interface- oder Arraytypen, sowie an Felder primitiver Typen kann nicht delegiert werden.

Semantik

Durch die Annotation delegatee wird die Schnittstelle der Klasse childClass (in der das Feld parent deklariert ist) um die public und protected Elemente des Typs parentClass erweitert. Darüberhinaus wird sie um paketweit sichtbare Elemente von parentClass erweitert, wenn childClass und parentClass im gleichen Paket deklariert sind. (s. Abb. 5.3)

In folgendem Beispiel:

```
class childClass {
  delegatee private parentClass parent; // Verweis auf ein Elternobjekt
}
```

wird demnach childClass um alle Elemente erweitert, die in der Deklaration von parentClass public, protected oder ggfs. paketweit sichtbar sind.

Die zusätzlichen Elemente behalten in childClass den Sichtbarkeitsstatus, den sie in parentClass haben, d.h. public Elemente von parentClass sind auch in childClass public, usw. Der Sichtbarkeitsstatus des Felds parent bezieht sich nur auf dieses Feld selbst, nicht auf die darüber „geerbten“ Elemente des Typs parentClass.

Die Alternative, daß die geerbten Elemente den Sichtbarkeitsstatus des Felds parent erhalten, wurde aus folgendem Grund verworfen: Wenn das delegatee Feld parent in obigem Beispiel den Sichtbarkeitsstatus private hätte, würden alle darüber geerbten Elemente ebenfalls den Sichtbarkeitsstatus private erhalten, und wären somit nicht in Unter- oder Kindklassen von childClass sichtbar. Somit würde keine objektbasierte Vererbung vorliegen, sondern lediglich eine neue Möglichkeit des Imports von Elementen aus anderen Klassen.

Die Semantik der delegatee-Annotation würde also von den Sichtbarkeitsannotationen eines Felds abhängig sein, was verwirrend sein kann. Insb. wenn die

Sichtbarkeit eines delegatee Felds während der Entwicklung eines Programms nachträglich geändert wird, würde dadurch die Sichtbarkeit anderer Elemente der Klasse ebenfalls geändert, was sich wiederum auf Klienten der Klasse auswirkt. Dies ist in der Regel nicht wünschenswert.

Multiplizität

In einer Klassendeklaration können mehrere Felder mit der Annotation delegatee versehen werden. Auf die Auflösung ggfs. resultierender Namenskonflikte wird in Kapitel 5.5 eingegangen. Im folgenden wird stets davon ausgegangen, daß sich keine Namenskonflikte ergeben, bzw. daß sie mit Hilfe der in Kapitel 5.5 beschriebenen Konstrukte aufgelöst werden.

Andere Annotationen

Es lassen sich nur Instanzvariablen als Referenzen auf Elternobjekte verwenden, daher kann delegatee nicht zusammen mit der Annotation static verwendet werden, mit der in Java Klassenelemente deklariert werden. Auch kann delegatee nicht zusammen mit consultee[2] verwendet werden.

Mit allen anderen bekannten Annotationen von Java (public, protected, private, final, transient und volatile) läßt sich delegatee verwenden.

Feldzugriffe

Wenn in einer Instanz aChild der Klasse childClass auf ein Feld aField zugegriffen wird, das über das delegatee Feld parent vom Typ parentClass geerbt wird, so wird das entsprechende Feld des Objekts adressiert, das an parent gebunden ist.

In folgendem Beispiel:

```
public static void main(String[] args) {
  childClass aChild; // Variable vom Typ "childClass"
  ...
  System.out.println(aChild.aField);
}
```

wird der Wert des Felds aChild.parent.aField ausgegeben.[3]

Es kann sein, daß aField im durch parent referenzierten Objekt ebenfalls aus einem weiteren Elterntyp geerbt wird. Dann wird das Feld des entsprechenden Elternverweises aus aChild.parent adressiert. Diese Semantik gilt transitiv für alle weiteren Objekte innerhalb einer Elternhierarchie.

[2] Siehe Kapitel 5.2.2.

[3] Dies gilt nur, wenn das Feld aField nicht in childClass oder einer Oberklasse von childClass ebenfalls deklariert ist. Ist dies doch der Fall, so handelt es sich ggfs. um einen Namenskonflikt, worauf in Kapitel 5.5 näher eingegangen wird.

Verstecken von Feldern

Eine Redefinition von Feldern ist in Kindklassen nicht möglich, da auch in Java kein Konzept für die Redefinition von Feldern in Unterklassen existiert. Stattdessen wird in Java durch die Definition eines Felds ein Feld mit gleichem Namen der Oberklasse *versteckt*. Dabei kann es sich um Felder verschiedener oder gleicher Typen handeln.[4] Das hat zur Folge, daß auf das Feld der Oberklasse nicht mehr mit diesem Namen zugegriffen werden kann.

In folgendem Beispiel:

```
class superClass {
  int aField;
}

class aClass extends superClass {
  float aField;

  public static void main(String[] args) {
    aClass aVar; // Variable vom Typ "childClass"
    ...
    System.out.println(aVar.aField);
  }
}
```

wird der Wert des Felds aField ausgegeben, der in aClass definiert ist. Durch eine Konversion kann der Wert des Felds aField ausgegeben werden, der in aClass definiert ist, und zwar wie folgt:

```
    System.out.println(((superClass) aVar).aField);
```

Innerhalb von Instanzmethoden der Klasse aClass kann auch mit Hilfe des Ausdrucks super.aField auf dieses Feld zugegriffen werden.

In Lava können sowohl delegatee Felder aus Oberklassen durch die Definition neuer Felder, als auch Felder aus Oberklassen durch delegatee Felder versteckt werden. Es lassen sich auch delegatee Felder durch delegatee Felder verstecken.

Das Verstecken eines delegatee Felds hat keine Auswirkung auf die Sichtbarkeit der darüber geerbten Elemente. Dies geschieht aus den gleichen Gründen, warum der Sichtbarkeitsstatus eines delegatee Felds keine Auswirkungen auf den Sichtbarkeitsstatus der darüber geerbten Elemente hat: Die Folgen wären verwirrend und in der Regel nicht wünschenswert.

Felder aus Elterntypen können in Lava ebenfalls durch die Definition eines Felds mit gleichem Namen in der Kindklasse versteckt werden.

Methodenaufrufe

Eine Methode aMethod, die aus parentClass geerbt wird, kann in childClass redefiniert werden, es sei denn, sie ist in parentClass als final deklariert.[5]

[4] Dies wird in Java *Hiding* genannt; s. [Gosling 96], S. 144.

[5] Wenn die Methode aMethod mit gleicher Signatur bereits in einer Oberklasse von childClass definiert ist, so wird keine der beiden Methoden als Redefinition der anderen aufgefaßt. Auf die Auflösung von Namenskonflikten wird in Kapitel 5.5 eingegangen.

Wenn aMethod in einer Instanz instance von childClass oder einer Unterklasse von childClass aufgerufen wird, so wird (a) bei einer Redefiniton von aMethod in der Klasse von instance die redefinierte Version ausgeführt. Ansonsten wird (b) die entsprechende Methode des Objekts aufgerufen, das an parent gebunden ist.[6] *In beiden Fällen bleibt* this *an* instance *gebunden.*

Im Fall (b) kann es sein, daß aMethod im durch parent referenzierten Objekt ebenfalls aus einem weiteren Elterntyp geerbt wird, und in der Deklaration von parentClass nicht redefiniert wurde. Dann wird weiterhin die Methode des entsprechenden Elternobjekts von parent aufgerufen, wobei ebenfalls this an instance gebunden bleibt. Diese Semantik gilt transitiv für alle weiteren Objekte innerhalb einer Elternhierarchie.

Explizite Delegation

In Java gibt es die Möglichkeit, aus einer Methode einer bestimmten Klasse heraus Methoden der jeweiligen Oberklasse aufzurufen. Dies geschieht durch das Versenden einer Nachricht an die Pseudovariable super. In folgendem Beispiel:

```
class D extends C {
  public void aMethod () {
    ...
    super.aMethod(); // Aufruf der Version von
    ...              // "aMethod" aus der Klasse "C"
  }
}
```

wird durch den Aufruf super.aMethod() die Definition von aMethod() aus der Klasse D aufgerufen. Redefinitionen von aMethod() in C oder einer Unterklasse von C werden dabei ignoriert.

Da bei objektbasierter Vererbung ebenfalls Methoden von Elternobjekten redefiniert werden können, müssen wir eine zu super-Aufrufen analoge Aufrufmöglichkeit für Methoden aus Elterntypen anbieten. Zu diesem Zweck führen wir die sogenannte explizite Delegation ein, die einem Methodenaufruf ähnelt, dabei aber die Bindung von this unverändert läßt.

Eine Methode aMethod, die über parent aus parentClass geerbt wird, kann in Methoden von childClass mit Hilfe von parent <- aMethod () aufgerufen werden. Bei einem solchen Elternaufruf wird die Definition von aMethod() ausgeführt, die an das Objekt gebunden ist, auf das parent verweist. Redefinitionen von aMethod() in childClass oder einer Kind- oder Unterklasse von childClass werden dabei ignoriert.

In folgendem Beispiel:

```
class childClass {
  delegatee parentClass parent;

  void aMethod () {
```

[6]Dabei kann es sich auch um eine Instanz einer Unterklasse von parentClass handeln.

```
    ...
    parent <- aMethod (); // Aufruf der Version von
                          // "aMethod" des Objekts,
                          // das durch das Feld "parent"
                          // referenziert wird
    ...
  }
}
```

wird die Version von aMethod() im Objekt ausgeführt, auf das parent verweist. Dabei bleibt die Bindung von this erhalten: Wenn eine Instanz instance1 von childClass die Nachricht aMethod() erhält, so bleibt also this an instance1 gebunden. Wenn eine Instanz instance2 einer weiteren Kindklasse von childClass die Nachricht aMetod() erhält, und diese an eine Instanz von childClass delegiert, so bleibt this an instance2 gebunden.

Elternaufrufe sind nur innerhalb von Instanzmethoden möglich. Darüberhinaus muß der Ausdruck links von der Zeichenkombination <- ein Feld innerhalb der aktuellen Klasse bezeichnen, das die Annotation delegatee hat.

5.2.2 consultee Felder

Ein Feld kann wie folgt als consultee deklariert werden:

```
class childClass {
  consultee parentClass parent; // "parent" verweist auf
                                // ein Elternobjekt vom Typ
                                // "parentClass" in Instanzen
                                // von "childClass"
}
```

Die Annotation consultee läßt sich nur bei Feldern verwenden, die von einem Klassen- oder Interfacetyp sind. Felder von Arraytypen, sowie Felder primitiver Typen können nicht konsultiert werden.

Semantik

Die Annotation consultee hat hinsichtlich folgender Punkte die gleiche Semantik wie die Annotation delegatee:

- Erweiterung der Schnittstelle einer Kindklasse um Elemente des Typs eines consultee Felds.
- Multiplizität. (Es können auch delegatee und consultee Felder in derselben Klasse definiert werden.)
- Kombinierbarkeit mit anderen Annotationen.
- Zugriff auf Felder aus Elternobjekten.
- Verstecken von Feldern.

- Redefinierbarkeit von Methoden.

Der einzige, aber wesentliche Unterschied zu delegatee Feldern besteht in der Bindung von this bei Methodenaufrufen.

Methodenaufrufe

Wenn aMethod in einer Instanz instance von childClass oder einer Unterklasse von childClass aufgerufen wird, so wird (a) bei einer Redefiniton von aMethod in der Klasse von instance die redefinierte Version ausgeführt. Dabei wird this an instance gebunden. Ansonsten wird (b) die entsprechende Methode des Objekts aufgerufen, das an parent gebunden ist.[7] *Dabei wird* this *an* parent *gebunden.*

Im Fall (b) kann es sein, daß aMethod im durch parent referenzierten Objekt ebenfalls aus einem weiteren Elterntyp geerbt wird, und in der Deklaration von parentClass nicht redefiniert wurde. Dann wird die Methode des entsprechenden Elternobjekts aus parent aufgerufen, wobei this an den Methodenträger gebunden wird. Diese Semantik gilt transitiv für alle weiteren Objekte innerhalb einer Elternhierarchie.

„Explizite Konsultation“

Ein spezielles Konstrukt, um „explizite Konsultation“ zu ermöglichen, ist nicht notwendig, da normale Methodenaufrufe dafür ausreichend sind.

In folgendem Beispiel:

```
class childClass {
  consultee parentClass parent;

  void aMethod () {
    ...
    parent.aMethod (); // Aufruf der Version von
                       // "aMethod" aus dem Feld "parent"
    ...
  }
}
```

wird die Version von aMethod() im Objekt ausgeführt, auf das parent verweist. Dabei wird this an parent gebunden, was der Semantik von Konsultation entspricht.

5.3 Erweiterung des Typsystems

Da durch die Deklaration eines Elternverweises die Schnittstelle einer Kindklasse durch die Elemente der Schnittstelle des Elterntyps erweitert wird, die Schnittstelle des Elterntyps also eine Teilmenge der Schnittstelle der Kindklasse

[7] Dabei kann es sich um eine Instanz einer Unterklasse von parentClass handeln.

ist, könnte gemäß den Ausführungen in Kapitel 2.4 festgelegt werden, daß die Kindklasse ein Subtyp des Elterntyps ist.

Dies erweist sich jedoch als problematisch, da Elternverweise mit dem Wert null belegt sein können. Ist ein Elternverweis tatsächlich mit dem Wert null belegt, so wird bei dem Versuch, eine Nachricht an dieses Feld zu delegieren, eine Ausnahme zur Laufzeit des Programms ausgelöst. M.a.W., obwohl eine Methode in der Schnittstelle der Kindklasse enthalten ist, wird ggfs. keine Methode bereitgestellt, die ausgeführt werden kann. Daher kann die Kindklasse kein Subtyp des Elterntyps sein, da sonst keine statische Typkonsistenz gewährleistet werden kann.

In Java existiert keine Möglichkeit, die garantiert, daß ein Feld nicht mit dem Wert null belegt ist.[8] Um dennoch eine Subtypbeziehung zwischen Kindklasse und Elterntyp zu etablieren, muß daher in Lava eine solche Möglichkeit geschaffen werden.

5.3.1 mandatory Felder

In Lava wird die Feldannotation mandatory eingeführt, die angibt, daß ein Feld zwar verändert, aber nicht mit null belegt werden kann. Um dies sicherzustellen, müssen besondere Vorkehrungen bei Zuweisungen und Initialisierungen getroffen werden.

Zuweisungen

Wird zur Laufzeit eines Lava-Programms der Versuch unternommen, einem mandatory Feld den Wert null zuzuweisen, so wird eine für diese Zwecke in Lava neu eingeführte AssignmentOfNullException ausgelöst. Es handelt sich dabei um eine Laufzeitausnahme, die nicht überprüft werden muß.

In folgendem Beispiel:

```
class aClass {
   mandatory String p;

   ...

   void dothis() {
     String q = null;
     p = q; // Hier wird versucht, "p" mit dem Wert "null" zu belegen.
   }
}
```

wird zur Laufzeit bei der Zuweisung p = q die AssignmentOfNullException ausgelöst.

[8]Selbst die Annotation final hilft hier nicht weiter, da der Initialisierungswert eines final Felds erst zur Laufzeit eines Programms ausgewertet wird.

Wenn ein Compiler für Lava zur Übersetzungszeit feststellt, daß ein derartiger Versuch unternommen wird, so muß er eine Fehlermeldung ausgeben. Zumindest die einfache Zuweisung p = null muß abgelehnt werden, wenn p mit der Annotation mandatory versehen ist.[9]

Initialisierungen

Darüberhinaus muß gewährleistet sein, daß ein mandatory Feld korrekt initialisiert wird. Dies kann durch einen Initialisierungswert bei der Deklaration des Felds geschehen, oder aber durch eine Zuweisung in einem Konstruktor.

Der Initialisierungswert wird als rechte Seite einer Zuweisung betrachtet. Eine Initialisierung mit dem Wert null muß daher vom Compiler abgelehnt werden.[10] Ggfs. wird bei der Erzeugung einer Instanz der Klasse oder einer Unterklasse eine AssignmentOfNullException ausgelöst.

Ist für ein mandatory Feld kein Initialisierungswert angegeben, muß in der Klasse, die das Feld deklariert, mindestens ein Konstruktor definiert sein. Der Standardkonstruktor, der in Java automatisch angelegt wird, wenn kein Konstruktor definiert ist, ist für mandatory Felder ohne Initialisierungswert nicht ausreichend, da im Standardkonstruktor der Wert null zugewiesen werden würde.

Das Konzept der *garantierten Zuweisung* wird für die Behandlung eines mandatory Felds in einem Konstruktor benötigt: Es ist in Java für die Initialisierung lokaler Variablen vor einem lesenden Zugriff innerhalb einer Methode definiert. Die Idee dabei ist, daß eine Zuweisung an eine Variable auf jedem möglichen Ausführungspfad innerhalb eines Konstruktors bis zu einem lesenden Zugriff erfolgt. Diese Analyse berücksichtigt die Struktur von Anweisungen und Ausdrücken, insbesondere von boolschen Ausdrücken. Genauere Ausführungen hierzu finden sich in in [Gosling 96], Kapitel 16 „Definite Assignment“, S. 383ff.

Eine Zuweisung an ein mandatory Feld innerhalb eines Konstruktors muß garantiert sein. Das ist dann der Fall, wenn das Feld vor jedem lesenden Zugriff und nach der letzten Anweisung innerhalb des Konstruktors garantiert zugewiesen ist. Dadurch ist sichergestellt, daß bei Ausführung des Konstruktors mindestens eine Zuweisung an ein mandatory Feld erfolgt.

In folgendem Beispiel:

```
class aClass {
  mandatory String p;

  //Konstruktor
  aClass (String s) {
    if (s == null) p = "";
      else p == s;
  }
}
```

[9] Dem Entwickler eines Compilers für Lava steht es frei, durch aufwendigere Analysen des Programmflusses mehr Fälle abzudecken.

[10] Auch hier kann ein Compiler mehr Aufwand betreiben.

ist im Konstruktor p garantiert zugewiesen, da in beiden Zweigen der if-Anweisung eine Zuweisung an p erfolgt. Der Konstruktor ist daher zulässig und wird von einem Compiler für Lava akzeptiert.

Andere Annotationen

Die Annotation mandatory kann nicht zusammen mit optional (s.u.) verwendet werden. Mit allen anderen Annotationen läßt sich mandatory verwenden.

5.3.2 optional Felder

Durch die Annotation optional wird explizit das bisherige Verhalten von Java realisiert, daß derart deklarierte Felder den Wert null annehmen können. Aus Kompatibilitätsgründen wird optional als Default betrachtet, d.h., wenn weder mandatory noch optional angegeben sind, wird optional angenommen. Diese Annotation wurde in Lava eingeführt, um Programmierern zu ermöglichen, explizit auf die Tatsache hinzuweisen, daß ein Feld den Wert null erhalten kann.

Andere Annotationen

Es gelten die gleichen Einschränkungen, die für die Annotation mandatory definiert sind.

5.3.3 Typsichere Delegation

Aus den obigen Ausführungen ergibt sich, daß für Nachrichten aus der Schnittstelle eines delegatee oder consultee Felds, das gleichzeitig mit der Annotation mandatory versehen ist, in Instanzen der Kindklasse garantiert Methoden des Elterntyps bereitgestellt werden. Daher wird für Lava festgelegt, daß in einem solchen Fall die Kindklasse ein Subtyp des Elterntyps ist.

Wenn beispielsweise childClass wie folgt deklariert ist:

```
class childClass {
  mandatory delegatee parentClass parent; // parent ist immer != null

  // Konstruktor
  childClass() {
    ...
  }
}
```

dann kann eine Variable vom Typ parentClass nicht nur

- Instanzen von parentClass
- Instanzen von Unterklassen von parentClass

referenzieren, sondern auch

- Instanzen von childClass
- Instanzen von Unter- oder Kindklassen von childClass.

5.3.4 Typunsichere Delegation

Wenn eine Klasse einen Elternverweis deklariert, der mit dem Wert null belegt sein kann, etwa wie folgt:

```
class childClass {
  optional delegatee parentClass parent; // parent ist evtl. == null
}
```

dann sind Felderzugriffe auf Felder bzw. Methodenaufrufe von Methoden des Elterntyps unsicher: Wenn parent mit dem Wert null belegt ist, wird bei einem solchen Feldzugriff oder Methodenaufruf eine in Lava neu eingeführte ParentIsNullException ausgelöst.

Es handelt sich dabei um eine überprüfte Ausnahme. Daher muß ein solcher Feldzugriff oder Methodenaufruf, sowie eine explizite Delegation entweder in einer entsprechenden try-catch-Anweisung enthalten sein, oder aber in einer Methode, in deren Liste der ausgelösten Ausnahmen die ParentIsNullException enthalten ist. Wird eine Methode, die über einen optionalen Elternverweis geerbt wird, in der Kindklasse redefiniert, so muß in deren Liste der ausgelösten Ausnahmen die ParentIsNullException enthalten sein.

Durch die Deklaration eines optionalen Elternverweises wird *keine* Subtypbeziehung zwischen der Kindklasse und dem Elterntyp festgelegt. Eine Variable vom Typ parentClass kann daher keine Instanzen von childClass referenzieren.

5.3.5 Zusammenfassung

Sowohl die typsichere, als auch die typunsichere Deklaration von Elternverweisen bieten Vorteile, die je nach Anwendung genutzt werden können.

- Durch mandatory Elternverweise ist die Möglichkeit gegeben, eine Subtypbeziehung festzulegen. Beispielsweise können Instanzen einer Klasse Student an mandatory Elternverweise vom Typ Person delegieren. Solche Instanzen können auch von Variablen vom Typ Person referenziert werden, was der Intuition entspricht, daß ein Student immer auch eine Person ist.
- Eigenschaften, die Objekte nur zeitweise haben, sind mit Hilfe von optional Elternverweisen modellierbar. Beispielsweise können Instanzen der Klasse Person an optional Elternverweise vom Typ Student delegieren: Eine Person ist möglicherweise ein Student. Wenn sie ein Student ist, hat sie die entsprechenden Eigenschaften. Da die ParentIsNullException überprüft werden muß, ist gewährleistet, daß die Möglichkeit des Fehlens der Eigenschaften immer berücksichtigt wird. In folgendem Beispiel:

```
public static void main(String[] args) {
  Person p; // ist möglicherweise "Student"

  ...

  try {
    System.out.println(p.MatrikelNr());
  } catch (ParentIsNullException e) {
    System.out.println("keine Matrikelnr. vorhanden")
  }
}
```

wird die Matrikelnummer des Studenten ausgegeben, der durch die Variable p referenziert wird. Im catch-Zweig wird der Fall behandelt, daß p auf ein Person-Objekt verweist, das nicht an ein Student-Objekt delegiert.

5.4 Behandlung von this

Bei der Ausführung eines Lava-Programms kommt es durch Delegation von Nachrichten zu Situationen, in denen der Empfänger einer Nachricht (this) und der Träger der aktuell ausgeführten Methode unterschiedliche Objekte sind. In diesem Abschnitt wird die Semantik von Zugriffen auf Felder von this und Nachrichten an this in solchen Situationen behandelt.

5.4.1 Zugriffe auf Felder von this

Explizite oder implizite Zugriffe auf Felder von this[11] werden immer direkt als Zugriffe auf die entsprechenden Felder des Trägers der aktuellen Methode realisiert. Dies ist zulässig, da Felder in Kindklassen nicht redefiniert werden können. Ein Versuch, auf ein entsprechendes Feld des Empfängers der Nachricht zuzugreifen, kann daher nicht gelingen.

5.4.2 Nachrichten an this

Bei Nachrichten an this treten zwei Probleme auf, für die semantische Festlegungen getroffen werden müssen.

Erstes Problem: Das this-Objekt kennt möglicherweise die versendete Nachricht nicht.

Dies ist z.B. der Fall, wenn das das this-Objekt an einen Elternverweis eines Typs delegiert, von dem die Klasse des Elternobjekts abgeleitet ist, und die versendete Nachricht nicht im Elterntyp enthalten ist. In Abb. 5.4 ist dieser Fall gegeben, wenn obj1 das this-Objekt und parent1 das Elternobjekt ist, das die Nachricht

[11] z.B. this.field oder field, wenn field Instanzvariable ist und nicht durch eine lokale Variable versteckt wird

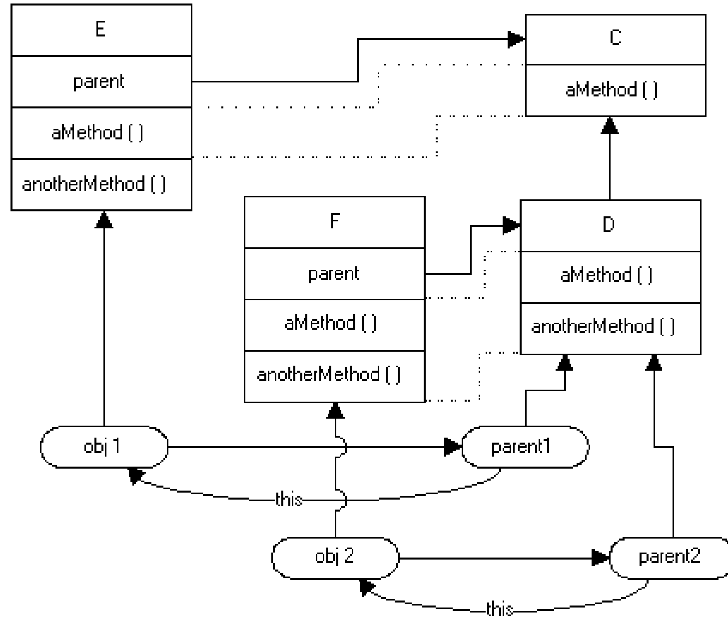

Abbildung 5.4: Nachrichten an this: Wenn parent1 die Nachricht anotherMethod() an this sendet, wird sie in obj1 nicht gefunden. Wenn parent2 diese Nachricht an this sendet, wird sie in obj2 gefunden. (Achtung: D und E sind voneinander unabhängige Klassen, die Methoden anotherMethod() aus diesen beiden Klassen müssen als unterschiedlich betrachtet werden.)

anotherMethod() an this sendet. Dabei ist E die Klasse des this-Objekts, C der Typ des Elternverweises und D die Klasse des Elternobjekts. Durch Delegation wird also this an ein Objekt gebunden, das nicht die Schnittstelle der aktuellen Klasse erfüllt. Im Beispiel erfüllt obj1 weder den Typ von D, noch einen Untertyp von D.

Ebenfalls in Abb. 5.4 ist der Fall enthalten, daß das this-Objekt an einen Elternverweis des Typs der aktuelle Klasse delegiert. Dann ist die Nachricht auch im this-Objekt bekannt, das evtl. eine Redefinition der passenden Methode bereitstellt.

M.a.W. kann an der Aufrufstelle *zur Übersetzungszeit* einer Lava-Klasse nicht entschieden werden, ob das this-Objekt eine zur Nachricht passende Methode bereitstellt oder nicht.

Der Safety Check

Es muß also bei einer Nachricht an this zur Laufzeit überprüft werden, ob das this-Objekt die Nachricht kennt. Dabei reicht es nicht aus, daß das this-Objekt zufällig eine Methode mit der richtigen Signatur bereitstellt, sondern es muß sich um eine Methode für exakt dieselbe Nachricht handeln, die versendet wurde; das kann aber nur der Fall sein, wenn sowohl Sender (Träger der aktuellen

Methode) als auch Empfänger (this-Objekt) die Nachricht jeweils aus demselben Typ geerbt haben.

Handelt es sich tatsächlich dieselbe Nachricht, so kann die entsprechende Methode ausgeführt werden, die das this-Objekt für diese Nachricht bereitstellt. Andernfalls muß derselbe Test für das nächste Objekt der Elternhierarchie ausgeführt werden, usw., bis ein Objekt mit einer passenden Methode gefunden wurde. Ggfs. ist erst der Träger der aktuellen, nachrichtenversendenden Methode in der Lage, eine passende Methode bereitzustellen.

Wenn die Semantik von Lava festlegen würde, daß ein zufälliges Vorhandensein einer Methode mit der richtigen Signatur im this-Objekt ausreicht, um sie in einem solchen Fall auszuführen, so würde in Abb. 5.4 das Senden der Nachricht anotherMethod() von parent1 an this zur Folge haben, daß die Definition der Methode anotherMethod() der Klasse E ausgeführt wird. Möglicherweise weiß der Programmierer, der die Klasse E implementiert, nichts von der Existenz der Klasse D und der darin definierten Methode anotherMethod(). Er hätte aber ohne jede Absicht genau diese Methode redefiniert. Eine solche Semantik ist offensichtlich nicht wünschenswert.

Zweites Problem: Der Träger der aktuellen Methode ist nicht in der Elternhierarchie enthalten.

Sei aMethod() die aktuelle Methode und anotherMethod() die Nachricht, die an das this-Objekt versendet wird. Die Nachricht aMethod() wurde ausgehend vom this-Objekt an das aktuelle Objekt entlang einer Folge von Elternobjekten bis hin zum Träger von aMethod() delegiert, entweder implizit oder explizit.[12]

Einfrieren des Suchpfads

Um zu verhindern, daß eine Änderung der Elternhierarchie während der Ausführung von aMethod() zur Folge hat, daß zu anotherMethod() gar keine passende Methode gefunden wird, weil in der aktuellen Elterhierarchie der Träger der aktuellen Methode nicht mehr enthalten ist, wird verlangt, daß bei einer Nachricht an this der Pfad ausgehend vom aktuellen this-Objekt nach einer passenden Methode durchsucht wird, der sich aus der Delegation von aMethod() ergeben hat. (s. Abb. 5.5)

Damit ist eindeutig festgelegt, welche Objekte bei einer Nachricht an this nach einer passenden Methode durchsucht werden, selbst wenn ein Objekt innerhalb des Suchpfads ggfs. auf mehrere Elternobjekte verweist. Dieser Suchpfad ist für die Ausführung von aMethod() festgelegt; selbst wenn in den Objekten innerhalb des Suchpfads die Elternverweise geändert werden, bleibt er unverändert. Eine Suche nach einer passenden Methode endet damit garantiert spätestens beim Träger von aMethod().

Diese Suche nach einer passenden Methode wird in Lava auch als Delegation aufgefaßt. Daher wird die passende Methode mit unverändertem this-Objekt

[12]Eine andere Möglichkeit gibt es in Lava nicht, daß this-Objekt und Träger der aktuellen Methode unterschiedliche Objekte sind.

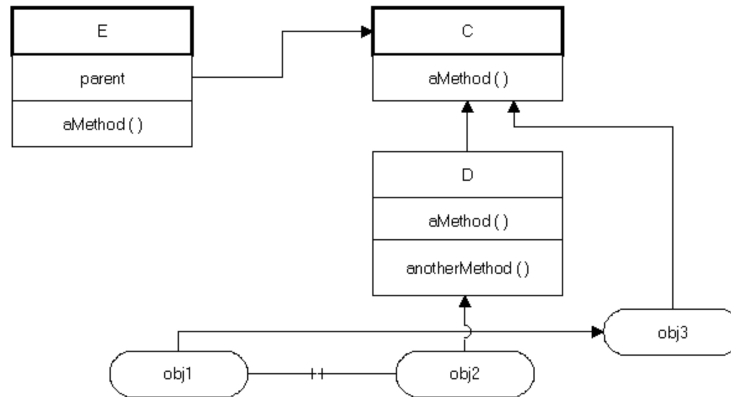

Abbildung 5.5: Wenn obj1 die Nachricht aMethod() an obj2 delegiert, und während der Ausführung von aMethod() der Elternverweis in obj1 auf obj3 gesetzt wird, wird dennoch beim Versenden von anotherMethod() and this zuerst in obj1 und dann in obj2 nach einer passenden Methode gesucht.

aufgerufen. Auch der Pfad, in denen nach Methoden zu Nachrichten an this gesucht wird, bleibt daher unverändert.

5.4.3 Zusammenfassung

Bei Zugriffen auf Felder von this wird direkt auf die entsprechenden Felder des Trägers der aktuellen Methode zugegriffen.

Bei Nachrichten an this wird im this-Objekt nach einer passenden Nachricht gesucht. Die Typsicherheit wird dabei durch folgende zwei Festlegungen sichergestellt:

Safety Check Es werden nur Methoden ausgeführt, die zur gesendeten Nachricht des verlangten Typs passen. Dies ist der Fall, wenn die gesendete Nachricht sowohl in der Klasse des Trägers der aktuellen Methode, als auch in der Klasse des this-Objekts aus einem gemeinsamen Typ geerbt wurde.

Wird im this-Objekt keine passende Methode gefunden, so wird in einem Elternobjekt von this weitergesucht. Dabei wird ebenfalls ein Safety Check durchgeführt, usw.

Einfrieren des Suchpfads Der Pfad, in dem nach einem Objekt mit einer passenden Methode gesucht wird, wird für die Ausführung der aktuellen Methode eingefroren. Damit ist sichergestellt, daß der Träger der aktuellen Methode in diesem Suchpfad enthalten ist, auch wenn die tatsächliche Elternhierarchie geändert wird. Daher wird bei der Suche für Nachricht an this spätestens beim Träger der aktuellen Methode eine passende Methode gefunden.

5.5 Auflösung von Namenskonflikten

Durch die Deklaration von Elternobjekten kann es innerhalb einer Klasse zu Namenskonflikten kommen, wenn Felddefinitionen bzw. Methodendefinitonen mit gleichen Namen mehr als einmal in der Oberklasse und den Elternklassen auftreten.[13]

Namenskonflikte treten bereits bei rein klassenbasierten Programmiersprachen auf, die multiple Vererbung ermöglichen. Da Java auf Klassenebene keine multiple Vererbung bietet, wurde dem Problem möglicher Namenskonflikte beim Design von Java keine besondere Beachtung geschenkt. Bei Lava müssen mögliche Namenskonflikte jedoch eindeutig aufgelöst werden. Dafür gibt es verschiedenen Designalternativen:

Reihenfolge der Deklaration Die Reihenfolge, in der Oberklasse und Elternvariablen deklariert werden, kann kein Kriterium dafür bieten, welches Element aus verschiedenen konfligierenden Elementen den Vorrang gegenüber den anderen hat. Zum Einen könnten Elemente eines Elterntyps keinen Vorrang vor Elementen der Oberklasse erhalten, da die Reihenfolge in der Deklaration der Oberklasse und der Elternvariablen nicht geändert werden können; zum Anderen können bei mehreren konfligierenden Elementen vom Programmierer unterschiedliche Prioritäten gewünscht oder verlangt sein, die so nicht ausdrückbar wären. Aus den genannte Gründen scheidet diese Alternative aus.

Auflösung bei Verwendung In C++ werden Namenskonflikte nicht an der Stelle, wo sie entstehen, also bei der Definition einer Klasse aufgelöst, sondern erst bei der Verwendung von Elementen der Klasse. Erbt z.B. eine Klasse `C` von zwei Oberklassen `A` und `B`, und sind in beiden Oberklassen Methoden mit dem Namen `m` und gleicher Signatur definiert, so muß beim Senden der Nachricht `m` an eine Variable `v` vom Typ `C` angegeben werden, aus welcher Oberklasse eine Definition gewählt werden soll. Ist z.B. die Definition aus `A` gewünscht, so sieht der Aufruf wie folgt aus: `v->A::m()`.

Dies ist jedoch nicht wünschenswert, da hierbei an der Aufrufstelle Informationen über die Oberklassen von `C` bekannt sein müssen, was dem Prinzip der Kapselung widerspricht und darüberhinaus bei Änderung der Vererbungshierarchie zur Folge haben kann, daß Klienten geändert werden müssen oder sich sogar das Verhalten eines Programms unerwartet ändert.

Eine weitreichendere Folge ist jedoch, daß bei einem solchen Aufruf die dynamische Nachrichtenbindung verloren geht: Durch `v->A::m()` wird *immer* die Defintion von `m()` aus `A` gewählt, unabhängig davon, ob `m()` in einer Unterklasse von `A` redefiniert wird oder nicht.

[13]Durch die Definition von Elternverweisen werden in einer Lava-Klasse nur Instanzelemente des Elterntyps geerbt, nicht aber Klassenelemente. Daher können keine Namenskonflikte zwischen Klassenelementen entstehen. Es kann jedoch ein Namenskonflikt zwischen einem Klassenelement einer Oberklasse und einem Instanzelement eines Elterntyps auftreten. Wir betrachten im folgenden jedoch der Einfachheit halber zunächst nur Namenskonflikte zwischen Instanzelementen.

Zwar wird in [Stroustrup 92] auf Seite 219 vorgeschlagen, daß in der erbenden Klasse (in obigem Beispiel die Klasse C) das betroffene Element neu definiert wird, damit der Aufruf v->m() wieder möglich ist. Das kann jedoch keine generelle Lösung sein, und zwar aus folgenden Gründen:

- Felder sind nicht redefinierbar, daher kann bei Feldern mit gleichen Namen dieses Verfahren nicht angewendet werden.
- Bei Methoden kann es sein, daß der betroffene Name in den verschiedenen Oberklassen unterschiedliche Aufgaben bezeichnen kann, die in keinem Zusammenhang zueinander stehen. Eine Redefinition zweier unterschiedlicher Methoden durch eine neue kann daher zu semantischen Problemen führen.
- Diese Probleme könnten umgangen werden, indem für die unterschiedlichen Aufgaben Methoden mit neuen Namen definiert werden, die die entsprechenden Methoden aus den jeweiligen Oberklassen aufrufen. Z.B. könnte die Methode Am in C wie folgt definiert werden: `void Am() {this->A::m();}`. Die Methode Bm könnte analog definiert werden. Dann ist in Klienten von C klar, welche Methode für welche Aufgabe steht.

  Dabei geht jedoch wiederum die dynamische Bindung verloren: Angenommen einer Variablen v vom Typ A wird eine Instanz einer Unterklasse von C zugewiesen, die eine Redefinition der Methode Am bereitstellt. Diese Redefinition ist jedoch beim Senden der Nachricht m an v nicht sichtbar, es wird die Definition aus der Klasse A ausgeführt.

Umbenennung In der Programmiersprache Eiffel wird bei Namenskonflikten verlangt, daß jedes betroffene Element in einer Unterklasse explizit umbenannt wird. Durch die eindeutige Zuordnung der neuen Namen zu den jeweiligen Elementen in den Oberklassen kann die dynamische Bindung der Methoden beibehalten werden.[14]

Dies ist jedoch noch nicht ausreichend, da unter bestimmten Umständen dennoch nicht eindeutig klar ist, welches Element mit einem bestimmten Namen gemeint ist: Angenommen von einer Klasse A werden die Klassen B und C abgeleitet, und eine weitere Klasse D erbt sowohl von B, als auch von C. Die Vererbungsstruktur „fließt“ also in der Klasse A zusammen, daher wird sie als *konfluenter Pfad* bezeichnet.

Eine in A definierte Methode m wird demnach in D zweimal, nämlich von B und von C geerbt. Durch Umbenennungen z.B. in Bm für die Version aus B und in Cm für die aus C kann der Namenskonflikt behoben werden. Wenn einer Variablen v vom Typ A eine Instanz von D zugewiesen wird, und anschließend die Nachricht m an v gesendet wird, kann trotz Umbenennungen nicht entschieden werden, welche der beiden Methoden Bm und Cm der Instanz ausgeführt werden soll. Für diesen Fall wird in

[14] Die eben dargestellte Definition von Methoden mit neuen Namen in C++ kann nicht als Umbenennung in diesem Sinn verstanden werden, da es sich um völlig neue Methoden handelt, die „zufällig“ eine andere Methode des this-Objekts aufrufen. Daraus kann aber keine besondere Beziehung zwischen den definierten und den aufgerufenen Methoden abgeleitet werden.

> Eiffel verlangt, daß bei der Deklaration von D zusätzlich angegeben wird, welche Methode ausgewählt werden soll.

Die Lösung, die Eiffel für Namenskonflikte bietet, ist zwar mit vergleichsweise viel Schreibaufwand bei der Deklaration einer Klasse verbunden, gewährleistet jedoch Eindeutigkeit unter Beibehaltung von Kapselung und dynamischer Bindung. Aus diesem Grund wurde für Lava eine ähnliche Lösung gewählt, die ich im folgenden beschreiben werde.[15]

5.5.1 Umbenennungen

Zwischen einem Feld und einer Methode können weder in Java, noch in Lava Namenskonflikte auftreten, da eine Methode immer mit einer in runden Klammern eingeschlossenen, möglicherweise leeren Parameterliste verwendet wird. Durch das Vorhandensein einer öffnenden runden Klammer nach einem Namen kann daher eindeutig zwischen Feldern und Methoden unterschieden werden, somit ist die Verwendung desselben Namens für ein Feld und eine Methode unproblematisch.

Ich beschreibe daher im folgenden der Klarheit wegen die neuen Sprachkonstrukte zunächst getrennt nach Feldern und Methoden.

5.5.2 Umbenennung von Feldern

Ein Namenskonflikt zwischen zwei oder mehr Feldern liegt vor, wenn in einer Oberklasse und mindestens einem Elterntyp, oder in mindestens zwei Elterntypen der gleiche Name für jeweils ein Feld definiert ist. Dabei ist es unerheblich, von welchem Typ die betroffenen Felder jeweils sind, bzw. ob sie von paarweise unterschiedlichem Typ sind, da bei Verwendung des Namens eines Felds nicht auf den Typ des Felds geschlossen werden kann. In folgendem Beispiel:

```
// In "superClass", "parentClass1" und "parentClass2"
// ist jeweils ein Feld "field" definiert

class aClass extends superClass {
    delegatee parentClass1 parent1;
    consultee parentClass2 parent2;

    void doSomething () {
      System.out.println(field);
    }
}
```

kann nicht entschieden werden, welches field in der Methode doSomething() auf der Standardausgabe ausgegeben werden soll. (s. Abb. 5.6)

[15] In Java treten Namenskonflikte möglicherweise zwischen konstanten Feldern mehrerer Interfaces auf, die gemeinsam von einer Klasse implementiert oder von einem weiteren Interface erweitert werden. Diese Namenskonflikte müssen ähnlich wie in C++ aufgelöst werden. Siehe dazu [Gosling 96], S. 153ff. Diese Möglichkeit wird in Lava aus Kompatibilitätsgründen für genau diese Fälle beibehalten; für die neuen, durch objektbasierte Vererbung eingeführten Namenskonflikte müssen jedoch die im folgenden beschriebenen Sprachkonstrukte verwendet werden.

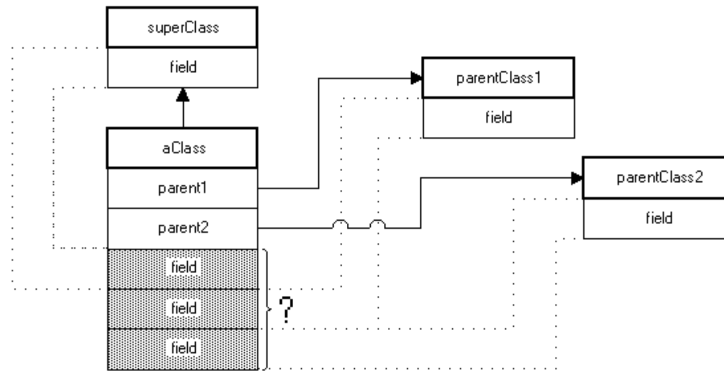

Abbildung 5.6: Namenskonflikte zwischen Feldern

Daher wird in Lava im Fall von Namenskonflikten zwischen Feldern, die durch objektbasierte Vererbung entstehen, verlangt, daß die betroffenen Felder mit Hilfe der Angabe renames umbenannt werden, und zwar wie folgt:

```
// In "superClass", "parentClass1" und "parentClass2"
// ist jeweils ein Feld "field" definiert

class aClass extends superClass {
    delegatee parentClass1 parent1;
    consultee parentClass2 parent2;

    type0 field renames super::field;
    type1 fieldFromParent1 renames parent1::field;
    type2 fieldFromParent2 renames parent2::field;

    void doSomething () {
      System.out.println(field);
    }
}
```

Auf die Angabe renames folgt die Angabe, aus welchem Elternverweis, oder ob aus der Oberklasse ein Feld umbenannt werden soll, gefolgt von einem :: und dem Namen des Felds im Elternverweis oder in der Oberklasse. Die Umbenennung ist Bestandteil einer Felddeklaration, jedoch werden keine neue Felder definiert, sondern bereits definierte Felder lediglich mit neuen Namen versehen. Dabei muß allerdings einer der betroffenen Felder den ursprünglichen Namen erhalten, damit der Name in der Klasse aClass nicht für andere Felder verwendet werden kann.

Damit ist in obigem Beispiel (Abb. 5.7) z.B. fieldFromParent1 eine neue Bezeichnung für field aus dem durch parent1 referenzierten Elternobjekt. Daher wird in folgendem Beispiel:

```
aClass       v0 = new aClass (...);
```

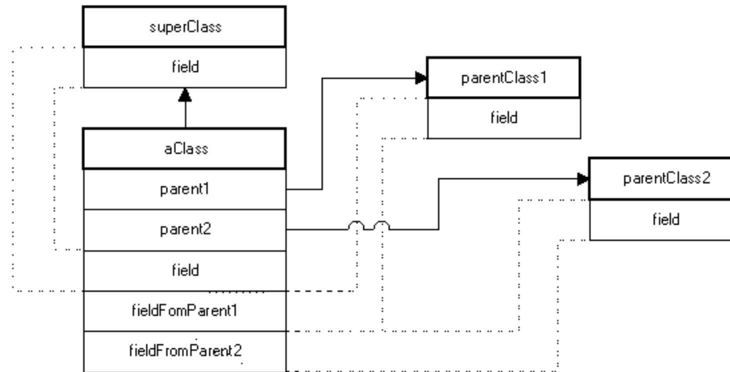

Abbildung 5.7: Auflösung von Namenskonflikten zwischen Feldern durch Umbenennungen

```
parentClass1 v1 = v0;

System.out.println(v0.fieldFromParent1 == v1.field);
```

der Wert true auf der Standardausgabe ausgegeben.

Konfluente Pfade

Möglicherweise reicht die Umbenennung von Feldern bei einem Namenskonflikt nicht aus: Angenommen in obigem Beispiel sind zwei der betroffenen Felder vom gleichen Typ, z.B. fieldFromParent1 und fieldFromParent2. Dann kann es sein, daß parentClass1 und parentClass2 von einem gemeinsamen Typ parentClass0 abgeleitet sind,[16] der die ursprüngliche Definition von field erhielt. M.a.W., field wird in aClass zweimal aus parentClass0, einmal via parentClass0 und einmal via parentClass1 geerbt (s. Abb. 5.8). Angenommen aVar sei eine Variable vom Typ parentClass0, der eine Instanz von aClass zugewiesen wird. Wird danach versucht, auf aVar.field zuzugreifen, so kann nicht ohne weiteres entschieden werden, auf welches der beiden Felder fieldFromParentClass1 und fieldFromParentClass2 tatsächlich zugegriffen werden soll.

Aus diesem Grund wird in Lava verlangt, daß zusätzlich zu den verlangten Umbenennungen für alle Felder aus der Oberklasse und aus Elternverweisen, die den gleichen Namen *und* den gleichen Typ haben, ein Feld angegeben wird, das bei konfluenten Pfaden für den Namen ausgewählt wird. Für das obige Beispiel (Abb. 5.8) wäre demnach folgende Deklaration möglich:

```
// In "parentClass0" ist ein Feld "field" definiert.
// "parentClass1" und "parentClass2" sind
// von "parentClass0" abgeleitet.
```

[16] Dabei ist es unerheblich, ob dies durch klassenbasierte oder objektbasierte Vererbung erfolgt.

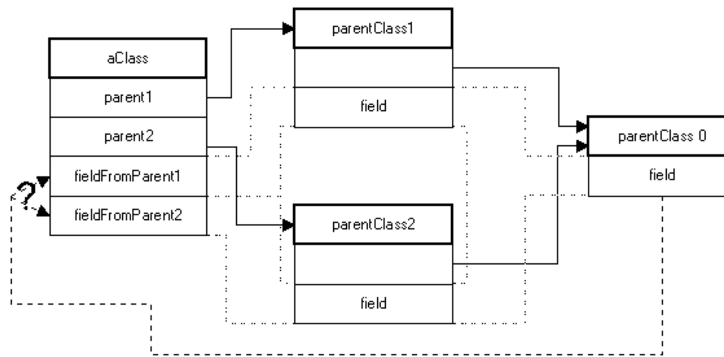

Abbildung 5.8: Konfluenter Pfad: field wird indirekt zweimal an aClass vererbt.

```
class aClass extends superClass {
    delegatee parentClass1 parent1;
    consultee parentClass2 parent2;

    type0 field renames super::field;
    type1 fieldFromParent1 renames parent1::field selected;
                                    // für den konfluenten Pfad
                                    // wird "fieldFromParent1"
                                    // ausgewählt
    type1 fieldFromParent2 renames parent2::field;

    void doSomething () {
      System.out.println(field);
    }
}
```

In folgendem Beispiel:

```
parentClass0 aVar = new aClass (...);

System.out.println(aVar.field);
```

wird der Wert von fieldFromParent1 der referenzierten Instanz auf der Standardausgabe ausgegeben.

Wichtig ist, daß eine Selektion nur bei konfluenten Pfaden verwendet wird, wenn nicht anders eindeutig feststellbar ist, welches Feld gemeint ist. In folgendem Beispiel

```
parentClass2 aVar = new aClass (...);

System.out.println(aVar.field);
```

wird der von Wert von fieldFromParent2 der referenzierten Instanz auf der Standardausgabe ausgegeben, unabhängig von der Selektion, die für field in der Deklaration von aClass getroffen wurde.

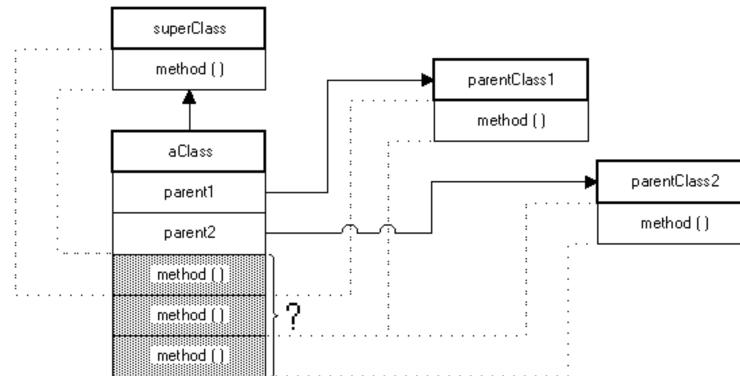

Abbildung 5.9: Namenskonflikte zwischen Methoden

5.5.3 Umbenennung von Methoden

Ein Namenskonflikt zwischen zwei oder mehr Methoden liegt vor, wenn in einer Oberklasse und mindestens einem Elterntyp, oder in mindestens zwei Elterntypen die gleiche Signatur für jeweils eine Methodendefinition verwendet wird. Die Signatur einer Methode besteht aus dem Namen, sowie die Anzahl und Typen der formalen Parameter der Methode.[17] Streng genommen müßte also im Fall von Methoden nicht von Namenskonflikten, sondern von Signaturkonflikten die Rede sein. Aus Gründen der Einfachheit wird aber der Begriff Namenskonflikt auch für Methoden verwendet.

In folgendem Beispiel:

```
// In "superClass", "parentClass1" und "parentClass2"
// ist jeweils eine Methode "method()" definiert

class aClass extends superClass {
    delegatee parentClass1 parent1;
    consultee parentClass2 parent2;

    void doSomething () {
      method();
    }
}
```

kann nicht entschieden werden, welche method() in der Methode doSomething() aufgerufen werden soll. (s. Abb. 5.9)

Daher wird in Lava im Fall von Namenskonflikten zwischen Methoden, die durch objektbasierte Vererbung entstehen, verlangt, daß die betroffenen Methoden mit Hilfe der Angabe renames umbenannt werden, z.B. wie folgt:

[17] Eine Klasse darf nicht zwei oder mehr Methoden mit der gleichen Signatur deklarieren, selbst wenn die Deklarationen sich im Typ des Rückgabewerts unterscheiden, weil bei Aufruf einer Methode der Rückgabetyp nicht zur Unterscheidung zwischen ggfs. konfligierenden Methodendefinitionen herangezogen werden kann. Daher gehört der Rückgabetyp einer Methode nicht zur Signatur der Methode.

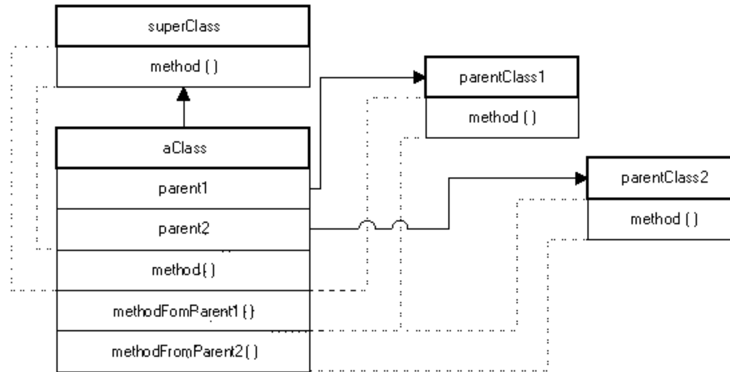

Abbildung 5.10: Auflösung von Namenskonflikten zwischen Methoden durch Umbenennungen

```
// In "superClass", "parentClass1" und "parentClass2"
// ist jeweils eine Methode "method" definiert.

class aClass extends superClass {
    delegatee parentClass1 parent1;
    consultee parentClass2 parent2;

    type0 method () renames super::method;
    type1 methodFromParentClass1 () renames parent1::method;

    type2 methodFromParentClass2 () renames parent2::method {
      ...
      return ...;
    }

    void doSomething () {
      method ();
    }
}
```

Die renames-Angabe wird genauso verwendet wie bei Feldern, ggfs. kann die entsprechende Methode zusätzlich wie im Beispiel methodFromParentClass2 redefiniert werden.

Damit ist in obigem Beispiel (Abb. 5.10) z.B. methodFromParent1 eine neue Bezeichnung für method aus dem durch parent1 referenzierten Elternobjekt. Daher wird in folgendem Beispiel:

```
aClass       v0 = new aClass (...);
parentClass1 v1 = v0;

v0.methodFromParent1();
v1.method();
```

bei beiden Methodenaufrufen dieselbe Methode auf der selben Instanz ausgeführt.

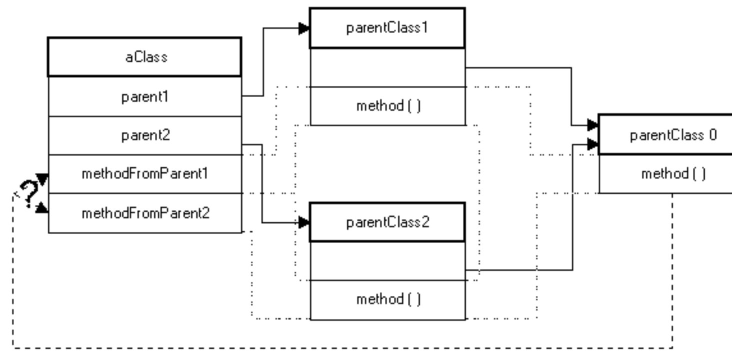

Abbildung 5.11: Konfluenter Pfad: method() wird indirekt zweimal an aClass vererbt.

Nachrichten an this

Bei Nachrichten an this werden Umbenennungen berücksichtigt. Ist bei obigem Beispiel während der Ausführung einer Methode von parentClass2 das this-Objekt eine Instanz von aClass, und sendet diese Methode die Nachricht method() an this, so wird die Definition von methodFromSuperClass2() aus aClass ausgeführt.

Konfluente Pfade

Möglicherweise reicht die Umbenennung von Methoden bei einem Namenskonflikt nicht aus: Angenommen in obigem Beispiel haben zwei der betroffenen Methoden den gleichen Rückgabetyp, z.B. für methodFromParent1() und methodFromParent2(). Dann kann es sein, daß parentClass1 und parentClass2 von einem gemeinsamen Typ parentClass0 abgeleitet sind, der die ursprüngliche Definition von method() enthielt (s. Abb. 5.11). Angenommen aVar sei eine Variable vom Typ parentClass0, der eine Instanz von aClass zugewiesen wird. Wird danach versucht, aVar.method() aufzurufen, so kann nicht ohne weiteres entschieden werden, welche der beiden Methoden methodFromParentClass1() und methodFromParentClass2() tatsächlich aufgerufen werden soll.

Aus diesem Grund wird in Lava verlangt, daß für alle Methoden aus der Oberklasse und aus Elterntypen, die die gleiche Signatur *und* den gleichen Rückgabetyp haben, eine Methode angegeben wird, die bei konfluenten Pfaden für die Signatur ausgewählt wird. Für das obige Beispiel wäre demnach folgende Deklaration möglich:

```
// In "parentClass0" ist eine Methode "method" definiert.
// "parentClass1" und "parentClass2" sind
// von "parentClass0" abgeleitet.

class aClass extends superClass {
    delegatee parentClass1 parent1;
    consultee parentClass2 parent2;
```

```
    type0 method () renames super::method;
    type1 methodFromParent1 () renames parent1::method selected;
                                          // für den konfluenten Pfad
                                          // wird "methodFromParent1"
                                          // ausgewählt
    type1 methodFromParent2 () renames parent2::method;

    void doSomething () {
      method ();
    }
}
```

In folgendem Beispiel:

```
parentClass0 aVar = new aClass (...);

aVar.method();
```

wird die Methode methodFromParent1() der referenzierten Instanz aufgerufen.

Wichtig ist, daß eine Selektion nur bei konfluenten Pfaden verwendet wird, wenn nicht anders eindeutig feststellbar ist, welche Methode gemeint ist. In folgendem Beispiel

```
parentClass2 aVar = new aClass (...);

aVar.method();
```

wird die Methode methodFromParent2() der referenzierten Instanz aufgerufen, unabhängig von der Selektion, die für method() in der Deklaration von aClass getroffen wurde.

Bei Nachrichten an this wird eine Selektion ebenfalls nicht berücksichtigt, da durch den eingefrorenen Suchpfad eindeutig festgelegt ist, welche Objekte nach einer passenden Methode durchsucht werden müssen.

5.5.4 Namenskonflikte zwischen Klassen- und Instanzelementen

Wie bereits erwähnt, kann zwischen einem Klassenelement einer Oberklasse und einem Instanzelement eines Elterntyps ein Namenskonflikt auftreten. Da Klassenelemente jedoch nicht dynamisch an Instanzen einer Klasse gebunden sind, sondern statisch über den Namen der Klasse, in der sie definiert sind, müssen solche Namenskonflikte nicht aufgelöst werden.

Innerhalb von Instanzmethoden einer solchen Klasse wird der entsprechende Name für das Instanzelement im Elternverweis verwendet. Das Klassenelement der Oberklasse muß via super oder den Namen der Klasse, in der es definiert ist, angesprochen werden.

5.5.5 Diskussion

Es gab beim Design von Lava eine Reihe von Alternativen, wie die Sprachkonstrukte für Umbenennungen festgelegt werden konnten. Die verschiedenen Aspekte der hier dargestellten Lösung sind wie folgt motiviert:

- Bei Umbenennungen muß jeweils eines der betroffenen Elemente den ursprünglichen Namen erhalten. Dies geschieht, wie bereits erwähnt, damit der jeweilige Name nicht in der gleichen Klasse und im Fall von Methoden nicht in Subtypen für andere Elemente verwendet wird. Würde die Verwendung der Namen für andere Elemente ermöglicht, die mit den ursprünglichen Elementen nicht im Zusammenhang stehen, so wären die sich ergebenden Quelltexte schwerer verständlich, was nicht wünschenswert ist.

  Die Tatsache, daß ein Elemente den ursprünglichen Namen erhält, muß in Lava explizit notiert werden, wie z.B. in:

  ```
  class aClass extends superClass {
    ...
    type0 method () renames super::method;
    ...
  }
  ```

  Dies erscheint auf den ersten Blick unnötig und redundant, und wurde in einer älteren Fassung von Lava nicht verlangt. Die Erfahrung hat jedoch gezeigt, daß ein Quelltext wiederum schwerer verständlich wird, wenn nicht explizit erwähnt wird, welches Element den Namen behält, sondern nur, welche Elemente umbenannt werden.

- Eine Alternative zur renaming-Angabe bei einer Deklaration eines Elements besteht darin, in einem separaten Abschnitt eines Quelltexts alle Umbenennungen tabellarisch anzugeben, und somit die Deklaration und die Umbenennung von Elementen textuell zu trennen. In Lava wurde jedoch der hier dargestellten Alternative Vorzug zu geben, damit die gesamte Bedeutung eines Namens an einer Stelle im Quelltext ersichtlich ist, und nicht verschiedenen Stellen berücksichtigt werden müssen. Dies geht mit dem Design von Java konform, daß z.B. keine Headerfiles für Java-Klassen geschrieben werden müssen.

- Es wäre denkbar gewesen, daß bei einem Namenskonflikt zwischen Feldern die Definition eines neuen Felds alle betroffenen Felder versteckt, und somit der Namenskonflikt aufgelöst wäre. Damit ist aber zum Einen das Problem der Selektion eines Felds bei konfluenten Pfaden noch nicht gelöst. Zum Anderen gäbe es keine dazu analoge Möglichkeit im Fall von Methoden: Wegen möglicherweise unterschiedlicher Rückgabetypen bei Namenskonflikten zwischen Methoden können nicht alle betroffenen Methoden durch eine einzige Methode mit gleicher Signatur redefiniert werden. Um Namenskonflikte zwischen Feldern und Methoden möglichst einheitlich handzuhaben wurde daher auf ein Verstecken von Feldern im Fall von Namenskonflikten verzichtet.

- Im Fall von konfluenten Pfaden, und dann nur in Situationen, in denen nicht entschieden werden kann, welches Element ausgewählt werden muß, wird die Angabe einer Selektion verwendet. Es wäre möglich gewesen, gründsätzlich für alle Oberklassen und Elterntypen das selektierte Element auszuwählen. In folgendem Beispiel, das sich auf Abb. 5.11 bezieht:

  ```
  parentClass2 aVar = new aClass (...);

  aVar.method();
  ```

  würde daher wegen der Selektion in aClass (s.o.) die Methode methodFromParent1() der referenzierten Instanz aufgerufen.

  Die gewählte Lösung bietet jedoch mehr Ausdrucksmöglichkeiten:

  1. Kontexterhaltung: In aClass und Subtypen von aClass kann die Methode methodFromParent2() redefiniert werden, und somit das Verhalten in obigem Beispiel beeinflußt werden. Es kann also ein nach verschiedenen Kontexten (parentClass1 oder parentClass2) differenziertes Verhalten spezifiziert werden, was mit der Alternative, immer das selektierte Element auszuwählen, nicht erreichbar wäre.
  2. Einheitliches Verhalten: Um zu erzwingen, daß method() immer einheitlich für beide Elterntypen in Subtypen von aClass redefiniert werden muß, kann aClass wie folgt deklariert werden:

     ```
     // In "parentClass0" ist eine Methode "method" definiert.
     // "parentClass1" und "parentClass2" sind
     // von "parentClass0" abgeleitet.

     class aClass extends superClass {
       delegatee parentClass1 parent1;
       consultee parentClass2 parent2;

       type0 method () renames super::method;

       final type1 methodFromParent1 ()
                               renames parent1::method selected {
         return newMethod();
       }

       final type1 methodFromParent2 () renames parent2::method {
         return newMethod();
       }

       type1 newMethod () {
         ...
         return ...;
       }

       void doSomething () {
         methodFromSuperClass ();
       }
     }
     ```

In den Definitionen für methodFromParent1() bzw. methodFromParent2() wird sichergestellt, daß newMethod() aufgerufen wird, und daß sie in Subtypen nicht mehr redefiniert werden können. Nur noch newMethod() kann redefiniert werden, wodurch indirekt auch methodFromParent1() und methodFromParent2 redefiniert werden.

Dies entspricht der Alternative, daß unabhängig vom Kontext, in dem ein Methodenaufruf erfolgt, genau eine Methode der Klasse aClass aufgerufen wird.[18]

5.5.6 Zusammenfassung

Bei Namenskonflikten zwischen Feldern bzw. Signaturkonflikten zwischen Methoden müssen in den Klassen, in denen die Konflikte auftreten, die betroffenen Felder / Methoden mit neuen Namen versehen werden. Jeweils eines der betroffenen Elemente muß dabei den ursprünglichen Namen erhalten. Durch Umbenennungen werden keine neuen Felder oder Methoden definiert; bei Methoden bleiben die Anzahl und Typen der formalen Parameter unverändert, ggfs. können sie redefiniert werden.

Für alle Felder, die gleichen Namen *und* gleichen Typ haben, bzw. für alle Methoden, die gleiche Signatur *und* gleichen Rückgabetyp haben, muß jeweils ein Feld / eine Methode aus der Menge der betroffenen Elemente für konfluente Pfade ausgewählt werden.

5.6 Kapselung bei objektbasierter Vererbung

In Java kann die Sichtbarkeit von Elementen einer Klasse durch die Annotationen public, protected und private definiert werden. In rein klassenbasiertem Java reicht es aus, vertrauenswürdige Daten einer Klasse aClass in einem Feld zu speichern, das als protected deklariert ist: Zwar kann in Unterklassen auf solche Felder zugegriffen werden, jedoch nur bei Instanzen dieser Unterklassen, nicht aber bei Instanzen von aClass.

Diese rein auf Sichtbarkeit basierende Zugriffseinschränkung kann in Lava durch Delegation umgangen werden, indem in einer weiteren Klasse anotherClass ein Feld der Klasse aClass als delegatee deklariert wird. Diesem Feld kann dann jede beliebige Instanz von aClass zugewiesen werden; Felder, die in aClass als protected deklariert sind, können ausgelesen werden, da Lesezugriffe an das Elternobjekt delegiert werden.

5.6.1 delegatee Klassen

Um zu verhindern, daß bereits exisitierende Java-Klassen durch diese Sicherheitslücke betroffen sind, führen wir eine zusätzliche Annotation delegatee für

[18] Für Felder ist die Definition eines „einheitlichen Verhaltens“ in der beschriebenen Art und Weise nicht möglich. Es empfiehlt sich generell, in solchen und anderen Fällen Felder mit der Annotation private zu versehen und Zugriffsmethoden für diese Felder zur Verfügung zu stellen. Diese Zugriffsmethoden können dann ggfs. in Subtypen redefiniert werden.

Klassen ein, die eine Klasse für Delegation freigibt. Fehlt diese Annotation, so darf keine Delegation an Instanzen dieser Klasse erfolgen. Für Konsultation wird keine derartige Annotation benötigt, weil protected Elemente in konsultierenden Kindklassen nicht sichtbar sind und nicht redefiniert werden können.

Andere Annotationen

Die Annotation delegatee läßt sich mit allen anderen Klassennotationen (public, abstract und final) kombinieren. Da in Lava Delegation an Interfacetypen nicht möglich ist, können sie nicht mit der Annotation delegatee versehen werden.

Eine Unterklasse einer delegatee Klasse muß ebenfalls mit delegatee versehen sein, damit Delegation an Instanzen der Unterklasse erfolgen kann, wenn sie von delegatee Elternverweisen des Typs der Oberklasse referenziert werden.

Aus diesem Grund ist auch die gleichzeitige Verwendung der Klassenannotationen final und delegatee erlaubt: Die Unterklasse einer delegatee Klasse kann zwar final sein, womit weitere Kind- und Elternklasse nicht mehr von dieser Unterklasse abgeleitet werden können. Dennoch muß die Delegation an Instanzen der Unterklasse durch delegatee freigegeben sein.

5.7 Zusammenfassung

Instanzvariablen können mit Hilfe der Schlüsselwörter delegatee und consultee als Verweise auf Elternobjekte deklariert werden. Zulässige Typen für delegatee Felder sind Klassen, für consultee Felder zusätzlich Interfacetypen. Die Klasse, in der ein solcher Elternverweis definiert wird, wird dadurch um Elemente des Elterntyps erweitert, bei consultee Feldern um public und paketweit sichtbare Elemente, bei delegatee zusätzlich um protected Elemente. Eine Klasse kann auf mehr als ein Elternobjekt verweisen.

Nachrichten eines Elterntyps, zu denen in der Kindklasse keine Methodendefinitionen vorliegen, werden an die durch die entsprechenden Felder referenzierten Elternobjekte weitergeleitet. Bei Delegation bleibt dabei this an den ursprünglichen Empfänger der Nachricht gebunden, bei Konsultation wird this an das Elternobjekt gebunden. Nachrichten können in Methodendefinitionen der Kindklasse auch explizit an Elternobjekte delegiert werden.

Um Typsicherheit bei Delegation und Konsultation zu gewährleisten, ist es möglich, mandatory Felder zu deklarieren, denen der Wert null nicht zugewiesen werden darf. Soll es möglich sein, daß ein Feld mit dem Wert null belegt ist, so ist dieses Feld (explizit oder implizit) optional. Bei Zuweisungen an mandatory Felder wird ggfs. eine AssignmentOfNullException ausgelöst.

Einer Variablen vom Typ eines mandatory Elternverweises können auch Instanzen der Kindklasse zugewiesen werden. Bei der Verwendung von Elementen, die über einen optional Elternverweis geerbt werden, wird ggfs. eine ParentIsNullException ausgelöst, die überprüft werden muß.

Bei Nachrichten an this in einer Methode, die durch Delegation aus einem Kindobjekt aufgerufen wurde, wird im Delegationspfad nach einer passenden Methode gesucht, die eine Definition für dieselbe Nachricht darstellt.

Namenskonflikte, die sich durch die Deklaration von Verweisen auf Elternobjekte ergeben, müssen in der Kindklasse durch Umbenennungen der betroffenen Elemente aufgelöst werden. Dabei muß ein Element den ursprünglichen Namen erhalten. Aus einer Menge mehrerer Felder mit gleichem Namen und gleichem Typ bzw. Methoden mit gleicher Signatur und gleichem Rückgabetyp muß jeweils ein Element selektiert werden, das bei konfluenten Pfaden verwendet werden soll. Bei Nachrichten an this werden Umbenennungen berücksichtigt, Selektionen jedoch nicht.

Um die Kapselung von protected Feldern in bestehenden Java-Klassen nicht zu verletzen, müssen Klassen durch die Annotation delegatee explizit für Delegation freigegeben werden. Konsultation ist bei allen Klassen möglich.

Kapitel 6

Das Strategy Pattern

Ich stelle im folgenden ein Programmierproblem vor, daß zunächst ohne Hilfe von objektbasierter Vererbung implementiert wird. Es handelt sich hierbei um das Beispiel, daß in dem Buch „Design Patterns“ [Gamma 95] verwendet wird, um das sog. *Strategy Pattern* zu verdeutlichen. Dabei beschreibe ich verschiedene Alternativen, die jeweils ihre Vor- und Nachteile haben. Zuletzt demonstriere ich, wie das Beispiel in Lava implementiert werden kann. Es zeigt sich, daß die Verwendung der neuen Sprachkonstrukte von Lava wesentliche Vorteile gegenüber eine Implementation in Java ohne objektbasierte Vererbung bietet.

6.1 Implementation in Java

Angenommen, eine Klasse Composition ist dafür zuständig, die Zeilenumbrüche eines Texts zu verwalten, der in einem Textfenster dargestellt wird. Änderung der Breite eines Textfensters führt dazu, daß die Zeilenumbrüche neu berechnet werden müssen. Die Strategien, die verwendet werden, um dies zu bewerkstelligen, werden allerdings nicht direkt in der Klasse Composition implementiert, sondern in Unterklassen der abstrakten Klasse Compositor (s. Abb. 6.1). Diese Klassen implementieren verschiedenen Strategien:

SimpleCompositor implementiert eine einfache Strategie, die eine Zeile solange füllt, bis kein Platz mehr vorhanden ist, einen Zeilenumbruch setzt, und anschließend mit der folgenden Zeile fortfährt.

TeXCompositor verwendet den TEX-Algorithmus, um Zeilenumbrüche zu finden. Dabei werden einzelne Wörter so lange in der Breite gedehnt oder gestaucht, bis ganze Absätze möglichst optimal formatiert sind.

ArrayCompositor implementiert eine Strategie, so daß jede Zeile eine feste Anzahl an Elementen (z.B. Icons) enthält.

Ein Composition Objekt enthält einen Verweis compositor auf ein Objekt vom Typ Compositor. Immer, wenn ein Text neu formatiert werden muß, wird das

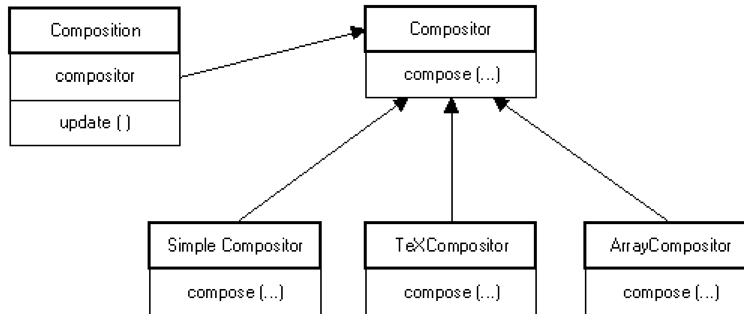

Abbildung 6.1: Beispiel für ein Strategy Pattern

Composition Objekt diese Aufgabe an compositor weiterleiten. Welche Strategie letztenendes zur Anwendung kommt, wird dadurch bestimmt, auf welches Objekt das Feld compositor zum Zeitpunkt der Formatierung verweist.

Erste Variante: Bereitstellung aller Informationen über Parameter

Die Klasse Composition könnte wie folgt implementiert sein:

```
class Composition {

  // Einige Deklarationen
  ...

  Compositor compositor;

  // Hier wird ein Text neu formatiert
  void update() {
    ...

    // Zeilenumbrüche berechnen lassen
    lineBreaks = compositor.compose(
      natural, stretchability, shrinkability,
      componentCount, lineWidth);

    ...
  }
}
```

Die Parameter natural, stretchability und shrinkability sind Arrays, die für jedes Zeichen im Text die natürliche Größe und die Anzahl der Pixel, um die es gedehnt bzw. gestaucht werden kann. In componentCount wird die Anzahl der Zeichen übergeben, in lineWidth wird die Breite in Pixeln übergeben, für die die Formatierung erfolgen soll.

Neben den Vorteilen, die diese Implementationstechnik bietet, gibt es einen entscheidenden Nachteil: Da nicht vorhersehbar ist, welche Strategie jeweils zum

Einsatz kommt, müssen der Methode compose eines Compositor Objekts immer alle Informationen (natural, stretchability und shrinkability) übergeben werden, die evtl. von ihr benötigt werden, um die Berechnung überhaupt durchzuführen. Man kann sich leicht vorstellen, daß die Bereitstellung dieser Informationen sehr aufwendig ist, und darüberhinaus sogar ggfs. unnötig, wie im Fall von SimpleCompositor, wo nur die „natürlichen“ Größen der Komponenten berücksichtigt werden müssen, oder im Fall von ArrayCompositor, wo keine dieser Informationen benötigt werden. Allein im Fall von TeXCompositor werden alle Informationen verwendet.

Zwei Variante: Statischer Rückverweis auf das Composition Objekt

Es wäre sehr wünschenswert, wenn die benötigten Informationen nur bei Bedarf ermittelt werden. Dies kann man erreichen, indem man im Compositor Objekt einen Verweis auf das Composition Objekt speichert, zu dem es gehört und die entsprechenden Informationen in einzelnen Methoden der Composition Klasse berechnen läßt. Im Composition Objekt muß z.B. im Konstruktor dieser Verweis in compositor belegt werden.

Die Klasse Composition sieht dann in etwa wie folgt aus:

```
class Composition {

  // Einige Deklarationen

  Compositor compositor;

  // Konstruktor
  Composition (Compositor compositor) {
     this.compositor = compositor;
     compositor.setComposition(this);
  }

  int[] getNatural() {...}
  int[] getStretchability() {...}
  int[] getShrinkability() {...}

  // Hier wird ein Text neu formatiert
  void update() {
    ...

    // Zeilenumbrüche berechnen lassen
    lineBreaks = compositor.compose(
      componentCount, lineWidth);

    ...
  }
}
```

Man beachte: Zum Einen muß im Konstruktor dem Compositor Objekt noch mitgeteilt werden, zu welchem Composition Objekt es gehört. Zum Anderen entfällt in der Methode update die Notwendigkeit, alle Informationen zu berechnen.

Die Klasse Compositor könnte wie folgt definiert sein:

```
abstract class Compositor {

  Composition myComposition;

  void setComposition(Composition composition) {
    myComposition = composition;
  }

  abstract int[] compose(int componentCount, int lineWidth);
}
```

Innerhalb der Methode compose in den einzelnen Unterklassen von Compositor muß dann jeweils die entsprechende Methode des Felds myComposition aufgerufen werden, um die jeweils benötigten Informationen zu erhalten, z.B. wie folgt:

```
    int natural[] = myComposition.getNatural();
```

Wie man sieht, wird der gewünschte Effekt erreicht: Es werden immer nur die von den jeweiligen Strategien tatsächlich benötigten Informationen berechnet. Es ergibt sich aber wiederum ein Nachteil, und zwar aufgrund der Tatsache, daß im Compositor Objekt eine Referenz auf das Composition Objekt enthalten ist, zu dem es gehört. Damit kann ein solches Compositor Objekt nicht mehr gemeinsam von mehreren Composition Objekten gleichzeitig genutzt werden.

Es wäre zwar möglich, daß vor dem Aufruf der Methode compose die Methode setComposition aufgerufen wird, um sicherzustellen, daß sich das Compositor Objekt auf das richtige Composition Objekt bezieht. Somit wäre eine Nutzung durch mehrere Composition Objekte möglich. Diese Vorgehensweise ist aber fehleranfällig – ein Programmierer könnte den Aufruf von setComposition evtl. vergessen – und darüberhinaus im Kontext von nebenläufigen Prozessen nicht sinnvoll, da ein Compositor Objekt während der Berechung von Zeilenumbrüchen für andere Composition Objekte in jedem Fall gesperrt werden müßte, was insb. beim aufwendigen TeXCompositor zu unnötigen Wartezeiten führen würde.

Dritte Variante: Dynamischer Rückverweis auf das Composition Objekt

Eine Alternative, um dieses Problem in den Griff zu kriegen, ist, das Composition Objekt, zu dem ein Compositor Objekt gehört, nicht fest in letzterem zu speichern, sondern es statt dessen beim Aufruf einer Methode zu übergeben. Es folgt wieder ein Quelltext, der verdeutlicht, was gemeint ist:

```
class Composition {

  // Einige Deklarationen

  Compositor compositor;

  int[] getNatural() {...}
```

```
  int[] getStretchability() {...}
  int[] getShrinkability() {...}

  // Hier wird ein Text neu formatiert
  void update() {
    ...

    // Zeilenumbrüche berechnen lassen
    lineBreaks = compositor.compose(
      this, componentCount, lineWidth);

    ...
  }
}
```

Man beachte, daß die Methode setComposition in der Klasse Compositor nicht mehr verwendet wird, und daher dort entfallen kann. Die Methode compose sieht dann in einer Unterklasse von Compositor etwa wie folgt aus:

```
class SimpleCompositor extends Compositor {

  // Berechnung von Zeilenumbrüchen
  int[] compose(Composition myComposition,
                int componentCount, int lineWidth) {
    int natural[] = myComposition.getNatural();

    ...
  }
}
```

Jetzt können mehrere Composition Objekte auf jeweils ein Compositor Objekt zurückgreifen.

Was jetzt noch fehlt ist die allgemeine Wiederverwendbarkeit der Klasse Compositor und deren Unterklassen: Da der Methode compose laut Deklaration ein Objekt des Typs Composition übergeben werden muß, ist die Verwendung dieser Klassen auf die Klasse Composition und deren Unterklassen eingeschränkt. (Z.B. könnte ein Text statt auf dem Bildschirm auf dem Drucker ausgegeben werden; für diese Fälle können grundsätzlich die gleichen Layout-Routinen verwendet werden.)

Vierte Variante: Deklaration eines Interfaces

Die Tatsache, daß die Klasse Compositor nur eingeschränkt verwendet werden kann, kann in Java umgangen werden, indem bei der Deklaration der Methode compose kein Klassentyp, sondern ein korrespondierender Interfacetyp verwendet wird, der z.B. wie folgt deklariert ist:

```
interface CompositorInterface {
   int[] getNatural();
   int[] getStretchability();
   int[] getShrinkability();
}
```

Die Klasse Composition (und alle anderen Klassen, die Compositor Objekte verwenden wollen) muß dann dieses Interface explizit implementieren.

Eine allgemeine Wiederverwendbarkeit der Klasse Compositor ist damit dennoch nicht erreicht. Zum Einen muß der Entwickler der Klasse Compositor diese Art der Wiederverwendung vorsehen, indem er den entsprechenden Interfacetyp deklariert und als Typ des entsprechenden Parameters der Methode compose einsetzt. Im Interfacetyp muß auch festgelegt werden, welche Methoden der Klasse Compositor zur Verfügung gestellt werden müssen. Das Verhalten von Compositor-Objekten kann also nicht beliebig angepaßt werden.

Darüberhinaus kann es erforderlich sein, daß die Unterklassen von Compositor ebenfalls auf diese Art und Weise wiederverwendet werden sollen, beispielsweise TeXCompositor verbunden mit einer gezielten Anpassung verschiedener Eigenschaften des TeX-Algorithmus.

Es müßte demnach für jede Unterklasse von Compositor, die derart wiederverwendbar sein soll, ein zusätzlicher Interfacetyp deklariert werden, was nicht nur sehr aufwendig ist, sondern weitere Typprobleme bezüglich der der Klassen aufwirft, die Compositor und deren Unterklassen wiederverwenden wollen. Ich verzichte auf eine eingehendere Darstellung dieser Typprobleme, und stelle im folgenden eine Lösung dar, bei der diese Probleme von vorneherein vermieden werden.

6.2 Implementation in Lava

Mit Hilfe von objektbasierter Vererbung läßt sich dieses Beispiel für die Anwendung eines Strategy Patterns direkter und natürlicher umsetzen. Zu diesem Zweck muß die Klasse Compositor durch die Angabe des Schlüsselworts delegatee für Delegation freigegeben werden. Außerdem werden die Methoden getNatural, getStretchability und getShrinkability mit in die Klasse Compositor aufgenommen. Somit kann die Methode compose diese Methoden direkt aufrufen und muß sich nicht darum kümmern, ob sie von einem Composition Objekt (oder einem Objekt einer anderen Klasse) zur Verfügung gestellt werden. Gegebenenfalls können Unterklassen von Compositor die Methoden derart redefinieren, daß Compositor Objekte eigenständig verwendet werden können.

Hier der Quelltext der Klasse Compositor:

```
abstract delegatee class Compositor {
   int[] getNatural() {return new int[0];}
   int[] getStretchability() {return new int[0];}
   int[] getShrinkability() {return new int[0];}

   abstract int[] compose(int componentCount, int lineWidth);
}
```

Damit ein Composition Objekt die Berechnung von Zeilenumbrüchen an ein Compositor Objekt weiterleiten kann, wird das Feld, das dieses Compositor Objekt referenziert mit der Annotation delegatee versehen. Auf diese Art und Weise erbt

ein Composition Objekt die Methoden des Compositor Objekts. Die Methoden getNatural, getStretchability und getShrinkability werden redefiniert. Da aufgrund von Delegation bei einem Aufruf der Methode compose innerhalb von update sichergestellt wird, daß Methodenaufrufe innerhalb von compose wieder zurück an das Composition Objekt geschickt werden, sind diese Redefinitionen ausreichend, um dem Compositor Objekt die benötigten Informationen zur Verfügung zu stellen.

Man beachte, daß beim Aufruf von compose innerhalb von update nicht mehr angegeben wird, daß es sich um eine Methode des Compositor Objekts handelt. Der Aufruf lautet also jetzt:

```
class Composition {

   delegatee Compositor compositor;

   void update() {
     ...

     // Zeilenumbrüche berechnen lassen
     lineBreaks = compose(componentCount, lineWidth);

     ...
   }
}
```

Ebenso wird beim Aufruf von getNatural, getStretchability und getShrinkability innerhalb der Methode compose in den einzelnen Unterklassen von Compositor nicht mehr angegeben, daß es sich um Methoden eines Composition Objekts handelt. Der Aufruf lautet also jetzt etwa:

```
class SimpleCompositor extends Compositor {

   int[] compose(int componentCount, int lineWidth) {
     int natural[] = getNatural();

     ...
   }
}
```

6.3 Zusammenfassung

Man sieht also, daß bei Verwendung von Delegation sich wesentliche Vorteile ergeben:

- In der Compositor Klasse müssen keine aufwendigen Vorkehrungen getroffen werden, um die Verwendung in der Compositor Klasse (oder anderen Klassen) zu ermöglichen. Lediglich das Schlüsselwort delegatee muß angegeben werden, um die Klasse für Delegation freizugeben. Programmierer können sich daher auf die wesentliche Funktionalität der beteiligten Klassen konzentrieren.

- Das Einbinden der Methoden getNatural, getStretchability und getShrinkability in die Compositor Klasse ist keine besondere Vorkehrung für eine Verwendung in einer Composition Klasse – im Gegenteil ist es natürlicher, diese Methoden dort zu definieren, wo sie benötigt werden, statt sich darauf zu verlassen, daß eine delegierende Klasse diese zur Verfügung stellt. Das Schreiben eines speziellen Interfaces entfällt ebenfalls.

- Die Methoden der Compositor Klasse werden an die Composition Klasse vererbt. Für Klassen, die die Composition Klasse verwenden, sehen sie daher aus wie „normale“ Methoden der Composition Klasse. Wird die Composition Klasse für Delegation freigegeben, können sie wiederum redefiniert werden. Würde z.B. getNatural() in einer solchen Klasse redefiniert, so wäre immer sichergestellt, daß je nach Kontext innerhalb von compose eine solche Redefinition von getNatural() aufgerufen wird. Es wird, wie bei objektorientierter Programmierung gewohnt, dynamisch zur Laufzeit bestimmt, welche Methode zur Ausführung kommt; ein Programmierer muß sich um die Details keine Gedanken machen.

- Die Compositor Klasse ist allgemein wiederverwendbar. Es ist nicht festgelegt, welche Methoden von Kindklassen „bereitgestellt“ werden müssen, damit Compositor Objekte ihre Aufgabe erfüllen können. Zum Einen kann bei Bedarf sogar die Methode compose in einer Kindklasse redefiniert werden; zum Anderen können Unterklassen von Compositor evtl. eigenständig verwendet werden. Darüberhinaus können auch beliebige Unterklassen von Compositor problemlos als Elterntyp verwendet werden. In den Unterklassen evtl. neu hinzugekommene Methoden können bei Bedarf ebenfalls redefiniert werden.

Es zeigt sich also, daß Design Patterns, die in tatsächlich existierenden Softwarelösungen verwendet werden, mit Hilfe der neuen Sprachkonstrukte in Lava für objektbasierte Vererbung wesentlich einfacher implementiert werden können. Die allgemeine Wiederverwendbarkeit der beteiligten Klassen ist dabei ohne besondere Vorkehrungen durch Programmierer gewährleistet.

Kapitel 7

Design der Lava Virtual Machine

Ich gehe jetzt ausführlich auf alle Erweiterungen der Spezifikation der Java Virtual Machine ein, die wir beim Design der Lava Virtual Machine vorgenommen haben. Von den Änderungen betroffen sind im Wesentlichen das Class File Format, sowie der Befehlssatz der JVM. Für eine zusammenfassende Darstellung der LVM verweise ich auf Anhang B.

Die Beispiele in Maschinensprache, die ich im folgenden verwende, werden in einer Notation gegeben, die in etwa der Notation aus [Lindholm 96], S. 340 entspricht. Aus Gründen der besseren Lesbarkeit weiche ich jedoch an verschiedenen Stellen ein wenig von dieser Notation ab, ohne daß ich näher darauf hinweise.

7.1 Feldannotationen

In der Sprache Lava wurden die neuen Annotationen delegatee, consultee, mandatory und optional für Felddeklarationen eingeführt. Diese werden im Class File Format abgebildet, wie es bereits in der JVM für die bekannten Annotationen für Felder geschieht. Im Class File Format ist für solche Annotationen das Element access_flags der Struktur field_info vorgesehen.[1] Diese Feld umfaßt zwei Bytes, darin wird jede Annotation durch ein Bit repräsentiert, das genau dann gesetzt ist, wenn sie in der Felddeklaration vorhanden ist.

Da die für die in Java bekannten Annotationen bereits reservierten Bitpositionen keinen Platz mehr für unsere neuen Annotationen lassen, wurde für Lava ein Attribut mit dem Namen ExtAccessFlags definiert. Dieses Attribut enthält zwei Bytes, in denen weitere Annotationen repräsentiert werden können. Für die jetzige Version von Lava sind vier Bits entsprechend den neuen Annotationen reserviert. Zusätzlich wird ein Bit für die Angabe selected bei einer Umbenennung verwendet. Die restlichen Bits werden ignoriert.

[1] s. [Lindholm 96], S. 101ff

Es wäre möglich gewesen, das bereits bestehende Feld access_flags der Struktur field_info um zwei weitere Bytes zu erweitern, um damit den nötigen Platz für die neuen Annotationen von Lava zu schaffen – damit wäre aber die Kompatibilität zur JVM verloren gegangen. Bei der gewählten Lösung wird eine Implementation der JVM die ExtAccessFlags ignorieren – damit sind Lava-Kompilate lauffähig, die die neuen Sprachkonstrukte von Lava nicht verwenden. Liest hingegen eine Implementation der LVM ein Java-Kompilat ein, so werden für die fehlenden ExtAccessFlags die sich aus der Spezifikation von Lava ergebenden Standardwerte angenommen.

7.2 Feldzugriffe

Feldzugriffe werden in Java mit Hilfe der Befehle *getfield* für Lesezugriffe und *putfield* für Schreibzugriffe übersetzt. Die Semantik dieser Befehle wurde erweitert, so daß Zugriffe auf Feldern aus Elternobjekten möglich ist. Ggfs. wird eine ParentFieldIsNullException ausgelöst. Das Befehlsformat konnte unverändert beibehalten werden.

7.3 Übersetzung von Methoden

Instanzmethoden werden in herkömmlichen Programmiersprachen (inkl. Java) wie eine Prozedur mit einem Parameter übersetzt, der zu den deklarierten Parametern hinzugefügt wird. Dieser Parameter wird bei Aufruf der Methode an das Objekt gebunden, für das die Methode ausgeführt werden soll. Im Quelltext einer Methode ist dieser Parameter als this sichtbar, es handelt sich also um das bereits erwähnte this-Objekt. Mit Hilfe dieses Parameters ist es möglich, innerhalb einer Methode eine Nachrichtensendung an this als weiteren Methodenaufruf zu implementieren: Der Wert von this wird auf den Parameterstack gelegt; danach werden die Werte für die weiteren Parameter der Methode ebenfalls auf den Parameterstack gelegt; zuletzt wird die Methode aufgerufen.

In Lava muß ggfs. in einer Methode zwischen dem this-Objekt und dem Träger der aktuellen Methode (im folgenden delegatee genannt) unterschieden werden, da es sich um verschiedene Objekte handeln kann. Das this-Objekt kann in Lava auf ein Kindobjekt von delegatee verweisen – es wird benötigt, um bei Nachrichten an this eine evtl. Redefinition einer Methode in einem Kindobjekt zu finden. Das delegatee-Objekt wird z.B. für Feldzugriffe oder Aufrufe von private Methoden benötigt.

Daher müssen Instanzmethoden in Lava mit einem weiteren Parameter übersetzt werden, der für delegatee-Objekt verwendet wird. Um die Kompatibilität mit der JVM zu bewahren, darf aber das bestehende Methodenformat nicht geändert werden. Daher müssen die Kompilate für Methoden in zwei Versionen vorliegen: Eine Version wird für herkömmliche, nicht-delegierte Methodenaufrufe bereitgestellt, bei denen das this-Objekt und das delegatee-Objekt identisch sind. Eine andere Version wird für delegierte Methodenaufrufe bereitgestellt, bei denen diese beiden Objekte unterschieden werden.

Im folgenden verwende ich die Bezeichnung *delegierte Methode* als Kurzform für die Version einer Methode, die für delegierte Nachrichten ausgeführt wird, entsprechend *nicht-delegierte Methode* als Kurzform für die Version einer Methode, die für Nachrichten ausgeführt wird, die explizit an den Träger der Methode gesendet werden. Der Code einer delegierten Methode wird als *delegierter Code* bezeichnet, entsprechend ist der Code einer nicht-delegierten Methode *nicht-delegierter* Code.

Das Kompilat einer Methode wird für die JVM im Attribut Code der Struktur method_info gespeichert.[2] Für die LVM wird in diesem Attribut die nicht-delegierte Version einer Methode gespeichert. Die delegierte Version einer Methode wird im neu definierten Attribut DelegatedCode gespeichert, dessen Struktur der des Code-Attributs entspricht – lediglich das Kompilat unterscheidet sich darin, daß bei DelegatedCode ein zusätzlicher Parameter wie oben beschrieben erwartet wird und daß dieser bei Zugriffen auf Felder von this und Nachrichten an this berücksichtigt wird.

Zugriffe auf Felder von this werden dabei als Zugriffe auf Felder des delegatee-Objekts übersetzt. (s. Kapitel 5.4.1) Die Übersetzung von Nachrichten an this wird in Kapitel 7.5.3 beschrieben.

Die Abwärtskompatibilität ist durch das Attribut DelegatedCode gewährleistet: Eine Implementation der JVM ignoriert es und ruft immer die nicht-delegierte Methode auf. Die Aufwärtskompatibilität erscheint zunächst problematisch, da Class Files, die den Spezifikationen der JVM entsprechen, kein DelegatedCode-Attribut kennen. Da jedoch aufgrund der Spezifikation der Sprache Lava nur an Instanzen von Klassen delegiert werden darf, die dafür durch ein neues Sprachkonstrukt explizit freigegeben werden, wirft auch dieser Aspekt der LVM keine Kompatibilitätsprobleme zur JVM auf.

7.4 Klassenannotationen

Die Klassenannotation delegatee, mit der in Lava eine Klasse für Delegation freigegeben wird, wird im Attribut ExtAccessFlags der Struktur ClassFile gespeichert.[3] Das Attribut entspricht dem Attribut ExtAccessFlags für Feldannotationen. (s. Kapitel 7.1)

7.5 Methoden aus Elterntypen

7.5.1 Unterscheidung zwischen this-Objekt und Träger der aktuellen Methode

Für jede public, protected oder ggfs. paketweit sichtbare Instanzmethode aus einem Elterntyp, die durch objektbasierte Vererbung in einer Kindklasse geerbt wird, wird im Class File eine neue method_info-Struktur[4] angelegt. Dabei werden

[2] s. [Lindholm 96] S. 104ff und S. 110ff
[3] s. [Lindholm 96], S. 84ff
[4] s. [Lindholm 96], S. 104ff

die access_flags ACC_PUBLIC und ACC_PROTECTED für public bzw. protected Methoden übernommen, sowie ACC_FINAL für final Methoden, die nicht redefiniert werden dürfen. Auch die Namen und Deskriptoren werden übernommen, es sei denn, es wurden Umbenennungen (s. Kapitel 7.6) vorgenommen, sowie die Liste der auslösbaren Ausnahmen[5]. Werden die Methoden über einen optional Elternverweis geerbt, so wird dieser Liste die ParentIsNullException hinzugefügt.

Falls für eine Methode aus einem Elterntyp in der erbenden Klasse eine Redefinition vorliegt, so wird in den Attributen Code und ggfs. DelegatedCode der nicht-delegierte bzw. delegierte Code der Methode gespeichert. Liegt keine Redefinition vor, oder darf im Fall einer final Methode keine Redefinition vorliegen, so muß gewährleistet sein, daß bei Aufruf der Methode die Methode des entsprechenden aktuellen Elternobjekts aufgerufen wird. Ein Compiler für Lava muß daher automatisch einen speziellen Code für diesen Fall erzeugen, der genau dieses gewünschte Verhalten bewirkt, und diesen in einer neu angelegten method_info-Struktur ablegen. Es handelt sich hierbei um den *Customized Lookup Code*, der bereits in [Kniesel 95] beschrieben wird.

Im Fall von Konsultation muß hierbei die nicht-delegierte Version der Methode des Elternobjekts aufgerufen werden, da das this für die Ausführung dieser Methode an das Elternobjekt gebunden wird. Aus diesem Grund kann der bereits in der JVM spezifizierte Methodenaufruf verwendet werden. Der Customized Lookup Code für eine Methode, wie z.B. void method() sieht daher in der nicht-delegierten Version wie folgt aus:

```
Methode void method()
    aload_0                                   // Lege this auf den Stack
    getfield parent                           // Ermittle parent
    invokevirtual ParentClass.method()        // Rufe method() auf
    return
```

Die delegierte Version unterscheidet sich darin, daß das Elternobjekt über delegatee, und nicht über this ermittelt wird. Statt *aload_0* wird also der Befehl *aload_1* verwendet. Im Fall von Parameterübergabe und Rückgabewerten muß entsprechend übersetzt werden.[6]

Im Fall von Delegation muß die delegierte Version der Methode des Elternobjekts aufgerufen werden, da das this die Bindung beibehält und nur der Träger der ausführenden Methode sich ändert. Für diesen Zweck stellt die JVM keinen geeigneten Befehl bereit, daher wurde für die LVM ein neuer Befehl *invokedelegated* definiert, der sich wie *invokevirtual* verhält, bis auf die Tatsache, daß statt einem zwei zusätzliche Parameter übergeben werden müssen, und daß die delegierte Version einer Methode ausgeführt wird. Der Customized Lookup Code für eine Methode, wie z.B. method() sieht dann in der nicht-delegierten Version wie folgt aus:

```
Method void method()
    aload_0                                   // Lege this auf den Stack
```

[5] Attribut Exceptions

[6] Ausführliche Beispiele sind in [Lindholm 96], S. 339ff zu finden.

```
                                            // (Parameter für method())
aload_0                                     // Lege this auf den Stack
getfield parent                             // Ermittle parent
invokedelegated ParentClass.method()        // Rufe method() auf
return
```

Die delegierte Version unterscheidet sich wieder darin, daß das Elternobjekt über delegatee, und nicht über this ermittelt wird.

7.5.2 Explizite Delegation

Mit Hilfe des Befehls *invokedelegated* kann auch explizite Delegation realisiert werden. Ein Ausdruck der Form:

```
parent <- method();
```

wird in nicht-delegiertem Code übersetzt in:

```
aload_0                                     // Lege this auf den Stack
                                            // (Parameter für method())
aload_0                                     // Lege this auf den Stack
getfield parent                             // Ermittle parent
invokedelegated ParentClass.method()        // Rufe method() auf
```

Diese Befehlssequenz entspricht der des Customized Lookup Codes für delegierte Methoden. Umgekehrt könnte also der Customized Lookup Code auch explizit in Lava wie folgt programmiert werden:

```
class aClass {
  ParentClass parent;

  void method () {
    parent <- method();
  }
}
```

Dies ist jedoch nicht notwendig, und dient hier nur zur Verdeutlichung.

7.5.3 Nachrichten an this

Nachrichten an this müssen unterschiedlich behandelt werden, je nachdem ob sie innerhalb der delegierten oder nicht-delegierten Version einer Methode vorkommen.

Innerhalb von nicht-delegiertem Code gilt immer this == delegatee. Daher kann für Nachrichten an this der normale Methodenaufruf via *invokevirtual* erfolgen.

Innerhalb von delegiertem Code zeigt this ggfs. auf ein anderes Objekt als delegatee. Daher muß zunächst beginnend bei this nach einer Methode innerhalb des

aktuellen Delegationspfads gesucht werden, die zur Nachricht paßt. Anschliessend kann ein Methodenaufruf erfolgen: Ist nach der Suche this == delegatee so kann der normale Methodenaufruf via *invokevirtual* erfolgen. Andernfalls muß ein Befehl verwendet werden, der Delegation berücksichtigt. Da die Nachricht evtl. an eine Instanz gesendet wird, deren Klasse aus Sicht des sendenden Objekts unbekannt ist, wird hierfür ein neuer Befehl *invokeself* definiert, der sich ähnlich wie *invokedelegated* verhält, der aber keine Annahmen über die Klassenzugehörigkeit der Instanz voraussetzt, für die eine Methode aufgerufen wird.

Der Code für eine Nachricht an this innerhalb einer delegierten Methode sieht dann z.B. für this.method(parameters) etwa wie folgt aus:

```
candidate = FindDelegatee(this, method); // finde Empfänger für Nachricht

if (candidate == this) then {
  invokevirtual(method, this, parameters); // optimierter Methodenaufruf
} else {
  invokeself(method, this, candidate, parameters); // Methodenaufruf
}
```

Die Routine FindDelegatee wird im folgenden Abschnitt erläutert.

7.5.4 Suche nach Delegatees in der Elternhierarchie

Entsprechend der Spezifikation der Sprache Lava muß bei Nachrichten an this innerhalb von delegiertem Code gewährleistet sein, daß bei der Suche nach einem Objekt, das diese Nachricht versteht, der gleiche Pfad gewählt wird, der ursprünglich von this durch Delegation zum Träger der aktuellen Methode geführt hat. (s. Kapitel 5.4.2)

Zu diesem Zweck muß zur Laufzeit ein *Delegationsstack* verwaltet werden, der die jeweils bei Delegation ausgewählten Elternobjekte speichert. Es handelt sich dabei um keinen reinen FIFO-Stack, sondern es kann aus pragmatischen Gründen über ganzzahlige Indizes auf Elemente innerhalb eines aktuellen Stacks zugegriffen werden; für das erste Element, das in einem Stack gespeichert wird, ist hierbei der Index 0 gültig, für alle weiteren Elemente wird der Index jeweils um 1 erhöht.

Die Elemente, die in dem Delegationsstack gespeichert werden, bestehen zum Einen aus einem Verweis auf ein Objekt – darin wird das jeweilige Elternobjekt bei einer Delegation gespeichert. Zum Anderen ist ein Index auf ein Folgeelement innerhalb des aktuellen Stacks enthalten. Mit Hilfe dieses follow-Indexes läßt sich ausgehend von einem Startelement ein Pfad von Elternobjekten auf dem Stack verfolgen. Dieser Pfad läßt sich durch Ändern des follow-Indexes in einem Element ändern.

Der Delegationsstack enthält zwei Indizes: Der *aktuelle Index* zeigt auf das jeweils oberste Element des Stacks; er wird mit 0 intialisiert. (Der follow-Index des Elements bei Index 0 wird mit 1 initialisiert.) Der *Iterationsindex* wird bei der Suche nach einem Delegatee verwendet und wird zunächst nicht initialisiert.

Für jeden Thread muß ein eigener Delegationsstack existieren, damit Delegation in mehreren Threads gleichzeitig möglich ist.[7]

Der Stack kann als Klasse aufgefaßt werden, die folgende Methoden versteht:

void push(Object object) legt ein Objekt auf dem Stack ab: Das übergebene Objekt wird im Element beim aktuellen Index gemerkt; der aktuelle Index wird um 1 inkrementiert; der follow-Index des Elements beim anschließend aktuellen Index wird auf den aktuellen Index + 1 gesetzt.

void pop() entfernt ein Objekt vom Stack: Der aktuelle Index wird um 1 dekrementiert.

Stack newPath() legt einen neuen Stack an, initialisiert diesen und gibt den alten Stack zurück.

void restorePath(Stack path) verwirft den aktuellen Stack und stellt den übergebenen Stack wieder her.

void startIterate() bereitet eine Iteration vor: Der Iterationsindex wird mit 0 initialisiert.

int getIterate() der Wert des Iterationsindexes wird zurückgegeben.

void setIterate(int index) dem Iterationsindex wird der übergebene Wert zugewiesen.

Object next() setzt die Iteration fort: Das Objekt des Elements beim Iterationsindex wird zurückgegeben; dem Iterationsindex wird der Wert des follow-Indexes dieses Elements zugewiesen.

int splitPath() der Delegationspfad wird aufgeteilt: Dem follow-Index des Elements beim Iterationsindex wird der Wert des aktuellen Indexes zugewiesen; der alte Wert dieses follow-Indexes wird zurückgegeben.

joinPath(int index) der Delegationspfad wird zusammengeführt: Dem follow-Index des Elements beim Iterationsindex wird der übergebene Wert zugewiesen.

static Stack getStackForThread() diese Klassenmethode gibt den Stack für den aktuellen Thread zurück.

Nicht-delegierter Code

Innerhalb von nicht-delegierten Versionen von Methoden muß der Stack nur bei Ausführung des Befehls *invokedelegated* berücksichtigt werden. Er muß zunächst initialisiert werden, da durch diese Delegation ein neuer Suchpfad beginnt. Ausserdem muß das Objekt, an das delegiert wird, auf dem Stack gespeichert werden. Der Code für eine Delegation parent<-method(parameters) sieht demnach wie folgt aus:

[7] Der jeweils aktuelle Thread läßt sich in der JVM durch die Klassenmethode Threads.currentThread() ermitteln.

```
ds = Stack.getStackForThread(); // hole aktuellen Stack

oldPath = ds.newPath();  // lege neuen Stack an
ds.push(parent);         // lege Elternobjekt ab
invokedelegated(method, this, parent, parameters); // Delegation
ds.pop();                // entferne Elternobjekt
ds.restorePath(oldPath); // stelle alten Stack wieder her
```

Für eine Reihe aufeinanderfolgender Delegationen in delegiertem Code ist das einmalige Anlegen eines neuen Stacks vorher und Wiederherstellen des alten Stacks nachher ausreichend. Darüberhinaus muß die Ermittlung des Stacks für den aktuellen Thread nicht bei jeder Delegation erfolgen, sondern nur bei der ersten innerhalb des Codes einer Methode. Bei jeder weiteren Delegation wird der erstmalig ermittelte Stack verwendet. Dies gilt auch für alle anderen Codefragmente, die weiter unten folgen.

Delegierter Code

Innerhalb von delegierten Versionen von Methoden muß bei der Ausführung des Befehls *invokedelegated* zur Fortsetzung des aktuellen Suchpfads das Objekt, an das delegiert wird, auf dem Stack gespeichert werden. Der Code für eine Delegation parent<-method(parameters) sieht demnach wie folgt aus:

```
ds = Stack.getStackForThread(); // hole aktuellen Stack

ds.push(parent);         // lege Elternobjekt ab
invokedelegated(method, this, parent, parameters); // Delegation
ds.pop();                // entferne Elternobjekt
```

Der Code für Nachrichten an this in delegiertem Code sieht in vollständiger Fassung z.B. für this.method(parameters) aus wie folgt:

```
ds = Stack.getStackForThread(); // hole aktuellen Stack

ds.startIterate();              // bereite Iteration vor
candidate = this;               // "this" ist erster möglicher
                                // Empfänger der Nachricht

while (!accesssafe(method, delegatee, candidate) {
  // Die Klasse von "candidate" kennt nicht die
  // Methode "method" aus der Klasse von "delegatee"
  candidate = ds.next();        // hole nächsten Kandidat
}

// Die Klasse von "candidate" kennt die Methode
// "method" aus der Klasse von "delegatee"

if (candidate == this) then {
  invokevirtual(method, this, parameters);  // optimierter Methodenaufruf
} else {
  index = getIterate();         // merke Iterationsindex
  follow = splitPath();         // teile Pfad auf
```

```
  invokeself(method, this, delegatee, parameters);  // Methodenaufruf
  setIterate(index);            // stelle Iterationsindex wieder her
  joinPath(follow);             // führe Pfad wieder zusammen
}
```

Das FindDelegatee(...) aus dem Codefragment des letzten Abschnitts entspricht den ersten Zeilen dieses Codefragments bis zum Ende der while-Schleife. Bei *accesssafe* handelt es sich um einen weiteren neuen Maschinenbefehl, der den Safety Check für die Nachricht method ausführt, wobei delegatee der Sender und candidate der mögliche Empfänger der Nachricht ist. Dieser Safety Check wird entsprechend der Semantik der Sprache Lava durchgeführt. (s. 5.4.2)

Die while-Schleife wird spätestens beendet, wenn der Iterationszeiger des Delegationsstacks auf das oberste Element zeigt, da darin der Träger der aktuellen Methode, also delegatee gespeichert ist: Die aktuelle Methode wurde nämlich durch Delegation aufgerufen, folglich wurde bei dem entsprechenden Methodenaufruf der aktuelle delegatee auf dem Delegationsstack abgelegt.

Ist candidate == this, so kann wie gewohnt der optimierte Methodenaufruf via *invokevirtual* verwendet werden. Andernfalls wird der Delegationspfad bei dem Index aufgeteilt, bei dem das Objekt mit der passenden Methode gefunden wurde. Dadurch wird bei einer Nachricht an this innerhalb dieser Methode der Delegationsstack bis zu diesem Index wiederverwendet, ab diesem Index können neue Objekte auf dem Stack abgelegt werden.

Der Iterationszeiger muß gemerkt und wiederhergestellt werden, da er ggfs. bei weiteren Nachrichten an this wiederverwendet wird.

Zusammenfassung

Durch die oben beschriebene Behandlung des Delegationsstacks ist die Kompatibilität zur JVM gewährleistet – nur Code, der im Zusammenhang mit Delegation steht, muß eine gesonderte Behandlung des Delegationsstacks enthalten. Methodenaufrufe, die der Spezifikation der JVM entsprechen, benötigen keine besondere Vorkehrungen, daher können existierende Class Files weiterverwendet werden.

7.6 Umbenennungen und Selektionen

Eine Umbenennung eines Elements wird als neu definiertes Attribut Renaming in einem Class File abgebildet, das dementsprechend sowohl bei Feldern, als auch bei Methoden vorkommen kann. Der Name, der in der field_info bzw. method_info Struktur angegeben ist, ist der neue Name für das entsprechende Element in der aktuellen Klasse, im Attribut Renaming wird zum Einen der Name der Elternreferenz angegeben, aus dessen Typ das umbenannte Element stammt, bzw. „super“ im Fall von Umbenennung von Elementen aus der Oberklasse. Zum Anderen wird der ursprüngliche Name des umbenannten Elements angegeben.

Durch ein field_info Element, das ein Renaming-Attribut enthält, wird kein neues Feld definiert: In Instanzen der entsprechenden Klasse wird daher keine Variable

für ein solches Feld angelegt; bei einem Zugriff auf ein solches Feld wird auf das umbenannte Feld im jeweiligen Elternobjekt bzw. auf das aus der Oberklasse geerbte und umbenannte Feld in der gleichen Instanz zugegriffen.

Ein method_info Element, das ein Renaming-Attribut enthält, redefiniert die umbenannte Methode des jeweiligen Elternobjekts bzw. der Oberklasse. Die Attribute Code und ggfs. DelegatedCode enthalten entweder das Kompilat einer Methodendefinition im Quelltext, oder aber den vom Compiler erzeugten Customized Lookup Code. Nur im Fall einer abstract Methode ist weder Code, noch DelegatedCode vorhanden – Instanzen der entsprechenden Klasse können dann nicht erzeugt werden.

Eine Selektion eines Elements für konfluente Pfade wird als neues Flag in den ExtAccessFlags eines Elements gespeichert.

Das Renaming-Attribut und das Selektionsflag darf nur bei Elementen erscheinen, die das access_flag ACC_STATIC nicht gesetzt haben. M.a.W. dürfen nur bei Instanzelementen einer Klasse diese Informationen erscheinen. Außerdem darf das Selektionsflag nur erscheinen, wenn gleichzeitig ein Renaming-Attribut vorhanden ist.

7.7 Die Annotation mandatory

Ist ein Feld mit der Annotation optional versehen, so muß bei Zugriffen auf dieses Feld kein besonderer Code erzeugt werden, da bei Dereferenzierung gegebenenfalls eine NullPointerException ausgelöst wird. Im Fall einer Delegation an ein Feld mit der Annotation optional wird entsprechend eine ParentFieldIsNullException ausgelöst.

Für Felder, die mit der Annotation mandatory versehen sind, müssen entsprechend der Semantik der Sprache Lava zusätzliche Bedingungen zur Laufzeit eingehalten werden:

1. Nach Erzeugung einer Instanz, die ein solches Feld enthält, darf dieses nicht mit null belegt sein.
2. Bei Zuweisungen an ein solches Feld darf nicht null zugewiesen werden.

Erzeugung von Instanzen

Die erste Bedingung ist durch die Einschränkungen abgedeckt, die von der Sprache Lava für die Initialisierung eines mandatory Felds festgelegt sind. (s. Kapitel 5.3.1)

Zuweisungen

Die zweite Bedingung für mandatory Felder muß bei den Befehlen *putfield* bzw. *putstatic* der JVM überprüft werden. Der Code für eine Zuweisung field = value, wenn die Instanzvariable field mit der Annotation mandatory versehen ist, sieht daher wie folgt aus:

```
// Das oberste Element auf dem Stack ist value
       dup                                // Dupliziere es für den nächsten Befehl
       ifnonnull label                    // Wenn ungleich null,
                                          // mache weiter bei „label“
       new AssignmentOffNullException     // Erzeuge eine AssignmentOfNullException
       invokespecial <init>               // Initialisiere sie
       athrow                             // Löse sie aus
label: putfield field                     // Zuweisung
```

Ist field eine Klassenvariable, so wird *putstatic* statt *putfield* verwendet, der restliche Code bleibt unverändert.

7.8 Zusammenfassung

Die neuen Annotationen für Felder und Klassen werden in einem neuen Attribut ExtAccessFlags repräsentiert. Für Zugriffe auf Felder aus Elterntypen wird die Semantik der Befehle *putfield* und *getfield* der JVM modifiziert.

Für Instanzmethoden von Klassen, die für Delegation freigegeben sind, exisitiert ein neues Attribut DelegatedCode, das das Kompilat der delegierten Version der Methode enthält. Einer solchen Methode werden zwei implizite Parameter this und delegatee übergeben, um zwischen this-Objekt und Träger der Methode zu unterscheiden.

Es werden drei neue Befehle *invokedelegated*, *invokeself* und *accesssafe* definiert: *invokedelegated* wird verwendet, um die delegierte Version einer Methode auszuführen; dabei müssen Werte für zwei implizite Parameter this und delegatee übergeben werden. *invokeself* und *accesssafe* werden bei Nachrichten an this in delegiertem Code verwendet: *invokeself* verhält sich wie *invokedelegated*, macht aber keine Annahmen über die Klassenzugehörigkeit des Objekts, dessen Methode ausgeführt wird. *accesssafe* testet, ob ein Objekt für eine Nachricht aus der Klasse eines sendenden Objekts eine Methode bereitstellt.

Um bei Nachrichten an this den Delegationspfad nach einem Objekt mit passender Methode absuchen zu können, der zum Träger der aktuellen Methode geführt hat, wird ein Delegationsstacks verwaltet, der für jeden Thread eines Programms existiert. Dieser Delegationsstack ermöglicht die Wiederverwendung von Teilpfaden, wenn ein Objekt mit passender Methode innerhalb des Delegationspfads gefunden wird. Der Delegationsstack muß bei Delegation von Nachrichten gepflegt werden. Nicht-delegierter Code, der Delegation nicht verwendet, muß keine besondere Pflege des Delegationsstacks enthalten.

Umbenennungen und Selektionen werden im Renaming-Attribut und einem Selektionsflag in den ExtAccessFlags repräsentiert, die für Instanzelemente einer Klasse verwendet werden können.

Für mandatory Felder wird zusätzlicher Code erzeugt, der sicherstellt, daß sie sie bei Zuweisungen nicht den Wert null erhalten.

Kapitel 8

Der Lava-Compiler

In diesem Kapitel werde auf einige wesentliche Aspekte der Implementation des Lava-Compilers eingehen. Der Lava-Compiler wurde nicht vollständig neu implementiert, sondern es wurden die Originalquelltexte des JDK 1.0.2 von Sun Microsystems verwendet und modifiziert.

Ich werde im folgenden zunächst den Java-Compiler von Sun Microsystems beschreiben und danach auf die Erweiterungen eingehen, die ich vorgenommen habe, damit die neuen Sprachkonstrukte von Lava korrekt entsprechend der Spezifikation der Lava Virtual Machine übersetzt werden.

8.1 Der Java-Compiler von Sun Microsystems

Der Java-Compiler wurde selbst in Java geschrieben. Er besteht aus den Paketen sun.tools.java, sun.tools.javac, sun.tools.tree, sun.tools.asm und sun.tools.zip.

Die Pakete sun.tools.java und sun.tools.javac enthalten zum Einen die Klassen, die benötigt werden, um Java-Klassen und deren Elemente als Objektbaum zu repräsentieren. Dabei können sowohl Klassen verarbeitet werden, die als Quelltexte vorliegen, als auch Klassen, die als Class Files vorliegen. Dementsprechend sind auch Klassen enthalten, die Java-Quelltexte einlesen und parsen können. Zum Anderen ist das eigentliche Hauptprogramm des Java-Compilers enthalten (sun.tools.javac.Main), das die Übersetzung von Quelltexten initiiert und eine übersetzte Java-Klasse in ein Class File schreibt.[1]

Das Paket sun.tools.tree enthält die Klassen, die benötigt werden, um geparste Anweisungen und Ausdrücke als Objektbaum zu repräsentieren.

In sun.tools.asm sind Klassen enthalten, mit deren Hilfe Folgen von Maschinenbefehlen erzeugt werden, die der Spezifikation der JVM entsprechen.

Das Paket sun.tools.zip enthält schließlich Klassen, die es ermöglichen, Class Files aus ZIP-Dateien auszulesen.

Im folgenden gehe ich auf wesentliche Klassen der verschiedenen Pakete ein:

[1] Das Paket sun.tools.java enthält die Klassen, die vom Programm javadoc mitverwendet werden, das ebenfalls im JDK 1.0.2 enthalten ist.

8.1.1 Die Pakete sun.tools.java und sun.tools.javac

Umgebung: In den Interfaces sun.tools.java.RuntimeConstants und sun.tools.java.Constants sind konstante Werte definiert, die in allen anderen Klassen des Compilers direkt oder indirekt sichtbar sind. Dabei enthält RuntimeConstants alle Konstanten, die auch einer Implementation der JVM bekannt sein müssen; es handelt sich z.B. um Werte für Access Flags, Typcodes oder Befehlscodes. In Constants sind hingegen nur Konstanten enthalten, die dem Compiler bekannt sein müssen, wie z.B. die Schlüsselwörter der Sprache Java.

In der Klasse sun.tools.java.Environment werden Methoden definiert, mit deren Hilfe Informationen ermittelt werden können, die auf allen Ebenen des Compilers bekannt sein müssen. Es handelt sich dabei z.B. um Methoden, die eine Klassendefinition für einen gegebenen Klassennamen zurückgeben, oder die feststellen, ob eine Konversion von einem Typ zu einem anderen Typ erlaubt ist. Außerdem sind Methoden enthalten, die ermitteln, welche Flags beim Aufruf des Compilers gesetzt wurden, z.B. ob Debug-Informationen oder optimierter Code erzeugt werden soll. Darüberhinaus sind Methoden zur Ausgabe von Fehlermeldungen des Compilers enthalten.

Von Environment ist die Unterklasse sun.tools.javac.BatchEnvironment abgeleitet, die zusätzlich in der Lage ist, die von Environment definierten Methoden für mehrere Quelltexte zur Verfügung zu stellen.[2]

Parser: Die Klasse sun.tools.java.Scanner und die davon abgeleitete Klasse sun.tools.java.Parser sind für das Einlesen eines Quelltexts und das Erzeugen eines korrespondierenden Objektbaums zuständig, der aus Instanzen anderer Klassen der Pakete sun.tools.java, sun.tools.javac sowie sun.tools.tree besteht. Von Parser ist wiederum eine Unterklasse sun.tools.javac.BatchParser abgeleitet, die die Funktionalität von Parser für die Verarbeitung mehrerer Quelltexte erweitert. Außerdem wird hier für je eine Java-Klasse eine Instanz der Klasse SourceClass (s. S. 91) erzeugt und gefüllt.

Instanzen der Klasse sun.tools.java.Identifier stehen für Namen, die in einer Java-Klasse vorkommen. Dabei kann es sich um einfache Namen handeln, wie toString, oder zusammengesetzte, wie System.out. Die Klasse Identifier stellt sicher, daß für jeden Namen genau eine Identifier-Instanz existiert, so daß Namensvergleiche durch einfache Vergleiche der Objektidentität zweier Identifier-Instanzen effizient implementiert werden können.

Typen: Die Klasse sun.tools.java.Type wird verwendet, um Typen zu repräsentieren und Zugriff auf Bestandteile eines Typs zu ermöglichen, wie z.B. der Komponententyp eines Arrays oder der Rückgabetyp einer Methode. Unterklassen hiervon sind ArrayType, ClassType und MethodType, die alle Elemente des Pakets sun.tools.java sind.

Klassen: Eine Java-Klasse wird im Compiler durch eine Instanz der Klasse sun.tools.java.ClassDeclaration repräsentiert. Eine Instanz von ClassDeclaration verweist wiederum auf eine Instanz von BinaryClass für Klassen, die aus einem

[2] Der Compiler erlaubt es, mehrere Quelltexte bei einem Aufruf zu übersetzen. Dabei können zum Einen mehrere Quelltexte beim Aufruf angegeben werden; zum Anderen kann der Compiler dazu veranlaßt werden, Java-Klassen implizit ebenfalls zu übersetzen, die von den beim Aufruf angegebenen Java-Klassen abhänggig sind.

Class File, bzw. SourceClass für Klassen, die aus einem Quelltext gelesen wurden. Diese Instanzen werden jedoch nur bei Bedarf angelegt und tatsächlich eingelesen.

Die Klasse sun.tools.java.ClassDefinition wird verwendet, um die Defintion von Klassen zu repräsentieren. Es sind Methoden vorhanden, um auf die direkte Oberklasse einer Klasse, auf die implementierten Interfaces, auf die Annotationen der Klasse sowie auf die Felder und Methoden einer Klasse zugreifen zu können. Darüberhinaus existieren Methoden, die andere Eigenschaften einer Klasse behandeln, wie z.B. ob eine bestimmte Klasse Ober- oder Unterklasse dieser Klasse ist.

Von ClassDefintion sind die Unterklassen sun.tools.java.BinaryClass und sun.tools.javac.SourceClass abgeleitet. In BinaryClass existiert eine Methode, um die Defintion einer Klasse aus einem Class File zu lesen. Eine Instanz von SourceClass wird, wie oben beschrieben, vom Parser des Compilers erzeugt und gefüllt.

In SourceClass sind die Methoden check und compile definiert: Die Methode check führt alle notwendigen Checks durch, um festzustellen, ob eine Klasse zulässig ist. Darin ist z.B. enthalten, ob die direkte Oberklasse und die implementierten Interfaces erreichbar sind, ob Methodenredefintionen zulässig sind oder ob alle Elemente korrekt definiert sind. Die check-Methode nimmt aber auch ggfs. Ergänzungen vor: Z.B. wird der Standardkonstruktor für die Klasse eingefügt, falls im Quelltext keine Konstruktoren definiert sind.

Die Methode compile startet die eigentliche Übersetzung der internen Repräsentation einer Klasse in eine interne Repräsentation eines Class Files. Anschließend wird bereits in dieser Methode diese interne Repräsentation in ein tatsächliches Class File geschrieben.

Elemente: Die Klasse sun.tools.java.FieldDefinition wird verwendet, um die Defintion von Elementen einer Klasse zu repräsentieren. Dabei kann es sich nicht nur um Felder, sondern auch um Methoden, Konstruktoren und statischen Initialisierungscode handeln.[3] Es sind Methoden vorhanden, um auf die Klasse, in der ein Element definiert ist, auf die Annotationen des Elements, auf den Namen und Typ eines Elements, auf die Argumenttypen und Ausnahmen einer Methode, auf den Wert eines Elements[4] und auf verschiedene andere Eigenschaften eines Elements zugreifen zu können.

Von FieldDefinition sind die Unterklassen sun.tools.java.BinaryField und sun.tools.javac.SourceField abgeleitet. Sie werden als Repräsentationen von Elementen in Instanzen von BinaryClass bzw. SourceClass verwendet.

In FieldDefinition sind die Methoden check sowie code und codeInit definiert: Die Methode check führt alle notwendigen Checks durch, um festzustellen, ob ein Element zulässig ist. Darin ist z.B. enthalten, ob alle Ausnahmen, die von der Methoden ausgelöst werden können, Unterklassen von java.lang.Throwable sind oder ob alle lokalen Variablen vor einem lesenden Zugriff initialisiert werden. Diese Methode wird von der check-Methode von SourceClass aufgerufen. Die Methoden code und codeInit nehmen die Übersetzung eines Elements vor – dabei

[3] Die Namensgebung FieldDefintion erscheint daher etwas verwirrend.
[4] Das ist der Wert eines final Felds, oder der Rumpf einer Methode.

wird code für Methoden verwendet, codeInit für die Initialisierung von Feldern. Diese Methoden werden von der compile-Methode von SourceClass aufgerufen.

Von FieldDefintion ist noch die weitere Unterklasse sun.tools.tree.LocalField abgeleitet, die verwendet wird, um Variablen zu repräsentieren, die lokal zu einer Methode sind.

Die Klasse sun.tools.javac.CompilerField wird ebenfalls für Elemente einer Klasse verwendet, ist aber nicht von FieldDefintion abgeleitet. Instanzen dieser Klasse werden verwendet, um die interne Repräsentation des Kompilats von Methoden zu speichern. Sie werden erst nach Beendigung aller Checks von der compile-Methode von SourceClass erzeugt.

8.1.2 Das Paket sun.tools.tree

Die Klasse Node ist Oberklasse für alle Klassen in diesem Paket. Ein Ausschnitt aus der Klassenhierarchie dieses Pakets ist in Abb. 8.1 ersichtlich.

Ausdrücke

Die Klasse Expression ist Oberklasse für alle Ausdrücke, die in einem Java-Quelltext formuliert werden können. Neben verschiedenen Methoden, die Eigenschaften eines Ausdrucks ermitteln, wie z.B. ob es sich um einen konstanten Ausdruck handelt, oder Methoden, die einen Ausdruck auswerten oder vereinfachen, existieren eine Reihe von Methoden für drei wesentliche Aufgabenbereiche: Typcheck, Inlining und Übersetzung.

Typchecks werden durch die check-Methoden durchgeführt; es handelt sich dabei um:

check für Ausdrücke, deren Wert nicht verwendet wird (z.B. der Rückgabewert von Methodenaufrufen).

checkAssignOp für Ausdrücke, die auf der linken Seite einer Zuweisung mit Operator (z.B. +=) stehen.

checkInitalizer für Ausdrücke, die als Initialisierungswert bei Variablendeklarationen dienen.

checkLHS für Ausdrücke, die auf der linken Seite einer einfachen Zuweisung stehen.

checkValue für alle sonstigen Ausdrücke.

Das *Inlining* eines Ausdrucks wird verwendet, um Ausdrücke in eine dem Typcheck entsprechenden korrekten Form umzuwandeln. Beispielsweise wird ein Ausdruck, der aus einem Bezeichner besteht (wie die Bezeichner a und b im Ausdruck a + b), vom Parser als Instanz der Klasse IdentifierExpression repräsentiert. Wird dann aber während des Typchecks festgestellt, daß a lediglich als Abkürzung für this.a gemeint ist, weil keine lokale Definition für den Namen

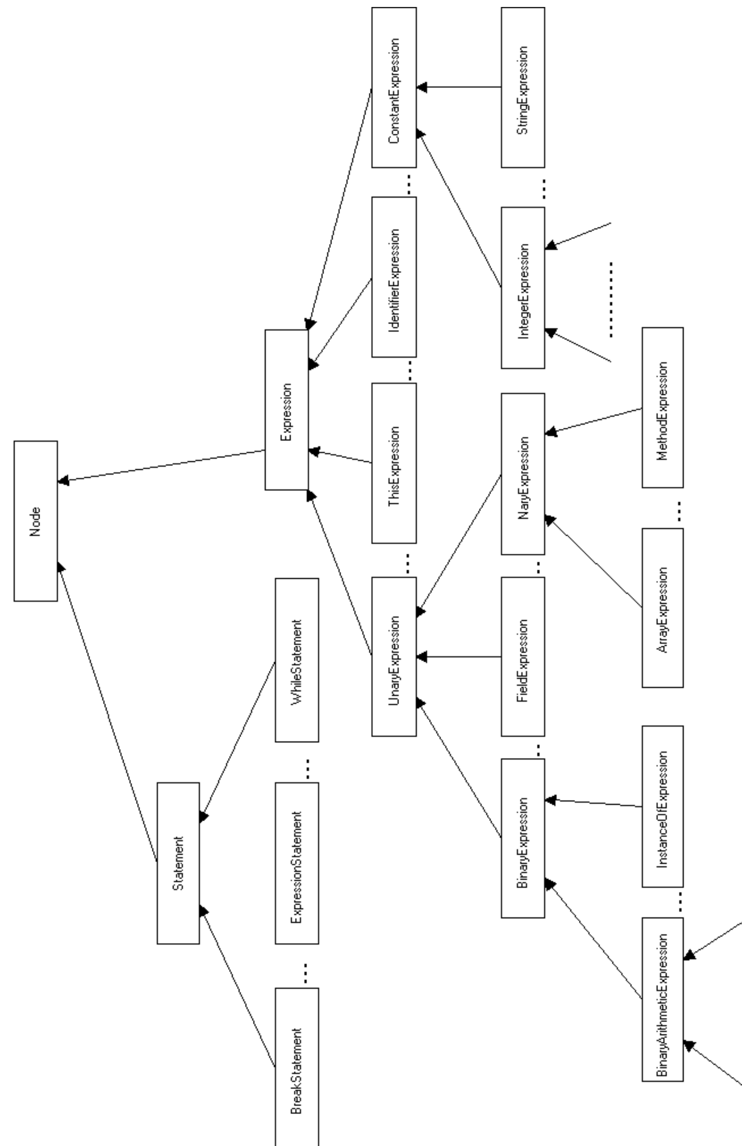

Abbildung 8.1: Die Klassenhierarchie des Pakets `sun.tools.tree`

a gegeben ist, dann wird durch das Inlining das ursprüngliche IdentifierExpression-Objekt durch eine entsprechende Instanz der Klasse FieldExpression mit den korrekten Verweisen auf this und a ersetzt.

Das Inlining wird in diesem Beispiel von einer der inline-Methoden des IdentifierExpression-Objekts durchgeführt: Dabei wird die FieldExpression-Instanz erzeugt, gefüllt und zurückgegeben. An der Stelle, an der die inline-Methode aufgerufen wurde, wird anschließend die Ersetzung des IdentifierExpression-Objekts durch das FieldExpression-Objekt vorgenommen.

In Expression existieren die folgenden inline-Methoden:

inline für Ausdrücke, deren Wert nicht verwendet wird.

inlineLHS für Ausdrücke, die auf der linken Seite einer Zuweisung stehen.

inlineValue für alle sonstigen Ausdrücke.

Die Übersetzung eines Ausdrucks wird von den code-Methoden durchgeführt, z.B.:

code für Ausdrücke, deren Wert nicht verwendet wird.

codeConversion für die Konversion eines Ausdrucks zu einem gewünschten Typ.

codeLValue für Ausdrücke, die auf der linken Seite einer Zuweisung stehen.

codeValue für alle sonstigen Ausdrücke.

Anweisungen

Die Klasse Statement ist Oberklasse für alle Anweisungen, die in einem Java-Quelltext auftreten können. U.a. sind auch hier verschiedene check-, inline- und code-Methoden für die Aufgaben Typcheck, Inlining und Übersetzung definiert.

Eine wichtige Unterklasse von Statement ist ExpressionStatement, mit deren Hilfe z.B. einen Methodenaufruf, der als MethodExpression-Objekt repräsentiert ist, als Anweisung aufgefaßt werden kann. Die Methoden für die verschiedenen Aufgaben werden dabei auf die Methoden der Klasse Expression abgebildet.

8.1.3 Das Paket sun.tools.asm

Eine Instanz der Klasse Assembler enthält das Kompilat für genau eine Methode. Dabei wird das Kompilat als einfach verkettete Liste von Instruction-Instanzen verwaltet. In Assembler sind Methoden enthalten, die z.B. das Hinzufügen von Befehlen ermöglichen, eine Optimierung des Kompilats vornehmen, die von den Befehlen verwendeten konstanten Werte bestimmen oder das Kompilat in eine Datei schreiben.

In der Klasse Instruction wird jeder Befehl durch seinen Opcode dargestellt. Desweiteren werden alle Argumente für einen Befehl in einer Instruction-Instanz

gespeichert. Es sind Methoden enthalten, die z.B. die Anzahl der Bytes, die ein Befehl belegt, oder die Anzahl der Worte ermitteln, um die der Operandenstack bei Ausführung des Befehls wächst oder schrumpft.

Die Klasse Label ist von der Klasse Instruction abgeleitet. Instanzen dieser Klasse repräsentieren Sprungziele innerhalb eines Kompilats. Für solche Instanzen wird kein echter Code im Class File erzeugt.

8.1.4 Der Übersetzungsvorgang

Die Übersetzung einer Java-Klasse läuft wie folgt ab:

In der Methode compile der Klasse sun.tools.javac.Main wird die Java-Klasse zunächst durch eine BatchParser-Instanz geparst. Dabei wird eine SourceClass-Instanz erzeugt, die wiederum auf eine Liste von SourceField-Instanzen für die Elemente der Klasse verweist. Anschließend wird die Methode compile des SourceClass-Objekts aufgerufen.

Darin wird die Nachricht check an this gesendet, was wiederum den Aufruf der entsprechenden check-Methoden der zur Klasse gehörigen SourceField-Objekte zur Folge hat, usw., bis hin zum Aufruf der check-Methoden der in den Anweisungen der Methoden enthaltenen Expression-Objekte.

Anschließend wird für jedes Element der Klasse eine CompilerField-Instanz erzeugt und die Übersetzung der Elemente durch Aufruf der entsprechenden code-Methoden der SourceField-Objekte gestartet. Dabei wird auch bei Bedarf das Inlining der Elemente vorgenommen.

Zuletzt werden die fertigen CompilerField-Objekte optimiert und ein Class File entsprechend dem Class File Format erzeugt.

8.2 Erweiterungen des Lava-Compilers

Ich beschreibe jetzt die Änderungen, die ich am Java-Compiler vorgenommen habe, um Lava-Klassen übersetzen zu können. Ich werde dabei nur auf die wesentlichen Aspekte eingehen. Auf Änderungen, die auf der Hand liegen, wie z.B. das Einfügen der neuen Opcodes und der neuen Schlüsselwörter in die Interfaces RuntimeConstants bzw. Constants, werde ich nicht besonders eingehen.

8.2.1 Änderungen in sun.tools.asm

Befehle

Im Paket sun.tools.asm mußte die Klasse Instruction um die neuen Befehle der LVM (accesssafe, invokeself und invokevirtual) ergänzt werden. Dabei mußten auch die Werte für Befehlsgröße und Stackzuwachs angepaßt werden.

8.2.2 Änderungen in sun.tools.tree

Explizite Delegation und Methodenaufrufe

Eine Möglichkeit mußte geschaffen werden, eine explizite Delegation in einem Objektbaum zu repräsentieren. Der Einfachheit halber habe ich die Klasse MethodExpression um ein Flag ergänzt, mit dem zwischen einem normalen Methodenaufruf und einer expliziten Delegation unterschieden werden kann, da diese beiden Sprachkonstrukte ähnlich übersetzt werden.

In den check-Methoden von MethodExpression mußte zusätzlich sichergestellt werden, daß explizite Delegation z.B. nur an Felder erfolgt, die mit der Annotation delegatee versehen sind.

Bei der Übersetzung eines MethodExpression-Objekts in den code-Methoden mußten Fallunterscheidungen eingebaut werden, um zwischen Methodenaufruf und expliziter Delegation innerhalb von delegiertem und nicht-delegiertem Code, und dort jeweils zwischen Nachrichten an this (Instanz der Klasse ThisExpression) und Nachrichten an sonstige Ausdrücke zu unterscheiden. Ob ein MethodExpression-Objekt für delegierten oder nicht-delegierten Code übersetzt wird, wird anhand der neuen Methode withinDelegatedCode() vom Typ boolean der Klasse Environment festgestellt.

Es wird in eine Folge von Instruction-Instanzen übersetzt. Das Kompilat entspricht den Codefragmenten aus Kapitel 7.5, die vorher von Hand in die Maschinensprache der LVM übersetzt wurden. Die Verwaltung des Delegationsstacks wurde dabei in eine Java-Klasse des Pakets java.lang verlagert. Die Stackelemente werden in einem Array ausreichender Länger gespeichert, die Indizes sind Werte vom Typ int. Die Spezifikation des Delegationsstacks (s. Kapitel 7.5.4) konnte mehr oder weniger direkt als Java-Klasse implementiert werden, die vom erzeugten Code verwendet wird.

Zuweisungen an mandatory Felder

Die Klasse AssignExpression mußte geändert werden, damit bei der Übersetzung von Zuweisungen an mandatory Felder die erforderlichen Überprüfungen zur Laufzeit eines Lava-Programms durchgeführt werden.

In den check-Methoden wird zunächst festgestellt, ob zusätzlicher Code erzeugt werden muß; das ist der Fall, wenn die linke Seite einer Zuweisung ein mandatory Feld ist und die rechte Seite nicht this, ein konstanter String oder ein Ausdruck zur Erzeugung einer Instanz[5] ist. Ist die rechte Seite einer solchen Zuweisung konstant null, so wird hier eine Fehlermeldung ausgegeben.

In den code-Methoden wird schließlich abhängig vom Ergebnis der check-Methoden eine Folge von Instruction-Instanzen erzeugt, die dem Codefragment aus Kapitel 7.7 entspricht.

[5] Das sind Instanzen von ThisExpression, StringExpression bzw. NewInstanceExpression.

8.2.3 Änderungen in sun.tools.java und sun.tools.javac

Scanner und Parser

Die Erweiterungen der Klassen Scanner, Parser und BatchParser konnten mehr oder weniger direkt aufgrund der Grammatik von Lava (s. Anhang A) vorgenommen werden.

Umbenennungen

Um Umbenennungen von Elemten via renames bei der Übersetzung einer Lava-Klasse zu berücksichtigen, wurde die Klasse FieldDefinition um ein Feld ergänzt, das den ursprünglichen Namen eines Elements enthält. Ist dieses Feld mit dem Wert null belegt, so liegt keine Umbenennung des entsprechenden Elements vor.

In den check-Methoden der Klasse SourceField wird überprüft, ob eine Umbenennung zulässig ist. Z.B. wird überprüft, ob das im Quelltext nach renames angegebene Element mit der entsprechenden Signatur existiert.

Virtuelle Felder

Felder, die aus Elterntypen geerbt werden, dürfen nicht im Class File einer Klasse erscheinen, es sei denn sie werden umbenannt. Um zwischen solchen „virtuellen“ Feldern und anderen Elementen unterscheiden zu können, wurde in der Klasse FieldDefinition ein Flag ergänzt, das mit der Methode isVirtual() vom Typ boolean abgefragt werden kann.

Customized Lookup Code

Bei der Erzeugung einer SourceField-Instanz kann angegeben werden, ob es sich um eine Methode handelt, für die Customized Lookup Code erzeugt werden muß. Dieser Code wird dann innerhalb der check-Methoden von SourceField *vor* den eigentlichen Checks der Methode erzeugt. Damit ist sichergestellt, daß auch der Customized Lookup Code den Checks und den darin enthaltenen Inlinings unterworfen wird.

Der Customized Lookup Code wird als Baum von Instanzen verschiedener Unterklassen von Statement und Expression angelegt, die dem entsprechen, was der Parser für einen in Lava explizit programmierten Customized Lookup Code entsprechend dem Codefragment aus Kapitel 7.5.2 erzeugen würde. Die Übersetzung in eine Folge von Instruction-Instanzen kann dadurch den entsprechenden code-Methoden überlassen werden.

Änderungen in SourceClass

An der Klasse SourceClass sind die umfangreichsten Änderungen vorgenommen worden:

- In der check-Methode werden die Elemente aus den Elterntypen einer Klasse eingefügt. Zu diesem Zweck werden alle Felder der Klasse überprüft, ob sie mit der Annotation delegatee oder consultee versehen sind. Dabei wird gleichzeitig überprüft, ob die Felddeklarationen zulässig sind, z.B. daß keine Klassenvariablen mit der Annotation delegatee versehen sind, und ggfs. Fehlermeldungen ausgegeben.

  Wird ein delegatee oder consultee Feld gefunden, so werden alle Instanzelemente des Feldtyps in die aktuelle Klasse eingefügt, die public, für delegatee Felder auch protected, oder ggfs. paketweit sichtbar sind. Dabei findet gleichzeitig eine Überprüfung auf Namenskonflikte mit evtl. Fehlermeldungen statt. Ist ein Instanzelement eines Elternverweises in der Umbenennung eines bereits definierten Elements erwähnt, so muß es nicht in die aktuelle Klasse eingefügt werden. Bei der Einfügung der Elemente wird festgelegt, ob es sich jeweils um virtuelle Felder oder um Methoden handelt, für die Customized Lookup Code erzeugt werden muß.

- Anschließend wird in der check-Methode Code für die Überprüfung aller mandatory Klassenvariablen auf null erzeugt, ähnlich wie für mandatory Instanzvariablen. Dieser Code wird als statischer Initialisierungscode für die aktuelle Klasse angelegt.

- Im Java-Compiler wird ebenfalls in der check-Methode überprüft, ob es sich bei einer Methode um eine Redefintion einer Methode mit gleicher Signatur aus einer Oberklasse handelt, und ob diese Redefinition zulässig ist. Z.B. darf die Methode der Oberklasse nicht final sein.

  Diese Überprüfung mußte für Lava um folgende Punkte ergänzt werden:

  - Umbenennungen müssen berücksichtigt werden, da die Methode der Oberklasse ggfs. einen anderen Namen hat.
  - Felder müssen in die Überprüfung miteinbezogen werden, da es sich ebenfalls um Umbennenungen von Feldern aus einer Oberklasse handeln kann. Umbenanntes und umbenennendes Feld müssen z.B. exakt den gleichen Typ haben.
  - Die Überprüfungen müssen auch für Elterntypen durchgeführt werden. Ist ein Element in der aktuellen Klasse eine Umbenennung eines Elements eines Elternverweises, so muß diese Überprüfung für den Typ des Elternverweises durchgeführt werden. Ist keine Umbenennung angegeben, so muß die Überprüfung für alle Elterntypen durchgeführt werden, die in der aktuellen Klasse vorkommen.

- In der compile-Methode muß zum Einen verhindert werden, daß virtuelle Felder übersetzt werden. Zum Anderen müssen im Fall einer delegatee Klasse für jede Instanzmethode zwei Versionen – die delegierte und die nicht-delegierte – übersetzt werden.

  Zu diesem Zweck wurde die Klasse CompilerField um eine Instanzvariable ergänzt, in der die delegierte Version einer Methode gespeichert wird. Diese wird den code-Methoden der SourceField-Objekte übergeben, wo die Erzeugung des Codes stattfindet. Vor dem Aufruf der code-Methode wird in der Umgebung jeweils ein Flag umgeschaltet, wodurch festgelegt wird, ob delegierter oder nicht-delegierter Code erzeugt werden soll. Dieses Flag

wird mit Hilfe der Methode withinDelegated() der Klasse Environment z.B. in den code-Methoden der Klasse MethodExpression abgefragt. (s. S. 96)

- Zuletzt wird in der compile-Methode das Class File für die aktuelle Klasse erzeugt. Die Änderungen an dieser Stelle konnten mehr oder weniger direkt entsprechend der Spezifikation der LVM (s. Anhang B) vorgenommen werden.

Einlesen von Lava Class Files

Um Class Files entsprechend der LVM einlesen zu können, mußte die Klasse BinaryClass geändert werden.

Die Access Flags von Klassen und Feldern wurden dabei derart um die ExtAccessFlags der LVM erweitert, so daß in den restlichen Quelltexten des Lava-Compilers nicht mehr zwischen diesen beiden Repräsentationen der Access Flags unterschieden werden mußte. Erst beim Schreiben eines Class Files in der compile-Methode von SourceClass werden die Access Flags und die ExtAccessFlags voneinander getrennt.

Da das Konzept der virtuellen Felder, die nur intern im Lava-Compiler für eine Klasse erzeugt werden, aber nicht in das Class File geschrieben werden, im Java-Compiler nicht existierte, mußte BinaryClass derart geändert werden, daß auch hier ein Aufruf der check-Methode erfolgt, und darin virtuelle Felder angelegt werden.

8.3 Zusammenfassung

Ich gebe im folgenden einen Überblick über alle Sprachkonstrukte von Lava und die durchgeführten Änderungen am Java-Compiler, der der Einteilung des Kapitels 7 entspricht:[6]

Feldannotationen Lesen der ExtAccessFlags in BinaryClass, Schreiben in der Methode compile der Klasse SourceClass.

Übersetzung von Methoden Übersetzung der delegierten Version einer Methode: Ergänzung der Klasse CompilerField, Setzen eines Flags in der Klasse Environment, Abfrage des Flags in den code-Methoden der Klasse MethodExpression, Wechsel zwischen Übersetzung von delegiertem und nicht-delegiertem Code in der Methode compile der Klasse SourceClass.

Klassenannotationen wie Feldannotationen.

Methoden aus Elterntypen Einfügen von Methoden aus Elterntypen in der Methode check der Klasse SourceClass. Erzeugung von Customized Lookup Code in den check-Methoden der Klasse SourceField.

[6]Zum Zeitpunkt der Fertigstellung meiner Diplomarbeit wurden leider noch nicht alle Aspekte der Sprache Lava durch den Lava-Compiler abgedeckt. Noch nicht implementiert sind die Subtypbeziehung zwischen einer Kindklasse und dem Typ eines mandatory Elternverweises, sowie Selektionen.

Explizite Delegation als Variante der Klasse MethodExpression.

Nachrichten an this Übersetzung verschiedener Aufrufvarianten in den code-Methoden von MethodExpression.

Suche nach Delegatees in der Elternhierarchie Aufruf von Methoden einer Java-Klasse zur Verwaltung des Delegationsstacks.

Umbenennungen und Selektionen Einlesen des Renaming-Attributs in BinaryClass, Speicherung als zusätzliches Feld in der Klasse FieldDefinition, Schreiben in der Methode compile der Klasse SourceClass.

Selektionen sind noch nicht implementiert.[7]

Die Annotation mandatory

Zuweisungen Einfügung einer Überprüfung der rechten Seite einer Zuweisung auf null in den check- und code-Methoden der Klasse AssignExpression.

[7] Zur Zeit wird für konfluente Pfade das Element ausgewählt, das denselben Namen hat wie in der ursprünglichen Definition des Elements. Dies funktioniert jedoch nicht unter allen Umständen, da ggfs. in der Menge der umbenannten Elemente mit ursprünglich gleichem Namen und gleichem Typ keins existiert, das selektiert werden kann, da noch mindestens ein weiteres Element eines anderen Typs existiert, das den ursprünglichen Namen erhalten hat.

Kapitel 9

Evaluation

Eine Bewertung der Programmiersprache Lava und der Implementation des Lava-Compilers ist zum jetzigen Zeitpunkt der Fertigstellung meiner Diplomarbeit nicht leicht. Erfahrungsgemäß zeigen sich die Stärken und Schwächen einer Programmiersprache erst nach häufiger Verwendung insb. in großen Programmierprojekten. Dennoch will ich im folgenden einzelne Aspekte meiner Arbeit diskutieren.

9.1 Sprachdesign

Es hat sich als vorteilhaft erwiesen, Java als Ausgangspunkt für Spracherweiterungen zu wählen. Die Syntax und Semantik ist sauber und exakt spezifiziert und leidet nicht unter Sprachkonstrukten, die erst im Laufe der Jahre hinzugekommen sind, wie es z.B. bei C und C++ der Fall ist.

Ein Beleg hierfür ist die Tatsache, daß die Spezifikation von Lava (s. Anhang A), die nur aus Änderungen und Ergänzungen der Spezifikation von Java besteht, relativ knapp ist.

Wie sich die neuen Sprachkonstrukte in der Praxis bewähren, muß sich im Laufe der Zeit zeigen.

- Das Konzept der delegatee und consultee Felder (s. Kapitel 5.2) ist sehr einfach handhabbar und bietet wesentliche Vorteile, wie das Beispiel der Implementation des Strategy Patterns in Lava aus Kapitel 6.2 belegt.
- Das Konzept der mandatory Felder (s. Kapitel 5.3.1) trägt wesentlich zur Mächtigkeit objektbasierter Vererbung bei, weil hierdurch erst eine sichere Subtypbeziehung zwischen einer Kindklasse und einem Elterntyp ermöglicht wird. Darüberhinaus könnten sich mandatory Felder auch als nützlich in Kontexten erweisen, die in keinem Zusammenhang zu objektbasierter Vererbung stehen.
- Die Auflösung von Namenskonflikten mittels Umbenennungen und Selektionen (s. Kapitel 5.5) führt zu einer diesbezüglich komplexen Semantik

von Lava. Dies könnte die Akzeptanz von Lava bei Programmierern beeinträchtigen.

Namenskonflikte sind allerdings vom Standpunkt des Sprachdesigns prinzipiell nicht vermeidbar, da durch die Einführung objektbasierter Vererbung zumindest Namenskonflikte zwischen Elementen der Oberklasse und eines Elterntyps auftreten können. Darüberhinaus können sich wegen der Möglichkeit in Lava, von mehrereren Elternverweisen zu erben, auch Namenskonflikte zwischen Elementen aus Elterntypen ergeben. Es ist allerdings eine offene Frage, wie häufig oder selten Namenskonflikte tatsächlich auftreten und daher Umbenennungen und Selektionen erforderlich sind.

Es ist in Lava nicht möglich, durch eine Methode „zufällig" mehrere Methoden aus verschiedenen Elternverweisen und der Oberklasse gleichzeitig zu redefinieren, da immer explizit angegeben werden muß, welche Methode aus welchem Typ jeweils gemeint ist.[1] Dadurch werden Programmierfehler vermieden, die sich daraus ergeben können, daß bestimmte Methoden vom Programmierer schlicht übersehen werden. Dies muß als ein Vorteil gewertet werden, da hiermit entscheidend zur Sicherheit eines Programms beigetragen wird.

- Aus Gründen der Sicherheit müssen Klassen explizit für Delegation freigegeben werden, um die Sichtbarkeitseinschränkungen von Elementen aus bestehenden Java-Klassen nicht zu verletzen (s. Kapitel 5.6). Delegation an bestehende Java-Klassen, z.B. die Standardklassen von Java, ist daher nicht möglich.

  Diese Einschränkung könnte sich als hinderlich erweisen. Daher ist zu überlegen, ob sie ggfs. aufgehoben und die Verletzung bestehender Sichtbarkeitseinschränkungen in Kauf genommen, oder aber ein alternativer Ansatz für dynamische Sichtbarkeitseinschränkungen realisiert wird, wie er z.B. in [Kniesel 96b] beschrieben ist.

  Dies hätte jedoch weitreichende Folgen für das Design und eine Implementation der Lava Virtual Machine, da u.a. nicht davon ausgegangen werden könnte, daß für Methoden delegierter Code vorliegt.

9.2 Lava Virtual Machine

Das Design der Lava Virtual Machine, wie es in Kapitel 7 vorgestellt wurde, ergibt sich in weiten Teilen direkt aus dem Design der Sprache Lava und aus dem Design der Java Virtual Machine. Z.B. liegt das Speichern der neuen Annotationen in Lava als Access Flags für die entsprechenden Strukturen des Class File Formats auf der Hand.

Zu anderen Aspekten der Lava Virtual Machine sind alternative Ansätze denkbar. Beispielsweise könnten die Customized Lookup Codes für Methoden aus Elterntypen von einer Implementation der LVM erzeugt werden, anstatt zu verlangen, daß ein Compiler sie automatisch bereitstellt (s. Kapitel 7.5). Auch

[1] Wenn dies dennoch erwünscht ist, kann das in Kapitel 5.5.5 beschriebene Verfahren angewendet werden.

die Verwaltung des Delegationsstacks (s. Kapitel 7.5.4) oder die besondere Behandlung von mandatory Feldern bei Zuweisungen (s. Kapitel 7.7) könnte von entsprechenden Befehlen der LVM übernommen werden.

Diese Alternativen hätten die Vorteile, daß Lava Class Files kompakter wären und das Laden und die Ausführung eines Lava-Programms beschleunigt würde. Andererseits müßte eine Implementation der Lava Virtual Machine mehr Aufwand insb. bei der Initialisierung einer Lava-Klasse betreiben, um z.B. die Customized Lookup Codes zu erzeugen, was die Geschwindigkeitsvorteile beim Laden eines Lava-Programms zumindest zum Teil wieder aufheben würde.

In [Schickel 97] werden verschiedene Alternativen vorgestellt. Das Laufzeitverhalten der darin vorgestellten Implementation der Lava Virtual Machine, die der Spezifikation in Anhang B entspricht, wird ebenfalls in [Schickel 97] bewertet.

9.3 Lava-Compiler

Die Entscheidung, einen Compiler für Lava nicht von Grund auf neu zu programmieren, sondern auf dem Java-Compiler von Sun Microsystems aufzubauen, wurde in der Hoffnung gefällt, zügig ein lauffähiges Programm zu erhalten.

Behindert wurde dies jedoch durch die Tatsache, daß der Java-Compiler nur sehr dürftig dokumentiert ist. Zu den meisten Klassen und den darin enthaltenen Methoden sind so gut wie gar keine Beschreibungen vorhanden. Auch mit Kommentaren in den Quelltexten des Compilers wurde sehr sparsam umgegangen. Daher stand vor der eigentlichen Implementation der Sprachkonstrukte von Lava die Aufgabe an, die Quelltexte des Java-Compilers eingehend zu analysieren.

Es ergab sich, daß das Design des Java-Compilers offensichtlich nicht sehr sauber ist. Beispielsweise erfolgt die Deklaration der Klasse Parser als Unterklasse der Klasse Scanner nur aus dem Grund, Code wiederverwenden zu können, aber nicht, weil eine sinnvolle Subtypbeziehung zwischen diesen beiden Klassen definiert werden soll.

Dies alles – dürftige Dokumentation und unsauberes Design – resultiert vermutlich aus der Tatsache, daß die Version 1.0 des Java Development Kits von Sun Microsystems aus Marketinggründen unter Zeitdruck veröffentlicht wurde.

Dennoch konnten weite Teile der Spezifikation der Sprache Lava relativ schnell realisiert werden, da nicht gleichzeitig auch die Spezifikation von Java implementiert werden mußte.

Wie verschiedene Tests, z.B. für die Evaluation der Implementation der Lava Virtual Machine aus [Schickel 97], zeigen, erzeugt der Lava-Compiler korrekten Code. Die Integration des Lava-Compilers mit der Implementation der Lava Virtual Machine, die in [Schickel 97] beschrieben ist, in ein *Lava Development Kit* konnte ebenfalls erfolgreich durchgeführt werden.

Trotz der Nachteile zeigt sich also, daß die Entscheidung für eine Erweiterung des bestehenden Java-Compilers richtig war. Bei der Analyse des Java-Compilers und der Implementierung von Lava wurden darüberhinaus wertvolle Erfahrungen gesammelt.

9.4 Ausblick

In der Zwischenzeit wurde von der mittlerweile von Sun Microsystems gegründeten Tochterfirma JavaSoft die Versionen 1.1.x des Java Development Kits veröffentlicht. Die Sprache Java wurde darin um neue Sprachkonstrukte erweitert. Außerdem ist die Version 1.2 des JDK angekündigt, eine erste Beta-Version liegt zum Zeitpunkt der Fertigstellung meiner Diplomarbeit bereits vor. Auch darin wurden Erweiterungen der Sprache Java vorgenommen.

Es muß daher überprüft werden, ob die Änderungen Auswirkungen auf das Design der Sprache Lava und der Lava Virtual Machine haben und inwieweit Anpassungen vorgenommen werden müssen. Ob die bestehenden Quelltexte des Lava-Compilers weiterverwendet werden können, ist fraglich; zumindest können sie aber als Basis für die prototypische Implementation weiterer Konzepte verwendet werden.

Die Entwicklung eines vollständig neuen Compilers kann in Erwägung gezogen werden, wobei die Erfahrungen mit dem bestehenden Lava-Compiler sicherlich einfließen werden. Verschiedene, mittlerweile erhältliche Werkzeuge für Java könnten diese Aufgabe darüberhinaus vereinfachen, wie z.B. Parsergeneratoren[2].

Am Institut für Informatik III der Rheinischen Friedrich-Wilhelms-Universität Bonn sind verschiedene Diplomarbeiten in Vorbereitung, in denen die Praxistauglichkeit der neuen Sprachkonstrukte von Lava getestet werden. Eine differenziertere Bewertung der Programmiersprache Lava wird aufgrund dieser Diplomarbeiten möglich sein. Wahrscheinlich werden sie Einfluß auf weitere Versionen von Lava haben.

[2]s. z.B. http://www.suntest.com

Kapitel 10

Resümee

Mit Hilfe der in dieser Diplomarbeit vorgestellten Programmiersprache Lava wird gezeigt, daß die streng typisierte, auf klassenbasierter Vererbung beruhende Programmiersprache Java um Sprachkonstrukte erweitert werden kann, die objektbasierte Vererbung inklusive zusätzlicher Subtypbeziehung unterstützen. Die Implementation des Lava-Compilers zusammen mit der Implementation der Lava Virtual Machine, die in [Schickel 97] beschrieben wird, bilden eine funktionierende Plattform, für die Programme in der Sprache Lava entwickelt werden können.

Folgende Aufgaben mußten bei der Implementation von Lava gelöst werden:

Sprachdesign Ein Sprachkonstrukt mußte eingeführt werden, um Elternverweise definieren zu können. Daraus ergaben sich Probleme, für deren Lösung weitere Sprachkonstrukte eingeführt wurden.

Aufruf von Methoden aus Elternobjekten Durch die Definition eines Elternverweises müssen von Instanzen einer Klasse Nachrichten verstanden werde, für die die Klasse keine Methode bereitstellt, wohl aber das durch den Elternverweis referenzierten Objekt. Aus diesem Grund mußte ein Mechanismus eingeführt werden, der sicherstellt, daß tatsächlich die Methoden aus den Elternobjekten bei entsprechenden Nachrichten ausgeführt werden.

Typsichere Delegation Um die Typsicherheit delegierter Nachrichten zu gewährleisten, und damit die Subtypbeziehung zwischen Kindklasse und Elterntyp zu ermöglichen, mußte die Möglichkeit geschaffen werden, daß Elternverweise nie den Wert `null` erhalten.

Unterscheidung von `this`-Objekt und Träger einer Methode In Methoden muß zwischen dem `this`-Objekt und dem Träger einer Methode unterschieden werden, da es sich um unterschiedliche Objekte handeln kann. Daher mußten Vorkehrungen getroffen werden, um die Übergabe dieser zwei Objekte an eine delegierte Methode bei Berücksichtigung der Kompatibilität mit Java zu ermöglichen.

Nachrichten an this Aufgrund der Möglichkeit, daß das this-Objekt und der Träger einer Methode unterschiedliche Objekt sind, werfen Nachrichten an this Probleme auf, da das this-Objekt die Nachricht möglicherweise nicht kennt. Aus diesem Grund mußten in Lava Vorkehrungen getroffen werden, um die Typsicherheit einer Nachrichtenversendung an this zu gewährleisten.

Namenskonflikte Es mußte die Möglichkeit geschaffen werden, Namenskonflikte in Lava aufzulösen.

Aufgrund der Architektur von Java und der Java Virtual Machine konnte die Implementation von Lava in zwei unabhängigen Bereichen, der Übersetzungszeit und der Laufzeit, durchgeführt werden.

Der Bereich der Übersetzungszeit, der in dieser Diplomarbeit beschrieben wurde, umfaßt die Aspekte Sprachdesign, Design der Lava Virtual Machine und Implementation eines Lava-Compilers.

Sprachdesign

Die Sprache Java wurde um folgende Konstrukte erweitert (s. Kapitel 5):

- Deklaration von Elternverweisen in Klassendefinitionen
- Explizite Delegation an Elternverweise
- Deklaration von Feldern, die nie null sind
- Subtypbeziehung zwischen Kindklasse und Elterntyp durch Elternverweise, die nie null sind, wodurch Typsicherheit bei Delegation von Methoden gewährleistet ist
- Auflösung von Namenskonflikten, die durch objektbasierte Vererbung entstehen durch explizite Umbenennung und Selektion von geerbten Namen
- Aus Kompatibilitäts- und Sicherheitsgründen nur Delegation an Objekte von Klassen, die dies explizit erlauben

Design der Lava Virtual Machine

Die Java Virtual Machine mußte erweitert werden, um die Ausführung von Lava-Programmen zu ermöglichen (s. Kapitel 7).

Dazu wurde das Class File Format um folgende Elemente ergänzt:

- Access Flags für Felder, die sie als Elternverweise deklarieren
- Access Flags für Felder, die deklarieren, daß sie nie null sind
- Ein Attribut und ein Access Flag für Felder und Methoden, mit denen Umbenennungen und Selektionen festgelegt werden

- Ein Attribut für Methoden, das delegierten Code enthält
- Ein Access Flag für Klassen, um Delegation an Instanzen zu erlauben

Der Befehlssatz der Java Virtual Machine wurde wie folgt erweitert: werden:

- Die Semantik bestehender Befehle wurde erweitert, um Zugriffe auf Felder aus Elternverweisen zu ermöglichen.
- Ein neuer Befehl für Safety Checks wurde eingeführt.
- Zwei neue Befehle für Delegation wurden eingeführt.

Lava-Compiler

Der Java-Compiler wurde erweitert; dabei waren folgende Punkte bedeutend (s. Kapitel 8):

- Die neue Syntax/Semantik muß berücksichtigt werden; die entsprechenden neuen Access Flags und Attribute des Class File Formats müssen erzeugt werden.
- Für Methoden aus Elternverweisen, die nicht redefiniert werden, muß Customized Lookup Code erzeugt werden.
- Für jede Methode müssen zwei Kompilate, die delegierte und die nicht-delegierte Version einer Methode erzeugt werden.
- Es muß sichergestellt werden, daß immer die richtige Version einer Methode ausgeführt wird.
- Insb. für Nachrichten an this muß die Verwaltung eines Delegationsstacks berücksichtigt werden.
- Für Zuweisungen an Felder, die nicht null sind, muß zusätzlicher Code erzeugt werden, der dies sicherstellt.

Evaluation

Die Implementation von Lava konnte erfolgreich durchgeführt werden.

In Kapitel 6 wurde gezeigt, daß das Strategy Pattern aus [Gamma 95] in Lava wesentlich einfacher als in einer klassenbasierten Programmiersprache ohne objektbasierte Vererbung implementiert werden kann.

In Kapitel 9 wurden die Vor- und Nachteile der verschiedenen Aspekte der Übersetzungszeit von Lava diskutiert. Eine endgültige Bewertung ist zum jetzigen Zeitpunkt nicht möglich, da noch nicht genügend Erfahrungen mit Lava gesammelt werden konnten.

Grundsätzlich ist Lava aber durch die Einführung objektbasierter Vererbung eine wesentlich mächtigere Sprache als z.B. Java. Weitere Arbeiten mit Lava werden zeigen, wie erfolgreich die neuen Sprachkonstrukte tatsächlich eingesetzt werden können.

Anhang A

Spezifikation der Sprache Lava

A.1 Einleitung

Ich beziehe mich im folgenden auf „The Java Language Specification“, die Spezifikation der Sprache Java [Gosling 96].

Das Ziel der Ergänzungen ist, Programmierern Sprachkonstrukte an die Hand zu geben, mit deren Hilfe zwei Varianten objektbasierter Vererbung, namentlich Delegation und Konsultation, sauber und passend zu den anderen Sprachkonstrukten von Java eingesetzt werden können. Das Design und die Semantik der Sprache Lava ist ausführlich in Kapitel 5 beschrieben.

Ich gebe im folgenden eine Spezifikation der Sprache Lava wieder. Dabei werden alle Bestandteile der Syntax von Lava aufgelistet, die sich gegenüber der Syntax von Java geändert haben oder die neu hinzugekommen sind. Die Semantik von Lava wird hier nur angerissen, genauere Ausführungen finden sich in Kapitel 5.

A.2 Notation

Die Notation der Grammatik entspricht der in [Gosling 96], Kapitel 2. Bestandteile der Grammatik, die sich gegenüber der Grammatik von Java geändert haben, sind unterstrichen. Die Grammatik ist keine LALR(1) Grammatik, sondern liegt in einer Form vor, die das Verständnis der Syntax erleichtern soll.

Zu Beginn jedes folgenden Abschnitts ist in *kursiver Schrift* angegeben, auf welchen Abschnitt von [Gosling 96] Bezug genommen wird.

A.3 Ergänzungen

A.3.1 Schlüsselwörter

Abschnitt „3.9 Keywords“

Die Liste der Schlüsselwörter wird ergänzt um: consultee, delegatee, mandatory, optional, renames, selected.

A.3.2 Trennzeichen

Abschnitt „3.11 Separators“

Die Liste der Trennzeichen wird ergänzt um: :: und <-.

A.3.3 Konversionen

In diesem Abschnitt sind nur Kindklassen gemeint, die über einen mandatory Elternverweis definiert werden. Für optional Elternverweise gelten die folgenden Ausführungen nicht.

Abschnitte „5.1.4 Widening Reference Conversions“ und „5.1.5 Narrowing Reference Conversions“

Folgende zusätzlichen Konversionen sind erlaubt:

- Von einer beliebigen Klasse S zu einer beliebigen Klasse T, wenn S eine Kindklasse von T ist. Es handelt sich um eine „Widening Reference Conversion“, für die kein besonderer Test zur Laufzeit eines Programms erfolgen muß, und die folglich keine Ausnahme auslösen kann.
- Von einer beliebigen Klasse S zu einer beliebigen Klasse T, wenn S eine Elternklasse von T ist. Es handelt sich um eine „Narrowing Reference Conversion“, für die zur Laufzeit ein Test ausgeführt muß, ob tatsächlich ein Wert des verlangten Typs vorliegt. Ggfs. wird eine ClassCastException ausgelöst.

Abschnitt „5.2 Assignment Conversion“

Bei Zuweisungen eines Werts eines Referenztyps S an eine Variable eines Referenztyps T ist folgende zusätliche Bedingung für die Übersetzungszeit definiert:

- Falls S ein Klassentyp ist
  - Falls T ein Klassentyp ist und S eine Kindklasse von T ist, ist die Zuweisung erlaubt.

Abschnitt „5.5 Casting Conversion“

Bei einem Typecast eines Werts eines Referenztyps S zu einem Referenztyp T ist folgende zusätliche Bedingung für die Übersetzungszeit definiert:

- Falls S ein Klassentyp ist
  - Falls T ein Klassentyp ist und S eine Kindklasse oder Elternklasse von T ist, ist der Typecast erlaubt.

Bei einem Typecast eines Werts eines Referenztyps R zu einem Referenztyp T ist folgende zusätliche Bedingung für die Laufzeit definiert:

- Falls R ein Klassentyp ist
  - Falls T ein Klassentyp ist und R eine Kindklasse von T ist, ist der Typecast erlaubt.

A.3.4 Zugriffskontrolle

Abschnitt „6.6.1 Determining Accessibility“

- Zugriff auf ein protected Element ist zusätzlich in folgendem Fall erlaubt:
  - Der Zugriff erfolgt in einer delegierenden Kindklasse der Klasse, in der das protected Element deklariert ist.

A.3.5 Standardklassen

Abschnitt „7.5.3 Automatic Imports“

Folgende zusätliche Klassen sind im Paket java.lang enthalten: AssignmentOfNullException, ParentIsNullException.

A.3.6 Klassenannotationen

Abschnitt „8.1.2 Class Modifiers“

Das Nicht-Terminal *ClassModifier* wird erweitert:

ClassModifier: one of
public abstract final delegatee

Die Annotation delegatee deklariert eine Klasse als eine mögliche Elternklasse, an die delegiert werden darf. Fehlt diese Annotation, so darf keine Delegation an Instanzen dieser Klasse erfolgen. Unterklassen von delegatee Klassen müssen ebenfalls mit der Klassenannotation delegatee versehen sein.

Von final Klassen dürfen keine Kindklassen abgeleitet werden.

A.3.7 Felddeklarationen

Abschnitt „8.3 Field Declarations“

Das Nicht-Terminal *FieldDeclaration* wird geändert, um zwischen den zulässigen Typen für delegatee, consultee und herkömmlichen Feldern zu unterscheiden. Zu diesem Zweck werden auch die neuen Nicht-Terminale *NonDelegateeType*, *DelegateeType* und *ConsulteeType* eingeführt. Darüberhinaus wird zusätzlich die Selektion eines Felds für konfluente Pfade bei Namenskonflikten ermöglicht.

FieldDeclaration:
  FieldModifiers$_{opt}$ NonDelegateeType VariableDeclarators ;
  DelegateeFieldModifiers$_{opt}$ DelegateeType VariableDeclarators ;
  ConsulteeFieldModifiers$_{opt}$ ConsulteeType VariableDeclarators ;

NonDelegateeType:
  PrimitiveType
  ArrayType

DelegateeType:
  ClassType

ConsulteeType:
  ClassType
  InterfaceType

Das Nicht-Terminal VariableDeclarator wird um eine optionale *Renaming*-Angabe erweitert, um Felder aus einer Ober- oder aus Elternklassen umbenennen zu können. Der Typ der umbenennenden Felddeklaration muß dabei exakt mit dem Typ des umbenannten Felds übereinstimmen:

VariableDeclarator:
  VariableDeclaratorId Renaming$_{opt}$
  VariableDeclaratorId = VariableInitializer

Feldannotationen

Abschnitt „8.3.1 Field Modifiers“

Zu den bereits existierenden Feld Annotationen werden die neuen hinzugefügt:

FieldModifier: one of
  public protected private
  final static transient volatile
  mandatory optional

DelegateeFieldModifiers:
  DelegateeFieldModifier
  DelegateeFieldModifiers DelegateeFieldModifier

DelegateeFieldModifier: one of

public protected private
final transient volatile
mandatory optional
delegatee

ConsulteeFieldModifiers:

ConsulteeFieldModifier
ConsulteeFieldModifiers ConsulteeFieldModifier

ConsulteeFieldModifier: one of

public protected private
final transient volatile
mandatory optional
consultee

A.3.8 Methodendeklarationen

Abschnitt „8.4 Method Declarations“

Das Nicht-Terminal MethodHeader wird um eine optionale *Renaming*-Angabe erweitert, um Methoden aus einer Ober- oder aus Elternklassen umbenennen zu können. Der Rückgabetyp, sowie die Anzahl und Typen der formalen Parameter der umbenennenden Methodendeklaration müssen dabei exakt mit den entsprechenden Angaben des umbenannten Felds übereinstimmen:

MethodHeader:

$MethodModifiers_{opt}$ *ResultType MethodDeclarator* $\underline{Renaming_{opt}}$
$Throws_{opt} MethodModifiers_{opt}$

Wird der Rumpf einer umbenennenden Methodendeklaration leer gelassen (besteht er also nur aus einem Semikolon), so wird die umbenannte Methode an das in der Umbenennung angegebene Elternobjekt delegiert bzw. die entsprechende Methode der Oberklasse aufgerufen; andernfalls wird die umbenannte Methode redefiniert.

Methodenannotationen

Wird durch objektbasierte Vererbung eine abstract Methode aus einem Elterntyp geerbt, so ist sie in der erbenden Klasse nicht ebenfalls abstract, da dem Elternverweis eine Instanz einer Unterklasse des Elterntyps zugewiesen werden kann, in der die abstract Methode definiert ist.

A.3.9 Umbenennungen

Es wird das neue Nicht-Terminal *Renaming* eingeführt, daß sowohl bei Feld-, als auch bei Methodendeklarationen verwendet wird, um Felder bzw. Methoden aus der Oberklasse und aus Elternverweisen umbenennen zu können:

Renaming:
  renames *Identifier* :: *Identifier* selected$_{opt}$
  renames super :: *Identifier* selected$_{opt}$

Hierbei steht der erste Bezeichner für einen Elternverweis bzw. für die Oberklasse (im Fall des Schlüsselworts super). Der zweite Bezeichner steht für das umzubenennende Element des Elternobjekts. Das Schlüsselwort selected wählt die Deklaration für konfluente Pfade aus, wenn nicht eindeutig entschieden werden kann, auf welches Element zugegriffen werden soll.

A.3.10 Methodenaufrufe

Abschnitt „15.11 Method Invocation Expression“

Das Nicht-Terminal *MethodInvocation* wird erweitert:

MethodInvocation:
  MethodName (*ArgumentList*$_{opt}$)
  Primary . *Identifier* (*ArgumentList*$_{opt}$)
  Primary <- *Identifier* (*ArgumentList*$_{opt}$)
  super . *Identifier* (*ArgumentListopt*)

Ein Methodenaufruf via <- ist eine explizite Delegation. Der Ausdruck auf der linken Seite einer expliziten Delegation muß ein Feld des Trägers der aktuellen Methode ergeben, dessen Deklaration mit der Annotation delegatee versehen ist. Explizite Delegation ist nur im Rumpf einer Instanzmethode erlaubt.

Anhang B

Spezifikation der Lava Virtual Machine

B.1 Einleitung

Ich beziehe mich im folgenden auf „The Java Virtual Machine Specification“, die Spezifikation der Java Virtual Machine [Lindholm 96]. Die Notationen sind ebenfalls daraus entnommen.

Zu Beginn jedes folgenden Abschnitts ist in kursiver Schrift angegeben, auf welchen Abschnitt von [Lindholm 96] Bezug genommen wird.

B.2 Ergänzungen zum Class File Format

Abschnitt „4 The class *File Format“*

B.2.1 Klassenbezogene Ergänzungen

Abschnitt „4.1 ClassFile*“*

Das Element attributes der ClassFile Struktur kann das Attribut ExtAccessFlags (s. S. 116) enthalten, das die Semantik des Class Files beeinflußt.

B.2.2 Feldbezogene Ergänzungen

Abschnitt „4.5 Fields“

Das Element attributes der field_info Struktur kann die Attribute ExtAccessFlags (s. S. 116) und Renaming (s. S. 117) enthalten, die die Semantik des Class Files beeinflussen.

B.2.3 Methodenbezogene Ergänzungen

Abschnitt „4.6 Methods“

Das Element attributes der method_info Struktur kann die Attribute DelegatedCode (s. S. 118, ExtAccessFlags (s. S. 116) und Renaming (s. S. 117) enthalten, die die Semantik des Class Files beeinflussen.

B.2.4 Neue Attribute

Abschnitt „4.7 Attributes“

Attribut ExtAccessFlags

Das ExtAccessFlags Attribut wird in den Strukturen ClassFile (s. S. 115), field_info (s. S. 115) und method_info (s. S. 116) verwendet, um zusätzliche access_flags verwenden zu können, die zu den neuen Annotationen der Sprache Lava korrespondieren.

Das ExtAccessFlags Attribut hat das Format:

```
ExtAccessFlags_attribute {
   u2 attribute_name_index;  // "ExtAccessFlags"
   u4 attribute_length;      // 2
   u2 ext_access_flags;      // Neue Access Flags
}
```

attribute_name_index constant_pool[attribute_name_index] ist der CONSTANT_Utf8 String "ExtAccessFlags".

attribute_length Die Länge eines ExtAccessFlags_attribute (ohne die Einträge attribute_name_index und attribute_length) ist 2 Bytes.

ext_access_flags Der Wert ext_access_flags besteht aus Bits, die verschiedene Annotationen repräsentieren. Die Bedeutung der einzelnen Bits ist weiter unten erläutert.

Für die ClassFile Struktur ist nur das neue Flag ACC_DELEGATEE definiert. Es hat den Wert 0x0008 und kann nur für Klassen, nicht für Interfaces verwendet werden. Der Typ einer Klasse, bei der dieses Flag gesetzt ist, darf bei delegatee Felddeklarationen verwendet werden. Jede Methode dieser Klasse, die das Code-Attribut enthält, enthält auch das DelegatedCode-Attribut (s. S. 118).

Für die field_info Struktur sind die Flags aus Tabelle B.1 definiert.

Ein Feld kann entweder das Flag ACC_CONSULTEE oder ACC_DELEGATEE oder keines von beiden enthalten. Wenn eines dieser beiden Flags gesetzt ist, sind alle public und protected Elemente des Feldtyps sichtbar, sowie die paketweit sichtbaren Elemente des Feldtyps, wenn der Feldtyp und die Klasse des Class Files im gleichen Paket definiert sind. Ggfs. sind sie unter anderen Namen sichtbar.

Flag Name	Wert	Bedeutung	Verwendet von
ACC_CONSULTEE	0x0001	Ist ein Elternverweis; best. Nachrichten werden an dieses Feld konsultiert.	Instanzvariablen
ACC_MANDATORY	0x0002	Ist mandatory; Feld ist immer != null.	Alle Felder
ACC_OPTIONAL	0x0004	Ist optional; Feld kann == null sein.	Alle Felder
ACC_DELEGATEE	0x0008	Ist ein Elternverweis; best. Nachrichten werden an dieses Feld delegiert.	Instanzvariablen

Tabelle B.1: Access Flags für die field_info Struktur

Bei gesetztem ACC_CONSULTEE werden Methoden des Feldtyps, die im Class File nicht redefiniert sind, an das Objekt konsultiert, auf das das Feld verweist, im Fall von ACC_DELEGATEE werden sie delegiert.

Ist gleichzeitig das Flag ACC_MANDATORY nicht gesetzt, dann wird die Liste der auslösbaren Ausnahmen für alle Methoden, die aus dem Feldtyp geerbt werden, um ParentIsNullException (s. Kapitel 5.3.4) ergänzt. Ist es gesetzt, so ist die aktuelle Klasse eine Kindklasse des Feldtyps.

Wenn ACC_CONSULTEE oder ACC_DELEGATEE gesetzt ist, darf nicht gleichzeitig ACC_STATIC der Standard access_flags gesetzt sein.

Ein Feld kann entweder das Flag ACC_MANDATORY oder ACC_OPTIONAL oder keines von beiden enthalten. Wenn ACC_MANDATORY gesetzt ist, so ist im Code sichergestellt, daß das Feld nie mit dem Wert null belegt ist. Wenn ACC_OPTIONAL gesetzt ist, so kann das Feld mit dem Wert null belegt werden.

Sowohl für die field_info, als auch für die method_info Strukturen ist das neue Flag ACC_SELECTED definiert. Es hat den Wert 0x0010. Wenn ACC_SELECTED gesetzt ist, wird das Element für konfluente Pfade ausgewählt, wenn nicht eindeutig entschieden werden kann, welches Element verwendet werden soll. Es kann nicht gleichzeitig ACC_PRIVATE oder ACC_STATIC der Standard access_flags gesetzt sein. Es muß gleichzeitig eine Renaming-Attribut (s. S. 117) für das Element vorhanden sein.

Attribut Renaming

Das Attribut Renaming kann bei Instanzvariablen und -methoden vorkommen. Damit wird angegeben, welcher Bezeichner eines spezifizierten Elternverweises oder der Oberklasse durch die jeweilige Defintion umbenannt wird.

Das Renaming Attribut hat das Format:

```
Renaming_attribute {
   u2 attribute_name_index;  // "Renaming"
```

```
   u4 attribute_length;       // 4
   u2 parent_index;           // Name des Elternverweises
                              // oder "super"
   u2 name_index;             // Ursprünglicher Name
}
```

attribute_name_index constant_pool[attribute_name_index] ist der CONSTANT_Utf8 String "Renaming".

attribute_length Die Länge eines Renaming_attribute (ohne die Einträge attribute_name_index und attribute_length) ist 4 Bytes.

parent_index constant_pool[parent_index] ist der CONSTANT_Utf8 String mit dem Namen der Referenz auf das Elternobjekt, in dem der umzubenennende Bezeichner enthalten ist, bzw. "super", wenn ein Bezeichner aus der direkten Oberklasse umbenannt wird.

name_index constant_pool[name_index] ist ein CONSTANT_Utf8 String mit dem Namen des Bezeichners im Elterntyp oder in der direkten Oberklasse, der umbenannt wird.

Attribut DelegatedCode

Das DelegatedCode Attribut enthält den Code der delegierte Version einer Methode. Das DelegatedCode Attribut hat das gleiche Format wie das Code Attribut der JVM. (s. [Lindholm 96], S. 110ff)

B.3 Ergänzungen zum JVM Befehlssatz

Abschnitt „6 Java Virtual Machine Instruction Set“

Die Semantik der Befehle *putfield* und *getfield* wird dahingehend erweitert, daß auch auf Felder eines Objekts zugegriffen werden kann, die über einen Elternverweis in der Klasse des Objekts geerbt wurden. Bei Vererbung über einen optional Elternverweis wird ggfs. eine ParentIsNullException (s. Kapitel 5.3.4) ausgelöst.

Folgende neuen Befehle sind definiert:

accesssafe *accesssafe*

Operation Ausführung eines Safety Checks (s. Kapitel 5.4.2)

Format

accesssafe
indexbyte1
indexbyte2

Formen = 205 (0xcd)

Stack . . . , *sender_objectref*, *candidate_objectref* $\Longrightarrow$
. . . , *result*

Beschreibung Der Stack muß zwei Referenzen auf Objekte (sender und candidate) enthalten. Aus *indexbyte1* und *indexbyte2* wird ein weiterer Index in den Constant Pool der aktuellen Klasse erzeugt. An der so bestimmten Stellen im Constant Pool befindet sich ein Klassenname, ein Methodenname und ein Methodendeskriptor für eine Nachricht m.
accesssafe bestimmt, ob die Nachricht m bzgl. *sender* sicher an candidate versendet werden kann. Ist dies der Fall, so enthält *result* anschließend den Wert 1, ansonsten den Wert 0.

invokedelegated — *invokedelegated*

Operation Delegierter Aufruf einer Instanzmethode

Format

invokedelegated
indexbyte1
indexbyte2

Formen *invokedelegated* = 203 (0xcb)

Stack ..., *this_objectref*, *delegatee_objectref*, [*arg1*, [*arg2* ...]], ... $\Longrightarrow$
...

Beschreibung Der Stack muß zwei Referenzen auf Objekte (this und delegatee) und ggfs. Argumente enthalten. Aus *indexbyte1* und *indexbyte2* wird ein Index in den Constant Pool der aktuellen Klasse erzeugt. An der so bestimmten Stelle im Constant Pool befindet sich ein Klassenname, ein Methodenname und ein Methodendeskriptor. Ein Zeiger auf eine Methodentabelle wird über das Objekt delegatee ermittelt. Der Methodenname und Metodendeskriptor wird in der Methodentabelle gesucht. Durch den Compiler ist sichergestellt, daß Methodenname und Methodendeskriptor mit genau einem Eintrag in der Methodentabelle übereinstimmt.
Der Eintrag in der Methodentabelle enthält einen Verweis auf einen Methodenblock. Wenn die Methode mit der Annotation synchronized versehen ist, wird für das Objekt delegatee der entsprechende Monitor gestartet.
Die Objektreferenzen und die Argumente werden vom Stack geholt und werden als Initialwerte für die entsprechenden lokalen Variablen der neuen Methode übernommen. Die Ausführung wird mit der ersten Anweisung der neuen Methode fortgesetzt.

Laufzeitfehler Wenn delegatee == null ist, wird eine ParentIsNullException (s. Kapitel 5.3.4) ausgelöst. Wenn während der Ausführung der Methode der Stack überlauft, wird ein StackOverflowError ausgelöst.

invokeself ***invokeself***

Operation Aufruf einer Methode eines delegierenden Objekts

Format

invokeself
indexbyte1
indexbyte2

Formen *invokeself* = 204 (0xcc)

Stack ..., *this_objectref*, *delegatee_objectref*, [*arg1*, [*arg2* ...]], ... $\Longrightarrow$
...

Beschreibung Der Stack muß zwei Referenzen auf Objekte (this und delegatee) und ggfs. Argumente enthalten. Aus *indexbyte1* und *indexbyte2* wird ein Index in den Constant Pool der aktuellen Klasse erzeugt. An der so bestimmten Stelle im Constant Pool befindet sich ein Methodenname und ein Methodendeskriptor. Ein Zeiger auf eine Methodentabelle wird über das Objekt delegatee ermittelt. Der Methodenname und der Methodendeskriptor wird in der Methodentabelle gesucht. Durch den Compiler ist sichergestellt, daß Methodenname und Methodendeskriptor mit genau einem Eintrag in der Methodentabelle übereinstimmt.
Der Eintrag in der Methodentabelle enthält einen Verweis auf einen Methodenblock. Wenn die Methode mit der Annotation synchronized versehen ist, wird für das Objekt delegatee der entsprechende Monitor gestartet.
Die Objektreferenzen und die Argumente werden vom Stack geholt und werden als Initialwerte für die entsprechenden lokalen Variablen der neuen Methode übernommen. Die Ausführung wird mit der ersten Anweisung der neuen Methode fortgesetzt.

Laufzeitfehler Wenn delegatee == null ist, wird eine ParentIsNullException (s. Kapitel 5.3.4) ausgelöst. Wenn während der Ausführung der Methode der Stack überlauft, dann wird ein StackOverflowError ausgelöst.

Literaturverzeichnis

[Albano 93] A. Albano, R. Bergamini, G. Ghelli und R. Orsini, *An Object Data Model with Roles*, in Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

[Arnold 96] Ken Arnold und James Gosling, *The Java Programming Language*, Addison-Wesley, Reading, MA, USA, 1996.

[Booch 94] Grady Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, Redwood City, CA, USA, 1995. Zweite Auflage.

[Canning 89] Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff, *Interfaces for Strongly-Typed Object-Oriented Programming*, in OOPSLA '89 Conference Proceedings, Band 24 (10) von SIGPLAN Notices, Seiten 457–467, ACM Press, Oktober 1989.

[Chambers 89] C. Chambers, D. Ungar und E. Lee, *An Efficient Implementation of* SELF *a Dynamically-Typed Object-Oriented Language Based on Prototypes*, in OOPSLA '89 Conference Proceedings, Band 24 (10) von SIGPLAN Notices, Seiten 49–70, ACM Press, Oktober 1989.

[Chambers 91] C. Chambers und D. Ungar, *Making Pure Object-Oriented Languages Practical*, in OOPSLA '91 Conference Proceedings, Band 26 (11) von SIGPLAN Notices, Seiten 1–15, ACM Press, Oktober 1991.

[Chambers 92] C. Chambers, *Object-Oriented Multi-Methods in Cecil*, in *Proceedings of ECOOP'92*, Utrecht, Niederlande, Juli 1992.

[Dahl 68] O. J. Dahl, Myrhaug B. und K. Nygaard, *SIMULA 67 Common Base Language*, Oslo, 1968.

[Driesen 95] Karel Driesen, Urs Hölzle und Jan Vitek, *Message Dispatch on Pipelined Processors*, Technischer Bericht, University of California at Santa Barbara, 1995, accepted at ECOOP'95, to appear in Lecture Notes on Computer Science, Springer Verlag, 1995.

[Ellis 95] Margaret A. Ellis und Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, MA, USA, 1990, 1995.

[Gamma 95] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, USA, 1995.

[Goldberg 89] Adele Goldberg und David Robson, *Smalltalk-80: The Language*, Addison-Wesley, Reading, MA, USA, 1989.

[Gosling 95] James Gosling und Henry McGilton, *The Java Language Environment*, Technischer Bericht, Sun Microsystems Copmuter Corporation, October 1995, A White Paper.

[Gosling 96] James Gosling, Bill Joy und Guy Steele, *The Java Language Specification*, Addison-Wesley, Reading, MA, USA, August 1996.

[Hölzle 91] Urs Hölzle, C. Chambers und U. Ungar, *Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Chaches*, in Proceedings of ECOOP'91, Lecture Notes in Computer Science 512, Seiten 21–38, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong, 1991, Springer-Verlag.

[Hölzle 94] Urs Hölzle, *Adaptive Optimization for SELF: Reconceiling Responsiveness with Performance*, Dissertation, Stanford University, 1994.

[Kim 89] Won Kim und Frederik H. Lochovsky, Hrsg., *Object-Oriented Concepts, Databases and Applications*, ACM Press, Addison-Wesley, Reading, MA, USA, 1989.

[Kniesel 95] Günter Kniesel, *Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems*, Technischer Bericht, Institut für Informatik III, Universität Bonn, Bonn, Oktober 1994, (revised April 1995).

[Kniesel 96a] Günter Kniesel, *Objects don't migrate! Perspectives on Objects with Roles*, Technischer Bericht, Institut für Informatik III, Universität Bonn, Bonn, April 1996.

[Kniesel 96b] Günter Kniesel, *Encapsulation = Visibility + Access Rights*, Technischer Bericht, Institut für Informatik III, Universität Bonn, Bonn, November 1996.

[Lieberman 86] Henry Lieberman, *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems*, in OOPSLA '86 Conference Proceedings, Band von SIGPLAN Notices, Seiten 214–223, ACM Press, September 1986.

[Lindholm 96] Tim Lindholm und Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading, MA, USA, September 1996.

[Madsen 93] Ole Lehrmann Madsen, Birger Moller-Pedersen und Kristen Nygaard, *Object-Oriented Programming in the BETA Programming Language*, ACM Press, Addison-Wesley, Reading, MA, USA, 1993.

[Masini 91] Masini et al., *Object Oriented Languages*, Academic Press 1991.

[Meyer 92] B. Meyer, *Eiffel: The Language*, Prentice-Hall, Englewood Cliffs, N.J., 1992.

[Mössenböck 93] Hanspeter Mössenböck, *Objektorientierte Programmierung in Oberon-2*, Springer-Verlag Berlin, Heidelberg, New York, 1993.

[Moon 89] D. A. Moon, *The Common Lisp Object-Oriented Programming Language Standard*, in [Kim 89], Kapitel 4, S. 49ff.

[Palsberg 94] Jens Palsberg und Michael I. Schwartzbach, *Object-Oriented Type Systems*, John Wiley & Sons, New York, London, Sydney, 1994.

[Schickel 97] Matthias Schickel, *Lava – Konzeptionierung und Implementierung von Delegationsmechanismen in der Java Laufzeitumgebung*, Diplomarbeit, Institut für Informatik III, Universität Bonn, Bonn, Dezember 1997.

[Smith 95] Walter R. Smith, *Using a Prototype-Based Language for User Interfaces: The Newton Project's Experiences*, in *OOPSLA'95 Conference Proceedings*, SIGPLAN Notices, New York, New York, 1995, ACM, ACM Press.

[Stroustrup 87] Bjarne Stroustrup, *Multiple Inheritance for C++*, in Proc. of European Unix Users Group Conference, Helsinki, May 1987.

[Stroustrup 92] Bjarne Stroustrup, *Die C++ Programmiersprache*, Addison-Wesley, Bonn, 1992. Zweite Auflage, deutsche Übersetzung.

[Stroustrup 94] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, Reading, MA, USA, 1994.

[Ungar 87] U. Ungar und R. B. Smith, *SELF: The Power of Siplicity*, in OOPSLA '87 Conference Proceedings, Band 22 (12) von Special issue of SIGPLAN Notices, Seiten 227-242, ACM Press, December 1987.

[Vitek 92] Jan Vitek, R. Nigel Horspool und James S. Uhl, *Compile-time analysis of object oriented programs*, in Proceedings CC '92, 4th International Conference on Compiler Construction, Paderborn, Germany, Seiten 236–250, Berlin, Heidelberg, New York, London, Paris, Tokyo, Hong Kong, 1992, Springer-Verlag.

[Wegner 90] P. Wegner, *Concepts and paradigms of Object-Oriented Programming*, OOPS Messenger, 1:7-87, August 1990.

[Wirth 92] Niklaus Wirth und Jürg Gutknecht, *Project Oberon – The Design of an Operating System and Compiler*, Addison-Wesley, Wokingham, England, 1992.