Lava
Spracherweiterungen für Delegation in Java

Pascal Costanza, Günter Kniesel, Armin B. Cremers

Institut für Informatik III
Rheinische Friedrich-Wilhelms-Universität Bonn
Römerstraße 164
53117 Bonn
costanza@cs.uni-bonn.de, gk@cs.uni-bonn.de, abc@cs.uni-bonn.de

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, wodurch sich leicht dynamische Verhaltensänderungen modellieren lassen.
Anhand einer Erweiterung der Sprache Java haben wir nun gezeigt, daß sich das Konzept der objektbasierten Vererbung in bestehenden klassenbasierten, streng typisierten Sprachen effizient und bei Bewahrung der Typsicherheit implementieren läßt.

1 Einleitung

Objektorientierte Programmiersprachen sind im Laufe des letzte Jahrzehnts zu etablierten Werkzeugen für die Implementation von Softwarelösungen geworden. Am weitesten verbreitet sind jene Sprachen, die *Vererbung auf Klassenebene* verwirklichen, wie z.B. C++ [Ellis 95] und Java [Arnold 96]. In Verbindung mit einem *strengen Typsystem* eignen sie sich hervorragend für die Implementation flexibler, aber dennoch effizienter und sicherer Systeme.

Auf der anderen Seite gibt es Sprachen, die *Vererbung auf Objektebene (Delegation)* verwirklichen; am bekanntesten sind SELF [Ungar 87], NewtonScript [Smith 95] und Cecil [Chambers 92]. Objekte können in diesen Sprachen Referenzen auf sogenannte Elternobjekte besitzen. Über solche Referenzen werden Eigenschaften der Elternobjekte geerbt, genauso wie in klassenbasierten Sprachen Klassen Eigenschaften der jeweiligen Oberklasse erben. Referenzen auf Elternobjekte 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 die Modellierung von Anwendungsbereichen, in denen *dynamische Verhaltensänderungen* von Objekten eine Rolle spielen. 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 würde, zur Übersetzungszeit eines Programms festzustellen, welche Objekte welche Nachrichten verstehen. Daher können bestimmte Programmierfehler nicht bereits vom Compiler abgefangen werden.

In [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. Mit der Programmiersprache Lava haben wir diesen Vorschlag als Erweiterung der Sprache Java verwirklicht und ein lauffähiges System aus Compiler (s. [Costanza 98]) und Laufzeitumgebung (s. [Schickel 97]) implementiert.

Im folgenden werden wir die wesentlichen Erweiterungen von Java vorstellen, die in Lava vorgenommen wurden. Anschließend werden wir diskutieren, in welchen Anwendungsfällen Lava u.a. eingesetzt werden kann.

2 Sprachdesign

In diesem Kapitel wird die Sprache Lava aus Anwendersicht beschrieben. Die grundlegenden Konzepte werden in Abschnitt 2.1 eingeführt. Ihre Benutzung wird anhand eines Design Patterns aus [Gamma 95] in Abschnitt 2.2 erläutert. Anschließend werden die im Sprachdesign sichtbaren Vorkehrungen zur Typsicherheit beschrieben (Abschnitt 2.3). Die Art, wie Lava die konfliktfreie dynamische Komposition unabhängig voneinander entwickelter Softwarekomponenten unterstützt, wird im letzten Abschnitt dieses Kapitels beschrieben.

2.1 Elternverweise

Verweise auf Elternobjekte unterscheiden sich zunächst einmal nicht von gängigen Referenzen auf Objekte in der Sprache Java. Insbesondere das Ändern eines Elternobjekts in einem Kindobjekt kann als einfache Zuweisung an ein Feld des Kindobjekts aufgefaßt werden, das einen Verweis auf ein Elternobjekt darstellt.

Ein Feld kann unter Verwendung des Schlüsselworts delegatee wie folgt als Verweis auf ein Elternobjekt deklariert werden:

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

Semantik Durch die Annotation delegatee wird in obigem Beispiel die Schnittstelle der Klasse ChildClass (in der das Feld parent deklariert ist) wie in einer Klassenvererbungsbeziehung um die sichtbaren Elemente des Klassentyps ParentClass erweitert.[1] (s. Abb. 1 c)

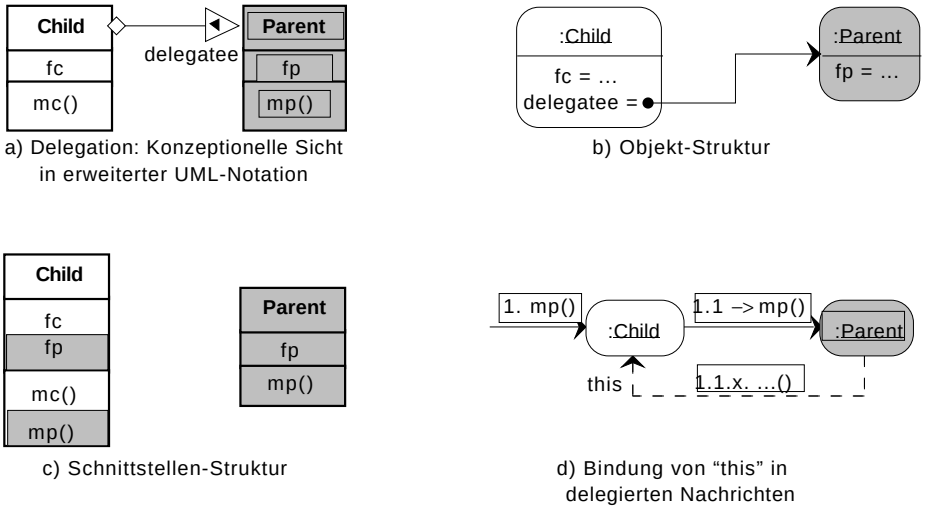

Abb. 1. Semantik der Annotation delegatee

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.

Multiplizität In einer Klassendeklaration können mehrere Felder mit der Annotation delegatee versehen werden. Ggfs. resultieren dadurch Namenskonflikte, die in der aktuellen Version von Lava durch explizite Umbenennung aufgelöst werden müssen (s. auch [Meyer 92]).[2] Im folgenden wird der Einfachheit halber stets davon ausgegangen, daß sich keine Namenskonflikte ergeben.

Delegation Im Gegensatz zur klassenbasierten Vererbung, wo Instanzen einer Unterklasse die Elemente (Felder und Methoden) aus der Oberklasse enthalten, stellt bei objektbasierter Vererbung ein Kindobjekt zwar die Schnittstelle seines

[1] Es handelt sich hierbei um die public und protected Elemente von ParentClass, sowie um die paketweit sichtbare Elemente von ParentClass, wenn ChildClass und ParentClass im gleichen Paket definiert sind.

[2] Für zukünftige Versionen von Lava wird aber auch über andere Alternativen zur Auflösung von Namenskonflikten nachgedacht.

Elternobjekts ergänzt durch seine eigene Schnittstelle zur Verfügung, enthält jedoch tatsächlich nicht die entsprechenden Elemente aus der Elternklasse; stattdessen werden bei Zugriffen auf Feldern oder Aufrufen von Methoden der Elternklasse die entsprechenden Elemente des aktuell referenzierten Elternobjekts adressiert, sie werden an das Elternobjekte *delegiert* (s. Abb. 1 b).[3]

Redefinitionen Eine Methode aMethod, die aus ParentClass geerbt wird, kann in ChildClass redefiniert werden. Wenn aMethod in einer Instanz instance von ChildClass aufgerufen wird, so wird bei einer Redefiniton von aMethod in ChildClass die redefinierte Version ausgeführt. Ansonsten wird wie gewohnt die entsprechende Methode des Objekts aufgerufen, das an parent gebunden ist.

In letzterem Fall wird jedoch die Pseudovariable this nicht an das parent-Objekt gebunden, wie man zunächst in einer klassenbasierten Sprache vermuten würde. Stattdessen bleibt this an den ursprünglichen Empfänger der Nachricht gebunden, in diesem Beispiel also an instance (s. Abb. 1 d). Dadurch hat eine Redefinition einer Methode aus einer Elternklasse einen ähnlichen Effekt wie eine Redefinition einer Methode aus einer Oberklasse: Immer wenn das Elternobjekt Nachrichten an this sendet, werden die im Kindobjekt redefinierten Methoden ausgeführt, sofern sie vorhanden sind.

Explizite Delegation Da Methoden von Elternobjekten redefiniert werden können, wird eine zu super-Aufrufen bei klassenbasierter Vererbung analoge Aufrufmöglichkeit für Methoden aus Elterntypen angeboten: Es handelt sich um die sogenannte *explizite Delegation*, 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 einer solchen expliziten Delegation wird die Definition von aMethod() ausgeführt, die an das Objekt gebunden ist, auf das parent verweist. Redefinitionen von aMethod() in ChildClass werden dabei ignoriert.

2.2 Anwendungsbeispiel *Strategy Pattern*

In Lava kann eine Klasse für Textformatierung (Formatting), die dynamisch zwischen verschiedenen Strategien für Zeilenumbrüche (Typ LineBreaking) wechseln können soll, mit Hilfe von Delegation wie folgt implementiert werden:

```
public class Formatting {
  // Zeilenumbrüche werden an "lineBreaker" weitergeleitet
  delegatee LineBreaking lineBreaker;
  // Erzeugen eines Objekts mit Default-Strategie
```

[3] Erst durch diese Semantik der objektbasierten Vererbung wird gewährleistet, daß eine solche Vererbungsbeziehung zur Laufzeit geändert werden kann.

```
  public Formatting() {
      lineBreaker = new SimpleLineBreaking();
  }
  // Wechseln der Strategie
  public void setLineBreaker(LineBreaking lb) {
      this.lineBreaker = lb;
  }
  // Um wieviele Pixel können einzelne Buchstaben gedehnt werden.
  // Redefiniert entsprechende Methode aus "LineBreaking".
  public int getStretchability(...) {
      ... lineBreaker <- getStretchability(...) ...
  }
}
```

Die Klasse Formatting kann alle Methoden aus LineBreaking verwenden, als wären sie lokal definiert, bzw. aus einer Oberklasse geerbt, jedoch mit dem wesentlichen Unterschied, daß hierfür andere Methoden ausgewählt werden können, indem einfach ein anderes Objekt eines Subtyps von LineBreaking dem Feld lineBreaker zugewiesen wird. Wie in Unterklassenbeziehungen kann eine Feinabstimmung des Verhaltens von LineBreaking-Objekten durch Redefinition einzelner Methoden erzielt werden. Beispielsweise wurde im obigen Beispiel angenommen, daß der Zeilenumbruch u.a. mit Hilfe der Methode getStretchability(...) bestimmt wird. Diese Methode legt fest, um wie viele Pixel einzelne Textelemente gedehnt werden können. Eine Redefinition dieser Methode kann demnach schon ausreichend sein, um das Verhalten des lineBreaker-Objekts an die Bedürfnisse der Klasse Formatting (und anderer Klassen, die den Typ LineBreaking verwenden wollen) anzupassen. Bei der Redefinition kann die Implementation von getStretchability(...) aus dem Elternobjekt via expliziter Delegation aufgerufen werden.

Wesentlich ist, daß bei der Implementation der Klassen vom Typ LineBreaking keine besonderen Vorkehrungen getroffen werden müssen, damit deren Objekte als Elternobjekte eingesetzt werden können.

2.3 Erweiterung des Typsystems

Da durch die Deklaration eines Elternverweises die Schnittstelle einer Kindklasse durch die Elemente der Schnittstelle des Elterntyps erweitert wird, ist die Schnittstelle des Elterntyps also eine Teilmenge der Schnittstelle der Kindklasse. Somit ist bereits eine wichtige Voraussetzung erfüllt, damit die Kindklasse ein Subtyp des Elterntyps sein kann. Dies reicht jedoch nicht aus, da Elternverweise mit dem Wert null belegt sein können. In diesem Fall wird bei dem Versuch, eine Nachricht an dieses Feld zu delegieren, eine Ausnahme zur Laufzeit des Programms ausgelöst.[4] 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.

[4] Es handelt sich um die in Lava neu eingeführte, überprüfte Ausnahme ParentIsNullException.

mandatory Felder Um dennoch eine Subtypbeziehung zwischen Kindklasse und Elterntyp zu etablieren, wurde in Lava eine Möglichkeit geschaffen, die garantiert, daß ein Feld zwar verändert, aber nicht mit dem Wert null belegt werden kann.[5] Dazu wurde die Feldannotation mandatory eingeführt, die genau dies sicherstellt. Zu diesem Zweck nehmen Lava-Compiler und -Laufzeitsystem besondere Vorkehrungen bei Zuweisungen und Initialisierungen vor.

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.[6] Darüberhinaus muß gewährleistet sein, daß ein mandatory Feld explizit initialisiert wird, entweder durch einen Initialisierungswert bei der Deklaration des Felds, oder aber durch eine Zuweisung in einem Konstruktor.

Typsicherheit Aus den obigen Ausführungen ergibt sich, daß für Nachrichten aus der Schnittstelle eines delegatee Felds, das gleichzeitig mit der Annotation mandatory versehen ist, in Instanzen der Kindklasse Methoden des Elterntyps bereitgestellt werden. Genau dann wird in Lava die Kindklasse als ein Subtyp des Elterntyps betrachtet.

Wenn beispielsweise ChildClass wie folgt deklariert ist:

```
class ChildClass {
  // parent ist nach Initialisierung in einem Konstruktor immer != null
  mandatory delegatee ParentClass parent;
  ...
}
```

dann kann eine Variable vom Typ ParentClass sowohl Instanzen von ParentClass und Unterklassen von ParentClass, als auch Instanzen von ChildClass und Unter- oder Kindklassen von ChildClass referenzieren.

2.4 Unabhängige Erweiterbarkeit

Eine zunehmend wichtige Eigenschaft von Programmiersprachen ist die Möglichkeit der *unabhängigen Erweiterbarkeit* von Systemen (s. [Szyperski 98]). Insbesondere bei komponentenbasierter Software werden Programme aus unabhängig entwickelten Bausteinen zusammengesetzt, die darüberhinaus von verschiedenen Herstellern stammen. Daher muß gewährleistet sein, daß diese Komponenten wie erwartet zusammenarbeiten, und nicht unerwünschte Seiteneffekte allein aus der Tatsache entstehen, daß sie gemeinsam eingesetzt werden.

[5] Die Annotation final hilft hier nicht weiter, da sie nur bewirkt, daß ein Feld nicht geändert werden kann; sie garantiert jedoch nicht, daß ein solches Feld nicht null ist.

[6] AssignmentOfNullException ist eine nicht überprüfte Laufzeitausnahme.

Diesem Aspekt wurde in Lava besondere Aufmerksamkeit geschenkt: Angenommen eine Klasse ParentClass wird zum Einen durch eine Klasse SubClass erweitert; zum Anderen wird an Objekte der Klasse ParentClass von Objekten der Klasse ChildClass delegiert. Darüberhinaus existiert sowohl in der Klasse ChildClass, als auch in der Klasse SubClass eine Methode namens open(...).

Möglicherweise sind die beiden Klassen ChildClass und SubClass unabhängig voneinander entwickelt und übersetzt worden, beispielsweise von unabhängigen Programmierteams, die jeweils nur die Klasse ParentClass, nicht jedoch den jeweils anderen Subtyp kannten. Daher ist es sehr unwahrscheinlich, daß die Methoden open(...) in beiden Fällen dieselbe Semantik haben, selbst wenn sie die gleiche Signatur besitzen. Beispielsweise könnte in einem Fall das Öffnen einer Datei, und im anderen Fall die Eröffnung eines Spiels gemeint sein. Die Methode ChildClass::open als Redefinition von SubClass::open anzusehen wäre daher nicht wünschenswert, da dadurch die Semantik von SubClass::open unbemerkt geändert würde, was höchstwahrscheinlich zu schwer auffindbaren Fehlern führt.

Wäre andererseits die Methode open(...) bereits in der Klasse ParentClass definiert, so wäre sowohl den Entwicklern von ChildClass, als auch denen von SubClass diese Methode bekannt, und eine Redefinition von SubClass::open durch ChildClass::open wäre wiederum wünschenswert.

Aus diesen Beobachtungen läßt sich die Regel ableiten, daß eine Methode des Elternobjekts genau dann in einem Kindobjekt redefiniert wird, wenn sie in einem gemeinsamen Obertyp der entsprechenden Kind- und Elternklasse enthalten ist. Genau dieses Verhalten wird von Lava realisiert. (s. Abb. 2)

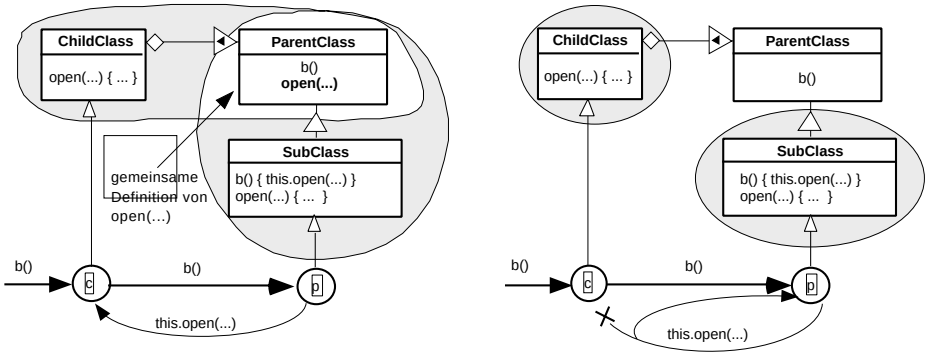

Abb. 2. SubClass::open wird genau dann durch ChildClass::open redefiniert, wenn open bereits in ParentClass definiert ist.

3 Diskussion

Die Art und Weise, wie das Konzept der Delegation in die Sprache Lava integriert wurde, bietet eine ganze Reihe von Vorteilen:

- Es ist generell anwendbar, da es sich mit beliebigen anderen Eigenschaften der Sprache kombinieren läßt (final, transient, mandatory Felder, Nutzung gemeinsamer Elternobjekte aus verschiedenen Kindobjekten (*sharing*), etc.)
- Es muß nicht vorhergesehen werden, an Objekte welcher Klasse delegiert werden kann, bzw. müssen solche Klassen zu diesem Zweck nicht geändert werden.
- Es müssen keine expliziten Methoden geschrieben werden, die lediglich Nachrichten an Elternobjekte delegieren. Außerdem müssen Nachrichten an this nicht explizit an ein möglicherweise vorhandenes Kindobjekt gesendet werden. Beides erspart erheblichen Kodierungsaufwand.
- Die Erweiterung des Typsystems bietet eine wesentlich erhöhte Flexibilität, da auch Objekte von Kindklassen zu einem Typ zuweisungskompatibel sind.
- Wegen der besonderen Berücksichtigung der unabhängigen Erweiterbarkeit ist es nicht möglich, daß Methoden unbeabsichtigt in Kindklassen redefiniert werden, sondern es ist immer klar ersichtlich, welche Methoden von einer Redefinition betroffen sind und welche nicht.

Das Konzept typsicherer dynamischer Delegation ist nicht nur eine Bereicherung des Wortschatzes objektorientierter Sprachen. Es bietet auch einen Beitrag zur konzeptuellen Modellierung: Die Verwendung von Delegation als Modellierungsmittel ist überall dort angebracht, wo ausgedrückt werden soll, daß verschiedene Objekte als eine Einheit zusammenwirken und für die Dauer des Zusammenwirkens nach Außen als ein einziges konzeptuelles Objekt erscheinen. Es gibt zwei typische Gründe für die Zerlegung eines konzeptuellen Objekts in Teilobjekte: Zum Einen die gemeinsame Nutzung eines Teilobjekts durch mehrere andere Teilobjekte und zum Anderen die dynamische Zusammensetzung und Austauschbarkeit von Teilobjekten.

Diese beiden Szenarien sind die Essenz einer Vielzahl von semantischen Beziehungen, die es in der Praxis zu modellieren gilt. Der Aspekt der gemeinsamen Nutzung von Teilobjekten taucht zum Beispiel bei der Modellierung von verschiedenen Sichtweisen / Perspektiven / Aspekten eines Konzepts auf (s. [Shilling 89, Marino 90, Richardson 91, Rieu 92]). Die dynamische Zusammensetzung von Teilobjekten ist wesentlicher Bestandteil bei der Modellierung von Rollen, die ein Objekt im Laufe seines Lebenszyklus annehmen und ablegen kann (s. [Smith 96, Pernici 90, Wieringa 94, Gottlob 96, Kniesel 96]).

Das vertrauteste Beispiel für beide Szenarien sind Personen: Personen können dynamisch neue Rollen und entsprechend neues Verhalten annehmen, z.B. als Studenten, Angestellte, Manager, Musikliebhaber, etc. Gleichzeitig können Personen zu jedem Zeitpunkt aus einer bestimmten, gerade dominierenden Rolle

heraus agieren. Durch die damit verbundenen Änderungen ihres Verhaltens werden sie von ihrer Umgebung unter verschiedenen Sichtweisen / Perspektiven / Aspekten wahrgenommen. In [Kniesel 96] und [Sielski 98] wurde gezeigt wie sowohl Rollen als auch verschiedenen alternative Sichtweisen in Lava durch ein sehr einfaches Entwurfsmuster modelliert werden können.

Insgesamt hat die Einführung von Delegation in einer objektorientierten Sprache eine doppelte Wirkung: Einerseits ermöglicht sie die Modellierung semantischer Konstrukte (wie Perspektiven und Rollen), für die sonst spezielle Spracherweiterungen vorgeschlagen worden sind. Andererseits vereinfacht Delegation die Umsetzung vieler allgemein empfohlener Entwurfsmuster: Strategy, State, Decorator, Chain of Responsibility und Flyweight sind nur einige Beispiele von bekannten Design Patterns, die sich als triviale Anwendungen von Delegation herausstellen. Der Vorteil einer Sprache wie Lava besteht dabei, neben der Reduzierung des Implementierungsaufwands auf die Benutzung des Schlüsselworts `delegatee`, vor allem in der leichten Wartbarkeit der entstehenden Programme (s. [Kniesel 98]).

Zur Zeit existiert eine prototypische Implementierung von Lava (s. [Costanza 98, Schickel 97]) als Erweiterung des Compilers und des Laufzeitsystems des Java Development Kits 1.0.2. Es wurden einige wenige neue virtuelle Maschinenbefehle definiert, die eine Übersetzung der neuen Sprachkonstrukte in Bytecode vereinfacht haben. Dieser Prototyp wurde in [Sielski 98] zur Realisierung eines Rollenmodells benutzt und dabei im Vergleich zu einem rein Java-basierten Ansatz evaluiert. Die Erfahrungen aus diesen Arbeiten werden zusammen mit dem in [Kniesel 99a] im Detail ausgearbeiteten Objektmodell die Basis für die Weiterentwicklung der Sprache und ihrer Implementierung bilden. Konkret soll auf der Sprachebene u.a. die Behandlung von Namenskonflikten vereinfacht werden; auf der Implementierungsebene wird auf eine Erweiterung der Java Virtual Machine um neue Maschinenbefehle verzichtet, da hierdurch entgegen unseren ursprünglichen Annahmen keine wesentlichen Performanzeinbußen zu erwarten sind. Lava bildet darüberhinaus einen der Ausgangspunkte des in Kürze startenden Tailor-Projekts[7], in dem Spracherweiterungen für dynamische Komponentenanpassung (s. [Kniesel 99b]) entwickelt werden sollen.

Literatur

[Arnold 96] K. Arnold und J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

[Chambers 92] C. Chambers, *Object-Oriented Multi-Methods in Cecil*, in Proceedings of ECOOP'92, Utrecht, Niederlande, Juli 1992.

[Costanza 98] P. Costanza, *Lava: Delegation in einer streng typisierten Programmiersprache – Sprachdesign und Compiler*, Diplomarbeit, Universität Bonn, 1998.

[Ellis 95] M.A. Ellis und B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1995.

[7] gefördert durch die Deutsche Forschungsgemeinschaft

[Gamma 95] E. Gamma, R. Helm, R. Johnson und J. Vlissides, *Design Patterns: elements of reusable object-oriented software*, Addison-Wesley, 1995.

[Gottlob 96] G. Gottlob, M. Schrefl und B. Röck, *Extending object-oriented systems with roles*, in ACM Transactions on Information Systems, Vol. 14, Seiten 268–196, 1996.

[Kniesel 95] G. Kniesel, *Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems*, Technischer Bericht, Institut für Informatik III, Universität Bonn, 1995.

[Kniesel 96] G. Kniesel, *Objects don't migrate! Perspectives on Objects with Roles*, Technischer Bericht, Institut für Informatik III, Universität Bonn, Bonn, April 1996.

[Kniesel 98] G. Kniesel, *Delegation for Java – API or Language Extension?*, Technischer Bericht, Universität Bonn, 1998.

[Kniesel 99a] G. Kniesel, *Darwin – Dynamic Object-Based Inheritance with Subtyping*, Dissertation (in Vorbereitung), Institut für Informatik III, Universität Bonn, 1999.

[Kniesel 99b] G. Kniesel, *Type-Safe Delegation for Runtime Component Adaptation*, in Proceedings of ECOOP'99, LNCS (in Vorbereitung), Springer Verlag, 1999.

[Marino 90] O. Marino, F. Rechenmann und P. Uvietta, *Multiple Perspectives and Classification Mechanism in Object-Oriented Representation*, in Proceedings of the European Conference on Artificial Intelligence, 1990.

[Meyer 92] B. Meyer, *Eiffel: The Language*, Prentice-Hall, 1992.

[Pernici 90] B. Pernici, *Objects with roles*, in Proceedings ACM-IEEE Conference of Office Information Systems (COIS), ACM Press, 1990.

[Richardson 91] J. Richardson und P. Schwarz, *Aspects: Extending Objects to Support Multiple, Independent Roles*, in Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 298–307. 1991.

[Rieu 92] D. Rieu und G.T. Nguyen, *Object Views for Engineering Databases*, in Proceedings of 3rd International Conference on Data & Knowledge Systems for Manufacturing & Engineering, 1992.

[Schickel 97] M. Schickel, *Lava – Konzeptionierung und Implementierung von Delegationsmechanismen in der Java Laufzeitumgebung*, Diplomarbeit, Institut für Informatik III, Universität Bonn, 1997.

[Shilling 89] J. J. Shilling and P. F. Sweeney, *Three Steps to Views: Extending the Object-Oriented Paradigm*, in ACM SIGPLAN Notices, Proceedings of OOPSLA '89, 24(10):353–361, 1989.

[Sielski 98] A. Sielski, *Vergleichende Implementierung eines Rollenkonzeptes in verschiedenen Zielsprachen*, Diplomarbeit, Universität Bonn, 1998.

[Smith 95] W.R. Smith, *Using a Prototype-Based Language for User Interfaces: The Newton Project's Experiences*, in *OOPSLA'95 Conference Proceedings*, SIGPLAN Notices, ACM Press, 1995.

[Smith 96] R.B. Smith und D. Ungar, *A Simple and Unifying Approach to Subjective Objects*, in Theory and Practice of Object Systems (TAPOS), 2(3):161-178, 1996, Special Issue on Subjectivity in Object-Oriented Systems.

[Szyperski 98] C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[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.

[Wieringa 94] R. Wieringa, W. de Jonge und P. Spruit, *Roles and Dynamic Subclasses: A Modal Logic Approach*, in Proceedings of ECOOP'94, LNCS 821, pages 32–59. Springer-Verlag, Bologna, Italy, 1994.