Gleichheitsabhängigkeiten zwischen Objekten durch parametrisierte Zuweisungen

Dissertation
zur Erlangung des Grades eines
Doktors der Naturwissenschaften

der Universität Dortmund
am Fachbereich Informatik

von

Daniel Chernuchin

2007

Danksagung

Mein großer Dank gilt Herrn Prof. Dr. Dittrich für die sehr intensive und konstruktive Betreuung, für lange Diskussionen und die Überzeugungsarbeit, formal sauber zu argumentieren. Herrn Prof. Dr. Padawitz sei ebenfalls gedankt für Vorschläge zu entscheidenden Verbesserungen und für die ständige Bereitschaft, meine Fragen zu beantworten.

Für die sorgfältige Korrektur der Arbeit und unzählige interessante Diskussionen bezüglich Rechtschreibung, Grammatik und Ausdrucksweise danke ich Fanny Weinert.

Außerdem gilt der Dank meiner Mutter für die Unterstützung während der Schreibphase.

Weiterhin danke ich Madan Sathe für die Durchsicht der Arbeit.

Besonders herzlich möchte ich meiner Frau Henrike für die vielen Stunden des Korrigierens und des Zuhörens und für ihren unerschütterlichen Glauben an das Gelingen der Dissertation danken.

Tag der mündlichen Prüfung:	9. Oktober 2007
Dekan:	Prof. Dr. Buchholz
Gutachter:	Prof. Dr. Dittrich
	Prof. Dr. Padawitz

Inhaltsverzeichnis

1 Einleitung

Zunächst wird ein Überblick über diese Arbeit gegeben. In Unterkapitel 1.2 folgt ihre Motivation. Danach wird das Ziel der Dissertation ausgearbeitet. Unterkapitel 1.4 stellt den Aufbau der Arbeit vor. Schließlich gibt es einige Anmerkungen zum Schriftstil.

1.1 Überblick

Diese Arbeit beschäftigt sich mit Gleichheitsabhängigkeiten im objektorientierten Umfeld. Dabei werden parametrisierte Zuweisungen zur Umsetzung von Abhängigkeiten vorgeschlagen.

Objektorientierte Entwicklung hat sich bereits als Standard durchgesetzt. Dennoch geht das Erforschen der Konzepte und ihrer Umsetzung in Programmiersprachen weiter. Ein Vorteil der Objektorientierung besteht darin, dass Aspekte der realen Welt durch Programmkonstrukte abgebildet werden. Dies macht die Software, die objektorientiert entwickelt wurde, verständlich.

Ein Phänomen der realen Welt ist, dass verschiedene Objekte als Eigenschaft ein und dasselbe andere Objekt besitzen. Die Änderung bestimmter Objekte kann sich somit auf andere Objekte auswirken. Mit anderen Worten: Es bestehen *Abhängigkeiten* zwischen Objekten.

Auf der Programmierebene wirkt sich das wie folgt aus: Nehmen wir an, dass zwei Variablen den gleichen Wert haben. Eine Änderung des Wertes einer Variablen bewirkt, dass die andere ebenfalls den neuen Wert annimmt. In diesem Zusammenhang spreche ich von *Gleichheitsabhängigkeiten* oder kurz von *Abhängigkeiten*.

Die Abhängigkeiten sind vielfältig. Manche Änderungen der Variablen sollen bewirken, dass abhängige Variablen weiterhin den gleichen Wert haben, andere dagegen, dass die Abhängigkeit aufgelöst wird. Um diese Anforderung zu erfüllen, werden das *3-Schichtenmodell* und darauf aufbauend *parametrisierte Zuweisungen* eingeführt.

Das wesentliche Merkmal des 3-Schichtenmodells liegt darin, dass die Länge des *Pfades* von einer Variablen zu ihrem Wert nicht vom Typ der Variablen bestimmt wird, sondern durch die Art der Zuweisungen, wie ein Wert an eine Variable gebunden wird. Das steht im Gegensatz zu Referenzen in Java oder Zeigern in C++. So sind z.B. Variablen der Typen `int*` und `int**` in C++ verschieden gespeichert und können nicht den gleichen Wert annehmen. Ein Typ gibt die Länge des Pfades zu einem Wert vor. Bei `int**` gilt z.B., dass eine ganze Zahl in der Adresse der Adresse steht, die in der Variablen gespeichert ist. Im 3-Schichtenmodell haben beide Variablen den Typ `int`, die Werte der Variablen sind jedoch durch verschiedene Zuweisungsarten an die Variablen gebunden. Die Abhängigkeit zwischen Variablen wird durch den gemeinsamen Teilpfad der Varia-

blen zum Wert ausgedrückt. Insbesondere ist es von Bedeutung, dass einige Zuweisungen *stärker* sind als andere, und dass es sowohl *symmetrische* als auch *unsymmetrische* Zuweisungen gibt. Mit dem 3-Schichtenmodell ist ein möglichst einfaches Modell entwickelt, das diese Zuweisungsarten unterstützt.

Die Formalisierung des Modells ist unverzichtbar, um wünschenswerte Eigenschaften aufzuzeigen. Deswegen wird eine operationale Semantik für das 3-Schichtenmodell spezifiziert. Auf der Basis dieser Semantik zeige ich diverse Eigenschaften der parametrisierten Zuweisungen und insbesondere die Typsicherheit einer objektorientierten Sprache, die um parametrisierte Zuweisungen erweitert ist.

Außerdem implementiere ich die parametrischen Zuweisungen, um einerseits deren Praxisrelevanz besser demonstrieren zu können und andererseits eine mögliche Umsetzung dieses Ansatzes vorzustellen.

1.2 Motivation

Der Erfolg der objektorientierten Programmierung basiert zum großen Teil auf der Verständlichkeit der Konzepte und des Codes. Diese Verständlichkeit kommt durch die Modellierung der realen Welt zustande. Die objektorientierte Entwicklung passt sich der realen Welt an, indem reale Phänomene auf Programmiersprachen übertragen werden. Auf diese Weise entstanden Klassen, Objekte, Methoden, Vererbung, Templates, usw.

Abhängigkeiten zwischen Entitäten sind ebenfalls ein Aspekt der realen Welt. In dieser Arbeit beschreibe ich eine Verallgemeinerung der Abhängigkeit, die ursprünglich bei Rollen eingeführt wurde. In [CD05a, CD05b] werden Abhängigkeiten bei Rollen einer Entität untersucht (siehe Anhang A). Diese Rollen können gemeinsame Teile haben, die üblicherweise in das so genannte Core Object ausgelagert werden [BRSW00]. Ein Nachteil dieses Verfahrens besteht darin, dass jedes Rollenpaar verschiedene gemeinsame Teile bzw. Abhängigkeiten besitzen kann. Also sollte jedes Paar ein individuelles Core Object besitzen. Dies erhöht quadratisch die Anzahl der Core Objects, was zu Unübersichtlichkeit führt. In meiner Lösung [CD05a, CD05b] referenzieren die Attribute der Rollen ohne den Umweg über das Core Object dieselben Objekte. Auf diesem Weg wirken sich die Veränderungen der Attribute einer Rolle auf die Attribute der anderen Rolle aus.

Abhängigkeiten treten nicht nur bei Rollen auf, sondern betreffen beliebige Entitäten. Schauen wir uns einige Beispiele aus der realen Welt an.

Ein Mann und seine Frau fahren immer dasselbe Auto. Obwohl der Zustand des Autos sich ändert, fahren die beiden weiterhin dasselbe Auto. Dies ist ähnlich den Referenzen in Java oder Zeigern in C++, aber Abhängigkeiten können weitergehen: Wenn ein Ehepartner sich ein neues Auto kauft, fährt der andere ebenfalls mit dem neuen Auto. Beide haben dabei die gleichen Rechte: Jeder von ihnen kann ein neues Auto kaufen, was dieselbe Auswirkung auf die andere Person hat.

In einem Beispiel mit ungleichen Rechten fahren ein Vater und sein Sohn dasselbe Auto. Wenn der Vater das Auto wechselt, fahren sie weiterhin dasselbe Auto; kauft sich jedoch der Sohn ein neues Auto, fahren sie nun verschiedene Autos.

Es gibt viele weitere Beispiele für Abhängigkeiten:

- Ein Konto hat mehrere Kontoinhaber.
- Nachbarn teilen eine Telefonleitung.
- Freunde gehen immer zu demselben Friseur.
- Ein Viereck und ein Dreieck haben dieselbe Kante.

Weitere Szenarien für Abhängigkeiten werden in den Kapiteln 3 und 4 beschrieben.

Abhängigkeiten kommen also immer dann vor, wenn mehrere Objekte *dieselbe* Entität nutzen. Diese Entität wird *Verbindungsobjekt* und die nutzenden Objekte *abhängige Objekte* genannt. Änderungen des Verbindungsobjekts von einem Objekt aus können auf andere Objekte Auswirkungen haben. Oft werden solche Auswirkungen als *Nebeneffekte* bezeichnet, denn ohne dass ein Objekt Änderungen durchgeführt hat, nimmt es Änderungen wahr. Nebeneffekte sind eine grundlegende Eigenschaft der objektorientierten Sprachen, wodurch es besonders schwer ist, diese Sprachen zu formalisieren. Dennoch haben Nebeneffekte eine positive Wirkung, da sie das Verhalten in der realen Welt widerspiegeln.

Wie dargestellt, sind Abhängigkeiten allgegenwärtig und vielfältig. Leider werden Abhängigkeiten heute auf unterschiedliche Arten modelliert und implementiert. Oft sind diese Umsetzungen sehr umständlich. Als Konsequenz ist es schwer, beim Anschauen von zwei Architekturen auf eine ähnliche Problemstellung zu schließen. Somit ist es notwendig, sich mit diesem Thema auseinander zu setzen.

Die angegebenen Beispiele haben die Gemeinsamkeit, dass Attribute verschiedener Objekte den gleichen Wert haben bzw. abhängig sind. Ich erweitere diese Problemstellung dadurch, dass nicht nur der Spezialfall *Attribute*, sondern *Variablen* abhängig sind. Bei der Änderung einer abhängigen Variablen muss die andere abhängige Variable reagieren. Dabei gibt es zwei Arten von Reaktionen: entweder die zweite Variable nimmt den Wert der ersten an, oder sie behält ihren Wert, und die Abhängigkeit wird aufgelöst.

Die *Wertgleichheit* von Variablen ist eine notwendige Voraussetzung für deren Abhängigkeit. Informell ausgedrückt spreche ich von *Gleichheitsabhängigkeiten*, wenn die Änderung einer Variablen die Änderung der anderen hervorrufen kann, so dass diese Variablen weiterhin den gleichen Wert haben. Im Vordergrund steht also, bei welchen Änderungen der ersten abhängigen Variablen die zweite Variable den Wert der ersten annimmt und bei welchen Änderungen sie ihren alten Wert behält.

Da die Wertgleichheit gewöhnlich durch Zuweisungen erzeugt wird, ist es sinnvoll, das Konzept der Zuweisungen zu erweitern und für Abhängigkeiten einzusetzen. Eine Zuweisung zwischen zwei Variablen kann nicht nur den Wert einer Variablen kopieren, sondern einen Teilpfad zu diesem Wert, so dass die Variablen einen gemeinsamen Endpfad haben. Wenn eine folgende Änderung im gemeinsamen Endpfad stattfindet, bleiben beide Variablen abhängig, da die Änderung beide Variablen betrifft. Durch die Länge des Pfades kann man die Abhängigkeit zwischen Variablen abstufen. Je mehr gemeinsamen Pfad zwei Variablen haben, umso mehr potentielle Änderungen betreffen diesen gemeinsamen Pfad und umso weniger die nicht gemeinsamen Anfangspfade der Variablen.

Es sind auch andere Arten von Abhängigkeiten denkbar. Z.B. hängt bei der Funktionsabhängigkeit der Wert einer Variablen durch eine Funktion von dem Wert einer

anderen Variablen ab; bei der Methodenabhängigkeit sind Methoden semantisch äquivalent [CD05a, CD05b]. Da sich diese Arbeit auf die Gleichheitsabhängigkeit der Variablen einschränkt, kürze ich Gleichheitsabhängigkeit durch *Abhängigkeit* ab.

1.3 Ziel

Zunächst soll gezeigt werden, dass Abhängigkeiten zwischen Objekten in der Softwareentwicklung oft auftreten. Es gilt diese Abhängigkeiten zu untersuchen und zu demonstrieren, dass es sinnvoll ist, zur Umsetzung von Abhängigkeiten objektorientierte Sprachen durch ein neues Sprachkonstrukt zu erweitern.

Des Weiteren muss dieses neue Sprachkonstrukt erarbeitet und formal spezifiziert werden. Die operationale Semantik der Erweiterung muss festgelegt und die Typsicherheit anhand dieser Semantik bewiesen werden.

Das neue Sprachkonstrukt soll sowohl Designlösungen als auch anderen Programmiersprachenerweiterungen in Bezug auf Abhängigkeiten überlegen sein. Die Erweiterung für Abhängigkeiten soll die Programmiersprache möglichst wenig verändern und verständlich sein. Aufgrund der Verständlichkeit der Software soll die Entwicklungszeit für Anwendungen mit Abhängigkeiten verringert werden. Es ist ausdrücklich kein Ziel dieser Arbeit, eine algorithmisch schnelle Lösung zu liefern, sondern Entwürfe und Programme sollen vereinfacht werden. Aufgrund der immer schnelleren Prozessoren erkauft man sich heute die Verständlichkeit der Sprachkonstrukte durch Laufzeiteinbußen [HDSM05].

Das Programmkonstrukt muss soweit entworfen und implementiert werden, dass es technisch realisierbar wird. Entwickler von Programmiersprachen sollen durch den dargestellten Entwurf und die gegebene Implementierung in der Lage sein, die spezifizierten Konzepte umzusetzen.

Schließlich gilt es, die Auswirkungen der erarbeiteten Lösung zu untersuchen. Es ist möglich, dass die entwickelte Erweiterung auch zu anderen Zwecken außer zur Abhängigkeit von Attributen einsetzbar ist.

1.4 Vorgehen

Nach der Einführung werden die Grundlagen für diese Arbeit vorgestellt. Ich gehe sowohl auf die objektorientierte Entwicklung als auch auf Zuweisungen ein. Außerdem gibt Kapitel 2 einen kurzen Überblick über Syntax und Semantik.

In Kapitel 3 werden verwandte Arbeiten ausführlich erläutert. Ich unterscheide zwischen Design Patterns und syntaktischen Erweiterungen. Während Design Patterns Abhängigkeiten mit gegebenen Sprachmitteln umschreiben, führen syntaktische Erweiterungen neue Schlüsselwörter für Abhängigkeiten ein. Außerdem wird eine klare Abgrenzung von dieser Arbeit zu anderen gegeben, die den Begriff *Abhängigkeiten* in anderen Zusammenhängen verwenden.

Kapitel 4 verdeutlicht die Allgegenwärtigkeit von Abhängigkeiten und Mängel bestehender Ansätze beim Umgang mit diesen. Es werden Szenarien aus verschiedenen Bereichen der Softwareentwicklung herausgegriffen. Die einzelnen Szenarien werden mit

Mitteln aus Kapitel 3 umgesetzt. Die Vor- und Nachteile der Umsetzungen werden herausgearbeitet. Für jedes Szenario wird ein Vergleich der Ansätze durchgeführt. Wie sich herausstellt, werden für das gleiche Phänomen der Abhängigkeit in verschiedenen Szenarien verschiedene Wege gegangen. Dies ist ein Zeichen für die Notwendigkeit weiterer Forschungen auf diesem Gebiet.

Kapitel 5 ist das Kernkapitel dieser Arbeit. Es führt parametrisierte Zuweisungen ein, mit denen Abhängigkeiten ausgedrückt werden können. Zuerst werden wichtige Eigenschaften von Zuweisungen ausgearbeitet, dann das 3-Schichtenmodell entwickelt. Dieses Modell ermöglicht parametrisierte Zuweisungen mit erwünschten Eigenschaften. Ich stelle die operationale Semantik für das Modell vor und beweise anschließend die Typsicherheit einer statisch typisierten, objektorientierten Sprache mit parametrisierten Zuweisungen.

Kapitel 6 untersucht die Eignung von parametrisierten Zuweisungen zur Umsetzung von Abhängigkeiten. Zunächst werden die Szenarien aus Kapitel 4 aufgegriffen und mit parametrischen Zuweisungen gelöst. Es wird ein Vergleich zu bestehenden Ansätzen vollzogen. Dann werden allgemeine Auswirkungen der Erweiterung einer Programmiersprache um das neue Konstrukt diskutiert.

In Kapitel 7 erläutere ich die wesentlichen Aspekte des Entwurfs und der Implementierung der parametrisierten Zuweisungen. Die komplette Beispielimplementierung befindet sich auf der beiliegenden CD. Auf die vorgestellte Weise kann man z.B. ein Eclipse Plugin erstellen, das Java um parametrisierte Zuweisungen erweitert. Die Implementierung eines Konzepts ist ein wichtiger Schritt auf dem Weg zur Akzeptanz. Außerdem hilft dieses Kapitel Lesern, die mit formaler Spezifikation nicht vertraut sind, die Semantik des Modells zu verstehen.

Das letzte Kapitel fasst die Ergebnisse zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

Anhang A beschäftigt sich mit dem Spezialfall von Abhängigkeiten im Kontext der Rollen. Insbesondere stellt er den von mir entwickelten Rollenansatz RCC vor [CD05a, CD05b].

1.5 Anmerkungen zum Schriftstil

Wie üblich werden wichtige Begriffe, die eingeführt werden, *kursiv* gekennzeichnet. Auch mathematische Formeln sind *kursiv* gedruckt.

Codebeispiele und -auszüge werden in `Schreibmaschinenschrift` vorgenommen. Für Variablen und Klassen wird die so genannte `KamelSchreibWeise` verwendet: Die Zeichenkette wird zusammen geschrieben, und jedes neue Wort fängt mit einem Großbuchstaben an. Dabei ist der erste Buchstabe bei Typen groß und bei Variablen klein. Somit bezeichnet `person` eine Variable, die üblicherweise ein Objekt repräsentiert, `Person` eine Klasse und Person einen normalen Menschen. Die Syntax der Programme entspricht weitgehend Java, wenn explizit nichts anderes angemerkt wird.

Definitionen, **Lemma**ta und *Beweis*e werden mit entsprechenden Überschriften versehen.

1.6 Veröffentlichungen

In dieser Arbeit werden die folgenden eigenen Veröffentlichungen referenziert:

[CD04] Chernuchin, D. und G. Dittrich: *Aspekte der Objektorientierten Modellierung am Beispiel Evolutionärer Algorithmen.* Forschungsbericht 791, University of Dortmund, 2004.

[CD05a] Chernuchin, D. und G. Dittrich: *Dependencies of Roles.* In: *Views, Aspects and Roles at ECOOP '05*, 2005.

[CD05b] Chernuchin, D. und G. Dittrich: *Role Types and their Dependencies as Components of Natural Types.* In: *2005 AAAI Fall Symposium: Roles, an interdisciplinary perspective*, 2005.

[Che03] Chernuchin, D.: *Zur objektorientierten Modellierung Evolutionärer Algorithmen.* Diplomarbeit, University of Dortmund, 2003.

[CLD05] Chernuchin, D., O. Lazar und G. Dittrich: *Comparison of Object-Oriented Approaches for Roles in Programming Languages.* In: *2005 AAAI Fall Symposium: Roles, an interdisciplinary perspective*, 2005.

Diese Veröffentlichungen spielen nur am Rande dieser Dissertation eine Rolle. Aus [Che03] und [CD04] werden nur einige allgemeine Erkenntnisse zur Objektorientierung entnommen. [CD04] basiert auf meiner Diplomarbeit, wobei der überwiegende Teil der Arbeit und der Ideen von mir geleistet wurde. Die Arbeiten [CD05a] und [CD05b] dienen als Grundlage für das Verfahren RCC (siehe Unterkapitel A.2). Auch zu dieser Arbeit habe ich den größeren Teil beigetragen. [CLD05] basiert auf der Diplomarbeit über den Vergleich verschiedener Rollenansätze meines Diplomanden [Laz05], unterscheidet sich jedoch maßgeblich von dieser (siehe Abschnitt 4.3.2 und Unterkapitel 6.4 und A.3). An dieser Veröffentlichung bin ich zur Hälfte beteiligt.

2 Grundlagen

Dieses Kapitel beschreibt die Grundlagen, auf denen die folgenden Kapitel aufbauen.

In dieser Arbeit werden Abhängigkeiten zwischen Objekten in objektorientierter Entwicklung behandelt. Diese Abhängigkeiten werden durch parametrisierte Zuweisungen gelöst. So stellt dieses Kapitel die Grundlagen der Objektorientierung in Unterkapitel 2.1 und die der Zuweisungen in Unterkapitel 2.2 vor. Außerdem gehe ich kurz auf Syntax und Semantik in Unterkapitel 2.3 ein, damit die entwickelten Konstrukte exakt spezifiziert werden können.

2.1 Objektorientierte Entwicklung

2.1.1 Grundbegriffe

In der objektorientierten Entwicklung sind die Begriffe Klasse und Objekt zentral. Eine *Klasse* beschreibt eine Menge von Objekten mit gleichen Attributen, Methoden, Beziehungen und gleicher Bedeutung [Boo95]. *Objekte* sind von ihrer Umwelt abgegrenzte und mit einer eindeutigen Identität versehene Entitäten, die einen individuellen, veränderbaren Zustand besitzen und ein vorgegebenes Verhalten aufweisen [Vin97]. Der Zustand eines Objekts wird durch *Attribute* dargestellt und das Verhalten durch *Methoden* [Mey97].

Eine Klasse wäre zum Beispiel der Mensch an sich, und ein Objekt wäre ein ganz konkreter Mensch. Man könnte sich eine Klasse als einen Begriff vorstellen und ein Objekt als einen Gegenstand, der mit dem Begriff umschrieben wird [Vin97].

Objekte werden über *Variablen* angesprochen. Jedes Objekt kann gleichzeitig von mehreren Variablen referenziert werden. Die Bindung eines Objekts an eine Variable ist in vielen Programmiersprachen eine Voraussetzung für seine Existenz: „To be is to be a value of a variable“ [Qui61].

Die grundlegenden Kompositionsmechanismen sind Vererbung und Delegation. Vererbung ist eine *is-a* Beziehung zwischen Klassen, so dass eine Klasse die Eigenschaften der anderen übernimmt. Man sagt, eine Unterklasse erbt von einer Oberklasse. Sei $\leq$ die transitive Hülle bezogen auf die Vererbung. Delegation ist eine *has-a* Beziehung zwischen Objekten. Dabei hat ein Objekt das andere als ein Attribut [PJ99].

Weiterhin ist die Unterscheidung zwischen statisch und dynamisch typisierten Programmiersprachen für diese Arbeit von Bedeutung. Eine Sprache ist *statisch typisiert*, wenn Variablen mit einem Typ deklariert werden müssen. Zu den statisch typisierten Sprachen gehören z.B. C# [RNW+03], C++ [Str97], Java [HR03] und Eiffel [Mey97]. Ein Vorteil solcher Sprachen liegt in der Typüberprüfung und entsprechender Fehlerfindung während der Compilezeit [Gra89]. Weiterhin kann der Programmierer den Code

besser verstehen, da er zu jeder Zeit den Typ der Variablen kennt, mit dem diese deklariert ist. Im Gegensatz dazu ist eine Sprache *dynamisch typisiert*, wenn Variablen nicht unbedingt deklariert werden müssen. Dazu zählen Smalltalk [Lew99], Python [Wei03] und Objective C [Pil06]. Zu deren Vorteilen gehört eine größere Flexibilität in der Programmiersprache, da keine Typbarrieren vorhanden sind. In dieser Arbeit werde ich mich hauptsächlich auf statisch typisierte Programmiersprachen beschränken.

Bei statisch typisierten Programmiersprachen hat jede Variable einerseits einen statischen Typ, mit dem sie deklariert ist, und andererseits einen dynamischen Typ, also den Typ des Objekts, das von der Variablen referenziert wird. Der dynamische Typ muss immer entweder eine Unterklasse oder gleich dem statischen Typ sein. Wichtig in diesem Zusammenhang sind die Begriffe *Upcast* und *Downcast*. Üblicherweise handelt es sich dabei um Zuweisungen, bei denen auf der rechten und linken Seite Variablen unterschiedlicher statischer Typen sind. Wenn eine Variable der Oberklasse einer Variablen der Unterklasse zugewiesen wird, ist es ein Downcast. Dabei kann es zu einem Typfehler kommen, wenn der dynamische Typ der rechten Variablen nicht eine Unterklasse oder gleich dem statischen Typs der Variablen auf der linken Seite ist. Wenn eine Variable der Unterklasse einer Variablen der Oberklasse zugewiesen wird, handelt es sich um einen Upcast. Dabei kann kein Typfehler auftreten, da jedes Objekt der Unterklasse auch ein Objekt der Oberklasse ist.

2.1.2 Bewertungskriterien

Softwareprojekte erreichen, sowohl durch ihre Größe als auch durch gestiegene Anforderungen, eine immer größere Komplexität. Bei der Entwicklung von Softwaresystemen soll eine hohe Qualität bei gleichzeitig möglichst niedriger Entwicklungszeit und geringen Entwicklungskosten erzielt werden [Bal98, TOHS99].

Die Methoden zur Softwareentwicklung haben der wachsenden Komplexität Rechnung getragen und unterschiedliche Prozesse wie den Rational Unified Process [Kru04] oder Extreme Programming [Bec03] hervorgebracht. Ohne einzelne Prozesse näher zu beleuchten, lassen sich Gemeinsamkeiten erkennen. Die Wiederverwendung einzelner Teile und die Verständlichkeit der Architektur und des Codes sind Voraussetzungen, um durch mehrfache Verwendung und weniger Eigenentwicklungen Entwicklungskosten und -zeit zu reduzieren [Bal98, SMC74].

Die Ziele kann man als einzelne Qualitätsmerkmale wie Benutzerfreundlichkeit, Geschwindigkeit, Robustheit usw. konkretisieren. Es wird zwischen externen und internen Merkmalen unterschieden. Interne Merkmale sind dabei einzig vom Softwareentwickler wahrnehmbar, externe Merkmale dagegen auch vom Benutzer einer Software. Für die Qualität des Softwaresystems spielen letztendlich nur die externen Merkmale eine Rolle, denn nur diese werden vom Anwender wahrgenommen. Für ihn ist es uninteressant, ob der Quelltext gut dokumentiert ist, wenn die Anwendung nicht wie gewünscht funktioniert [Mey97].

Um eine Architektur ebenso beurteilen zu können wie ein fertiges Softwaresystem werden in [Mey97, CD04] Kriterien für interne Merkmale vorgestellt. Zu diesen Kriterien gehören Zerlegbarkeit, Kombinierbarkeit, Kontinuität und als wichtigstes Kriterium Ver-

ständlichkeit.

Zerlegbarkeit und Kombinierbarkeit verlangen, dass ein Programm aufgeteilt werden kann. Dadurch sollen die entstandenen Teile selbst weniger komplex und damit besser beherrschbar und einfacher umzusetzen sein. Außerdem soll der Grad der Abhängigkeit zwischen den Modulen [Dij76, YC78] möglichst gering gehalten werden. Die erzeugten Teillösungen sollen zu der ursprünglichen Gesamtlösung rekombiniert werden können, aus der sie entstanden sind, und sich außerdem auch in neuen Umgebungen einsetzen lassen.

Eine Architektur erfüllt das Kriterium der Kontinuität, wenn kleine Änderungen in der Problemstellung nur Änderungen in wenigen Modulen nach sich ziehen [Bau04].

Die Verständlichkeit besagt, dass einzelne Module der menschlichen Intuition entsprechen. Sie werden unabhängig von anderen Modulen verstanden. Die Verständlichkeit lässt sich aber auch auf andere Ebenen übertragen. Für die Architektur ist sie erfüllt, wenn die statische Struktur der Module auf die dynamische Interaktion schließen lässt. Sind direkte oder indirekte Abhängigkeiten zwischen den Modulen vorhanden, sollten diese leicht nachvollziehbar sein. Für den Code bedeutet Verständlichkeit leichte Lesbarkeit.

Um das Kriterium Verständlichkeit möglichst gut zu erfüllen, versucht die objektorientierte Entwicklung, mit ihrem Klassenkonzept eine möglichst einfache Abbildung der realen Welt zu schaffen, denn Softwareentwickler sind mit der realen Welt vertraut [Che03]. Dort besitzen Objekte Eigenschaften und ein bestimmtes Verhalten. Eigenschaften der Objekte werden auf Attribute abgebildet und Handlungen auf Methoden. Die Grundidee der objektorientierten Programmierung beschreibt Grady Booch treffend: „Wenn Prozeduren und Funktionen Verben sind und Daten Substantive, richtet sich ein prozedurales Programm an den Verben aus, während ein objektorientiertes Programm an den Substantiven ausgerichtet ist.“ [Boo95]. Auch meine Arbeit versucht Programmiersprachen näher an die reale Welt zu bringen, indem ein Phänomen der realen Welt, nämlich Abhängigkeiten, direkt ausgedrückt wird.

2.1.3 Nebeneffekte

Nebeneffekte sind ein unabdingbarer Bestandteil der objektorientierten Programmierung; denn diese bildet die reale Welt ab, und Nebeneffekte sind ein Teil der realen Welt.

Man spricht von Nebeneffekten, wenn nicht alle Auswirkungen eines Programmfragments aus ihm direkt erschließbar sind. Einerseits kann die Funktionalität anderer Programmfragmente dadurch verändert werden, andererseits ist die Wirkung des gegebenen Fragments eine andere als die direkt ersichtliche.

Es gibt viele Beispiele für Nebeneffekte:

- Im Fall des call-by-reference wird die Variable nicht nur innerhalb der Methode, sondern auch außerhalb verändert (siehe Unterkapitel 4.2).
- Wenn zwei Variablen das gleiche Objekt referenzieren, haben bestimmte Veränderungen der einen Variablen Auswirkungen auf die andere (siehe Abschnitt 2.2.5).
- Durch Zeigerarithmetik werden implizit Variablen Werte zugewiesen, ohne dass eine Variable explizit erwähnt wird [Str97].

- Dank Polymorphie und dem dynamischen look-up kann man bei einem Methodenaufruf nicht erkennen, welche Methode welcher Klasse tatsächlich aufgerufen wird [RRH00, HR03].
- Aspektorientierte Programmierung basiert zum großen Teil auf Nebeneffekten. Die Funktionalität der Nebeneffekte ist aus den Kernmodulen ausgelagert und ist aus dem gegebenen Code oft nicht erkennbar [Voß05].
- Die berühmte goto-Anweisung führt dazu, dass der Programmfluss für den Entwickler nicht immer nachvollziehbar ist [Dij68].

Nebeneffekte haben sowohl Vor- als auch Nachteile. Nachteilig wirkt sich z.B. aus, dass sie das Testen der Programme erschweren [DRP01, SW02]. Weiterhin ist die Formalisierung deutlich schwieriger, wenn die Programmiersprache viele Nebeneffekte erlaubt. So sind funktionale Sprachen viel einfacher zu formalisieren als objektorientierte [SS05, SA97]. Auch die Wartung eines Systems mit vielen Nebeneffekten ist schwierig. Im Fall der Zeigerarithmetik z.B. werden Programme manchmal unverständlich. Deswegen wird z.B. in [HLR+99] versucht, Nebeneffekte zu vermeiden oder zu mindern.

Vorteile der Nebeneffekte leiten sich von der realen Welt ab. Viele Änderungen von Objekten verursachen Änderungen an ganz anderen Stellen. Dieses Prinzip in Programmiersprachen zu übernehmen, erhöht die Verständlichkeit der Software. Ein Programmierer wünscht sich oft die Wirkungen der Nebeneffekte, ohne dass er die expliziten Folgen genau angeben muss. Auch diese Arbeit führt Nebeneffekte ein: Objekte sind voneinander abhängig; wenn ein Objekt sich ändert, hat es gewisse Auswirkungen auf das andere Objekt, ohne dass diese explizit aufgeführt werden.

Meiner Meinung nach ist eine Abhängigkeit zwischen Variablen in vielen Fällen ein nützlicher Nebeneffekt, der der menschlichen Intuition entspricht.

2.2 Zuweisungen

In dieser Arbeit werden parametrisierte Zuweisungen entwickelt, um Abhängigkeiten zwischen Variablen umzusetzen. Deswegen ist der Begriff einer Zuweisung zentral.

2.2.1 Grundbegriffe

Als erstes müssen die Begriffe der Variablen und der Werte geklärt werden. Eine *Variable* bindet einen Namen an einen Speicherort. Ein *Wert* ist seinerseits in einem Speicherort gespeichert. Manchmal kann auf ihn über eine Variable zugegriffen werden [Vin97, HR03].

Man unterscheidet zwischen *expliziten* und *impliziten* Zuweisungen. Explizite Zuweisungen finden mittels eines Zuweisungsoperators, z.B. =, statt:

```
x = expression;
```

Dabei ist `x` eine Variable.

Bei impliziten Zuweisungen wird kein Zuweisungsoperator verwendet. Ein Beispiel für implizite Zuweisungen ist die Parameterübergabe beim Methodenaufruf. Dabei wird der

Wert des aktuellen Parameters einer Variablen in der Methode zugewiesen (siehe Unterkapitel 2.2.7).

Explizite Zuweisungen werden hauptsächlich in drei Arten angetroffen:

- Kopie eines Wertes,
- Zuweisungen zwischen Referenzen à la Java und
- Herstellung einer Referenz à la C++.

Allgemein lässt sich sagen, dass die Wirkung einer Zuweisung vom Datentyp der Variablen abhängig ist. Der Mechanismus bei Zuweisungen verschiedener Datentypen ist ähnlich, aber die Interpretation ist unterschiedlich. Mit Hilfe der Zeiger sind alle in diesem Unterkapitel vorgestellten Zuweisungen möglich: Auch die Zuweisungen bei Sprachen ohne Zeiger können durch Zeiger simuliert werden.

2.2.2 Zeiger

Dieser Abschnitt behandelt Zeiger am Beispiel von C++ [Str97].

Eine Variable vom Typ `T*` kann die Adresse eines Objekts vom Typ `T` speichern. Analog speichert eine Variable des Typs `T**` die Adresse eines Objekts vom Typ `T*`.

Sei `x` eine Variable vom Typ `T**`. Mit `&x` greift man auf die Adresse zu, mit der die Variable `x` assoziiert wird. In `x` ist die Adresse der Adresse des Objekts gespeichert. `*x` enthält die Adresse des Objekts. Schließlich erhält man mit `**x` das Objekt vom Typ `T`. Ein Sternchen vor der Variablen bedeutet *Dereferenzierung*, also Zugriff auf das mittels der Speicheradresse verwiesene Element.

Betrachten wir das folgende Codefragment:

```
1 int x = 5;
2 int* y = &x;
3 int** z = &y;
```

Ein möglicher Teilzustand des Speichers nach der Ausführung dieses Fragments ist in Tabelle 2.1 dargestellt. Die erste Zeile enthält die Variablennamen, die zweite die Speicheradressen und die dritte die Inhalte dieser Adressen. Die Werte von `&z`, `z`, `*z`, `**z`,

z	x	y	u	
$\mathbf{a_1}$	$\mathbf{a_2}$	$\mathbf{a_3}$	$\mathbf{a_4}$	$\mathbf{a_5}$
a_3	5	a_2	a_5	3

Tabelle 2.1: Zeiger

`&y`, `y`, `*y`, `&x` und `x` sind in Tabelle 2.2 dargestellt.

Bei einer Zuweisung wird der Wert der Variablen auf der rechten Seite in die Speicheradresse geschrieben, wo der Wert der Variablen auf der linken Seite stand. Z.B. wird bei

&z	z	*z	**z	&y	y	*y	&x	x
a_1	a_3	a_2	5	a_3	a_2	5	a_2	5

Tabelle 2.2: Werte

```
*y = *u;
```

in Tabelle 2.1 der Wert 3 in die Speicherzelle a_2 geschrieben, wo früher 5 stand. Bei

```
y = u;
```

wird a_5 in die Speicherzelle a_3 geschrieben.

Zeiger bringen sowohl Vor- als auch Nachteile mit sich. Zuerst gehe ich auf die Vorteile ein. Die Verwendung von Zeigern kann in bestimmten Fällen den Programmablauf beschleunigen oder helfen, Speicherplatz zu sparen:

- Dynamische Speicherverwaltung ist möglich: Ist die von einem Programm im Speicher zu haltende Datenmenge am Programmstart unbekannt, so kann genau so viel Speicher allokiert werden, wie benötigt wird.
- Bei der Verwendung von Arrays kann man mittels Zeigern schnell innerhalb des Arrays springen und navigieren.
- Verweise auf Speicherbereiche können geändert werden, z.B. zur Sortierung von Listen, ohne die Elemente verschieben zu müssen. Dies ist von Nutzen für dynamische Datenstrukturen.
- Bei Funktionsaufrufen kann es durch die Übergabe eines Zeigers auf einen Wert vermieden werden, diesen Wert zu kopieren. Dies wird *call-by-reference* genannt.
- Anstatt Variablenwerte zu kopieren und so jedes Mal erneut Speicherplatz zur Verfügung zu stellen, kann man in manchen Fällen mehrere Zeiger auf ein und dasselbe Objekt verweisen lassen.

Zeiger bringen auch Nachteile und Gefahren mit sich. Die meisten modernen Programmiersprachen wie Java und C# verzichten bewusst auf den Einsatz von Zeigern. Dies hat vor allem die folgenden Gründe:

- Der Umgang mit Zeigern ist schwierig zu erlernen, kompliziert und fehleranfällig. Vor allem im Sinne von Zeigern zu denken, bereitet Programmieranfängern Schwierigkeiten. Auch bei erfahrenen Programmierern kommen Flüchtigkeitsfehler im Umgang mit Zeigern häufig vor.
- Programmierfehler bei der Arbeit mit Zeigern können schwere Folgen haben. So kommt es z.B. zu Programmabstürzen, unbemerkter Beschädigung von Daten, Pufferüberläufen oder Speicherlecks.

- Zeiger werden oft falsch und unnötig eingesetzt. Dies verkompliziert unnötig den Programmcode.

Um die Vorteile der Zeiger beizubehalten, die Nachteile jedoch zu vermeiden, unterstützen viele Programmiersprachen einen Teil der Zeigerfunktionalität. Dies erhöht vor allem die Verständlichkeit dieser Programmiersprachen.

2.2.3 Referenzierungsgrad

Wie bereits erläutert, wird in jeder Variablen ein Wert gespeichert. Dieser Wert kann eine Zahl, ein Objekt oder eine Adresse sein. Wenn der Wert eine Adresse ist, dann kann man meistens den *eigentlichen Wert* der Variablen mit Hilfe dieser Adresse finden. So ist der Wert von z in Tabelle 2.1 a_3, während der eigentliche Wert, also der uns interessierende Wert, 5 ist. Dieser eigentliche Wert wird durch einen *Pfad* aus Adressen erreicht. Er wird dadurch gebildet, dass Speicherzellen Adressen von anderen Speicherzellen als Inhalt haben. Bei z sind es die Adressen a_1, a_3 und a_2. Ich kürze im Folgenden „eigentlichen Wert“ durch „Wert“ ab.

Der *Referenzierungsgrad* benennt die Anzahl der Dereferenzierungsschritte, die benötigt werden, bis der Wert einer Variablen erreicht wird. Je höher der Referenzierungsgrad einer Variablen, desto länger ist ihr Pfad.

Bei Zeigern in C++ ist der Referenzierungsgrad gleich der Anzahl der Sternchen im Typ der Variablen. Z.B. ist der Referenzierungsgrad

```
int*** x;
```

der Variablen x gleich 3.

In allgemein bekannten Programmiersprachen hängt der Referenzierungsgrad von dem Typ der Variablen ab. In dieser Arbeit wird hingegen ein Ansatz mit einem vom Typ unabhängigen und bei einer Variablen veränderbaren Referenzierungsgrad vorgestellt (siehe Kapitel 5).

In der Literatur wird zwischen Werttypen und Referenztypen unterschieden [Knu97]. Instanzen eines Werttyps enthalten die Daten direkt, haben also den Referenzierungsgrad 0. Sie sind üblicherweise auf einem Stack angeordnet. Werttypen können sowohl Aufzählungen sein als auch vom Programmierer definiert werden. Zu den Werttypen gehören

- primitive Datentypen in Java und C#,
- Enumerations in C# und
- Structs in C#.

Instanzen von Referenztypen speichern die Speicheradresse des Wertes, haben also den Referenzierungsgrad 1. Sie sind üblicherweise auf einem Heap angeordnet. Zu den Referenztypen gehören

- Klassen in Java, C# und Smalltalk und
- Arrays in Java, C# und C++.

2.2.4 Kopie eines Wertes

Bei einer Wertkopie wird der Wert einer Variablen kopiert. Sollte der Wert der Variablen mehrere Speicherzellen umfassen, oder sogar Speicheradressen anderer Variablen beinhalten, wie z.B. bei einem Array, wird der ganze Inhalt kopiert.

Wenn man mit Zeigern arbeitet, hat man z.B. eine Kopie des Wertes bei zwei Variablen vom Typ `T` mit n Sternchen, wenn beide Variablen bei einer Zuweisung n Mal dereferenziert werden:

```
1 T** x;
2 T** y;
3 ...
4 **x = **y;
```

In einer Programmiersprache ohne Zeiger findet eine Kopie des Wertes statt, wenn auf beiden Seiten der Zuweisung Instanzen von Werttypen stehen, z.B Structs in C#:

```
struct T {...}
T x;
T y;
...
x = y;  // Wertkopie
```

Eigentlich passiert bei einer Zuweisung zwischen Werttypen technisch gesehen das Gleiche wie bei einer Zuweisung zwischen Referenztypen oder Zeigern. Der Inhalt einer oder mehrerer Speicherzellen wird kopiert. Nur handelt es sich bei diesem Inhalt bei Werttypen um Werte und bei Referenztypen und Zeigern um Adressen.

Bei einer Wertkopie entsteht zwar Wertgleichheit, aber keine Abhängigkeit zwischen den Variablen. Dadurch, dass es sich nicht um denselben Wert handelt, kann die Änderung des Wertes einer Variablen keine Änderung des Wertes der anderen verursachen.

Leider sind Wertzuweisungen nur zwischen Instanzen von Werttypen möglich. Oft ist es jedoch gewünscht, dass Instanzen von Referenztypen durch eine Zuweisung zwar wertgleich, aber nicht abhängig sind (siehe Kapitel 4).

2.2.5 Zuweisungen zwischen Referenzen à la Java

Leider ist der Begriff *Referenz* von verschiedenen Communities verschieden definiert. Um dennoch exakt zu sein, verwende ich die Begriffe *j-Referenz* für Referenzen à la Java und *c-Referenz* für Referenzen à la C++.

Referenzen in Java und C# entsprechen den C++-Zeigern vom Typ `T*`, wobei die Variable ohne Sternchen angegeben wird. Der folgende Code in Java

```
class T {...}
T x;
T y;
...
x = y;  // Zuweisung zwischen j-Referenzen
```

entspricht semantisch dem folgenden Code in C++

```
class T {...}
T* x;
T* y;
...
x = y;
```

Nebeneffekte werden dadurch erreicht, dass mehrere Variablen das gleiche Objekt referenzieren. Betrachten wir das folgende Codefragment in Java

```
1 MyObject o = new MyObject();
2 MyObject x = o;
3 MyObject y = o;
4 y.changeState();
5 System.out.print(x.getState());
6 y = new MyObject();
7 x.changeState();
```

Ein Objekt wird von zwei Variablen `x` und `y` infolge der Zuweisungen in den Zeilen 2 und 3 referenziert. Wenn eine der Variablen den Zustand des Objekts durch einen Methodenaufruf ändert (siehe Zeile 4), referenzieren beide Variablen das veränderte Objekt. Diese Änderung erkennt man an der Ausgabe in Zeile 5. Es ist also eine Abhängigkeit zwischen `x` und `y` geschaffen. Diese Abhängigkeit bleibt so lange bestehen, bis eine neue Zuweisung an eine der Variablen gemacht wird. Dies ist in Zeile 6 der Fall. Weitere Veränderungen des Objektzustands, die an einer der Variablen durchgeführt werden, haben keine Auswirkungen auf die andere Variable.

Mehrere Variablen von Referenztypen können also das gleiche Objekt referenzieren. Sie sind abhängig, weil sie einen gemeinsamen Endpfad haben. Die Abhängigkeit drückt sich dadurch aus, dass Methodenaufrufe oder Attributveränderungen an einer Variablen dieses Objekt modifizieren, und alle anderen Variablen nun das modifizierte Objekt referenzieren. Eine weitere Zuweisung zu einer dieser Variablen kann die Abhängigkeit auflösen.

j-Referenzen sind bequem und verständlicher als Zeiger. Sie erlauben eine Art von Abhängigkeiten ohne zu viel Flexibilität zuzulassen. Sie ermöglichen z.B. Garbage Collection. In Programmiersprachen ohne Zeiger kann erkannt werden, wann ein Objekt nicht mehr zugreifbar ist. In diesem Fall referenziert keine einzige Variable so ein Objekt. Das Objekt wird dann automatisch vom *Garbage Collector* aus dem Speicher gelöscht [HR03]: „To be is to be a value of a variable“ [Qui61]. Das erspart dem Programmierer die Arbeit, sich um das Löschen jedes Objekts zu kümmern. In Programmiersprachen mit Zeigern muss ein Objekt explizit vernichtet werden, sonst läuft der Speicher mit ungenutzten Objekten voll.

2.2.6 Referenzen à la C++

Wie im vorigen Abschnitt bereits erwähnt, verwende ich für Referenzen in C++ den Begriff *c-Referenz*. Stroustroup definiert c-Referenz als alternativen Namen für ein Objekt

[Str97]. Ursprünglich dienen die c-Referenzen zur Angabe von Argumenten und Rückgabewerten von Funktionen im Allgemeinen und im Besonderen für überladene Operatoren. Die c-Referenzen können jedoch auch zur Implementierung der Abhängigkeiten zwischen Variablen verwendet werden. Die Schreibweise `T&` in C++ bedeutet Referenz auf `T`.

```
1 int x = 1;
2 int& y = x;
3 x = 2;
4 y ++;
```

In den Zeilen 1 und 2 werden die Variablen `x` und `y` deklariert. Beide Variablen haben den Wert `1`. Nach der Zuweisung zu `x` in der Zeile 3 haben beide Variablen den Wert `2`. Ein möglicher Speicherzustand ist in Tabelle 2.3 dargestellt. Dabei ist der Unterschied zu Tabelle 2.1 zu beachten, in der alle Werte in der zweiten Zeile unterschiedlich sind. Schließlich haben nach der Änderung von `y` beide Variablen den Wert `3`. Egal welche der beiden Variablen man verändert, verändert sich der Wert der anderen entsprechend.

x	y
$\mathbf{a_1}$	$\mathbf{a_1}$
1	

Tabelle 2.3: Referenzen in C++

Wenn die Abhängigkeit zwischen `x` und `y` durch eine Zuweisung zwischen zwei j-Referenzen hergestellt wäre (siehe Abschnitt 2.2.5), würde die Abhängigkeit durch die neue Zuweisung Zeile 3 aufgelöst werden.

Leider ist es nicht möglich, eine Gleichheitsabhängigkeit zwischen c-Referenzen wieder aufzulösen. `x` und `y` werden immer den gleichen Wert haben.

2.2.7 Parameterübergabe

Zunächst spezifiziere ich die Begriffe *formaler* und *aktueller Parameter*. Der aktuelle Parameter ist ein Ausdruck beim Methodenaufruf, wie z.B. `x+y` beim Methodenaufruf

```
method(x+y);
```

Durch die folgende Umformung des Codes

```
int expression = x+y;
method(expression);
```

wird der Ausdruck `x+y` durch eine Variable ersetzt. Im Folgenden gehe ich ohne Beschränkung der Allgemeinheit davon aus, dass aktuelle Parameter Variablen sind. Während der Laufzeit einer Methode werden für formale Parameter im Methodenrumpf lokale Variablen angelegt. Statt des vollständigen und korrekten Ausdrucks „für formale Parameter angelegte lokale Variablen“ spreche ich wegen der Kürze von „formalen Parametern“.

Es gibt drei Parameterübergabemechanismen: *call-by-value* und zwei Variationen von *call-by-reference*. Ich unterscheide bei call-by-reference zwischen dem *starken* und dem *schwachen call-by-reference*.

Call-by-value basiert auf der Wertkopie, schwacher call-by-refernce auf j-Referenzen und starker call-by-reference auf c-Referenzen. Bei call-by-value werden die an eine Funktion übergebenen Daten kopiert. Es besteht dann keine Verbindung mehr zwischen den aktuellen und formalen Parametern. Bei schwachem call-by-reference werden die Adressen der Variablen übergeben, anstatt die Daten zu kopieren. Bei starkem call-by-reference wird ein neuer Name für eine bestehende Variable angelegt.

Dementsprechend werden Instanzen von Werttypen durch call-by-value übergeben, während Instanzen von Referenztypen durch schwachen call-by-reference übergeben werden. In vielen Programmiersprachen ist es dagegen nicht möglich Instanzen von Werttypen by-reference und Instanzen von Referenztypen by-value zu übergeben.

Der starke call-by-reference kommt nicht in allen Programmiersprachen vor. In C++ wird dazu die c-Referenz verwendet:

```
1 void increment(int& x) {
2   x ++;
3 }
```

Die Semantik der Parameterübergabe ist definiert als die der Initialisierung. Daher wird beim Aufruf

```
increment(y);
```

ein weiterer Name `x` für `y` angelegt. Nach der Funktionsausführung ist `y` um `1` erhöht.

Der gleiche Effekt wird in C# durch das Schlüsselwort `ref` erreicht [Mic05, RNW+03]:

```
void increment(ref int x) { ... }
```

Der als `ref` übergebene Parameter muss initialisiert sein, bevor er an die Methode weitergeleitet wird. Weiterhin hat C# das Schlüsselwort `out`. Dieses ist für zusätzliche Rückgabewerte gedacht und unterscheidet sich von `ref` dadurch, dass der aktuelle Parameter nicht initialisiert sein muss, dafür ist es notwendig, dass bis zum Ende der Methodenausführung der formale Parameter initialisiert wird.

Pascal [KK96] und Ada [Bar06] erlauben ebenfalls starken call-by-reference.

2.3 Syntax und Semantik

Da in dieser Arbeit neue Sprachkonstrukte vorgestellt werden, ist deren Syntax und Semantik zu spezifizieren. Syntax regelt das Aussehen und die Struktur und Semantik die Bedeutung der Ausdrücke.

2.3.1 Syntax

Um die Syntax zu spezifizieren, setzt man oft Grammatiken ein, am häufigsten handelt es sich dabei um kontextfreie Grammatiken [Cho59].

Eine kontextfreie Grammatik ist ein Tupel

$$G = (V, T, P, S),$$

wobei

- V eine endliche Menge von Nichtterminalzeichen,
- T eine endliche Menge von Terminalzeichen,
- P eine endliche Menge von Produktionen und
- $S \in V$ das Startsymbol ist.

Dabei gilt $V \cap T = \emptyset$. Eine Produktion beschreibt die Überführung von einem Nichtterminalzeichen in Wörter aus Terminal- und Nichtterminalzeichen. Die Sprache $L(G)$ besteht aus Ausdrücken, die durch mehrfaches Anwenden der Produktionen entstehen. Dabei startet man immer mit dem Startsymbol, und die erzeugten Ausdrücke dürfen ausschließlich Terminalzeichen enthalten.

Die Grammatik beschreibt man üblicherweise in der Backus-Naur-Form [Nau60]. Produktionen in der Backus-Naur-Form sehen z.B. so aus:

$$\begin{aligned} \mathit{assign} ::= &\quad \mathit{var}\ {}_1{=}_1\ \mathit{expr} \\ \mid &\quad \mathit{var}\ {}_3{=}_3\ \mathit{expr} \end{aligned}$$

Dabei sind *assign*, *var* und *expr* Nichtterminalzeichen und ${}_1{=}_1$ und ${}_3{=}_3$ Terminalzeichen. *assign* kann sowohl in

$$\mathit{var}\ {}_1{=}_1\ \mathit{expr}$$

als auch in

$$\mathit{var}\ {}_3{=}_3\ \mathit{expr}$$

überführt werden.

Gängige Programmiersprachen wie Java, C, C++, Pascal, Ada, etc. sind so kompliziert, dass man sie nicht vollständig mit einer kontextfreien Grammatik beschreiben kann. Das liegt vor allem an den Kontextbedingungen, die die Programme der meisten Sprachen erfüllen müssen. Hier sind ein paar typische Kontextbedingungen:

- Vor der Variablennutzung, muss die Variable deklariert werden.
- Eine Methode kann nur dann aufgerufen werden, wenn sie deklariert ist.
- Wenn eine Methode mit einer bestimmten Zahl von formalen Parametern deklariert ist, muss man bei jedem Aufruf die richtige Zahl aktueller Parameter angeben.
- Wenn eine Methode mit Parametern von bestimmten Typen deklariert ist, müssen die Typen der aktuellen Parameter bei jedem Aufruf passen.

Dass Sätze einer bestimmten Programmiersprache solche Kontextbedingungen erfüllen müssen, kann man mit keiner kontextfreien Grammatik beschreiben [Sip97]. Trotzdem beschreibt man in der Praxis Programmiersprachen durch kontextfreie Grammatiken und gibt die Kontextbedingungen zusätzlich an. Dabei ist zu beachten, dass die kontextfreie Grammatik von z.B. Java nicht genau die Sprache Java beschreibt, sondern eine etwas umfangreichere Sprache Java+. Zur Sprache Java+ gehören alle korrekten Java-Programme, aber zusätzlich auch Zeichenketten, die große Ähnlichkeit mit Java-Programmen haben, aber bestimmte Kontextbedingungen verletzen [Gru05]. Die komplette kontextfreie Grammatik von Java findet man in [AFF99].

2.3.2 Semantik

Die formale Spezifikation der Semantik der neuen Sprachkonstrukte ist von großer Bedeutung. Diese liefert exakte Anforderungen an die Implementierung. Weiterhin erlaubt sie eine Analyse der Konstrukte; viele Eigenschaften können gezeigt und bewiesen werden. Schließlich dient die formale Spezifikation als Dokumentation und Designhilfe.

Es gibt drei Hauptarten der Semantik: axiomatische [Sch97], denotationale [Sch97, JP04] und operationale [GS02].

Für die erarbeiteten Konstrukte habe ich die operationale Semantik gewählt. Sie verwendet einen Interpreter zur Definition der Sprache. Die Evaluation ist eine Folge von Interpreterkonfigurationen. Die Wahl fällt auf diese Semantik, da die Regeln dieser Semantik an Anweisungen in Programmiersprachen erinnern und für Programmierer imperativer Sprachen am verständlichsten ist.

Der Zustand des Programms wird durch einzelne Anweisungen geändert. Sei s der Startzustand und s' der Zustand nach der Ausführung der Anweisung a. Die Notation ist:

$$s \xrightarrow{a} s'$$

In Kapitel 5 wird der Zustand als ein Tupel von partiellen Funktionen repräsentiert

$$s = (s_4, s_3, s_2, s_1)$$

Dies eignet sich zur Spezifikation von Zuweisungen besonders gut, denn bei Zuweisungen wird ein Wert in genau eine Speicherstelle geschrieben, d.h. eine der Funktionen wird genau an einer Stelle geändert. Nehmen wir an, dass eine Zuweisung

```
x = y;
```

die Funktion s_3 ändert.

$$(s_4, s_3, s_2, s_1) \xrightarrow{x=y} (s_4, s_3', s_2, s_1)$$

Die Funktion s_3' kann z.B. auf die folgende Weise notiert werden:

$$s_3'(a) = \begin{cases} s_3(s_4(y)) & \text{falls } a = s_4(x) \\ s_3(a) & \text{sonst.} \end{cases}$$

Das bedeutet, dass der Inhalt von `y` in `x` kopiert wird und alle anderen Werte in s_3' gleich denen von s_3 bleiben.

3 Verwandte Arbeiten

Dieses Kapitel stellt die verwandten Arbeiten vor. Zuerst werden in Unterkapitel 3.1 mehrere sowohl bekannte als auch neue Design Patterns besprochen, also Wege, mit gegebenen objektorientierten Mitteln bestimmte Abhängigkeiten auszudrücken. In Unterkapitel 3.2 werden syntaktische Erweiterungen von Programmiersprachen betrachtet. Im Gegensatz zu Design Patterns werden neue Sprachkonstrukte eingeführt. So gelingt es dank zusätzlicher Schlüsselwörter, die Verkomplizierung des Designs zu vermeiden. Schließlich erläutert Unterkapitel 3.3 den Begriff *Abhängigkeiten* aus verwandten Forschungsgebieten. Dies ist notwendig, um Verwechslungen zu vermeiden, da Abhängigkeiten allgegenwärtig sind, und dieser Begriff in verschiedenen Kontexten unterschiedliche Bedeutungen hat.

3.1 Design Patterns

Design Patterns sind einheitliche Lösungen für Design Probleme, die immer wieder vorkommen [GHJV97]. In diesem Unterkapitel werden Design Patterns vorgestellt, die bei Abhängigkeiten angewendet werden können, obwohl sie ursprünglich für andere Probleme entworfen sind.

3.1.1 Group of Objects

Group of Objects ist ein bisher nicht in der Literatur explizit vorgestelltes Design Pattern. Es handelt sich dennoch um eine Modellierung, die allgemein bekannt ist und immer wieder eingesetzt wird.

Abbildung 3.1 zeigt eine vereinfachte Version dieses Design Patterns. Bestehende Klassen werden zu einer Klasse `GroupOfObjects` zusammengefügt, so dass `GroupOfObjects` diese Klassen als Attribute enthält. Dabei soll `GroupOfObjects` als eine Entität existieren und nicht nur eine Vereinigung der Bestandteile sein.

Ein Beispiel dafür ist die Klasse `Family`, die Attribute der Typen `Husband`, `Wife`, `Children`, `Account` und `House` enthält. Die Objekte, die als Attribute fungieren, könnten auch ohne die Klasse `Family` existieren. Erst aufgrund der zusätzlichen Anforderungen, die die Sicht auf diese Objekte als Einheit verlangen, wird die `Family` eingeführt. Dieses Design Pattern ähnelt zwar dem Pattern Mediator (siehe Abschnitt 3.1.2), unterscheidet sich jedoch vor allem dadurch, dass die Abstraktion `GroupOfObjects` einer Entität entsprechen muss.

Group of Objects ist nützlich zur Behandlung von Abhängigkeiten. Es ist oft der Fall, dass zwei Objekte mehrere Verbindungsobjekte haben und die Gesamtheit dieser Verbindungsobjekte wieder eine Entität darstellt. In diesem Fall kann man die Verbindungsob-

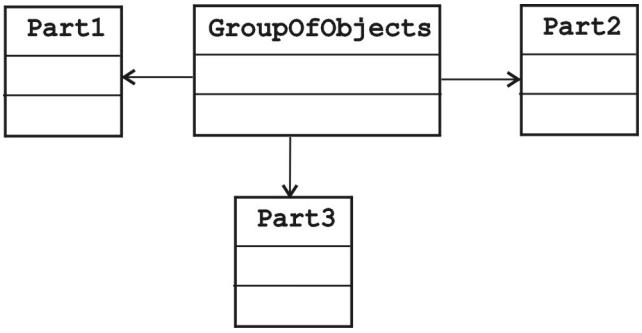

Abbildung 3.1: Group Of Objects

jekte mittels einer neuen Klasse zu einem Objekt zusammenfassen, so dass dieses nun von abhängigen Objekten referenziert wird. Wenn eins der Teile verändert oder ausgetauscht wird, haben die abhängigen Objekte weiterhin den Zugriff auf den aktuellen Teil, da sie `groupOfObjects` referenzieren und nicht alle Teile einzeln.

3.1.2 Mediator

Beim Design Pattern *Mediator* wird ein Objekt definiert, das eine Menge von Objekten als seine Attribute kapselt; es wird als *Mediator* bezeichnet [GHJV97]. Es wird eingesetzt, wenn viele Objekte, genannt *Kollegen*, einander kennen müssen. Konzeptionell stellt jeder Kollege ein Verbindungsobjekt dar, das von anderen abhängigen Kollegen geteilt wird. Das Pattern *Mediator* vermeidet, dass die Kollegen einander direkt referenzieren. Sie brauchen nur den Mediator zu kennen. Wenn eins der betroffenen Objekte ausgetauscht wird, muss nur der Mediator aktualisiert werden. Dies führt zu einer losen Kopplung zwischen den Kollegen.

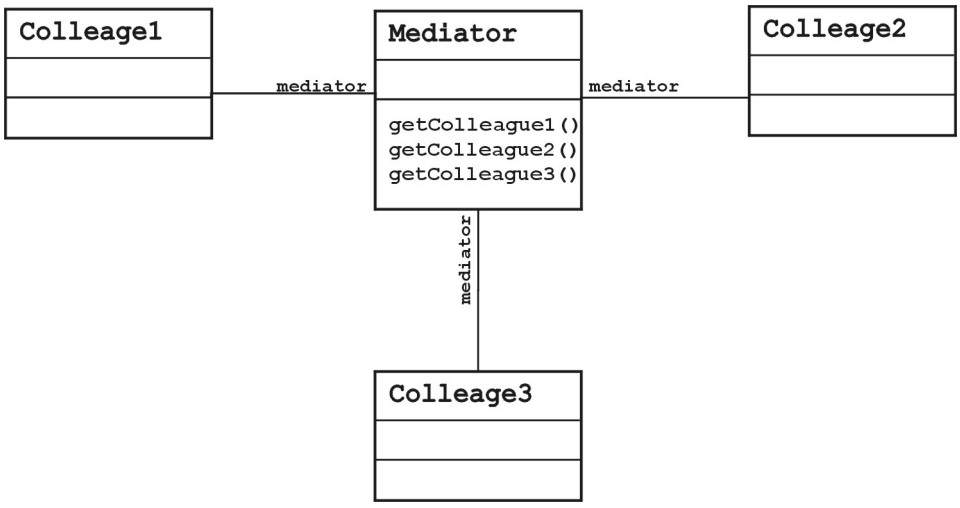

Abbildung 3.2: Mediator

Abbildung 3.2 zeigt eine vereinfachte Version des Mediators. Dieses Design Pattern ähnelt Group of Objects (siehe Abschnitt 3.1.1). Im Gegensatz dazu muss Mediator keine existierende Entität darstellen, sondern dient ausschließlich der Kommunikation zwischen Objekten. Darüber hinaus kennt nicht nur der Mediator die Kollegen, sondern diese auch ihn.

3.1.3 Decorator

Das Design Pattern *Decorator* fügt zusätzliche Funktionalität für Objekte dynamisch hinzu. Es handelt sich um eine flexible Alternative zur Vererbung [GHJV97].

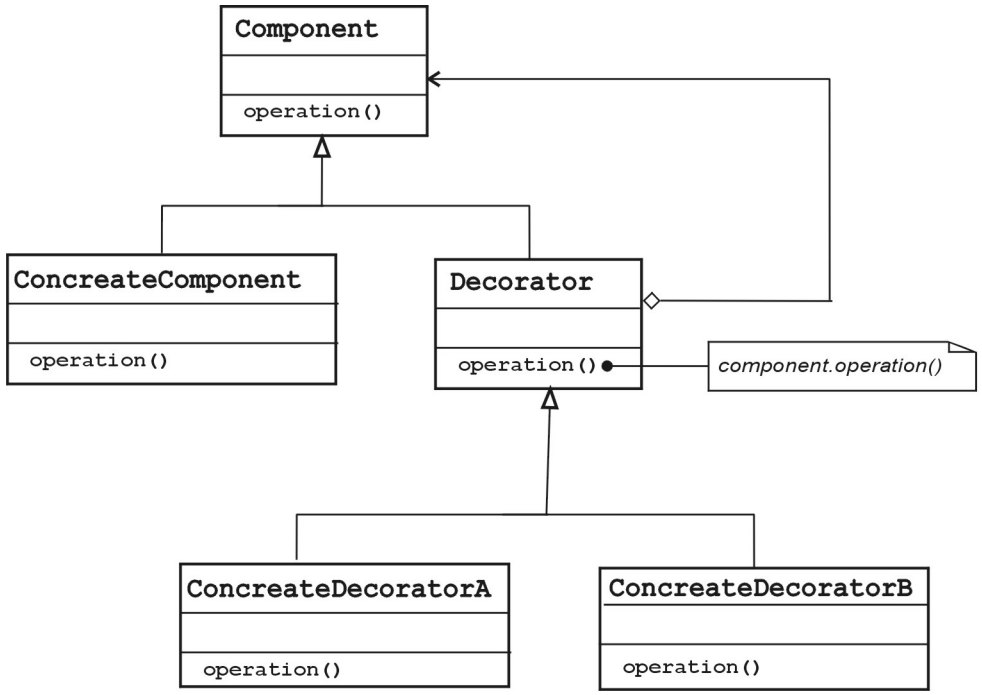

Abbildung 3.3: Decorator

Abbildung 3.3 zeigt ein Klassendiagramm eines Decorators. In [OM01] wird die Eignung dieses Design Patterns zur Behandlung von Abhängigkeiten untersucht (siehe Unterkapitel 4.1). Dabei referenziert `Decorator` das Verbindungsobjekt, während andere Objekte `Decorator` referenzieren. Es wird also eine Zwischenstufe zwischen abhängigen Objekten und dem Verbindungsobjekt eingebaut. So kann die Beziehung zum Verbindungsobjekt flexibel reguliert werden; insbesondere ist das Austauschen des Verbindungsobjekts möglich, ohne dass die Referenz bei allen abhängigen Objekten geändert werden muss. Diese Idee ähnelt den Design Patterns Group of Objects und Mediator (siehe Abschnitte 3.1.1 und 3.1.2). Nur diesmal wird von der Zwischenstufe, also dem Decorator, nur ein Objekt referenziert und nicht mehrere.

Decorator wird ausführlich in Abschnitt 4.1.2 mit seinen Vor- und Nachteilen an einem

Beispiel besprochen.

3.1.4 Observer

Das Design Pattern *Observer* wird bei einer eine-zu-viele Beziehung zwischen Objekten eingesetzt [GHJV97]. Es gibt üblicherweise ein beobachtetes Objekt *Subject* und mehrere beobachtende Objekte *Observer*. Wenn Subject seinen Zustand ändert, werden alle Observer benachrichtigt. Dabei soll die Bindung zwischen Beobachter und Beobachteten möglichst lose sein. Damit lassen sich die Einschränkungen, welche Objekte wen beobachten dürfen, minimieren, und so viele Objekte wie möglich können zu Observern werden.

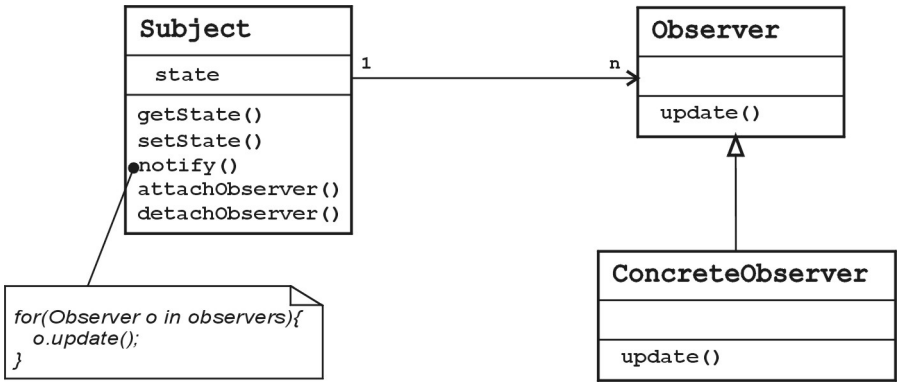

Abbildung 3.4: Observer

Abbildung 3.4 zeigt eine vereinfachte Version des Design Patterns. Subject speichert alle seine Observer. Wenn sich sein Zustand ändert, wird mittels der Methode `notify` in `Subject` die `update` Methode jedes Observers aufgerufen.

Dies ist eine unsymmetrische Abhängigkeit, da nur Subject vom Observer und nicht zusätzlich Observer vom Subject beobachtet wird. Die Änderungen des Zustands von Subject haben Einfluss auf die Observer. Die Änderungen des Zustands eines Observers haben dagegen keine Auswirkungen auf das Subject (siehe Abschnitt 5.1.2).

In dieser Arbeit beschränke ich mich auf die Gleichheitsabhängigkeit der Attribute. Dabei wird sichergestellt, dass bestimmte Attribute in Observer den gleichen Wert annehmen wie entsprechende Attribute in Subject. Das Design Pattern Observer erlaubt jedoch eine beliebige Reaktion auf die Änderung des Zustands, so dass auch eine funktionale Abhängigkeit möglich ist [CD05b].

Ein Nachteil des Observers besteht darin, dass das Design Pattern in statisch typisierten Programmiersprachen unübersichtlich wird, wenn mehrere Subjects beobachtet werden. Alle Observer müssen geeignete Interfaces implementieren. Ein Observer, der drei Subjects beobachtet, implementiert drei Interfaces und hat drei `update` Methoden mit verschiedenen Signaturen.

Nehmen wir an, es gibt folgende Subjects: `SubjectA`, `SubjectB` und `SubjectC`. Dann ist es sinnvoll, Interfaces `ObserverA`, `ObserverB` und `ObserverC` zu definie-

ren, die jeweils Methoden `updateA`, `updateB` und `updateC` deklarieren. Durch die Interfaces gibt es eine zur Compilezeit feststehende Menge von Klassen, die andere Klassen beobachten dürfen. Denn nur Objekte der Klassen, die `ObserverA` implementieren, können Objekte der Klasse `SubjectA` beobachten, was eine schwer wiegende Einschränkung darstellt. Bei Delegates ist dieses Problem gelöst (siehe Abschnitt 3.2.1). Ein weiterer interessanter Ansatz zur Observerimplementierung basiert auf aspektorientierter Programmierung [APS+05] (siehe Abschnitt 3.2.5).

In dynamisch typisierten Programmiersprachen ist der Umweg über Interfaces nicht nötig. Der Einsatz von Observer in einer dynamisch typisierten Sprache ist so erfolgversprechend, dass Observer in Smalltalk zur Grundausstattung der Standardbibliothek gehört. Beim *Dependency Mechanism* in Smalltalk handelt es sich um ein implementiertes Observer Design Pattern [Lew99]. Objekte können mittels Methoden `addDependent:` und `removeDependent:` Beobachter hinzufügen oder entfernen. Mit Hilfe der Methode `changed: with:` benachrichtigt das Subject seine Observer. Dabei berichtet `changed:`, was geändert wird, z.B. der Name des Attributs, und `with:`, wie es geändert wird, z.B. der neue Wert. Nach dem Aufruf von `changed:` am Subject werden die `update` Methoden an allen Observern aufgerufen. Die Hauptmethode in Observern ist `update: with: from:`. Dabei steht `update:` für den Aspekt, der geändert wird, `with:` für das Wie und `from:` für das geänderte Objekt.

Die Tatsache, dass diese Methoden ganz oben in der Klassenhierarchie auftauchen, zeigt, wie wichtig die Umsetzung von Abhängigkeiten ist. Ein Nachteil des Dependency Mechanism besteht darin, dass alle Observer eines Objekts benachrichtigt werden, und nicht nur die, für die die aktuelle Änderung relevant sind. Nehmen wir an, dass ein Subject verschiedene Arten von Observern hat, die an verschiedenen Arten einer Änderung interessiert sind, so werden bei jeder Änderung trotzdem alle Observer benachrichtigt. Die verschiedenen Arten der Änderungen werden durch Argumente vorgenommen: In den `update` Methoden müssen die Objekte zuerst feststellen, ob das Ereignis für sie von Bedeutung ist.

3.1.5 Singleton

Das Design Pattern *Singleton* stellt sicher, dass nur eine Instanz einer bestimmten Klasse global vorhanden ist [GHJV97].

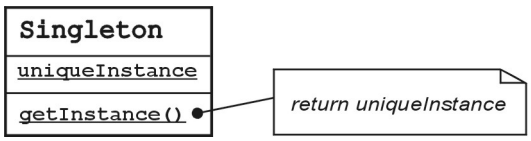

Abbildung 3.5: Singleton

Abbildung 3.5 zeigt die übliche Modellierung des Singletons. Das Unterstreichen der Methoden und Attribute weist darauf hin, dass diese statisch sind [OMG05].

Singleton ist ein Weg Informationen global zur Verfügung zu stellen. Globale Instanzen haben für alle Objekte den gleichen Wert. Die abhängigen Objekte referenzieren das

Singleton und werden somit aktualisiert, wenn Singleton seinen Zustand ändert.

Der größte Nachteil dieses Design Patterns liegt darin, dass das Geheimnisprinzip nicht befolgt wird. Trotzdem wird Singleton aufgrund der einfachen Implementierung oft eingesetzt. Es eignet sich zur Umsetzung von Abhängigkeiten, wenn es sehr viele abhängige Objekte gibt, die global zur Verfügung stehende Informationen benötigen [GHJV97].

3.1.6 Role Object

Das Design Pattern *Role Object* stellt die Standardlösung für flexible Rollendesigns dar. Es ermöglicht eine kontextspezifische Sicht auf das so genannte *Core Object* mittels *Rollen*. Das Core Object enthält eine Menge von Rollen, die dynamisch hinzugefügt und entfernt werden können.

Das Role Object wird nicht für Abhängigkeiten eingesetzt, sondern für Rollen. Abhängigkeiten sind bei Rollen wichtig, denn verschiedene Rollen einer Entität können gemeinsame Eigenschaften haben. Deswegen wird die Umsetzung von Rollen und somit auch das Role Object in dieser Arbeit untersucht (siehe Unterkapitel 4.3, 6.4 und Anhang A).

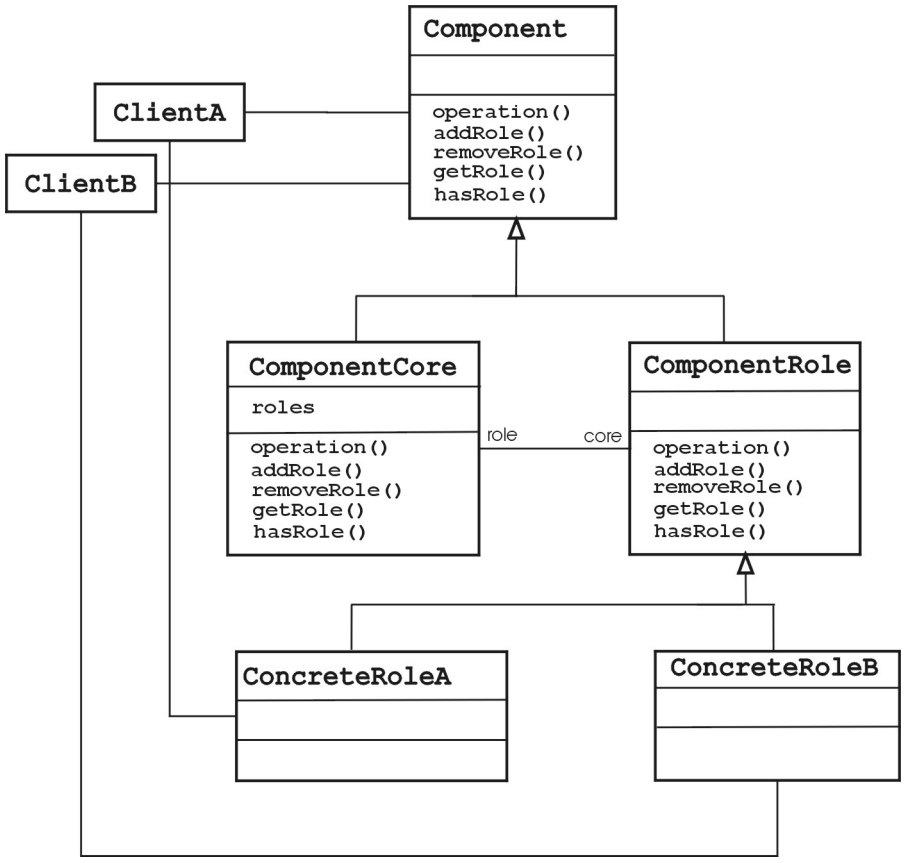

Abbildung 3.6: Role Object

Abbildung 3.6 zeigt das Klassendiagramm des Design Patterns [Fow97, BRSW00]. Im Core Object werden die innewohnenden Eigenschaften gekapselt. Dagegen beinhalten Rollenobjekte zusätzliche Rolleninformationen und -funktionalität. Der Ansatz hat viele Vorteile zur Umsetzung von Rollen [CLD05], bewältigt die Abhängigkeiten zwischen den Rollen jedoch nicht ausreichend.

3.2 Syntaktische Erweiterungen

Der allgemeine Nachteil von Design Patterns besteht darin, dass sie zusätzliche Klassen und somit zusätzliche Komplexität hinzufügen. Viele Programmiersprachenentwickler versuchen durch neue Sprachmittel diese Komplexität zu vermeiden. Manchmal dient ein Sprachmittel lediglich als eine verkürzte Schreibweise für ein Design Pattern. In diesem Kapitel betrachte ich syntaktische Erweiterungen der Programmiersprachen, die mit Abhängigkeiten in Beziehung stehen.

3.2.1 Delegate

Delegate ist ein Konstrukt, das in C# verwendet wird [RNW+03]. Im Unterschied zum Observer Design Pattern (siehe Abschnitt 3.1.4) wird ermöglicht, dass ein Subject verschiedene Methoden seiner Observer aufruft.

Beim Delegate handelt es sich um einen Typ, dessen Objekte auf Methoden verweisen, ähnlich zu Funktionszeigern in C und C++ [Str97]. Delegates sind im Gegensatz zu Funktionszeigern objektorientiert, typsicher [Mic05] (siehe Unterkapitel 5.5) und können mehrere Methoden referenzieren und aufrufen. Diese Methoden müssen mit den Parametern und dem Rückgabewert des Delegates kompatibel sein. Neue Methoden können mithilfe des Additionsoperators hinzugefügt werden. Zur Verdeutlichung gebe ich ein einfaches Beispiel an:

```
1  delegate void SampleDelegate(string message);
2  static void SampleDelegateMethod1(string message) { ... }
3  static void SampleDelegateMethod2(string message) { ... }
4  static void Main() {
5          SampleDelegate sampleDelegate;
6          sampleDelegate += SampleDelegateMethod1;
7          sampleDelegate += SampleDelegateMethod2;
8          sampleDelegate("Hello");
9      }
10 }
```

In Zeile 1 wird der Typ `SampleDelegate` mittels des Schlüsselworts `delegate` deklariert. In den Zeilen 2 und 3 werden zwei Methoden definiert, deren Signaturen bis auf den Namen identisch sind und zu `SampleDelegate` passen. In Zeile 5 wird ein Objekt `sampleDelegate` deklariert. Danach werden in den Zeilen 6 und 7 die beiden definierten Methoden dem Delegate hinzugefügt. Schließlich werden mit

```
sampleDelegate("Hello");
```

die beiden Methoden aufgerufen.

Im Gegensatz zu parametrisierten Zuweisungen können Variablen nicht direkt voneinander abhängig sein. Im Observer muss es eine Methode geben, die ein Attribut des Subjects dem entsprechenden Attribut des Observers gleichsetzt. Diese Methode muss einem Delegate hinzugefügt werden. Erst dadurch, dass Subject bei einer Attributänderung ein Event auslöst und alle Methoden des Delegates aufgerufen werden, kommt es zur Gleichsetzung der Attribute. Das stellt im Vergleich zu parametrisierten Zuweisungen einen Overhead dar.

3.2.2 Compound References

In [OM01] werden *Compound References* eingeführt. Es handelt sich um eine neue Abstraktion für Objektreferenzen. In Kontrast zu j-Referenz (siehe Unterkapitel 2.2) ist eine Bindung der Compound Reference zu einem Objekt nicht absolut, sondern relativ zu einer anderen Referenz. Die Funktionsweise soll hier an einem Beispiel verdeutlicht werden.

```
1  class Person {
2    private Account account;
3    Account getAccount() {
4      return account;
5    }
6    void setAccount(Account newAccount) {
7      account = newAccount;
8    }
9    Account getPersonsAccount() {
10     return this<-account;
11   }
12 }
13 class Main {
14   public static void main()(String[] args) {
15     Person jack = ...;
16     Account ubsAccount = new Account("UBS", "123");
17     Account dbAccount = new Account("Deutsche Bank", "321");
18     jack.setAccount(ubsAccount);
19     Account anAccount = jack.getAccount();
20     Account jacksAccount = jack.getPersonsAccount();
21     // anAccount, jacksAccount referenzieren das UBS-Konto
22     jack.setAccount(dbAccount);
23     // anAccount referenziert noch das UBS-Konto
24     // jacksAccount referenziert das DB-Konto
25   }
26 }
```

Wir sehen eine Klasse `Person`, die ein Attribut `account` und zugehörige getter- und setter-Methoden hat (siehe Zeilen 2 bis 8). Es gibt zusätzlich die Methode `getPersonsAccount`. Diese Methode unterscheidet sich von `getAccount` durch den Gebrauch der Compound Reference mit der Notation `<-` in Zeile 10. Durch

```
return this<-account;
```

wird erreicht, dass `getPersonsAccount` die Compound Reference zum Attribut `account` der Person liefert, während `getAccount` nur eine j-Referenz zurück gibt. Der Effekt der Compound Reference besteht darin, dass sie immer auf den aktuellen Wert des Attributs `account` in einem Objekt der Klasse `Person` zeigt.

Betrachten wir nun den Ablauf des Programms. Zuerst werden in den Zeilen 16 und 17 zwei Konten erzeugt und in den Variablen `ubsAccount` und `dbAccount` gespeichert. Nach der Zuweisung eines neuen UBS-Kontos an das Objekt `jack` in Zeile 18 und der Initialisierung der Variablen in den Zeilen 19 und 20 referenzieren erwartungsgemäß sowohl `anAccount` als auch `jacksAccount` das UBS-Konto. Nun wird dem Objekt `jack` in Zeile 22 ein neues Konto zugewiesen, ohne dass Variablen `anAccount` und `jacksAccount` explizit aktualisiert werden. `anAccount` referenziert weiterhin das UBS-Konto. Dagegen referenziert `jacksAccount` das aktuelle Konto von `jack`, also das DB-Konto, denn es wird mittels einer Compound Reference an das aktuelle Konto des Objekts `jack` gebunden.

Die Coumpound References definieren den zusätzlichen Zugriff auf ein Attribut um. Neben `o.f` kann man `o<-f` schreiben. So ist es möglich, mehrstufige Zugehörigkeiten auszudrücken.

```
x = a<-b<-c<-d;
```

Diese Zuweisung bedeutet, dass `d` ein Attribut des Objekts `c` ist, `c` ein Attribut des Objekts `b` und `b` ein Attribut von `a`. `x` referenziert somit das Objekt, das in `d` gespeichert ist. Sollte `a.b` oder `a.b.c` oder `a.b.c.d` sich ändern, wird `x` automatisch aktualisiert. Ein Beispiel für diesen Sachverhalt wäre

```
x = bestFriend<-car<-luggageBoot<-content<-price;
```

`x` bezeichnet damit den Preis von dem Inhalt des Kofferraums im Auto des besten Freundes. Dies wechselt mit dem besten Freund, seinem Auto, dem Inhalt des Kofferraums und den aktuellen Marktpreisen. Bei parametrisierten Zuweisungen werden dagegen zusätzliche Zuweisungsoperatoren definiert: Zusätzlich zur Standardzuweisung

```
x = y;
```

kann man Zuweisungen der Form

```
x i=j y;
```

für bestimmte i und j anwenden.

Es gibt Gemeinsamkeiten zwischen den beiden Ansätzen. Der Zugriff auf ein Attribut ist eine bestimmte Art, den Speicher auszulesen. Zuweisung ist eine bestimmte Art, in den Speicher zu schreiben. Mit dem speziellen Auslesen des Speichers und dem speziellen Schreiben in den Speicher kann man ähnliche Effekte hervorrufen. Genau genommen

ist es so, dass beide Ansätze sowohl beim Schreiben als auch beim Lesen von üblichen Speichermechanismen der Objektorientierung abweichen.

Der größte Nachteil von Compound References besteht darin, dass der Ansatz ohne zusätzlichen Aufwand nicht typsicher ist. Das folgende Beispiel demonstriert dies. In diesem Codefragment wird eine Implementierung der vier Klassen `A`, `B`, `BSub`, `C` und `Other` in Abbildung 3.7 gezeigt.

```
1  class A {
2    B b;
3    void p(Other o) {
4      b<-m(o);
5    }
6    void setB(B b) {
7      this.b = b;
8    }
9  }
10 class B {
11   void m(Other o) {}
12 }
13 class BSub extends B {
14   C c;
15   void m(Other o) {
16     o.store(this<-c);
17   }
18 }
19 class Other {
20   C c;
21   store(C c) {
22     this.c = c;
23   }
24   void q() {
25     c.danger();
26   }
27 }
28 class C {
29   void danger() {...}
30 }
```

Es folgt die Betrachtung des Programmablaufs.

```
1 A a = new A();
2 B b = new B();
3 BSub bs = new BSub();
4 Other o = new Other;
5 a.setB(bs);
6 a.p(o);
```

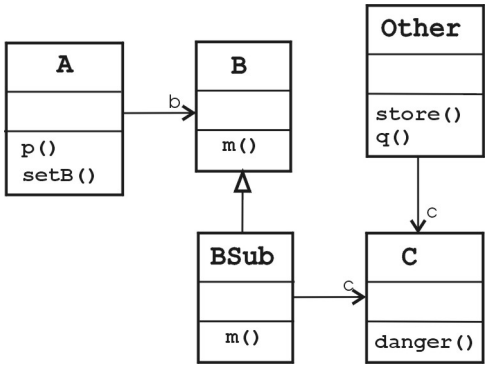

Abbildung 3.7: Verletzung der Typsicherheit

```
7 a.setB(b);
8 o.q();
```

Die ersten vier Zeilen initialisieren die Variablen. Alle folgenden Methodenaufrufe sind statisch gesehen korrekt. Sowohl der Aufruf

```
a.setB(bs);
```

als auch

```
a.setB(b);
```

sind gültig, denn `b` ist vom Typ `B`, und `bs` vom Typ `BSub`, welcher von `B` erbt. Es handelt sich also bei `a.setB(bs)` um einen Upcast. Der Aufruf

```
a.p(o);
```

in Zeile 6 führt dazu, dass die Methode `m(o)` in der Klasse `BSub` ausgeführt wird. Dadurch wird das Attribut `c` von `BSub` in `o` gespeichert. Genauer gesagt, in `o` wird eine Compound Reference `a<-b<-c` gespeichert. Zunächst wird also das Objekt `a` genommen, dann sein Attribut `b` und schließlich davon das Attribut `c`. In Zeile 7 wird in `a` das Attribut `b` neu gesetzt, und zwar mit einem Objekt, das kein Attribut `c` hat. Wenn nun

```
o.q();
```

aufgerufen wird, soll auf `a<-b<-c` die Methode `danger` aufgerufen werden. Diesmal besitzt `b` in `a` gar kein Attribut `c`, was zu einem Fehler führt. Der Laufzeitfehler findet bei

```
c.danger();
```

in Klasse `Other` statt. In dieser Zeile wird kein Typfehler erwartet, denn es gibt keinen Downcast. Das bedeutet die Verletzung der Typsicherheit und insbesondere, dass die Typüberprüfung durch den Compiler keine zuverlässigen Ergebnisse mit sich bringt.

In [Kni00] wird eine ähnliche Problematik behandelt. Dabei werden verschiedene Lösungen diskutiert. In [OM01] wird dem Problem dadurch begegnet, dass bei Zuweisungen mit Up- und Downcasts einer Compound Reference sowohl der neue als auch der alte

Wert der Variablen gespeichert wird. Wenn es, wie im vorgestellten Beispiel zu einer Situation kommen sollte, in der das entsprechende Attribut oder die entsprechende Methode nicht existiert, wird nicht der zuletzt gespeicherte Wert verwendet. Stattdessen wird einer der zwischengespeicherten Werte der Variablen benutzt, der das gewünschte Attribut bzw. Methode aufweist. Diese Lösung hat zwei Nachteile:

- Erstens werden bei Variablen viele zusätzliche Werte gespeichert, was in anderen Programmiersprachen nicht der Fall ist.
- Zweitens ist es schwer, dem Code zu entnehmen, welchen Wert eine Compound Reference liefert. Ein Programmierer sieht nicht unmittelbar, ob durch einen Up- oder Downcast die Typsicherheit verletzt wird, und die Variable einen unerwarteten Wert hat.

Compound References bieten einen interessanten Ansatz für Abhängigkeiten. Aufgrund der Verletzung der Typsicherheit ist er jedoch nicht praktikabel.

3.2.3 SuperGlue

In [MH06] wird ein Weg erarbeitet, Abhängigkeiten zwischen Variablen in verschiedenen Komponenten umzusetzen. Wie bei parametrisierten Zuweisungen wird erlaubt, dass Variablen voneinander über lange Zeit abhängig sind, daher der Name *SuperGlue*.

Betrachten wir die Funktionsweise von SuperGlue zunächst am Beispiel. Ein gutes Anwendungsszenario dafür ist der Model View Controller [MH06, KP88]. Der View soll erfahren, wenn im Model bestimmte Änderungen stattfinden. Ein Beispiel stammt aus dem Bereich des graphischen User Interface. Gegeben sind ein Editor und eine Tabelle; der Eintrag im Editor hängt davon ab, welche Zelle in der Tabelle ausgewählt wird. Wenn der Benutzer eine andere Zelle in der Tabelle, also dem Model, auswählt, soll der Editor, also der View, angepasst werden.

Ein weiteres Beispiel für Model View Controller ist ein Thermometer: Wenn der Sensor eine veränderte Temperatur misst, soll die Temperaturanzeige aktualisiert werden. Schauen wir uns die Umsetzung mit SuperGlue an.

```
1 atom Sensor {
2   export temperature : Int;
3 }
4 atom Label {
5   import text : String;
6   import color : Color;
7 }
```

Sowohl der Sensor, als auch die Anzeige sind als Klassen implementiert. Die Temperatur aus dem Sensor wird ausgelesen, und der Text und die Farbe in der Anzeige werden eingelesen.

Sehen wir uns nun das Verhalten eines laufenden Programms an.

```
1 let model = new Sensor;
2 let view = new Label;
3 view.text = "" + model.temperature + " C";
4 if (model.temperature > 30)
5   view.color = red;
6 else if (model.temperature < 0)
7   view.color = blue;
8 else
9   view.color = black;
```

In den ersten beiden Zeilen werden Objekte `model` und `view` erzeugt. Die Semantik der nächsten Zeilen 3 bis 9 ist nicht imperativer sondern rein deklarativer Natur. In Zeile 3 wird eine permanente Abhängigkeit zwischen `text` in `view` und `temperature` in `model` hergestellt. Dabei besteht der Text aus der Temperatur und der angehängten Zeichenkette `" C"`. Die Abhängigkeit zwischen der Temperatur und der Farbe der Anzeige ist komplexer. Wenn die Temperatur über 30 ist, ist die Farbe rot, unter 0 blau und ansonsten schwarz (siehe Zeilen 4 bis 9). Wenn die Temperatur sich ändert, wird die Anzeige aktualisiert und nicht umgekehrt.

Auf diese Weise kann man komplexe Abhängigkeiten zwischen zwei oder mehreren Variablen aufstellen. In diesem Zusammenhang spreche ich von funktionalen Abhängigkeiten [CD05a].

Es gibt wesentliche Unterschiede zu parametrisierten Zuweisungen:

- Die Interaktion zwischen zwei Variablen bei SuperGlue ist vielfältiger als bei parametrisierten Zuweisungen. Mein Ansatz beherrscht lediglich die Gleichheitsabhängigkeit, in SuperGlue können Variablen durch einen komplexen deklarativen Ausdruck verbunden werden.

- Bei SuperGlue werden Abhängigkeiten deklarativ formuliert. Diese gelten für die ganze Lebenszeit der Objekte. Parametrisierte Zuweisungen sind in diesem Punkt flexibler: Abhängigkeiten können u.a. verschieden stark sein, sie können hergestellt und wieder aufgelöst oder überschrieben werden.

- Bei SuperGlue werden die Variablen, bei denen Abhängigkeiten möglich sind, mit entsprechenden Schlüsselwörtern markiert: `export`, `import` oder `port`. Alle anderen Variablen können nicht voneinander abhängig sein. Auf diese Weise stellt SuperGlue sicher, dass der Missbrauch der Abhängigkeitsbeziehungen für nicht vorgesehene Fälle eingeschränkt wird. Parametrisierte Zuweisungen stellen keine Einschränkungen in dieser Hinsicht dar: Jede Variable kann an einer Abhängigkeitsbeziehung teilnehmen. Meiner Meinung nach sind die Sichtbarkeitsmodifier der Mechanismus, der den Zugriff auf Variablen und damit die Möglichkeit der Abhängigkeiten regelt. Man kann nicht alle Abhängigkeiten vorhersehen, so dass die SuperGlue Lösung in diesem Punkt eine unerwünschte Einschränkung darstellt.

- SuperGlue ermöglicht die Abhängigkeit zwischen zwei Variablen immer nur in eine Richtung: Sollte sich die erste Variable ändern, ändert sich die zweite. Der Fall

der eigenständigen Änderung der zweiten Variablen ist nicht vorgesehen. So ist eine Zuweisung zum Attribut `color` in `View` nicht möglich, da `color` importiert wird. Parametrisierte Zuweisungen erlauben dagegen, dass sowohl die erste Variable Einfluss auf die zweite hat als auch umgekehrt.

3.2.4 Immutable References

Im Gegensatz zu den bereits in diesem Unterkapitel vorgestellten Ansätzen weisen Immutable References die Analogie zu parametrisierten Zuweisungen nicht bei Abhängigkeiten, sondern bei Unabhängigkeiten auf. In [TE05] wird das Schlüsselwort `readonly` eingeführt. Es handelt sich dabei um einen Modifier, der dafür sorgt, dass Objekte nicht über die entsprechende Variable verändert werden, weder direkt noch über die Attribute dieses Objekts. Wenn z.B. ein Objekt von Variablen `x` und `y` referenziert wird und `x` dabei als `readonly` deklariert ist, sind weder Zuweisungen zu `x` noch zu `x.f` zulässig. Die Variable `y` ist von dieser Einschränkung nicht betroffen.

Zur Motivation dieser Vorgehensweise gebe ich hier zwei Beispiele wieder [TE05]. Die Methode `tabulate` soll Wahlergebnisse in tabellarischer Form darstellen. Die Änderung der Wahlergebnisse soll in dieser Methode ausgeschlossen werden. Dies ist in Java aufgrund des zwingenden call-by-reference nicht möglich. Eine Lösung dafür ist eine immutable Reference beim formalen Parameter:

```
ElectionResults tabulate(readonly Ballots votes) { ... }
```

Das folgende Codefragment stammt aus dem JDK 1.1.1 mit dem Unterschied, dass `readonly` im ursprünglichen Codefragment nicht vorhanden ist.

```
1 class Class {
2   private Object[] signers;
3   readonly Object[] getSigners() {
4     return signers;
5   }
6 }
```

In JDK führt das Fehlen von `readonly` zu einer Sicherheitslücke, denn das Attribut `signers` wird nach dem Aufruf der Methode von außerhalb der Methode referenziert. Dadurch kann die eigentlich geheime Variable von außen verändert werden. Das `readonly` Schlüsselwort erlaubt diese Veränderung nicht.

Wie diese Beispiele zeigen, geht es bei Immutable References in erster Linie darum, unerwünschte Änderungen zu vermeiden. Es soll nicht passieren, dass ein Objekt über eine Referenz geändert wird, und diese unerwünschte Änderung über andere Referenzen zum Vorschein kommt. Parametrisierte Zuweisungen lösen einen Teil dieser Anforderung durch das Kopieren des Wertes und nicht lediglich einer Referenz einer Variablen in die andere. Danach haben beide Variablen zwar den gleichen Wert, sind jedoch unabhängig. Der Wert einer Variablen kann nicht durch die andere Variable geändert werden. So können im ersten Beispiel die Wahlergebnisse call-by-value übergeben werden. Jegliche Änderungen der Variablen `votes` bleiben in diesem Fall ohne Folgen. Im zweiten Beispiel

kann die Methode `getSigner` eine Kopie von `signers` zurückgeben und den Wert einer unabhängigen Variablen zuweisen. Eine weitere Alternative zu Immutable References ist in [KT01] beschrieben.

3.2.5 Aspektorientierte Programmierung

Aspektorientierte Programmierung beschäftigt sich mit Crosscutting Concerns [TOHS99]. Dabei handelt es sich um ähnliche Funktionalität, die über viele Module gestreut ist. Aspektorientierte Programmierung bereinigt die Module von diesem zerstreuten und sich wiederholenden Code und kapselt ihn in Aspekte. Zur Laufzeit wird der Code aus Aspekten an richtigen Stellen eingewoben [KLM+97].

So ist es denkbar, dass Abhängigkeiten zwischen Variablen in Aspekte ausgelagert werden. Denn dabei handelt es sich um einen Crosscutting Concern. Sollte eine abhängige Variable sich ändern, kann durch einen Aspekt eine Änderung der anderen abhängigen Variablen ausgelöst werden. Bisher werden Crosscutting Concerns üblicherweise bei Methodenaufrufen und nicht bei Variablenänderungen eingewoben [Voß05, Bau05, KM05a, KM05b]. Die Verfolgung von Variablenänderungen ist im aspektorientierten Sinne: So wird z.B. in [APS+05] das Observer Design Pattern mit aspektorientierten Mitteln umgesetzt.

3.3 Abhängigkeitsbegriff in der Literatur

Abhängigkeit und Dependency sind abstrakte Begriffe. Es gibt viele wissenschaftliche Artikel, die darüber schreiben, aber keinen direkten Bezug zu Abhängigkeiten haben, wie sie in dieser Arbeit eingeführt werden. Der Zweck dieses Unterkapitels besteht darin, die verschiedenen Definitionen in der Literatur von meiner Abhängigkeitsdefinition abzugrenzen.

In [KMP95] wird eine Beziehung zwischen Objekten eingeführt, so dass ein Objekt nicht ohne ein anderes Objekt existieren kann. Dieses Phänomen wird *Lifetime Dependency* genannt. Eine Rolle kann z.B. nur so lange existieren wie die dazu gehörende Entität; manche Teile können nicht ohne ihr zugehöriges Ganzes existieren.

Die Autoren von [JDAO04] beschäftigen sich mit *Dependent Types*. Dabei werden Typen unabhängig von dem statischen Kontext zueinander in Beziehung gesetzt. Zur Verdeutlichung dient folgendes Beispiel: Die Klasse `ColoredNode` erbt von `Node`. `Node` enthält die Methode

```
void connect(Node n) {...}
```

Dabei soll sichergestellt werden, dass die Methode von `Node` nur mit dem Parameter des dynamischen Typs `Node` aufgerufen wird und die Methode von `ColoredNode` nur mit dem Parameter des Typs `ColoredNode`. In statisch typisierten Programmiersprachen ist das nicht möglich. Bei `Node` und `ColoredNode` handelt es sich um Dependent Types.

Dependency Injection ist ein Design Pattern, auch bekannt unter dem Namen *Inversion of Control*. Dabei werden Typen als abhängig bezeichnet, wenn ein Typ ein Attribut des anderen Typs hat [Fow04].

Ein weiteres aktuelles Forschungsgebiet sind Abhängigkeiten zwischen Modulen. Das bedeutet, dass ein Modul Komponenten des Anderen verwendet [Par78]. Um solche Komponenten aufzuzeigen, wird u.a. *Dependency Structure Matrix* verwendet [SJSJ05]. Sogar die Definitionen bei Abhängigkeiten zwischen Modulen variieren [Jac02].

Auch bei Datenbanken kommt der Abhängigketsbegriff vor. *Functional Dependency* bedeutet, dass ein Schlüssel den Rest eines Eintrags in einer Tabelle bestimmt [Bee80].

4 Motivation

Dieses Kapitel macht deutlich, wie verbreitet Abhängigkeiten in existierender Software sind und dass Bedarf an weiteren Untersuchungen und Verbesserungen besteht. Ich stelle verschiedene Szenarien und ihre gegenwärtig möglichen Umsetzungen vor. In allen diesen Szenarien spielen Abhängigkeiten eine entscheidende Rolle. Es wird gezeigt, dass die Lösungen in mancher Hinsicht nicht befriedigend sind. Das motiviert weitere Ansätze und Strategien, die in den folgenden Kapiteln erarbeitet werden.

Die präsentierten Szenarien sind aus sehr unterschiedlichen Bereichen gewählt. Zunächst werden in Unterkapitel 4.1 Zugehörigkeiten vorgestellt. Das nächste Unterkapitel beschäftigt sich mit den in der Literatur bereits gründlich untersuchten Parameterübergabemechanismen. Dabei stehen die Abhängigkeiten zwischen formalen und aktuellen Parametern im Vordergrund. Unterkapitel 4.3 geht auf Abhängigkeiten im Zusammenhang mit Rollen, Perspektiven und Gesichtspunkten ein. Im nächsten Unterkapitel werden weitere Anforderungen an Ansätze für Abhängigkeiten angerissen. In der Schlussfolgerung begründe ich, warum weitere Auseinandersetzungen mit Abhängigkeiten notwendig sind.

4.1 Zugehörigkeiten

Dieses Unterkapitel beschäftigt sich mit folgendem Phänomen: dem Zugriff auf ein Attribut eines Objekts außerhalb dieses Objekts. Problemstellungen mit dieser Anforderung werden *Zugehörigkeiten* genannt. Sie treten nämlich dann auf, wenn ein Attribut benötigt wird, das *zu* einem Objekt *gehört*, aber das Objekt selbst und dessen andere Attribute nicht betrachtet werden müssen. Beispielsweise sei `dogField` ein Attribut des Objekts `karlHeinz`. Von Interesse ist der Hund, der aber zu Karl-Heinz gehört. Wechselt der Hund von Karl-Heinz, so ist der neue Hund von Interesse; die Zugehörigkeit des Hundes zu Karl-Heinz ist also wichtig. Beispiele für Zugehörigkeiten sind „der Hund von Karl-Heinz“, „Jacks Konto“ oder „der Mann der Englischlehrerin von Sonja“. Dieser Zugriff kann über eine Variable erfolgen, die den gleichen Wert besitzt wie das Attribut. Dabei befindet sich diese Variable im Gegensatz zum Attribut außerhalb des Objekts.

An dem folgenden Codebeispiel möchte ich den Sachverhalt verdeutlichen.

```
1 karlHeinz.dogField = v1;
2 dogVariable =f karlHeinz.dogField;
3 karlHeinz.dogField = v2;
```

`dogVariable` soll die Variable außerhalb des Objekts `karlHeinz` darstellen, und `dogField` ist ein Attribut innerhalb des Objekts. v_1 und v_2 sind zwei Werte, die von den Variablen `dogField` und `dogVariable` angenommen werden können. Nach dem Erstellen der Abhängigkeit über $=_f$ zwischen `dogField` und `dogVariable` in Zeile

2 soll `dogVariable` immer den gleichen Wert annehmen wie `dogField`. Damit hat `dogVariable` vorerst den Wert v_1, da `karlHeinz.dogField` diesen Wert hat. Nach der Änderung von `dogField` in Zeile 3 nimmt auch die Variable `dogVariable` den neuen Wert v_2 an, obwohl `dogVariable` an dieser Stelle nicht explizit erwähnt wird. Es besteht also wegen der andauernden Wertgleichheit eine Abhängigkeit zwischen dem Attribut `dogField` und der Variablen `dogVariable`. Dieses Phänomen wird durch die übliche Zuweisungssemantik von j-Referenzen in Zeile 2 nicht erreicht.

4.1.1 Beispiel Konto

Dieser Abschnitt demonstriert, dass Zugehörigkeiten in realen Designs vorkommen. In [OM01] wird ein Szenario mit Abwicklung von Überweisungen eingeführt (siehe Abschnitt 3.2.2). Ich wandle das Beispiel leicht ab, um es mehr an die Realität anzupassen.

Eine Bank habe zwei Arten von Kunden, nämlich Personen und Unternehmen. Betrachtet werden Konten und Daueraufträge. Es gibt zwei Arten von Konten: Kontoart 1 ermöglicht höhere Zinsen, Kontoart 2 günstigere Überweisungen. Ein Unternehmen kann mehrere Konten haben und seine Daueraufträge über verschiedene Konten abwickeln. Eine Person kann ebenfalls mehrere Konten besitzen, jedoch kann sie nur ein Konto für Daueraufträge nutzen. Weiterhin kann ein Konto mehrere Eigentümer haben, z.B. eine Person und ein Unternehmen.

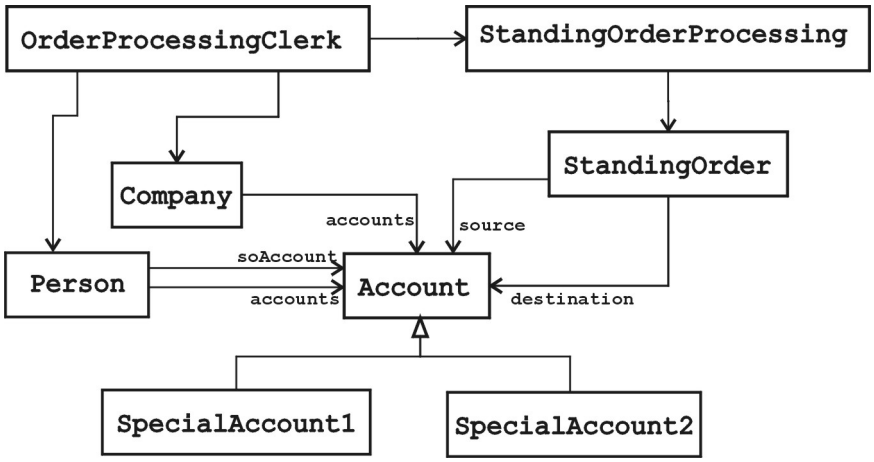

Abbildung 4.1: Klassendiagramm für das Kontobeispiel

Ein intuitives und einfaches Design für dieses Beispiel ist in Abbildung 4.1 dargestellt. Das Attribut `soAccount` in `Person` bezeichnet das Konto für Daueraufträge der jeweiligen Person. Der `orderProcessingClerk` bekommt die Kontoobjekte von der betreffenden Person bzw. dem Unternehmen und die Anweisung, einen Dauerauftrag zu erzeugen. `orderProcessingClerk` weist die Dauerauftragsausführung `standingOrderProcessing` an, den Dauerauftrag zu erzeugen und zu registrieren. Nun entsteht ein neues Objekt der Klasse `StandingOrder`, welches dann mit `standingOrderProcessing` registriert wird (siehe Abbildung 4.2).

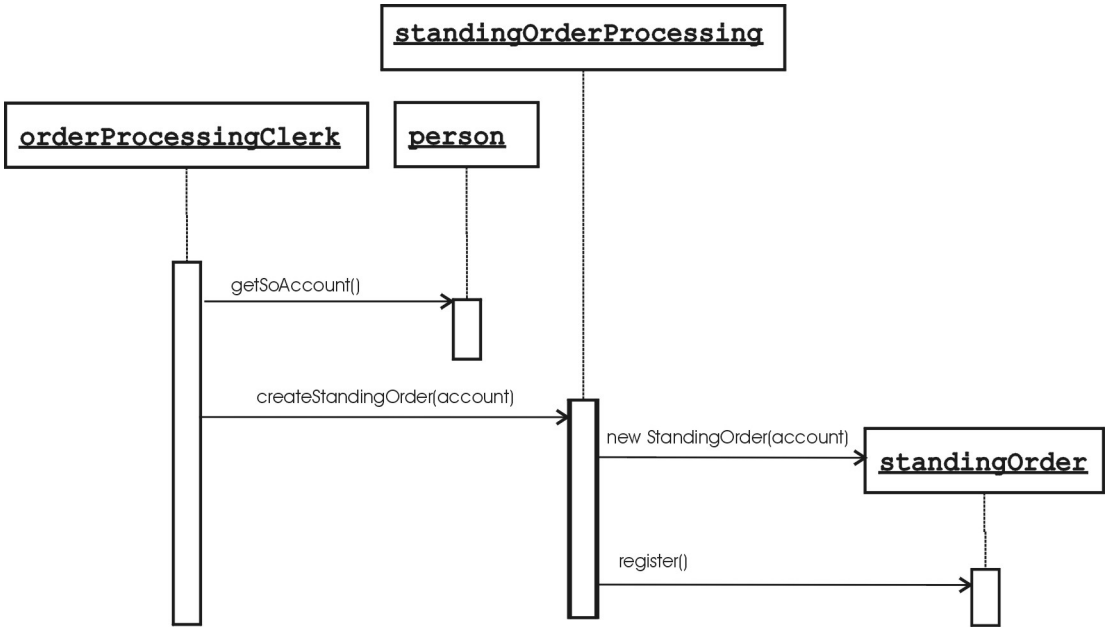

Abbildung 4.2: Sequenzdiagramm für das Kontobeispiel

Die Bank habe ein neues Angebot entwickelt. Es sei nun möglich, dass eine Person ihr Konto für Daueraufträge leicht wechseln kann, z.B. werden Daueraufträge zunächst vom Konto der Kontoart 2 ausgeführt, weil sie hier günstiger sind. Das Dauerauftragskonto kann nun automatisch wechseln, wenn kein Geld mehr auf diesem Konto ist. Dieses Angebot führt dazu, dass das Attribut `soAccount` für Dauerauftragskonto in `Person` häufig einen neuen Wert annimmt.

Mit dem gegebenen Design entsteht nun folgendes Problem: Wenn sich das Konto einer Person ändert, wird ein vorher registrierter Dauerauftrag weiterhin von dem ehemaligen Konto ausgeführt. Die Veränderung des Kontos im bestehenden Dauerauftrag kann beim Kontowechsel einer Person nicht direkt erfolgen. Das erkennt man im Klassendiagramm (siehe Abbildung 4.1) daran, dass es keine Verbindung zwischen der Klasse `Person` und der Klasse `StandingOrder` bzw. der Klasse `StandingOrderProcessing` gibt. `person` ist vorerst das einzige Objekt, das den Kontowechsel wahrnimmt. Sie kann den bestehenden `standingOrder` nicht direkt davon in Kenntnis setzen, da die beiden Objekte sich nicht kennen. Die Kommunikation muss über andere Objekte ablaufen: über `orderProcessingClerk` und über `standingOrderProcessing`. `orderProcessingClerk` muss `person` nach dem neuen `account` fragen und dieses dem `standingOrderProcessing` mitteilen, das seinerseits `standingOrder` aktualisiert. Dabei entsteht das Problem, dass `orderProcessingClerk` nicht weiß, wann er `person` nach dem neuen `account` fragen soll, denn `person` kann den Zeitpunkt der Änderung dem `orderProcessingClerk` nicht mitteilen, da die Assoziation zwischen diesen Objekten nur in einer Richtung verläuft. Dieser umständliche Prozess der Aktualisierung des Dauerauftrags ist nicht erwünscht.

Wenn es eine Referenz `person` zu einem Objekt der Klasse `Person` gibt, ist „`person`'s Dauerauftragskonto“ von Interesse. Es ist ein Sprachkonstrukt erwünscht, das folgende zusammengesetzte Referenz darstellt: Sie zeigt über `person` auf das Konto und wird nach `StandingOrderProcessing` weitergeleitet. Damit ergibt sich eine Abhängigkeit zwischen der Variablen, die das Konto in `StandingOrderProcessing` repräsentiert, und dem Attribut `soAccount` in der Klasse `Person`. Mit dieser Abhängigkeit ist es nicht mehr nötig, die Kontoreferenzen per Hand zu verändern.

4.1.2 Lösungsansätze

Das im vorherigen Abschnitt gewünschte sprachliche Mittel existiert in gängigen Programmiersprachen nicht. Also muss das Problem auf eine andere Art und Weise gelöst werden. Folgende bestehende Ansätze können zur Lösung beitragen.

Ein möglicher Lösungsansatz ist der Einsatz des Design Patterns Decorator (siehe Abschnitt 3.1.3). Decorator soll ein Kontoobjekt enthalten und alle Methodenaufrufe an das Objekt weiterleiten. Der Decorator und nicht das Konto selbst soll der Klasse `StandingOrderProcessing` bekannt sein. Das Hauptobjekt des Decorators, also ein Objekt der Klasse `Account`, kann ohne weitere manuelle Updates geändert werden.

Dieser Ansatz hat folgende Nachteile:

- Das Hinzufügen von Hilfsklassen ist eine für die Entwickler monotone und fehleranfällige Aufgabe.
- Es besteht das Syntactic Fragile Base Class Problem [Szy98]: Wenn man die Methodensignaturen in der Klasse `Account` ändert, muss die Klasse `Decorator` angepasst werden.
- Wenn zwei Personen Eigentümer des gleichen Kontos sind, vergleicht `StandingOrderProcessing` verschiedene Decoratoren mit dem gleichen Hauptobjekt. Also kann man beim Vergleich der Decorateren nicht feststellen, ob es sich um das gleiche Hauptobjekt handelt.
- Es ist keine transparente Weiterleitung gegeben. Der Aufruf der Methoden an `this` in `Account` betrifft nur die lokalen Methoden und nicht die Methoden der kapselnden Klasse `Decorator`. Dieser Sachverhalt ist als Self Problem [Lie86] oder Broken Delegation [HOT97] bekannt.
- Alle Decoratoren vererben ihren Zustand. Auf diese Weise entsteht Redundanz und die Klassen werden heavy (siehe Abschnitt 3.1.3) [OM01].

Im zweiten Ansatz wird ein neues Interface `AccountOwner` eingefügt. Es soll sowohl `Person` als auch `Company` als Unterklassen haben. Dieses Interface verfügt über eine einzige Methode `getAccount`. Die Klasse `StandingOrderProcessing` wird so geändert, dass sie das Interface `AccountOwner` statt der Klasse `Account` referenziert. In diesem Fall müssen die Konten für Daueraufträge nicht aktualisiert werden, da Kontoinhaber und nicht Konten von `StandingOrderProcessing` referenziert werden. Die

Einführung des Interfaces ist im Fall von `Company` kompliziert, da ein Unternehmen verschiedene Konten für Daueraufträge haben kann. Deswegen ist eine zusätzliche Unterklasse von `AccountOwner` nötig, die das Unternehmen repräsentiert. Dieser Ansatz hat ebenfalls Nachteile:

- Diese Lösung ist wegen zusätzlicher Klassen komplex.
- Die Klasse `StandingOrderProcessing` muss geändert werden. Das ist vielleicht nicht gewünscht oder gar nicht möglich, da es sich um eine Komponente einer Klassenbibliothek für Banken handeln kann.
- Dieser Ansatz hat die Einschränkung, dass nur einfach zusammengesetzte Referenzen, wie z.B. Jacks Konto, gelöst werden. Wenn mehrfach zusammengesetzte Referenzen erwünscht sind, wie z.B. Konto von Jacks Frau, müsste das Design komplett überarbeitet werden.

Eine dritte Möglichkeit besteht darin, die Klasse `StandingOrderProcessing` als Observer von Personen und Unternehmen einzurichten (siehe Abschnitt 3.1.4). `StandingOrderProcessing` wird bei einem Kontowechsel von der Klasse `Person` benachrichtigt. Daraufhin muss `StandingOrderProcessing` die Konten für alle bestehenden Daueraufträge aktualisieren.

Die Bindung zwischen `Person` und `StandingOrderProcessing` kann man hauptsächlich auf zwei Weisen herstellen. Bei der schnittstellenorientierten Vorgehensweise muss es zwei neue Interfaces `Subject` für `Person` und `Observer` für `StandingOrderProcessing` geben. Bei einer leichten Änderung der Anforderungen, z.B. wenn `StandingOrderProcessing` noch mehr Klassen beobachten soll, muss `StandingOrderProcessing` jeweils ein neues Interface implementieren. Bei einer schnellen und nicht so eleganten Lösung referenzieren sich die Klassen `Person` und `StandingOrderProcessing` gegenseitig. Dazu müssen sie entsprechend verändert werden. Die Observerlösung ist somit ebenfalls sehr kompliziert.

Weiterhin kann ein Delegate (siehe Abschnitt 3.2.1) erzeugt werden, mit dessen Hilfe eine Methode der Klasse `StandingOrderProcessing` bei `Person` registriert wird. Sollte das Konto einer Person geändert werden, wird ein Event aufgerufen, und somit eine entsprechende Methode in `StandingOrderProcessing` ausgeführt, die alle Daueraufträge aktualisiert. Dies ist zwar eine Vereinfachung im Vergleich zu Observer; die Umsetzung bleibt jedoch komplex.

Die betrachteten Design Patterns und Delegate weisen zwei grundlegende Probleme auf. Erstens werden sie sehr speziellen Anforderungen oft nicht gerecht, da sie nur Lösungen für Standardprobleme liefern. Bei vielen Anforderungen an Flexibilität erweisen sie sich als zu starr. Zweitens verkomplizieren sie das Design durch zusätzliche Klassen und deren Beziehungen. Damit sind die Design Patterns und Delegate für die Programmierung von Zugehörigkeiten mit ihrer speziellen Abhängigkeitsstruktur weniger geeignet. Es ist gewünscht, dass bestimmte Variablen andere Variablen *referenzieren*. Es gilt also, Sprachmittel zu finden, die Zugehörigkeiten bewältigen. Ein solcher Versuch ist der Ansatz Compound References. Dieses Sprachkonstrukt wird ausführlich u.a. am Kontobeispiel in Abschnitt 3.2.2 diskutiert. Compound References haben einen entscheidenden

Nachteil: Um die Typsicherheit zu gewährleisten, muss bei jeder Zuweisung einer Compound Reference sowohl der alte als auch der neue Wert gespeichert werden. Dies führt zur großen Redundanz. Außerdem ist es nicht transparent, welcher der beiden Werte in einem bestimmten Kontext durch die Referenz geliefert wird. Weiterhin existiert dieses syntaktische Konstrukt in keiner der geläufigen Programmiersprachen.

SuperGlue bietet für das Kontoszenario eine akzeptable Lösung (siehe Abschnitt 3.2.3). In der SuperGlue Syntax hätte die Klasse `StandingOrder` folgendes Aussehen:

```
1 atom StandingOrder {
2   import source : Account;
3   ...
4 }
```

Während `StandingOrder` das Absenderkonto importiert, exportiert `Person` das eigene Konto.

```
1 atom Person {
2   export soAccount : Account;
3   ...
4 }
```

Im folgenden Programmsegment wird `source` von `standingOrder` auf `soAccount` von `person` gesetzt.

```
1 let standingOrder = new StandingOrder;
2 let person = new Person;
3 let anAccount = new Account;
4 let anotherAccount = new Account;
5 standingOrder.source = person.soAccount;
6 let person.soAccount = anAccount;
7 let person.soAccount = anotherAccount;
```

Dabei ist die Zeile 5 deklarativ zu verstehen. Die Abhängigkeit zwischen `source` in `standingOrder` und `account` in `Person` besteht für die restliche Laufzeit des Programms: Sowohl in Zeile 6 als auch in Zeile 7 sind die Konten von `standingOrder` und `person` gleich.

Durch SuperGlue kann eine Zugehörigkeit mit einer zusätzlichen Codezeile ausgedrückt werden. Diese Lösung ist kurz und verständlich. Eine Übersichtstabelle aller besprochenen Ansätze im Vergleich ist in Abschnitt 6.2 dargestellt.

In den nächsten Kapiteln werden Zugehörigkeiten als ein Fall von Abhängigkeiten behandelt, denn bei Zugehörigkeiten geht es um Variablen, die gleichheitsabhängig sind. Die eingeführten Mittel für Abhängigkeiten eignen sich somit auch für Zugehörigkeiten.

4.2 Parameterübergabe

In diesem Unterkapitel werden Parameterübergabevariationen aufbauend auf dem Abschnitt 2.2.7 untersucht. Ich zeige, dass es sich hierbei wieder um Abhängigkeiten handelt.

Wie bereits in Abschnitt 2.2.7 erläutert, gehe ich ohne Beschränkung der Allgemeinheit davon aus, dass aktuelle Parameter Variablen sind. Am Anfang der Ausführung des Methodenrumpfes haben sowohl der aktuelle als auch der formale Parameter unabhängig von der Parameterübergabevariation den gleichen Wert. Gemäß der Parameterübergabevariation kann sich dies im Verlauf der Methode ändern. Es gibt verschiedene Stufen der Abhängigkeit zwischen formalem und aktuellem Parameter: von nicht abhängig bis derart abhängig, dass formaler und aktueller Parameter während der ganzen Methodenlaufzeit den gleichen Wert haben. Ich unterscheide zwischen *starkem* und *schwachem* call-by-reference und call-by-value. Bei der Parameterübergabe findet eine implizite Zuweisung zwischen dem formalen und dem aktuellen Parameter statt. Im Fall des call-by-value basiert diese Zuweisung auf einer Wertkopie, im Fall des schwachen call-by-reference auf j-Referenzen und im Fall des starken call-by-reference auf c-Referenzen.

Bei den Code-Beispielen verwende ich meistens Java-Syntax. Dies bedeutet jedoch nicht, dass Java-Semantik gegeben ist. Die Semantik der Codeblöcke variiert mit den betrachteten Parameterübergabemechanismen.

4.2.1 Variationen

In diesem Abschnitt wird verdeutlicht, dass alle drei Parameterübergabevariationen ihre Berechtigung haben.

Beim schwachen call-by-reference gilt für die Abhängigkeit von aktuellem und formalen Parameter Folgendes: Wenn der Wert eines Parameters durch den Aufruf einer seiner Methoden oder durch eine Zuweisung zu einem Attribut verändert wird, wird der Wert des anderen Parameters genauso verändert. Wenn einem der Parameter ein Wert *zugewiesen* wird, wird der Gegenspieler nicht verändert, und es besteht keine weitere Abhängigkeit.

Beim starken call-by-reference besteht kein Unterschied zum schwachen in Bezug auf Methodenaufrufe bzw. Attributveränderungen. Im Gegensatz zum schwachen call-by-reference bekommt bei einer Zuweisung eines Wertes zu einem der Parameter auch der andere Parameter den neuen Wert.

Listing 4.1: Methodenaufruf auf dem Parameter

```
1 public void changeByCallingMethod(MyClass o) {
2   o.changeState();
3 }
```

Listings 4.1 und 4.2 verdeutlichen diesen Sachverhalt. Die Beispielmethode in Listing 4.1 enthält einen Methodenaufruf auf dem Parameter; die Beispielmethode in Listing 4.2 beinhaltet eine Zuweisung. Im ersten Beispiel hängt die Semantik nicht von der call-by-reference Variation ab: Nach der Methodenausführung hat der aktuelle Parameter einen neuen Wert. Im zweiten Codebeispiel ist die Situation anders. Im Fall des schwachen call-by-reference wird der formale Parameter bei der Zuweisung vom aktuellen entkoppelt, so dass der aktuelle Parameter nicht verändert wird; bei dem starken call-by-reference

haben der aktuelle und der formale Parameter auch nach der Zuweisung den gleichen Wert, so dass der aktuelle Parameter verändert wird.

Listing 4.2: Zuweisung zu einem Parameter

```
1 public void changeByAssignment(MyClass o) {
2   o = ...;
3 }
```

Wenn diese Methoden z.B. in Java mit dem da einzig möglichen schwachen call-by-reference aufgerufen werden, wird der aktuelle Parameter im ersten Beispiel geändert, im zweiten jedoch nicht. Im nachfolgenden Beispiel wird demonstriert, dass die Änderung des aktuellen Parameters auch bei der Zuweisung sinnvoll sein kann. Starker call-by-reference ist also berechtigt.

```
1 public void exchangeIntValues(int x, int y) {
2   int z = x;
3   x = y;
4   y = z;
5 }
```

Mit Hilfe dieser Methode sollen die Werte der aktuellen Parameter vertauscht werden. Dieser Code würde in Java nicht das gewünschte Ergebnis erzeugen, sondern nur in einer Sprache mit starkem call-by-reference.

Ein weiteres Thema, das hier behandelt werden soll, handelt von Rückgabewerten. Manchmal bestehen die Anforderungen darin, dass eine Methode mehrere Rückgabewerte liefert. Ein Beispiel dafür ist die Methode zum Lösen einer quadratischen Gleichung. Da eine quadratische Gleichung zwei Lösungen haben kann, ist es gewünscht, dass zwei Ergebnisse zurückgegeben werden. Das hat auf den ersten Blick nichts mit Abhängigkeiten zu tun. Aber in Abschnitt 4.2.2 wird gezeigt, dass zusätzliche Rückgabewerte mit Hilfe von call-by-reference abgedeckt werden, was ein Spezialfall von Abhängigkeiten ist.

Bei call-by-value hingegen ist es so, dass die Veränderung des formalen Parameters keine Auswirkung auf den aktuellen Parameter hat. Das nächste Beispiel stellt eine Methode dar, in der keine Veränderungen der Parameter stattfinden dürfen, die Parameter sind also readonly. Betrachten wir in Java in der Klasse `PrintStream` die Methode:

```
public void print(Object o) {...}
```

Diese Methode sollte das Objekt nicht verändern, da sie nur Informationen darüber ausgibt. Da die Methode aus der Standardbibliothek von Java [Sun07a] gut getestet sind, kann man davon ausgehen, dass das Objekt `o` hier nicht verändert wird. Wenn eine ähnliche Aufgabe in einem kleineren Softwareprojekt erledigt wird, können dem Programmierer von einer Methode `print` mit ähnlicher Funktionalität in dieser Hinsicht Fehler unterlaufen (siehe auch das Beispiel in Abschnitt 3.2.3). Man möchte das entsprechende Objekt by-value übergeben, um die Möglichkeit einer Veränderung des aktuellen

Parameters auszuschließen. Es geht also um die Unabhängigkeit zwischen formalen und aktuellen Parametern.

Parameterübergabemechanismus	Abhängigkeit beim Methodenaufruf	Abhängigkeit bei einer Zuweisung
starker call-by-reference	ja	ja
schwacher call-by-reference	ja	nein
call-by-value	nein	nein

Tabelle 4.1: Abhängigkeit zwischen aktuellen und formalen Parametern

Wie gezeigt, sind alle drei Parameterübergabevariationen sinnvoll. Tabelle 4.1 demonstriert die Abhängigkeiten bei den Parameterübergabemechanismen. Die erste Spalte enthält den Namen der Mechanismen. Die zweite Spalte sagt aus, ob die Abhängigkeit nach einem Methodenaufruf weiterhin besteht (siehe Listing 4.1). In der dritten Spalte geht es nicht um einen Methodenaufruf, sondern um eine Zuweisung (siehe Listing 4.2). Man erkennt, dass alle Abhängigkeitsstufen durch die drei Parameterübergabevariationen abgedeckt werden.

4.2.2 Umsetzungen

Die Umsetzungen für die Parameterübergabevariationen sind im höchsten Maße programmiersprachenabhängig. Hier bespreche ich die bekannten Umsetzungen des starken call-by-reference und des call-by-value. Auf den schwachen call-by-reference gehe ich nicht ein, da er in fast jeder Programmiersprache zur Grundausstattung gehört.

Schauen wir uns zunächst call-by-reference am Beispiel der Methode

```
public void exchangeIntValues(int x, int y) {...}
```

an. In Java gibt es keinen Weg, das gewünschte Resultat direkt zu erzielen. Auch das implizite Wechseln vom primitiven Datentyp `int` zur Klasse `Integer` oder umgekehrt, das ab Java 5.0 [Sun07b] möglich ist, ergibt nicht den gewünschten starken call-by-reference. Die gewünschte Semantik kann mit Verwendung des Design Patterns Decorator oder des einfacheren Design Pattern Adapter, auch Wrapper genannt, erzielt werden [GHJV97] (siehe Abschnitt 3.1.3). Man erzeugt eine neue Klasse, die ein Attribut vom Typ `int` enthält.

```
1 class Adapter {
2   public int number;
3 }
```

Aus Gründen der Einfachheit verwende ich an dieser Stelle ein öffentliches Attribut. Eleganter wäre es, ein geheimes Attribut zu deklarieren und auf dieses mittels öffentlicher Methoden zuzugreifen. Objekte der Klasse `Adapter` sollen an die Methode übergeben werden. Die Zuweisungen richten sich an die Attribute dieser Objekte und nicht direkt an das Objekt selbst:

```
1 public void exchangeIntValues(Adapter ax, Adapter ay) {
2   int z = ax.number;
3   ax.number = ay.number;
4   ay.number = z;
5 }
```

Der Code mit dem Methodenaufruf sieht so aus:

```
1 Adapter a1 = new Adapter(x);
2 Adapter a2 = new Adapter(y);
3 exchangeIntValues(a1, a2);
4 x = a1.number;
5 y = a2.number;
```

Leider ist diese Lösung umständlich und nicht intuitiv.

Im Gegensatz zu Java ist der starke call-by-reference in C# möglich. Dies geschieht mittels des Schlüsselworts `ref` (siehe Abschnitt 2.2.7):

```
void exchangeIntValues(ref int x, ref int y){...}
```

Der Lösungsweg über Schlüsselwörter darf nicht verallgemeinert werden, da die Lösung dem Minimalitätsprinzip widerspricht. Es ist nicht erwünscht, für viele spezielle Anforderungen neue Schlüsselwörter der Programmiersprache hinzuzufügen [Str97].

In C++ kann die Aufgabe mit c-Referenzen gelöst werden (siehe Abschnitt 2.2.6):

```
void exchangeIntValues(int& x, int& y){...}
```

Diese Lösung erfordert den Einsatz von Zeigern. Die Vor- und Nachteile werden in Abschnitt 2.2.2 diskutiert.

Kommen wir nun zu dem Fall, dass eine Methode mehrere Rückgabewerte hat. Wir betrachten als Beispiel eine Methode zur Lösung einer quadratischen Gleichung mit drei Parametern und zwei Rückgabewerten.

Die eleganteste Lösung bietet meiner Meinung nach Python [Wei03]. Dadurch, dass Listen und Tupel in Python zur Grundausstattung gehören, können mehrere Rückgabewerte zurückgegeben werden. `(x,y)` stellt ein Tupel aus den Variablen `x` und `y` dar. Durch den Aufruf der Methode

```
(x, y) = solveQuadraticEquation(a,b,c)
```

wird dem Tupel aus den Variablen `x` und `y` das Tupel der Rückgabewerte zugeordnet. `x` wird der erste Wert im Tupel zugewiesen, `y` der zweite.

C# bietet wieder eine Schlüsselwortlösung. Das Schlüsselwort `out` simuliert einen zusätzlichen Rückgabewert:

```
void solveQuadraticEquation(double a, double b, double c,
    out double x1, out double x2) {...}
```

Die Einführung eines neuen Schlüsselworts verletzt genau wie im `ref`-Fall das Minimalitätsprinzip.

Auch in C++ maskiert man üblicherweise Rückgabewerte als Parameter durch c-Referenzen und geht wie in C# vor.

In Java kann man mit den gegebenen Sprachmitteln nicht direkt mehrere Rückgabewerte behandeln. Ein Ausweg wäre wieder, Hilfsklassen entsprechend dem beschriebenen Adapter Design Pattern zu erzeugen, die als Attribute die Rückgabewerte enthalten.

Die letzte hier zu behandelnde Variation ist call-by-value. Betrachten wir die Methode `print` aus dem letzten Abschnitt. Das Festlegen des Parameterübergabemechanismus ist in Java nur durch die Typen der Parameter möglich. Primitive Datentypen sind Werttypen und deren Instanzen werden by-value übergeben; Klassen sind dagegen Referenztypen, so dass Objekte by-reference übergeben werden. Es ist nicht möglich, das Festlegen des Parameterübergabemechanismus beim Definieren oder beim Aufruf der Methode zu verändern. Eine Idee, um Objekte by-value zu übergeben, wäre, sie vor dem Aufruf zu clonen:

```
print(myObject.clone());
```

Dies geht nur in dem Fall, wenn die Klasse von `myObject` das Interface `Cloneable` implementiert. Viele Klassen sowohl in den Enterprise Anwendungen als auch in der Standard Java Bibliothek [Sun07a] implementieren `Cloneable` nicht, so dass diese Lösung nicht immer angewandt werden kann.

In C# ist die Situation ähnlich. Es gibt Referenz- und Werttypen (siehe Abschnitt 2.2.3). Klassen gehören zu den ersten und primitive Datentypen, Structs und Enumerations zu den letzten. Man muss sich also schon während des Schreibens des Konstrukts `class` oder `struct` überlegen, ob man deren Objekte später by-reference oder by-value übergeben möchte. Während des Methodenaufrufs oder der Methodendeklaration ist es nicht mehr regulierbar. Wie die Beispiele gezeigt haben, ist es erwünscht, den Parameterübergabemechanismus gerade bei der Methoden- und nicht bei der Klassendeklaration zu bestimmen.

In C++ löst man die Aufgabe wieder durch die geeignete Verwendung von Zeigern, in dem sie dereferenziert werden.

Für das gegebene Beispiel eignet sich das Konstrukt mit dem Schlüsselwort `readonly` von Immutable References am besten (siehe Abschnitt 3.2.4).

```
1 public void print(readonly Object o) {
2   ...
3   o.f1.f2 = x; // Compilerfehler
4 }
```

Dabei kann über den formalen Parameter `o` weder das Objekt selbst verändert werden, noch seine Attribute, noch die Attribute der Attribute, usw. Dagegen erzeugt call-by-value lediglich eine flache Wertkopie. Wenn eine Zuweisung Attribute von Attributen verändert (siehe Zeile 3), betrifft die Änderung sowohl den formalen als auch den aktuellen Parameter:

```
1 public void print(Object o) {
2   ...
3   o.f1.f2 = x;
```

```
4 }
```

Wie in diesem Abschnitt gezeigt, existiert bei den so gut erforschten und jederzeit gebrauchten Parameterübergabemechanismen keine einheitliche Lösung. Dabei bewältigen Zeiger die Variationen bei der Parameterübergabe am besten. Jedoch versuchen die Entwickler der neueren Programmiersprachen, auf Zeiger wegen der Fehleranfälligkeit beim Umgang mit diesen zu verzichten. Dies führt dazu, dass die Anforderungen an die Parameterübergabe entweder umständlich oder gar nicht gelöst werden oder man der Programmiersprache neue Schlüsselwörter hinzufügen muss.

Es stellt sich heraus, dass der Parameterübergabemechanismus oft von den Parametertypen festgelegt wird. Daher ist es, obwohl erwünscht, nicht möglich, den Parameterübergabemechanismus von Methode zu Methode unterschiedlich zu regulieren.

Die große Varianz der Lösungen und die Tatsache, dass viele Programmiersprachen bei einem so häufig vorkommenden Konstrukt verschiedene Wege gehen, zeigt, dass es keine etablierte Lösung gibt. Dies ist ein Indiz für die Unzulänglichkeiten der vorhandenen Lösungen.

Auch bei der Parameterübergabe geht es um Abhängigkeiten oder Unabhängigkeiten von Variablen, nämlich denen innerhalb und außerhalb der Methoden. Mit einem Mechanismus für Abhängigkeiten könnten u.a. Variationen bei der Parameterübergabe umgesetzt werden.

4.3 Rollen

Rollen sind in der Literatur stark verbreitet [CLD05], wobei auf mitunter sehr verschiedene Aspekte eingegangen wird. Grundsätzlich wird versucht, die verschiedenen Rollen einer Entität konzeptuell oder programmtechnisch umzusetzen. Rollen einer Entität haben oft gemeinsame bzw. voneinander abhängige Eigenschaften. Daher ist es sinnvoll, Rollen bei der Untersuchung von Abhängigkeiten mit einzubeziehen. In der Literatur werden die Abhängigkeiten der Rollen nur unzureichend behandelt [CLD05, CD05b].

4.3.1 Beispielszenario

Hier wird ein leicht abgewandeltes Beispiel präsentiert, das ursprünglich aus [BD96] stammt. In [BD96] sprechen die Autoren nicht von Rollen, sondern von Gesichtspunkten. Ich beschränke mich auf die Diskussion der Attribute.

Betrachten wir eine Entität, die einen Menschen namens Joe repräsentiert. Zuerst sehen wir Joe einfach als Person. Seine Eigenschaften Adresse, Alter, Name, Telefon und Ehefrau sind von Interesse. Außerdem ist Joe Sportler. Dabei sind außer den bereits genannten Eigenschaften Ausdauer und Gewicht wichtig. Die Ehefrau ist in diesem Kontext irrelevant. Joe hat also eine zweite Rolle, die des Sportlers. Weiterhin ist Joe ein Filmenthusiast, seine dritte Rolle. Er hat einen Lieblingsschauspieler, einen Lieblingsfilm und einen Lieblingsregisseur. Die übrigen Eigenschaften mit Ausnahme von Ehefrau und Alter werden von seiner ersten Rolle, Person, übernommen. In Joes Filmgemeinde, die

sich bei Joe trifft, werden Spitznamen verwendet, die Eigenschaft Name wird durch die Eigenschaft Spitzname ersetzt.

Obwohl Joe aus verschiedenen Perspektiven betrachtet wird, ist er natürlich ein und dieselbe Person. So ist z.B. der Wert der Adresse aus allen Perspektiven gleich. Wenn er umzieht, egal in welcher Rolle, ändert sich die Adresse bei allen Rollen. Auch der Name als Person und der Spitzname als Filmenthusiast sind gleich, weil Joe auch von anderen Filmenthusiasten mit seinem Vornamen angesprochen wird.

Joe ist berufstätig, seine vierte Rolle ist Angestellter. Eine neue, für diese Rolle relevante Eigenschaft ist sein akademischer Grad. Die Adresse und die Telefonnummer unterscheiden sich von anderen Rollen, da es sich um die Arbeitsadresse und die geschäftliche Telefonnummer handelt.

Abbildung 4.3 stellt den Sachverhalt in einem Objektdiagramm dar. Die Rollen sind dabei als Objekte und die Abhängigkeiten als Linien dargestellt, wobei die Verbindungen, die bereits durch Transitivität gegeben sind, nicht noch einmal aufgezeichnet werden. So gibt es z.B. keine Linie zwischen den Adressen von `joeFilmEnthusiast` und `joeSportsman`.

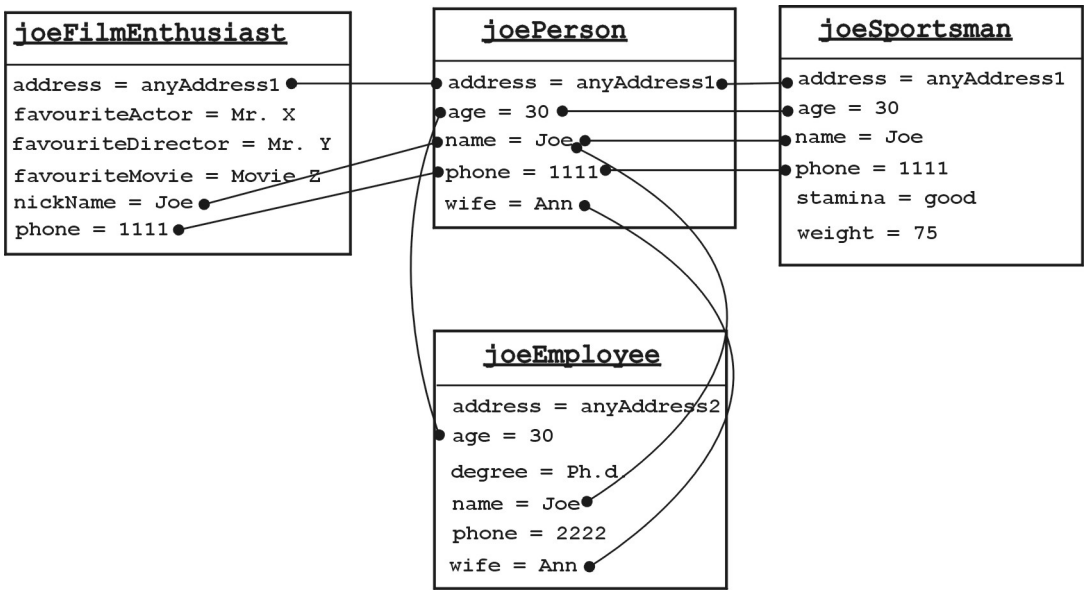

Abbildung 4.3: Joes Rollen

Um Joe und seine verschiedenen Rollen zu modellieren, erscheint es sinnvoll, eine Vererbungshierarchie der Klassen dieser Objekte einzusetzen. In diesem konkreten Beispiel ist die Vererbung allerdings irrelevant, da es um Abhängigkeiten der Objekte untereinander geht. Durch Vererbung kann die Struktur wiederverwendet, aber nicht die Werte der Attribute in Abhängigkeiten gebracht werden [Ast96, BD96].

Die Anforderungen an den Entwurf können geschärft werden, indem man verschiedene Stufen von Abhängigkeiten einführt. Wenn z.B. `joePerson` umzieht, soll `address` von `joeFilmEnthusiast` ebenfalls geändert werden. Wenn sich die Adresse von

`joeFilmEnthusiast` ändert, weil Joe andere Enthusiasten in einem Club treffen möchte, bleibt die Adresse von `joePerson` die alte. In Kapitel 5 wird dieser Sachverhalt als unsymmetrische Abhängigkeit eingeführt. Die Adresse von `joeSportsman` soll trotz möglicher Änderungen mit der Adresse von `joePerson` übereinstimmen. Das Überladen oder Nichtüberladen der Eigenschaften soll also möglichst flexibel gehalten werden.

Die Rollen werden ausführlich im Anhang A behandelt. Weitere Beispiele für Abhängigkeiten bei Rollen findet man in [CD05b, CD05a, CLD05, Ken99]. Nicht nur bei Rollen sind Abhängigkeiten ein wichtiger Aspekt. In [BK00] beschäftigen sich die Autoren mit gemeinsamen Teilen von Perspektiven, bei denen ähnliche Abhängigkeiten auftreten.

4.3.2 Lösungsansätze

Einen ausführlichen Vergleich verschiedener Lösungen findet man in [CLD05, Laz05] und in Anhang A.

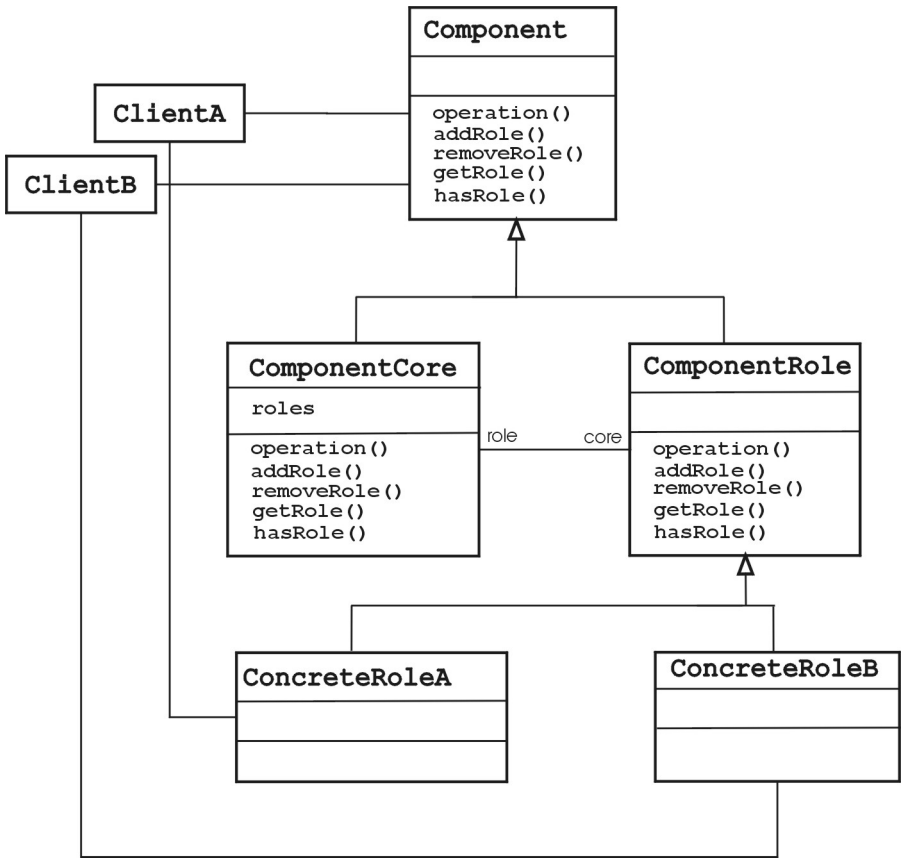

Abbildung 4.4: Role Object

Das Role Object Pattern eignet sich normalerweise gut zur Umsetzung von Rollen (siehe Abschnitt 3.1.6 und Abbildung 4.4). Im Core Object werden die innewohnenden Eigenschaften gekapselt, während Rollenobjekte zusätzliche Rolleninformationen und

-funktionalität beinhalten. Der Ansatz bewältigt die Abhängigkeiten jedoch nicht gut genug:

- Es ist schwer, die Attribute des Core Objects zu bestimmen. Die Attribute der Rollen variieren und lassen sich schlecht in innewohnende und zusätzliche unterteilen. Es gibt keine einzige Eigenschaft, die bei allen Rollen vorkommt und deren Wert in allen Rollen übereinstimmt. Dafür gibt es viele Eigenschaften, die von mehreren Rollen geteilt werden.
- Die Schnittmenge der einzelnen Rollen wird normalerweise in das Core Object ausgelagert. Da hier die einzelnen Schnittmengen unterschiedlich sind, braucht man mehrere Core Objects, was das Design verkompliziert. Im ungünstigsten Fall werden $\frac{n(n-1)}{2}$ Core Objects für n Rollen gebraucht, um die Schnittmengen zwischen je zwei Rollen auszulagern.
- Die unsymmetrischen Abhängigkeiten bei Attributen werden durch das Design Pattern nicht gelöst.

Eine weitere Möglichkeit, Rollen umzusetzen, besteht darin, Interfaces einzusetzen [Ste01, CLD05]. Eine Klasse soll alle Rollen als Interfaces implementieren. Dabei werden Abhängigkeiten zwischen den Attributen der Rollen besser gelöst, weil sich alle Attribute in der gleichen Klasse befinden. Dieser Ansatz hat folgende Nachteile:

- Interfaces deklarieren Methoden und keine Attribute. Also muss jedes Attribut durch eine öffentliche Methode zugreifbar sein.
- Den Anforderungen nach sollen Methoden mit gleichen Namen, wie z.B. `getAddress`, verschiedene Werte liefern. Denn die Adresse des Angestellten unterscheidet sich von der privaten Adresse der Person. Dies ist nur in wenigen Programmiersprachen möglich [Mic05].
- Die Interfaces implementierende Klasse kann sehr komplex werden, da sie die Funktionalität aller Rollen implementieren muss.

Ein weiterer Ansatz ist die Mehrfachvererbung. Sie funktioniert ähnlich wie die Lösung mit Interfaces, nur werden hier Interfaces durch Klassen ersetzt. Die Vor- und Nachteile der Mehrfachvererbung werden seit Jahrzehnten diskutiert. In Anhang A und in [CLD05] gehe ich näher darauf ein.

Neben den vorgestellten Ansätzen zu Rollen gibt es viele weitere. So werden neue Sprachkonstrukte für Rollen z.B. in [Her05] eingeführt. Andere bauen auf vorhandenen Kompositionsschemata in Programmiersprachen auf, wie z.B. auf Templates [Van97]. Dabei werden Abhängigkeiten zwischen Rollen nicht als Schwerpunkt betrachtet (siehe Anhang A).

Wie in diesem Abschnitt erläutert, ist das Forschungsgebiet für Rollen und Perspektiven sehr ausgeprägt. Dabei werden viele verschiedene Ziele verfolgt [Kri96, Ste00, CLD05, Laz05]. Dem zentralen Punkt der Abhängigkeit der Rollen wird von den meisten Ansätzen zu wenig Beachtung geschenkt. So ist es nur durch umständliche Programmierung möglich, komplexe Abhängigkeitsszenarien umzusetzen.

Mit der Einführung von Abhängigkeiten in den nächsten Kapiteln wird es möglich, die bereits vorhandenen Rollenansätze wieder zu verwenden und sie leicht mit Abhängigkeiten zu kombinieren.

4.4 Weitere Aspekte

Zusätzlich zu den in dieser Arbeit ausführlich diskutierten Abhängigkeitsbeispielen, existieren in der Literatur viele andere. Dazu gehört die Modellierung von Gebäudekomplexen mit objektorientiertem Ansatz [MB98]. Teile des Modells werden durch Objekte repräsentiert. Eine Änderung eines Objekts zieht Änderungen der anderen Objekte, z.B. Verschiebungen, nach sich. Ein weiteres bereits in Abschnitt 3.2.3 dargestelltes Beispiel ist ein Temperatursensor und die zugehörige Anzeige. Diese zeigt die Temperatur an, die der Sensor misst [MH06]. Es lassen sich viele weitere Beispiele finden und konstruieren.

Ein wünschenswerter Aspekt von Abhängigkeiten liegt darin, dass man die hergestellten Abhängigkeiten auflösen oder durch andere ersetzen kann. Allgemeiner ausgedrückt braucht man verschieden starke Abhängigkeiten. Dieses Phänomen ist auch in der realen Welt zu beobachten: Manche Beziehungen zwischen Entitäten sind stärker als andere.

Eine weitere Anforderung an einen Abhängigkeitsansatz besteht darin, dass sowohl symmetrische als auch unsymmetrische Abhängigkeiten ermöglicht werden. Diese Symmetrie bezieht sich auf die Auflösung der Abhängigkeiten zwischen Variablen. Ein Beispiel für eine symmetrische Abhängigkeit sind Ehepartner, die dasselbe Auto fahren. Sollte einer der Ehepartner ein neues Auto kaufen, fahren beide mit dem neuen. Die Ehepartner sind gleichberechtigt. Wenn wir einen Vater und seinen Sohn betrachten, die dasselbe Auto fahren, ist die Abhängigkeit unsymmetrisch. Sollte der Vater ein neues Auto kaufen, fahren die Beiden weiterhin dasselbe Auto; sollte der Sohn ein eigenes Auto bekommen, fahren sie verschiedene Fahrzeuge. Obwohl unsymmetrische Abhängigkeiten allgegenwärtig sind, sind alle gängigen existierenden Zuweisungsoperatoren symmetrisch (siehe Abschnitt 5.3.1). Eine formale Definition der Symmetrie folgt in Abschnitt 5.1.2.

Wie in Unterkapitel 2.2 dargestellt, werden Abhängigkeiten oft mit Zuweisungen gelöst. Der Referenzierungsgrad und somit die hergestellte Abhängigkeit hängt dabei immer von den Typen der Variablen ab. Man muss sich somit bereits bei der Deklaration eines Typs entscheiden, in welchen Abhängigkeitsarten die Variablen dieses Typs vorkommen dürfen. Die Abhängigkeiten sollen flexibler gestaltet werden: Für jede Variable soll es mehrere Abhängigkeitsarten geben und nicht nur eine, die durch den Typ bestimmt ist.

Die durch Zuweisungen hergestellten Abhängigkeiten basieren auf Zeigern, die sowohl explizit wie in C++ als auch implizit wie in Java und C# sein können. Wie bereits in Abschnitt 2.2.2 argumentiert, versuchen viele Programmiersprachenentwickler Zeiger zu vermeiden. In dieser Arbeit soll eine zu Zeigern alternative Lösung erarbeitet werden, die für Programmierer intuitiver ist.

Der letzte Aspekt, den ich hier ansprechen möchte, handelt von Rückgabewerten bei Methoden. Wie in Abschnitt 4.2.1 diskutiert, ist es für Methodenparameter wichtig, mit welcher Variation diese übergeben werden. Wie in Abschnitt 3.2.4 erläutert, gilt eine ähnliche Problematik für Rückgabewerte. Diese sollen sowohl by-value als auch by-

reference zurückgegeben werden können. Anders formuliert: Die Abhängigkeit zwischen der Variablen in der Methode und der Variablen beim Methodenaufruf soll reguliert werden.

4.5 Schlussfolgerung

In allen untersuchten Fällen geht es um Abhängigkeiten zwischen Variablen. Es wird gezeigt, dass Abhängigkeiten allgegenwärtig sind und in sehr unterschiedlichen Situationen vorkommen. Dabei stellen die Abhängigkeiten selten die Hauptanforderung dar. Normalerweise verfolgt man ein anderes Ziel mit der Nebenbedingung, dass die Abhängigkeiten konsistent gehalten werden. Hauptziele sind z.B. die Bearbeitung von Daueraufträgen oder das Programmieren von Rollen.

Wie geschildert, können die semantischen Anforderungen an die Programmierung der Abhängigkeiten zwar erreicht werden, jedoch auf sehr unterschiedlichen Wegen. Die meisten resultierenden Designs sind aufgrund vieler Hilfsklassen komplex. Die eigentliche Funktionalität verschwindet hinter diesen Klassen.

Außerdem gibt es keinen Standardweg, sondern nur Lösungen, die für Spezialfälle erarbeitet werden. Dies ist ein Indiz dafür, dass das grundsätzliche Problem bislang nicht befriedigend gelöst ist.

Die vorhandenen Mittel für Abhängigkeiten verändern in den meisten Fällen die Architektur. Einerseits spricht das gegen das Kriterium Kontinuität, da kleine Änderungen der Anforderung große Änderungen in der Umsetzung erfordern. Andererseits ist die Primärarchitektur oft nicht mehr gut erkennbar, so dass das Kriterium Verständlichkeit ebenfalls nicht erfüllt ist (siehe Abschnitt 2.1.2). Man braucht also ein Sprachmittel für Abhängigkeiten, das subtil, ohne die Architektur zu verändern, in die Implementierung eingewoben werden kann. Deswegen beschäftigen sich die folgenden Kapitel mit der Lösung dieses Problems: Abhängigkeiten sollen durch Sprachkonstrukte ausgedrückt werden.

5 3-Schichtenmodell des Speichers

Dieses Kapitel präsentiert die Grundgedanken meines Ansatzes zur Umsetzung von Abhängigkeiten. Ich führe ein Modell des Speichers ein, das aus drei Schichten von Speicherzellen besteht. Es beschreibt die Speicherung von Variablen und Werten. Ein *Pfad* kann dadurch beschrieben werden, dass Speicherstellen Adressen von anderen Speicherzellen als Inhalt haben. Von Interesse ist der Pfad von einer Variablen zu ihrem Wert. Während Variablen mit der dritten Schicht assoziiert werden, werden Werte in der ersten Schicht gespeichert. Dabei können verschiedene Variablen einen Teil des Pfades mit anderen Variablen teilen. Durch einen gemeinsamen Teilpfad werden Abhängigkeiten zwischen Variablen ausgedrückt.

Im 3-Schichtenmodell bewirkt eine parametrisierte Zuweisung $_i{=}_j$ zwischen zwei Variablen, dass der Wert in der Schicht j im Pfad einer Variablen in die Schicht i im Pfad der anderen Variablen geschrieben wird. Auf diese Art können verschieden lange gemeinsame Teilpfade der Variablen erzeugt und damit verschiedene Abhängigkeiten zwischen Variablen hergestellt werden.

Das Kapitel ist folgendermaßen aufgebaut: Zuerst werden in Unterkapitel 5.1 Variablen und Werte unabhängig vom 3-Schichtenmodell eingeführt. Auf den Variablen werden Zuweisungen definiert. Schließlich werden verschiedene Arten und nützliche Eigenschaften von Zuweisungen erläutert. Unterkapitel 5.2 führt in die Grundlagen des 3-Schichtenmodells ein. Dieses wird sowohl anschaulich beschrieben als auch formal definiert. Unterkapitel 5.4 erweitert das 3-Schichtenmodell um Arrays und Objekte. Danach definiere ich die Typregeln für Zuweisungen und beweise, dass das 3-Schichtenmodell mit diesen Regeln typsicher ist. Schließlich diskutiere ich einige weniger zentrale Aspekte im Zusammenhang mit parametrisierten Zuweisungen.

5.1 Eigenschaften von Zuweisungen

Im Laufe dieses Kapitels wird ein Modell entwickelt, das Abhängigkeiten handhaben soll. Dieses Modell basiert auf Zuweisungen. Um zu motivieren, warum ausgerechnet das 3-Schichtenmodell und kein anderes auf Zuweisungen basierendes Modell gewählt wird, werden hier Zuweisungen zunächst unabhängig von einem konkreten Modell betrachtet. Es wird gezeigt, welche Zuweisungseigenschaften wünschenswert sind.

5.1.1 Einführung

Zunächst führe ich eine Menge von Variablen *Vars* und eine Menge von Werten *Vals* ein. Die Variablen werden üblicherweise mit x, y, z und Werte mit v_1, v_2, v_3 bezeichnet.

Der Zustand eines Programms wird durch eine partielle Funktion

$$s : \mathit{Vars} \nrightarrow \mathit{Vals}$$

repräsentiert, die Variablen auf Werte abbildet. Dabei steht s für *state* und der Pfeil $\nrightarrow$ für eine partielle Funktion. In diesem Unterkapitel wird davon abstrahiert, dass ein Zustand möglicherweise Zusatzinformationen speichert. Hier bezeichnet der Zustand lediglich die Belegung von Variablen. Da manche Variablen keinen Wert haben, ist die Funktion s partiell.

Das Kernthema dieser Arbeit sind Abhängigkeiten. *Wertgleichheit* ist eine notwendige Voraussetzung für die *Abhängigkeit*. Zwei Variablen `x` und `y` sind *wertgleich*, wenn für definierte $s(x)$ und $s(y)$ gilt

$$s(x) = s(y)$$

oder, wenn $s(x)$ und $s(y)$ beide undefiniert sind; es handelt sich also um starke Äquivalenz [Wir90]. Anders ausgedrückt:

Definition 1. *Sei* $s \in$ Vars $\nrightarrow$ Vals *ein Zustand. Die Variablen* `x` *und* `y` *sind genau dann* wertgleich*, wenn*

$$(x, y) \in \mathit{ker}(s)$$

ker bezeichnet den Kern einer Funktion. In der Mengentheorie [Jec02] und universeller Algebra [DW02] ist $\mathit{ker}(f)$ als folgende Menge definiert:

$$\mathit{ker}(f) = \{(x, y) | f(x) = f(y)\}$$

Auf dem Zustand operieren *Aktionen*

$$a_1, \ldots, a_n : \mathit{Pars} \times (\mathit{Vars} \nrightarrow \mathit{Vals}) \nrightarrow (\mathit{Vars} \nrightarrow \mathit{Vals})$$

Diese überführen einen Zustand in einen neuen dadurch, dass sie Werte von Variablen ändern. a steht für *action*; bei Pars handelt es sich um Parameter, mit denen die Aktionen durchgeführt werden. Die Anwendung einer Aktion wird syntaktisch durch einen Pfeil notiert:

$$s \xrightarrow{a_i(p_1,\ldots,p_m)} s'$$

Diese Formel sagt aus, dass durch die Anwendung der Aktion a_i mit den Parametern $(p_1, \ldots, p_m) \in \mathit{Pars}$ der Zustand s in den Zustand s' übergeht.

Von Interesse sind dabei bestimmte Aktionen, die zwischen zwei Variablen Wertgleichheit herstellen, indem die erste Variable den Wert der zweiten Variablen zugewiesen bekommt. Diese Aktionen werden *Zuweisungen* zwischen Variablen genannt.

$$f, g : (\mathit{Vars} \times \mathit{Vars}) \times (\mathit{Vars} \nrightarrow \mathit{Vals}) \nrightarrow (\mathit{Vars} \nrightarrow \mathit{Vals})$$

Anstatt $f(x, y, s)$ schreibe ich beim gegebenen Zustand s

`x` $=_f$ `y;`

um die Ähnlichkeit zu der Syntax einer konventionellen Programmiersprache zu bewahren. Nach der Ausführung der Zuweisung von y zu x

$$s \xrightarrow{(x =_f y)} s'$$

hat x den Wert von y

$$s'(x) = s'(y).$$

Solch eine Zuweisung kann Nebeneffekte haben, so dass nicht nur x, sondern auch andere Variablen einen neuen Wert bekommen. Insbesondere kann die Änderung von x, y so verändern, dass $s'(y)$ undefiniert ist. Ansonsten behält y seinen alten Wert

$$s'(y) = s(y).$$

In diesem Kapitel wird oft über die Komposition von partiellen Funktionen argumentiert. Dabei ist die Komposition wie üblich definiert [Lep92]: Seien f und g partielle Funktionen. Wenn $g(x)$ undefiniert ist, ist auch $f(g(x))$ undefiniert. Ansonsten wird f auf den definierten Wert $g(x)$ angewendet.

5.1.2 Symmetrie

Wie bereits erläutert, stellt eine Zuweisung eine Wertgleichheit zwischen zwei Variablen her. Dies passiert unabhängig davon, ob eine Variable links oder rechts steht: sowohl

x $=_f$ y;

als auch

y $=_f$ x;

erzwingen eine Wertgleichheit zwischen x und y. Durch eine Folgeaktion kann solch eine Wertgleichheit aufgelöst oder beibehalten werden. Eine Eigenschaft von Zuweisungen besteht darin, dass bestimmte Folgezuweisungen die Wertgleichheit zwischen Variablen x und y gleich behandeln, unabhängig davon, ob x bzw. y rechts oder links von dem Gleichheitszeichen stehen. Es stellt sich heraus, dass es sich dabei um Symmetrie handelt (siehe Definition 2).

Solch eine symmetrische Zuweisung ist der Zuweisungsoperator = in C++. Betrachten wir das folgende Codefragment, bei dem sich der Wert von z von x und y unterscheidet:

```
1  x = y;
2  x = z;
```

Die in Zeile 1 aufgestellte Wertgleichheit wird durch die Aktion in Zeile 2 aufgelöst, da x einen neuen Wert bekommt und y den alten behält. Dabei könnte in Zeile 1 auch

```
y = x;
```

stehen. Die Aussage über die Auflösung der Wertgleichheit in Zeile 2 bleibt unverändert.

Betrachten wir eine andere Folgezuweisung $* = *$. Damit bezeichne ich die Zuweisungen in C++ der Form

```
*x = *y;
```

x und y sollen dabei Zeiger sein, z.B. vom Typ `int*`.

```
1  x = y;
2  *x = *z;
```

Nun zeigen x und y auf den gleichen neuen Wert, nämlich den von *z. Die Wertgleichheit wird beibehalten, obwohl die Werte der Variablen sich geändert haben. Sie würde auch beibehalten werden, wenn die Positionen von x und y in Zeile 1 vertauscht wären.

Unsymmetrische Zuweisungsoperatoren gibt es in gängigen Programmiersprachen nicht. Den Effekt kann man durch andere Aktionen simulieren. Intuitiv kann man sich so eine Zuweisung in C++ durch verschieden viele Sternchen an den beiden Seiten des Gleichheitszeichens vorstellen. Unsymmetrische Zuweisungen sind wichtig, da es einerseits Pendants in der realen Welt gibt und andererseits Variablen und Attribute in Softwareprojekten oft verschieden gewichtet werden.

Nun wird Symmetrie einer Zuweisung formal definiert. Sei $s \in (\mathit{Vars} \nrightarrow \mathit{Vals})$ ein Zustand und $x, y, z \in \mathit{Vars}$ Variablen. Die Zuweisung f von y zu x ausgehend vom Zustand s wird durch

$$f(x, y, s)$$

ausgedrückt. Auf den resultierenden Zustand wird nun eine Zuweisung g mit Parametern x, z mit $s(z) \neq s(x)$ und $s(z) \neq s(y)$ angewendet:

$$g(x, z, f(x, y, s)).$$

Schließlich überprüfen wir, ob x und y weiterhin wertgleich sind. Die Ausführung dieser Zuweisungen entspricht dem Programmfragment

```
1  x =f y;
2  x =g z;
```

Definition 2. *Eine Zuweisung f ist genau dann symmetrisch, wenn für alle Variablen* x, y, z *mit einem gültigen Wert, für alle Zustände s und für alle Zuweisungen g mit den Einschränkungen, dass der Wert von* z *sich von den Werten von* x *und* y *unterscheidet*

$$\forall x, y, z, s \quad \text{mit} \quad (x, z), (y, z) \notin ker(s)$$

gilt:

$$(x, y) \in ker(g(x, z, f(\mathbf{x}, \mathbf{y}, s))) \Leftrightarrow (x, y) \in ker(g(x, z, f(\mathbf{y}, \mathbf{x}, s))).$$

Eine nicht symmetrische Zuweisung wird unsymmetrisch *genannt.*

Was bereits für symmetrische und unsymmetrische Zuweisungen am Anfang des Abschnitts intuitiv beschrieben ist, wird nun für Variablen mit konkreten Belegungen gezeigt. Sei f eine symmetrische Zuweisung. Seien die Werte von x, y und z am Anfang jedes Codefragments v_1, v_2 und v_3. Nehmen wir an, dass die Variablen x, y und z nach der Ausführung der Codefragmente in Listings 5.1 und 5.2 definiert bleiben. Es gibt nur zwei Möglichkeiten für die Belegung am Ende der Codefragmente

Listing 5.1: Symmetrie 1

```
1  x =f y;
2  x =g z;
```

Listing 5.2: Symmetrie 2

```
1  y =f x;
2  x =g z;
```

- In der zweiten Zeile jedes Fragments wird Wertgleichheit zwischen x und y beibehalten.
  - In Listing 5.1 haben x und y nach der ersten Zeile den Wert v_2. Nach der zweiten Zeile haben x, y und z den Wert v_3.
  - In Listing 5.2 haben x und y nach der ersten Zeile den Wert v_1. Nach der zweiten Zeile haben x, y und z den Wert v_3.
- In der zweiten Zeile jedes Fragments wird die Wertgleichheit zwischen x und y aufgelöst.
  - In Listing 5.1 haben x und y nach der ersten Zeile den Wert v_2. Nach der zweiten Zeile haben x und z den Wert v_3. y behält seinen alten Wert v_2.
  - In Listing 5.2 haben x und y nach der ersten Zeile den Wert v_1. Nach der zweiten Zeile haben x und z den Wert v_3. y behält seinen alten Wert v_1.

Wäre die Zuweisung f dagegen unsymmetrisch, könnte es abhängig von der Zuweisung g passieren, dass nach dem Codefragment in Listing 5.1 x den Wert v_3 hat, während y weiterhin den Wert v_2 behält. Nach der Ausführung des Fragments in Listing 5.2 würden x und y beide den neuen Wert v_3 annehmen und somit wertgleich bleiben.

5.1.3 Stärke

Eine weitere wichtige Eigenschaft bei Zuweisungen ist deren Stärke. Intuitiv kann man sagen, dass eine Zuweisung f stärker ist als eine Zuweisung g, wenn die Wertgleichheit, die durch f hergestellt wird, durch g nicht aufgelöst werden kann. Andererseits soll die durch g hergestellte Wertgleichheit durch f aufgelöst werden können.

Als Beispiel nehmen wir die C++-Zuweisungen $=$ und $* = *$ bei Variablen des Typs `int*` (siehe Abschnitt 5.1.2). $=$ ist stärker als $* = *$, denn nach der Ausführung des Fragments

```
1  x = y;
2  *x = *z;
```

zeigen x und y zwar auf einen neuen, aber weiterhin den gleichen Wert. Auch

```
*y = *z;
```

in Zeile 2 löst die Wertgleichheit zwischen x und y nicht auf. Wenn die Zuweisungen in anderer Reihenfolge stattfinden

```
1  *x = *y;
2  x = z;
```

haben x und y bei passendem z anschließend verschiedene Werte. = überschreibt also $* = *$.

Definition 3. *Die Zuweisung f ist* stärker *als die Zuweisung g bzw. g ist* schwächer *als f genau dann, wenn die folgenden Bedingungen gelten:*

$$\forall x, y, z, s : (x, y) \in ker(g(x, z, f(x, y, s)))$$
$$\exists x, y, z, s : (x, y) \notin ker(f(x, z, g(x, y, s)))$$

Die erste Bedingung sagt aus, dass g niemals f überschreibt. Dagegen sagt die zweite Bedingung aus, dass f g überschreiben kann.

5.2 Grundlagen des 3-Schichtenmodells

5.2.1 Syntax

Tabelle 5.1 beschreibt die Syntax einer Programmiersprache mit parametrisierten Zuweisungen des 3-Schichtenmodells. Diese Syntax beschränkt sich nur auf die Darstellung von Variablen, Werten und Zuweisungen. Die Erweiterungen um Typen und Methodenaufrufe werden in Unterkapitel 5.4 präsentiert. Die einzelnen Produktionen der Grammatik entsprechen der Backus-Naur-Form [Nau60] (siehe Abschnitt 2.3.1).

assign	::=	*var assignOp expr*	Zuweisung
assignOp	::=	$= \mid {}_2{=}_2$	${}_2{=}_2$ Zuweisungsoperator
	|	${}_1{=}_1$	${}_1{=}_1$ Zuweisungsoperator
	|	${}_2{=}_3$	${}_2{=}_3$ Zuweisungsoperator
	|	${}_3{=}_3$	${}_3{=}_3$ Zuweisungsoperator
expr	::=	*val*	Ausdruck
	|	*var*	
val	::=	$v \mid v_1 \mid v_2 \mid v_3 \mid \ldots$	Wert
var	::=	$x \mid y \mid z \mid u \mid \ldots$	Variable

Tabelle 5.1: Syntax

Zunächst werden Terminalzeichen für Werte und für Variablen eingeführt. Ein Ausdruck ist entweder ein Wert oder eine Variable. Die erste Produktion stellt eine Zuweisung dar. Links von dem Zuweisungsoperator steht eine Variable, rechts ein Ausdruck.

Die Zuweisungsoperatoren stellen das Neue an der gegebenen Syntax dar. Im Gegensatz zur üblichen Notation gibt es nicht einen solchen Operator, sondern mehrere. Es werden vier Arten vorgestellt, wobei = nur eine zweite Bezeichnung für $_2{=}_2$ ist. Diese Zuweisungsoperatoren haben jeweils zwei Parameter, die sich links und rechts von dem Gleichheitszeichen befinden. Gelesen werden die Zuweisungen als eins-eins-Zuweisung, zwei-zwei-Zuweisung, zwei-drei-Zuweisung und drei-drei-Zuweisung.

5.2.2 Anschauliche Darstellung des Modells

Veranschaulichen wir uns nun die Semantik der gegebenen Syntax. Die exakte Bedeutung folgt im nächsten Abschnitt.

Das Speichermodell für parametrisierte Zuweisungen hat 3 disjunkte Schichten. Schicht 1 hat Speicheradressen $a_{11}, a_{12}, a_{13}, \ldots$. Als Inhalt können nur Werte gespeichert werden. Schicht 2 hat Adressen $a_{21}, a_{22}, a_{23}, \ldots$. Als Inhalte können beliebige Adressen gespeichert werden, jedoch keine Werte. In Schicht 3 mit Adressen $a_{31}, a_{32}, a_{33} \ldots$ haben die Speicherzellen als Inhalte Adressen der zweiten Schicht. Die dritte Schicht wird mit Variablen assoziiert.

Ein beispielhafter Zustand des Speichers ist in Tabelle 5.2 dargestellt. Die erste Zeile jeder Schicht enthält die Namen der Speicherzellen, die zweite die gespeicherten Werte. Die dritte Schicht ist um eine Zeile mit Variablen erweitert. Die Variable `x` hat beispielsweise den Wert v_3, was aus ihrem Pfad hervorgeht: Der *Pfad* führt über die Speicherzellen a_{31}, a_{22} und a_{12}, denn die Speicherzelle von `x` ist a_{31} und enthält a_{22}; a_{22} enthält ihrerseits a_{12}; schließlich enthält a_{12} den Wert v_3. Entsprechend hat die Variable `y` den Wert v_2; der Pfad führt über a_{32}, a_{21}, a_{24} und a_{15}.

Schicht 3:

x	y	k	z	m
a_{31}	**a_{32}**	**a_{33}**	**a_{34}**	**a_{35}**
a_{22}	a_{21}	a_{24}	a_{22}	a_{25}

Schicht 2:

a_{21}	**a_{22}**	**a_{23}**	**a_{24}**	**a_{25}**
a_{24}	a_{12}	a_{12}	a_{15}	a_{11}

Schicht 1:

a_{11}	**a_{12}**	**a_{13}**	**a_{14}**	**a_{15}**
v_4	v_3	v_1	v_4	v_2

Tabelle 5.2: Speichermodell

Bei einer Zuweisung soll der Inhalt einer bestimmten Speicherzelle in eine andere Speicherzelle geschrieben werden. Vorerst beschränke ich mich auf den Fall, dass auf beiden Seiten der Zuweisung Variablen stehen. Die Zuweisungen werden parametrisiert.

`x` $_i{=}_j$ `y;`

bedeutet Folgendes: i auf der linken Seite steht für die Schicht, in die geschrieben wird; j auf der rechten Seite steht für die Schicht, aus der gelesen wird. Auf dem Pfad von der

Variablen y zu ihrem Wert wird der Inhalt der Speicherzelle in der Schicht j ausgelesen, wenn diese Schicht zum ersten Mal erreicht wird. In Tabelle 5.2 ist das a_{21} für die Schicht 3, a_{24} für die Schicht 2 und v_2 für die Schicht 1. Analog wird auf dem Pfad von der Variablen x zu ihrem Wert der Inhalt in die Speicherzelle in der Schicht i geschrieben, wenn diese Schicht zum ersten Mal erreicht wird. Für die Schicht 3 wird in die Speicherzelle a_{31} geschrieben, für die Schicht 2 in a_{22} und für die Schicht 1 in a_{12}. Somit wird bei der Zuweisung

```
x 2=2 y;
```

der Wert a_{24} in die Speicherzelle a_{22} geschrieben; bei

```
x 3=3 y;
```

wird a_{21} in a_{31} geschrieben.

Zwischen den hier eingeführten und bereits in Programmiersprachen bestehenden Zuweisungen gibt es Analogien. Die $_2{=}_2$ Zuweisung ist ähnlich wie die Zuweisung in Java oder C++, wobei auf beiden Seiten des Zuweisungsoperators j-Referenzen stehen (siehe Unterkapitel 2.2). Die Änderung des Wertes einer j-Referenz kann die andere beeinflussen:

```
1 MyObject x;
2 MyObject y = new MyObject();
3 x = y;
4 x.changeState();
```

Die Abhängigkeit von x und y besteht ab der Zeile 3. Durch die Zustandsänderung von x in Zeile 4 wird auch der Zustand von y geändert.

Die $_1{=}_1$ Zuweisung erinnert an Zuweisungen zwischen Variablen von primitiven Datentypen in Java. Nach der Durchführung dieser Zuweisung hat die Änderung des Wertes einer Variablen keinerlei Auswirkungen auf die andere.

```
1 int x;
2 int y = 1;
3 x = y;
4 y = 2;
```

Trotz der Herstellung der Wertgleichheit in Zeile 3 kann eine Änderung von y in Zeile 4 keinerlei Auswirkungen auf x haben.

Die $_3{=}_3$ Zuweisung hat eine entfernte Ähnlichkeit zur Initialisierung einer c-Referenz in C++.

```
1 int y = 1;
2 int& x = y;
3 y = 2;
```

Die Abhängigkeit zwischen x und y wird in Zeile 2 aufgebaut. Die Änderung von y durch eine Zuweisung in Zeile 3 ändert x ebenfalls.

Die $_2{=}_3$ Zuweisung hat keine große Ähnlichkeit mit existierenden Zuweisungsoperatoren. Die Besonderheit dieser Zuweisung ist, dass man damit beliebig lange Pfade von

einer Variablen zu ihrem Wert erzeugen kann, da der Pfad von der Variablen zu ihrem Wert beliebig lange in der zweiten Schicht bleiben kann.

```
1 x 2=3 y;
2 y 2=3 z;
```

Nach der Ausführung dieses Codefragments ist der Pfad von `x` länger als der von `y`, und der von `y` länger als der von `z`. Durch die unterschiedlich langen Pfade wird die Unsymmetrie erreicht, die formal in Unterkapitel 5.3 gezeigt wird.

Wegen der möglichen langen Pfade steigt der Zeitaufwand, der nötig ist, um auf den Wert einer Variablen zuzugreifen. Jedoch beeinflusst dies die Performance der meisten Programme nur unwesentlich, da die Prozessorleistungen heute sehr hoch sind. Außerdem zeigen die Untersuchungen, dass die Pfade üblicherweise kurz bleiben (siehe Kapitel 6). Auch für andere Sprachkonstrukte gilt, dass Laufzeiteinbußen zugunsten der Verständlichkeit in Kauf genommen werden, wie es z.B. bei virtuellen Methoden in Java der Fall ist [HR03].

In Abschnitt 5.2.1 sind vier Zuweisungsoperatoren vorgestellt: $_3{=}_3$, $_2{=}_3$, $_2{=}_2$ und $_1{=}_1$. In einem 3-Schichtenmodell sind Zuweisungen $_i{=}_j$ $\forall i, j \in \{1, 2, 3\}$ denkbar. Jedoch verletzen viele davon die oben genannten Einschränkungen der Inhalte der Speicherzellen verschiedener Schichten. Betrachten wir die Zuweisung

```
x 3=1 y;
```

Beim Anfangszustand in Tabelle 5.2 wird der Wert v_2 in die Speicherzelle a_{31} geschrieben, wo bislang a_{22} steht. Diese Zuweisung verletzt die Regel, dass die Schicht 3 nur Adressen der Schicht 2 speichern darf. Also wird dieser Zuweisungsoperator verboten. Analog sind vier andere Zuweisungsarten nicht erlaubt. Tabelle 5.3 zeigt alle potentiellen Zuweisungsoperatoren. Besonders interessant ist der Fall $_3{=}_2$. Es kann sein, dass in einer Speicherzelle der Schicht 2 eine Adresse der Schicht 2 gespeichert ist. So wird keine der Invarianten verletzt, denn eine Adresse der Schicht 2 kann regelkonform in die Schicht 3 geschrieben werden. Aber es kann auch vorkommen, dass der Inhalt eine Speicherzelle einer anderen Schicht ist, so dass in die dritte Schicht ein unerlaubter Inhalt geschrieben wird. Also wird die $_3{=}_2$ Zuweisung verboten.

5.2.3 Operationale Semantik

Die intuitive Semantikbeschreibung aus dem letzten Abschnitt soll nun spezifiziert und formalisiert werden.

Da ich mit Existenz- und Allquantoren operieren werde, brauche ich eine Mengennotation für die in Tabelle 5.1 dargestellte Grammatik. In manchen Artikeln wie [BS99] werden Nichtterminalzeichen verwendet, als ob es sich um Mengen handeln würde. Hier wird eine explizite Umwandlung vorgenommen. Tabelle 5.4 beschreibt die Mengen, auf denen die operationale Semantik operiert. Ein kleingeschriebenes Wort z.B. *val* beschreibt das Nichtterminalzeichen, während ein großgeschriebenes Wort z.B. *Vals* die Menge der Zeichenketten beschreibt, die aus *val* abgeleitet werden können. Das Gleiche gilt für *var* und *assign*.

Parameter 1	Parameter 2	Zulassung
1	1	+
1	2	−
1	3	−
2	1	−
2	2	+
2	3	+
3	1	−
3	2	−
3	3	+

Tabelle 5.3: Potentielle Zuweisungsoperatoren

Werte und Variablen werden bereits in Unterkapitel 5.1 angesprochen, nun werden zusätzlich Speicheradressen eingeführt, die in Abschnitt 5.2.2 motiviert sind. Dabei sind A_1, A_2, A_3 endliche disjunkte Mengen von Speicheradressen der drei Schichten.

Vals	=	$\{v, v_1, v_2, v_3, \ldots\}$
Vars	=	$\{x, y, z, u, \ldots\}$
A_1	=	$\{a_{11}, a_{12}, a_{13}, \ldots\}$
A_2	=	$\{a_{21}, a_{22}, a_{23}, \ldots\}$
A_3	=	$\{a_{31}, a_{32}, a_{33}, \ldots\}$

Tabelle 5.4: Basismengen

Auf diesen Basismengen wird der Zustand aufgebaut. Da jede Speicherzelle einen Namen und einen Inhalt hat, ist es sinnvoll mit partiellen Funktionen zu arbeiten (siehe Tabelle 5.5). Die Funktionen s_1, s_2 und s_3 entsprechen den einzelnen Schichten im Modell. Wie in Abschnitt 5.2.2 motiviert, bildet s_1 Speicherzellen der ersten Schicht auf die Werte ab. Bei der Funktion s_2 ist zu bemerken, dass s_2 nur einen Spezialfall des in Abschnitt 5.2.2 vorgestellten Spielraums ausnutzt. Ursprünglich durften in der zweiten Schicht alle Arten von Speicheradressen gespeichert werden. Durch die vier definierten Zuweisungen kommt es jedoch nie dazu, dass Adressen der dritten Schicht in die zweite geschrieben werden. Also fehlt A_3 im Bildbereich von s_2. s_3 bildet vorschriftsgemäß die Adressen der dritten Schicht auf die der zweiten Schicht ab. Die Funktion s_4 drückt die Assoziation der Variablen zu den Zellen in der dritten Schicht aus. Der Zustand des Programms s erfasst alle drei Schichten und ist ein Tupel aus den vier Funktionen

$$s = (s_1, s_2, s_3, s_4).$$

s_1	:	$A_1 \nrightarrow \mathit{Vals}$
s_2	:	$A_2 \nrightarrow (A_1 \cup A_2)$
s_3	:	$A_3 \nrightarrow A_2$
s_4	:	$\mathit{Vars} \nrightarrow A_3$
s	$\in$	$(A_1 \nrightarrow \mathit{Vals}) \times (A_2 \nrightarrow (A_1 \cup A_2)) \times (A_3 \nrightarrow A_2) \times (\mathit{Vars} \nrightarrow A_3)$

Tabelle 5.5: Zustand

Zu jedem Zeitpunkt besitzt ein Programm einen genau definierten Zustand. Ein konkreter Zustand s kann folgendermaßen aussehen:

$$
\begin{aligned}
& s = (s_1, s_2, s_3, s_4) \quad \text{mit} \\
& s_4(x) = a_{31} \\
& s_4(y) = a_{33} \\
& s_3(a_{31}) = a_{22} \\
& s_3(a_{33}) = a_{23} \\
& s_2(a_{22}) = a_{23} \\
& s_2(a_{23}) = a_{12} \\
& s_1(a_{12}) = v_3
\end{aligned}
$$

Dieser Zustand ist in Tabelle 5.6 dargestellt.

Schicht 3:

x		y
a_{31}	**a_{32}**	**a_{33}**
a_{22}		a_{23}

Schicht 2:

a_{21}	**a_{22}**	**a_{23}**
	a_{23}	a_{12}

Schicht 1:

a_{11}	**a_{12}**	**a_{13}**
	v_3	

Tabelle 5.6: Exemplarischer Zustand

Nachdem die einzelnen Schichten definiert sind, kann man nun eine Verbindung zwischen einer Variablen und einem Wert definieren. Dazu ist zunächst eine Menge von Hilfsfunktionen s_2^i hilfreich. Dabei handelt es sich um eine mehrfache Anwendung von

s_2. Der Definitionsbereich von s_2 muss in A_2 sein, ansonsten ist das Ergebnis undefiniert.

$$s_2^1(a) = s_2(a)$$
$$s_2^{i+1}(a) = \begin{cases} s_2(s_2^i(a)) & \text{falls } s_2^i(a) \in A_2 \\ \perp & \text{sonst} \end{cases}$$

Die Einschränkung der Definitionsbereiche gilt auch für die Komposition $s_1(s_2(a))$. Dieser Wert ist nur dann definiert, wenn $s_2(a) \in A_1$.

Definition 4. *Gegeben sei ein Zustand* $s = (s_1, s_2, s_3, s_4)$. *Variable* x *hat den* Wert v *genau dann, wenn:*

$$\exists k : s_1(s_2^k(s_3(s_4(x)))) = v$$

Wir notieren diesen Sachverhalt als

$$s(x) = v.$$

Diese Definition weicht von dem Sachverhalt in Unterkapitel 5.1 ab, da s keine Funktion ist, die auf eine Variable angewendet werden kann. Jedoch kann man auf der Basis von s eine Funktion

$$s' : \mathit{Vars} \rightharpoonup \mathit{Vals}$$

konstruieren, die Variablen auf ihre Werte abbildet. Somit wird hier weiterhin ein Spezialfall der Zuweisungen in Unterkapitel 5.1 behandelt. Auch bei Wertgleichheit von Variablen x und y schreibe ich

$$(x, y) \in \mathit{ker}(s)$$

und meine eigentlich die Funktion s', die mit Hilfe von s konstruiert wird.

Der *Pfad* fängt bei x an, endet bei v und führt über die Speicherzellen $a_{3i}, a_{2j_1}, \ldots, a_{2j_k}, a_{1l}$.

Definition 5. *Gegeben sei ein Zustand* $s = (s_1, s_2, s_3, s_4)$. *Ein* Pfad *ist eine Folge*

$$p = (d_1, \ldots, d_n) \quad \text{mit} \quad d_i \in \mathrm{Vars} \cup A_1 \cup A_2 \cup A_3 \cup \mathrm{Vals},$$

so dass gilt

$$\forall i \in \{1, \ldots, n-1\}\, \exists j \in \{1, \ldots, 4\} : d_{i+1} = s_j(d_i).$$

Wir notieren den Pfad als

$$d_1 \xrightarrow[d_2,\ldots,d_{n-1}]{} d_n.$$

Wir sagen *Pfad von* x, wenn es um den Pfad geht, der bei x startet. Die *Länge* des Pfades ist die Anzahl seiner Elemente. In Tabelle 5.6 sind die Pfade von x und y

$$x \xrightarrow[a_{31},a_{22},a_{23},a_{12}]{} v_3,$$
$$y \xrightarrow[a_{33},a_{23},a_{12}]{} v_3.$$

Die Variablen haben also einen gemeinsamen Teilpfad ab der Speicherzelle a_{23}.

An dieser Stelle möchte ich eine graphische Pfaddarstellung einführen. Diese soll vor allem der Intuition dienen. Es wird von exakten Adressen und Werten abstrahiert; wichtig ist lediglich, wie oft der Pfad in welcher Schicht bleibt und welche Endpfade von mehreren Variablen geteilt werden. Die Pfade sind von oben nach unten zu lesen. Die Abbildung

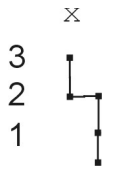

Abbildung 5.1: Pfad von x

5.1 zeigt den Pfad der Variablen x. Dieser enthält eine Adresse der dritten Schicht, zwei Adressen der zweiten Schicht, eine Adresse der ersten Schicht und schließlich einen Wert, den man durch einen Punkt ganz unten erkennt. In Abbildung 5.2 erkennt man, dass die

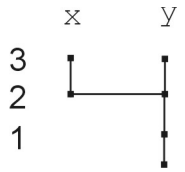

Abbildung 5.2: Gemeinsamer Teilpfad von x und y

Variablen x und y einen gemeinsamen Teilpfad haben. In den Speicherzellen der zweiten Schicht wird bei beiden Variablen die gleiche Adresse der dritten Schicht gespeichert.

Nach der Definition eines Zustands kommen wir zum Übergang von einem Zustand in einen anderen. Für eine gegebene Anweisung wird ein Zustand in einen neuen Zustand überführt. Dieser Übergang wird durch einen waagerechten Pfeil notiert. Über dem Pfeil steht die entsprechende Anweisung

$$s \xrightarrow{x_i =_j y} s'$$

bzw.

$$(s_1, s_2, s_3, s_4) \xrightarrow{x_i =_j y} (s_1', s_2', s_3', s_4').$$

Definition 6. *Gegeben sei ein Zustand* $s = (s_1, s_2, s_3, s_4)$.

a) Die $_3=_3$ *Zuweisung wird durch den folgenden Zustandsübergang definiert:*

$$(s_1, s_2, s_3, s_4) \xrightarrow{x_3 =_3 y} (s_1, s_2, s_3', s_4) \quad \text{mit}$$

$$s_3'(a) = \begin{cases} s_3(s_4(y)) & \text{falls } a = s_4(x) \\ s_3(a) & \text{sonst.} \end{cases}$$

b) Die $_2{=}_3$ *Zuweisung wird durch den folgenden Zustandsübergang definiert:*

$$(s_1, s_2, s_3, s_4) \xrightarrow{x\ {}_2{=}_3 y} (s_1, s_2', s_3, s_4) \quad \text{mit}$$
$$s_2'(a) = \begin{cases} s_3(s_4(y)) & \text{falls } a = s_3(s_4(x)) \\ s_2(a) & \text{sonst.} \end{cases}$$

c) Die $_2{=}_2$ *Zuweisung zwischen Variablen wird durch den folgenden Zustandsübergang definiert:*

$$(s_1, s_2, s_3, s_4) \xrightarrow{x\ {}_2{=}_2 y} (s_1, s_2', s_3, s_4) \quad \text{mit}$$
$$s_2'(a) = \begin{cases} s_2(s_3(s_4(y))) & \text{falls } a = s_3(s_4(x)) \\ s_2(a) & \text{sonst.} \end{cases}$$

d) Die $_1{=}_1$ *Zuweisung wird durch den folgenden Zustandsübergang definiert:*

$$(s_1, s_2, s_3, s_4) \xrightarrow{x\ {}_1{=}_1 y} (s_1', s_2, s_3, s_4) \quad \text{mit}$$
$$s_1'(a) = \begin{cases} s_1(s_2^j(s_3(s_4(y)))) & \text{falls } a = s_2^i(s_3(s_4(x))) \\ s_1(a) & \text{sonst.} \end{cases}$$

Jede dieser Zuweisungen schreibt genau einen Wert in genau eine Speicherzelle. s_4 als Bindung einer Variablen an einen Speicherplatz bleibt unbeeinflusst. Es wird jeweils eine der Funktionen s_1 bis s_3 geändert. Die Funktion s_i wird durch s_i' ersetzt. Darin entsprechen alle Werte außer einem von s_i' der Funktion s_i. Die Definitionen der verschiedenen s_i' haben folgenden Aufbau:

$$s_i'(a) = \begin{cases} h(y) & \text{falls } a = h'(x) \\ s_i(a) & \text{sonst.} \end{cases}$$

Die Funktion $h(y)$ liefert den Wert, der gelesen wird. $h'(x)$ zeigt, wohin dieser Wert geschrieben wird. h und h' sind durch mehrfache Anwendung der verschiedenen s_i's aufgebaut. Dabei kommen die Pfade von x bzw. y zum Ausdruck: Jede Anwendung von s_i liefert das nächste Element im Pfad. So wird bei der $_2{=}_3$ Zuweisung ein Element von A_2 in der dritten Schicht gelesen, denn

$$h(y) = s_3(s_4(y)),$$

und in die zweite Schicht geschrieben, denn

$$h'(x) = s_3(s_4(x)).$$

Der Wert wird beim ersten Erreichen der geforderten Schicht gelesen oder geschrieben. Die $_1{=}_1$ Zuweisung hat dabei die Besonderheit, dass es nicht klar ist, wie viele Schritte zum Erreichen der ersten Schicht notwendig sind, denn die Anwendung von s_2 kann wieder ein Element in A_2 ergeben. Der Pfad kann also lange in der zweiten Schicht

bleiben. Bei allen anderen Zuweisungen steht dagegen fest, nach wie vielen Schritten die jeweilige Schicht erreicht wird.

Die Zuweisungen erzeugen verschieden lange gemeinsame Teilpfade der beiden Variablen. Wenn die nächste Änderung einer der beiden Variablen in dem gemeinsamen Teilpfad stattfindet, werden beide Variablen von dieser Änderung beeinflusst. Sie bleiben wertgleich. Wenn eine Änderung im Pfad passiert, bevor der gemeinsame Teilpfad erreicht wird, ändert sich gewöhnlich der Pfad der betreffenden Variablen so, dass kein gemeinsamer Teilpfad und keine Abhängigkeit mehr existiert. Je früher der gemeinsame Teilpfad anfängt, desto stärker ist die Abhängigkeit der beiden Variablen.

Definition 7. *Zwei Variablen sind genau dann* gleichheitsabhängig, *wenn sie einen gemeinsamen nicht leeren Teilpfad haben.*

Bereits durch bloßes Betrachten der Definition 6 erkennt man, dass der gemeinsame Teilpfad bei der $_3{=}_3$ Zuweisung sehr früh anfängt und bei der $_1{=}_1$ Zuweisung sehr spät.

Während Definition 6 nur auf Zuweisungen zwischen Variablen eingeht, betrachtet Definition 8 die Zuweisung eines Wertes zu einer Variablen. Ausschließlich die $_2{=}_2$ Zuweisung mit Werten auf der rechten Seite ist möglich.

Definition 8. *Gegeben sei ein Zustand $s = (s_1, s_2, s_3, s_4)$. Sei $a_{1i} \in A_1$ die Adresse einer unbenutzten Speicherzelle, d.h. a_{1i} kommt nicht im Pfad einer Variablen vor.*

Die $_2{=}_2$ Zuweisung zwischen einer Variablen und einem Wert wird durch den folgenden Zustandsübergang definiert:

$$(s_1, s_2, s_3, s_4) \xrightarrow{x\ {}_2{=}_2 v} (s_1', s_2', s_3, s_4) \quad \text{mit}$$

$$s_1'(a) = \begin{cases} v & \text{falls } a = a_{1i} \\ s_1(a) & \text{sonst} \end{cases}$$

$$s_2'(a) = \begin{cases} a_{1i} & \text{falls } a = s_3(s_4(x)) \\ s_2(a) & \text{sonst.} \end{cases}$$

Bei dieser $_2{=}_2$ Zuweisung wird der Wert v in eine unbenutzte Speicherzelle geschrieben. Diese neue Speicherzelle a_{1i} wird in der zweiten Schicht referenziert. Dies zeige ich nun am folgenden Beispiel. Gegeben sei ein Zustand in Tabelle 5.6 und die Zuweisung

x $_2{=}_2$ v_1;

Der neue Zustand ist in Tabelle 5.7 präsentiert. In die unbenutzte Speicherzelle a_{11} wird v_1 geschrieben. In a_{22} im Pfad von x in der zweiten Schicht wird nun die Adresse a_{11} gespeichert.

5.3 Eigenschaften von Zuweisungen im 3-Schichtenmodell

Symmetrie und Stärke von Zuweisungen werden bereits in Unterkapitel 5.1 definiert und diskutiert. Hier werden jetzt die Zuweisungen des 3-Schichtenmodells auf Symmetrie und Stärke im Vergleich zueinander untersucht.

Schicht 3:	x		y
	$\mathbf{a_{31}}$	$\mathbf{a_{32}}$	$\mathbf{a_{33}}$
	a_{22}		a_{23}
Schicht 2:	$\mathbf{a_{21}}$	$\mathbf{a_{22}}$	$\mathbf{a_{23}}$
		a_{11}	a_{12}
Schicht 1:	$\mathbf{a_{11}}$	$\mathbf{a_{12}}$	$\mathbf{a_{13}}$
	v_1	v_3	

Tabelle 5.7: Zustand nach der Zuweisung x $_2{=}_2$ v_1

5.3.1 Symmetrie

Intuitiv gilt bei symmetrischen Zuweisungen Folgendes: Wenn zwischen x und y eine Abhängigkeit durch eine symmetrische Zuweisung

x $=_f$ y;

oder

y $=_f$ x;

hergestellt wird, gelten für die Auflösung oder das Beibehalten der Abhängigkeit in beiden Fällen ähnliche Bedingungen. Diese Symmetrie stammt daher, dass sich die Pfade der beiden Variablen in Bezug auf den gemeinsamen Teilpfad ähnlich verhalten. Die Abbildung 5.3 zeigt einen typischen Zustand nach einer symmetrischen Zuweisung. In diesem

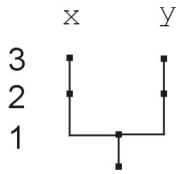

Abbildung 5.3: Symmetrische Zuweisung

Fall handelt es sich um die $_2{=}_2$ Zuweisung. Ein Zustand nach einer nicht symmetrischen Zuweisung, hier $_2{=}_3$ Zuweisung, ist in Abbildung 5.4 dargestellt. Wie man erkennt, ist

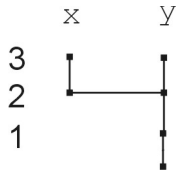

Abbildung 5.4: Unsymmetrische Zuweisung

hier der Anfangspfad von x bis zum gemeinsamen Teilpfad länger als der Pfad von y. D.h. es gibt Änderungen, die sich auf den Pfad von x anders auswirken, als sie sich auf den Pfad von y auswirken würden.

Lemma 1. $_1{=}_1$, $_2{=}_2$ *und* $_3{=}_3$ *sind* symmetrische *Zuweisungen.*

Beweis. Da die Beweise bei allen Zuweisungsoperatoren analog verlaufen, zeige ich hier nur die Symmetrie der $_2{=}_2$ Zuweisung. Laut Definition 2 ist Folgendes zu zeigen:

$$\forall s \text{ mit } (x,z),(y,z) \notin ker(s), f \in \{\,{}_3{=}_3,\ {}_2{=}_2,\ {}_2{=}_2,\ {}_1{=}_1\} :$$
$$(x,y) \in ker(f(x,z,\ {}_2{=}_2\ (x,y,s))) \Leftrightarrow (x,y) \in ker(f(x,z,\ {}_2{=}_2\ (y,x,s))).$$

Schauen wir uns den Spezialfall mit f als $_3{=}_3$ Zuweisung an. Dabei gilt

$$_3{=}_3\ (x,z,\ {}_2{=}_2\ (x,y,s)) =\ {}_3{=}_3\ (x,z,s')$$

mit

$$s' =\ {}_2{=}_2\ (x,y,s).$$

Ohne Beschränkung der Allgemeinheit seien die Pfade von x, y und z in s

$$x \xrightarrow[a_{31},a_{21},\ldots]{} v_1,$$
$$y \xrightarrow[a_{32},a_{22},a_{kl},\ldots]{} v_2,$$
$$z \xrightarrow[a_{33},a_{23},\ldots]{} v_3.$$

Dabei gelte $v_1 \neq v_3$ und $v_2 \neq v_3$. s' unterscheidet sich von s nur im Pfad von x. Die Adresse a_{kl} wird in die Speicherzelle a_{21} geschrieben (siehe Definition 6). Der Pfad von x in s' hat nun folgende Elemente:

$$x \xrightarrow[a_{31},a_{21},a_{kl},\ldots]{} v_2.$$

Als nächstes wird die Zuweisung

```
x 3=3 z;
```

ausgeführt. Wieder wird nur der Pfad von x geändert:

$$x \xrightarrow[a_{31},a_{23},\ldots]{} v_3.$$

Der Pfad von y bleibt gleich, denn die Änderung von x findet vor dem gemeinsamen Teilpfad statt. Nun ist der Wert von x v_3 und der von y v_2; die Werte unterscheiden sich:

$$(x,y) \notin ker(\ {}_3{=}_3\ (x,z,\ {}_2{=}_2\ (x,y,s))).$$

Analog gilt

$$(x,y) \notin ker(\ {}_3{=}_3\ (x,z,\ {}_2{=}_2\ (y,x,s)))$$

was zu zeigen war.

Schauen wir uns exemplarisch noch den Fall f als ${}_1{=}_1$ Zuweisung an. Dabei gilt

$$ {}_1{=}_1\ (x, z,\ {}_2{=}_2\ (x, y, s)) = {}_1{=}_1\ (x, z, s') $$

mit

$$ s' = {}_2{=}_2\ (x, y, s). $$

Ohne Beschränkung der Allgemeinheit seien die Pfade von `x`, `y` und `z` in s

$$ \begin{aligned} x &\xrightarrow[a_{31},a_{21},\ldots]{} v_1, \\ y &\xrightarrow[a_{32},a_{22},a_{kl},\ldots,a_{12}]{} v_2, \\ z &\xrightarrow[a_{33},a_{23},\ldots,a_{13}]{} v_3. \end{aligned} $$

Dabei gelte $v_1 \neq v_3$ und $v_2 \neq v_3$. Wieder unterscheidet sich s' von s nur im Pfad von `x`:

$$ x \xrightarrow[a_{31},a_{21},a_{kl},\ldots,a_{12}]{} v_2. $$

Als nächstes wird die Zuweisung

```
x 1=1 z;
```

ausgeführt. Diesmal betrifft die Änderung sowohl `x` als auch `y`. Denn die Änderung findet im gemeinsamen Teilpfad statt:

$$ \begin{aligned} x &\xrightarrow[a_{31},a_{21},a_{kl},\ldots,a_{12}]{} v_3, \\ y &\xrightarrow[a_{31},a_{21},a_{kl},\ldots,a_{12}]{} v_3. \end{aligned} $$

Also sind die Werte von `x` und `y` gleich und

$$ (x, y) \in ker(\ {}_1{=}_1\ (x, z,\ {}_2{=}_2\ (x, y, s))). $$

Analog zeigt man, dass

$$ (x, y) \in ker(\ {}_1{=}_1\ (x, z,\ {}_2{=}_2\ (y, x, s))). $$

Somit gilt

$$ (x, y) \in ker(\ {}_1{=}_1\ (x, z,\ {}_2{=}_2\ (x, y, s))) \Leftrightarrow (x, y) \in ker(\ {}_1{=}_1\ (x, z,\ {}_2{=}_2\ (y, x, s))). $$

□

Lemma 2. ${}_2{=}_3$ *ist eine* unsymmetrische *Zuweisung.*

Beweis. Hierzu reicht es, ein Beispiel anzugeben, in dem die Symmetrie verletzt wird. Ich zeige, dass

$$\exists x, y, z, s : (x, y) \notin ker(\ {}_2{=}_3\ (x, z,\ {}_2{=}_3\ (x, y, s)))$$
$$(x, y) \in ker(\ {}_2{=}_3\ (x, z,\ {}_2{=}_3\ (y, x, s))).$$

Dabei seien die Werte von x, y und z v_1, v_2 und v_3. Diese Werte sind paarweise verschieden. Nach den Zuweisungen

```
1  x  2=3  y;
2  x  2=3  z;
```

haben Werte von x und y unterschiedliche Werte, nämlich v_3 und v_2. Wenn man x und y in der ersten Zuweisung vertauscht:

```
1  y  2=3  x;
2  x  2=3  z;
```

haben x und y den gleichen Wert, nämlich v_3. □

Somit erlaubt das 3-Schichtenmodell sowohl symmetrische als auch unsymmetrische Zuweisungen.

5.3.2 Stärke

Intuitiv gilt Folgendes: Diejenige Zuweisung ist stärker, die einen kürzeren Anfangspfad vor dem gemeinsamen Pfad erzeugt. Je geringer die Gemeinsamkeit der Pfade zweier Variablen ist, desto weniger potenzielle Änderungen betreffen diese Anfangspfade und desto mehr potentielle Änderungen betreffen den gemeinsamen Teilpfad. Wenn eine Änderung im gemeinsamen Pfad stattfindet, bleiben die Variablen abhängig. Die Abbildungen 5.5 und 5.6 zeigen diesen Sachverhalt. Die Abhängigkeit in Abbildung 5.5 kann durch die ${}_3{=}_3$ Zuweisung hergestellt werden und die Abhängigkeit in Abbildung 5.6 durch die ${}_2{=}_2$ Zuweisung. Die erste Abhängigkeit ist stärker als die zweite. Eine Änderung in der zweiten Schicht kann die Abhängigkeit in Abbildung 5.5 nicht auflösen, dafür aber die Abhängigkeit in Abbildung 5.6.

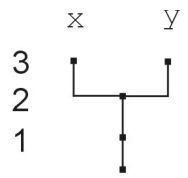

Abbildung 5.5: Stärkere Abhängigkeit

Wenn also zwischen x und y eine Abhängigkeit durch eine Zuweisung hergestellt wird, so dass die beiden Variablen den gleichen Wert haben, vermag eine schwächere Zuweisung diese Abhängigkeit nicht aufzulösen, d.h. die Werte von x und y bleiben gleich. Diese

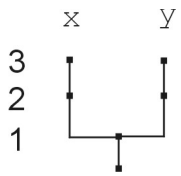

Abbildung 5.6: Schwächere Abhängigkeit

Abhängigkeit kann durch eine stärkere Zuweisung aufgelöst werden, so dass x und y wieder verschiedene Werte haben.

Die folgenden Lemmata 3 und 4 zeigen die Stärkebeziehungen zwischen den $_3{=}_3$, $_2{=}_2$, $_1{=}_1$ und $_2{=}_3$ Zuweisungen.

Lemma 3.

a) Die $_3{=}_3$ *Zuweisung ist stärker als die* $_2{=}_2$ *Zuweisung.*

b) Die $_3{=}_3$ *Zuweisung ist stärker als die* $_2{=}_3$ *Zuweisung.*

c) Die $_3{=}_3$ *Zuweisung ist stärker als die* $_1{=}_1$ *Zuweisung.*

d) Die $_2{=}_2$ *Zuweisung ist stärker als die* $_1{=}_1$ *Zuweisung.*

e) Die $_2{=}_3$ *Zuweisung ist stärker als die* $_1{=}_1$ *Zuweisung.*

Beweis. Exemplarisch zeige ich die Aussage b. Zu zeigen:

$$\forall x, y, z, s : (x, y) \in ker(\,_2{=}_3\,(x, z,\,_3{=}_3\,(x, y, s))),$$
$$\exists x, y, z, s : (x, y) \notin ker(\,_3{=}_3\,(x, z,\,_2{=}_3\,(x, y, s))).$$

Fangen wir mit der ersten Aussage an. Es gilt

$$_2{=}_3\,(x, z,\,_3{=}_3\,(x, y, s)) =\,_2{=}_3\,(x, z, s'))$$

mit

$$s' =\,_3{=}_3\,(x, y, s).$$

Ohne Beschränkung der Allgemeinheit seien die Pfade von x, y und z in s

$$x \xrightarrow[a_{31},a_{21},\ldots]{} v_1,$$
$$y \xrightarrow[a_{32},a_{22},\ldots]{} v_2,$$
$$z \xrightarrow[a_{33},a_{23},\ldots]{} v_3$$

und $v_1 \neq v_2$ sowie $v_2 \neq v_3$. s' unterscheidet sich von s im Pfad von x. Die Adresse a_{22} wird in die Speicherzelle a_{31} geschrieben (siehe Definition 6). Der Pfad von x in s' hat nun folgende Elemente:

$$x \xrightarrow[a_{31},a_{22},\ldots]{} v_2.$$

Als nächstes kommt die Zuweisung

```
x 2=3 z;
```

Hier wird a_{23} in die Speicherzelle a_{22} geschrieben. So wird sowohl der Pfad von x als auch der Pfad von y geändert, da die Änderung im gemeinsamen Teilpfad stattfindet, der bereits mit a_{22} startet:

$$x \xrightarrow[a_{31},a_{22},a_{23},\ldots]{} v_3,$$
$$y \xrightarrow[a_{32},a_{22},a_{23},\ldots]{} v_3.$$

Obwohl die Werte von x und y sich geändert haben, sind diese Variablen weiterhin wertgleich:

$$(x, y) \in ker(\ {}_2{=}_3\ (x, z,\ {}_3{=}_3\ (x, y, s))).$$

Zeigen wir nun die Aussage

$$\exists x, y, z, s : (x, y) \notin ker(\ {}_3{=}_3\ (x, z,\ {}_2{=}_3\ (x, y, s))).$$

Seien die Pfade von x, y und z wieder:

$$x \xrightarrow[a_{31},a_{21},\ldots]{} v_1,$$
$$y \xrightarrow[a_{32},a_{22},\ldots]{} v_2,$$
$$z \xrightarrow[a_{33},a_{23},\ldots]{} v_3.$$

Nach den Zuweisungen

```
1 x 2=3 y;
2 x 3=3 z;
```

ist der Wert von x v_3 und der von y v_2. Diesmal unterscheiden sich die Werte von x und y, da die in Zeile 2 verursachte Änderung außerhalb des gemeinsamen Pfades der Variablen x und y stattfindet.

□

Lemma 4. *Weder ist die* $_2{=}_2$ *Zuweisung stärker als die* $_2{=}_3$ *Zuweisung noch umgekehrt.*

Beweis. Die $_2{=}_2$ Zuweisung kann die $_2{=}_3$ Zuweisung überschreiben wie auch umgekehrt:

$$\exists x, y, z, s : (x, y) \notin ker(\ {}_2{=}_3\ (x, z,\ {}_2{=}_2\ (x, y, s))),$$
$$\exists x, y, z, s : (x, y) \notin ker(\ {}_2{=}_2\ (x, z,\ {}_2{=}_3\ (x, y, s))).$$

Seien die Pfade von x, y und z in s:

$$x \xrightarrow[a_{31},a_{21},a_{11}]{} v_1,$$
$$y \xrightarrow[a_{32},a_{22},a_{12}]{} v_2,$$
$$z \xrightarrow[a_{33},a_{23},a_{13}]{} v_3.$$

und $v_1 \neq v_2$ sowie $v_2 \neq v_3$. Nach den Zuweisungen

```
1  x  2=2  y;
2  x  2=3  z;
```

ist der Wert von x v_3 und der von y v_2. Im Fall der Zuweisungen

```
1  x  2=3  y;
2  x  2=2  z;
```

gilt ebenfalls, dass der Wert von x v_3 und der Wert von y v_2 ist. □

Es stellt sich also heraus, dass $_3{=}_3$ der stärkste Zuweisungsoperator und $_1{=}_1$ der schwächste ist. Die beiden Zuweisungsoperatoren $_2{=}_2$ und $_2{=}_3$ befinden sich dazwischen. Dabei kann man nicht sagen, welcher der beiden stärker ist. Es handelt sich um eine Halbordnung (siehe das Hasse-Diagramm in Abbildung 5.7) [Has80].

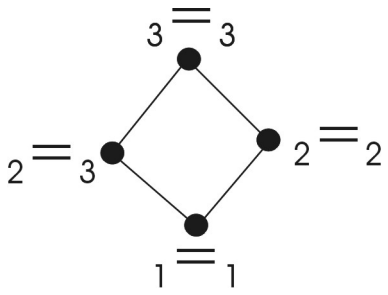

Abbildung 5.7: Stärke der Zuweisungen

Wie bereits gesagt, hängt die Stärke einer Zuweisung unmittelbar mit dem hergestellten gemeinsamen Teilpfad der beiden Variablen zusammen. Wenn die Änderung im gemeinsamen Pfad stattfindet, haben die Variablen weiterhin den gleichen Wert, da durch ihre Gemeinsamkeit beide Pfade trotz der Änderung gleich enden. Wenn die Änderung vor dem gemeinsamen Pfad stattfindet, wird die Abhängigkeit der Variablen aufgelöst. Denn durch die Änderung eines Pfades besitzen die Variablen keinen gemeinsamen Teilpfad mehr. Entsprechend erzeugt die $_3{=}_3$ Zuweisung den längsten gemeinsamen Teilpfad, während die $_1{=}_1$ Zuweisung den kürzesten erzeugt.

Das 3-Schichtenmodell ermöglicht also 3 symmetrische Zuweisungen von drei verschiedenen Stärken $_3{=}_3$, $_2{=}_2$ und $_1{=}_1$ und eine unsymmetrische Zuweisung $_2{=}_3$.

5.4 Erweiterungen des 3-Schichtenmodells

Die vorherigen Unterkapitel behandeln Abhängigkeiten zwischen *einfachen* Variablen (siehe Definition 9). Dieses Unterkapitel zeigt, wie die Regeln auf Arrays und Objekte erweitert werden können.

Definition 9. *Eine Variable ist* einfach, *wenn nur die ersten beiden Elemente ihres Pfades in der dritten Schicht gespeichert sind.*

Eine Variable ist zusammengesetzt, *wenn außer den ersten beiden Elementen ihres Pfades mindestens ein weiteres Element in der dritten Schicht gespeichert ist.*

5.4.1 Arrays

Die zusätzliche Syntax für Arrays entspricht dem Standard und wird aus [AFF99] übernommen. Die Definition der Syntax ist zwar nicht kompliziert, aber die Angabe aller Regeln ist sehr umfangreich und für diese Arbeit nicht von entscheidender Bedeutung. Sei `x` ein Array der Länge n. Syntaktisch wird der Zugriff auf einen Index m mit `x[m]` ausgedrückt. Die Variable `x` ist wie üblich in der dritten Schicht gespeichert und zeigt auf eine Adresse der zweiten Ebene a_{2j}. Die Neuerung besteht darin, dass der Pfad von a_{2j} wieder in die dritte Schicht zur Speicherzelle a_{3i} führt. Dort sind n Speicherplätze $a_{3i}, \ldots, a_{3(i+n-1)}$ für dieses Array reserviert. Die Variable `x[m]` für $0 \leq m \leq n-1$ ist in a_{3i+m} gespeichert, d.h. $x[0]$ in a_{3i}, $x[1]$ in $a_{3(i+1)}$, $x[2]$ in $a_{3(i+2)}$, usw. Der Pfad ab der Speicherzelle a_{3i+m} kann einem Pfad einer einfachen Variablen entsprechen (siehe Definition 4).

Schicht 3:	x			y		
	$\mathbf{a_{31}}$	$\mathbf{a_{32}}$	$\mathbf{a_{33}}$	$\mathbf{a_{34}}$	$\mathbf{a_{35}}$	$\mathbf{a_{36}}$
	a_{22}	a_{21}	a_{23}	a_{24}	a_{25}	a_{26}
Schicht 2:	$\mathbf{a_{21}}$	$\mathbf{a_{22}}$	$\mathbf{a_{23}}$	$\mathbf{a_{24}}$	$\mathbf{a_{25}}$	$\mathbf{a_{26}}$
	a_{23}	a_{35}	a_{12}	a_{32}	a_{11}	a_{21}
Schicht 1:	$\mathbf{a_{11}}$	$\mathbf{a_{12}}$	$\mathbf{a_{13}}$	$\mathbf{a_{14}}$	$\mathbf{a_{15}}$	$\mathbf{a_{16}}$
	v_4	v_3	v_1	v_4	v_2	v_5

Tabelle 5.8: Beispiel für Arrays

Schauen wir uns ein Beispiel mit Arrays `x` und `y` der Länge 2 an (siehe Tabelle 5.8). Für `x` sind die Speicherzellen a_{35} und a_{36} reserviert, die den Variablen `x[0]` und `x[1]` entsprechen, da der Pfad von `x` zur Speicherzelle a_{35} führt. Entsprechend ist `y[0]` in a_{32} und `y[1]` in a_{33} gespeichert. Bei dem gegebenen Zustand ist der Wert von `x[0]` v_4, von `x[1]` v_3, von `y[0]` v_3 und von `y[1]` v_3. Die Variablen `x` und `y` ohne Index haben dagegen keinen Wert.

Ein Array ist eine zusammengesetzte Variable (siehe Definition 9). Wenn der Pfad eines Arrays die zweite Schicht verlässt, geht er nicht in der ersten, sondern in der dritten Schicht weiter. Beim Pfad eines Elementes eines Arrays kommt vorne ein Teilpfad dazu. Dieser startet in der dritten Schicht z.B. bei `x` und endet ebenfalls in der dritten Schicht z.B. bei `x[m]`. Der Pfad ab `x[m]` entspricht einem Pfad einer weiteren Variablen. Eine einfache Variable hat einen Wert, während eine zusammengesetzte Variable keinen Wert besitzt. Man beachte, dass Definition 4 nicht einen Pfad von einer zusammengesetzten Variablen zu ihrem Wert beschreibt, da in dieser Definition der Pfad beim Verlassen der zweiten Schicht nicht in die dritte Schicht zurückkehren kann.

Schauen wir uns die Zuweisungen zwischen Arrayelementen zunächst am Beispiel an. Gegeben sei der Zustand in Tabelle 5.8. Da Elemente dieser Arrays einfache Variablen

sind, können die Zuweisungen zwischen diesen Elementen wie in Definition 6 ausgeführt werden. Nur müssen die Elemente durch einen zusätzlichen Anfangspfad gefunden werden. So führt z.B. die Zuweisung

```
x[1] 3=3 y[0];
```

dazu, dass in die Speicherzelle a_{36} der Wert a_{21} geschrieben wird. Bei der Zuweisung

```
y[0] 2=2 x[0];
```

wird a_{11} in die Speicherzelle a_{21} geschrieben.

Um keine neuen Zuweisungsregeln für Arrayelemente zu entwickeln, werden Arrayelemente syntaktisch durch Hilfsvariablen ersetzt. Zuerst veranschauliche ich diesen Prozess an einem Beispiel. Sei `x` ein Array und und `y` und `z` einfache Variablen. Aus einem Codeblock

```
1 x[4] 2=2 y;
2 z 2=2 x[2];
```

entsteht

```
1 var1 2=2 y;
2 z 2=2 var2;
```

mit neuen Variablen var_1 und var_2. Um sicher zu stellen, dass var_1 sich genauso verhält wie $x[4]$ und var_2 wie $x[2]$, werden implizit die Zuweisungen

```
1 var1 3=3 x[4];
2 var2 3=3 x[2];
```

vorgenommen.

Die folgende Definition beschreibt formal diese Ersetzung.

Definition 10. *Gegeben sei der Zustand $s = (s_1, s_2, s_3, s_4)$. Sei* `x` *ein Array und* `m` *ein Index in diesem Array. Weiterhin sei $var \in Vars$ eine neue Variable und $s_4(var)$ eine unbenutzte Speicherzelle. Die folgende Umformung, genannt* Dereferenzierung, *findet statt:*

a) Syntaktisch wird `x[m]` *durch* `var` *ersetzt.*

b) Der Zustand s geht in den Zustand $s' = (s_1, s_2, s'_3, s_4)$ über mit

$$s'_3(a) = \begin{cases} a_{2i+m} & \text{mit} \quad a_{2i} = s_3(s_4(x)) \quad \text{falls } a = s_4(var) \\ s_3(a) & \qquad\qquad\qquad\qquad\quad \text{sonst.} \end{cases}$$

Auf diese Weise kann man in k Schritten auf Elemente k-dimensionaler Arrays durch die Einführung von k neuen Variablen zugreifen. Aus

```
x[2][3] = y;
```

wird

```
var1[3] = y;
```

und daraus wird wiederum

var_2 = y;

Implizit werden die folgenden Zuweisungen vorgenommen:

1 var_1 $_3{=}_3$ x[2];
2 var_2 $_3{=}_3$ var_1[3];

Wenn Elemente eines Arrays einfache Variablen sind, so werden diese durch einfache Hilfsvariablen ersetzt. Die Zuweisungssemantik, die bereits für einfache Variablen definiert ist, kann auf diese Weise auf Elemente von Arrays angewendet werden.

Zur Performanceverbesserung kann man Speichermechanismen wie Heaps und Frame Stacks nutzen. In dieser Arbeit wird davon abstrahiert, da es um die Beschreibung der Konzepte geht.

5.4.2 Objekte

Bei Objekten verhält es sich ähnlich wie bei Arrays. Während bei Arrays auf die einzelnen Variablen durch Indizes zugegriffen werden kann, geschieht dies bei Objekten durch Qualifikation der Attributnamen. Sei x ein Objekt und $f_1, \ldots, f_n$ seine Attribute. Mit $x.f_i$ greift man auf das Attribut f_i des Objekts x zu. f steht dabei für *field*. Üblicherweise haben Attribute aussagekräftigere Namen als f_i. Es ist jedoch keine Beschränkung der Allgemeinheit, wenn ich die Attribute durchnummeriere und nur von Attributen f_1 bis f_n spreche. Wie bei Arrays wird die Syntax von Objekten aus Java übernommen [AFF99], da die Angabe aller Regeln zu umfangreich ist.

Die Speicherung und der Zugriff auf Objekte findet analog zu Arrays statt. Das Objekt x wird in der dritten Schicht gespeichert und zeigt auf die zweite Schicht. Der Pfad führt von der dritten Schicht in die zweite und wieder zurück in die dritte Schicht zur Speicherzelle a_{3i}. Dort sind n Speicherzellen für Attribute $f_1, \ldots, f_n$ reserviert. Das k-te Attribut befindet sich in der Speicherzelle $a_{3(i+k-1)}$. Der Pfad vom Attribut $x.f_k$ ist wiederum ein Pfad einer Variablen.

Schauen wir uns folgendes Beispiel an. Gegeben sei die Klasse ListElement:

```
1 class ListElement {
2   int content;
3   ListElement next;
4 }
```

Dabei ist content das erste und next das zweite Attribut.

Ein beispielhafter Speicherzustand ist in Tabelle 5.9 dargestellt. Dieser Zustand ist graphisch in Abbildung 5.8 abgebildet. Wie bei Arrays handelt es sich auch bei Objekten um zusammengesetzte Variablen. Die Variable y hat 1 als value und als next ein Objekt, dass ebenfalls von x referenziert wird. Dieses Objekt hat seinerseits als value 2 und als next den Wert null. Der Pfad von y zum Wert von y.next.value verläuft über Speicherzellen a_{34}, a_{24}, a_{36}, a_{26}, a_{21}, a_{32}, a_{22}, a_{11} und endet bei 2. Wie üblich startet der Pfad von y in a_{34} und führt zu a_{24}. Nach der Speicherzelle a_{24} führt der Pfad wieder in die dritte Schicht. Obwohl der Inhalt von a_{24} a_{35} lautet, ist das nächste

Schicht 3:

x			y		
a_{31}	**a_{32}**	**a_{33}**	**a_{34}**	**a_{35}**	**a_{36}**
a_{21}	a_{22}	a_{23}	a_{24}	a_{25}	a_{26}

Schicht 2:

a_{21}	**a_{22}**	**a_{23}**	**a_{24}**	**a_{25}**	**a_{26}**
a_{32}	a_{11}	a_{12}	a_{35}	a_{13}	a_{21}

Schicht 1:

a_{11}	**a_{12}**	**a_{13}**	**a_{14}**	**a_{15}**	**a_{16}**
2	null	1			

Tabelle 5.9: Beispiel für Objekte

Element des Pfades a_{36}. Das liegt daran, dass es sich dabei um das Attribut `next` handelt, welches das zweite Attribut des Objekts ist. Das nächste Pfadelement ist somit $a_{3(5+2-1)}$, also a_{36}. Danach bleibt der in der zweiten Schicht und verläuft über die Speicherzelle Speicheradresse a_{21}, denn es handelt sich um ein Listenelement, das referenziert wird und nicht um einen üblichen Wert wie v_i. Nun muss das Attribut `value`, also das erste Attribut erreicht werden. Also ist das nächste Element des Pfades $a_{3(2-1+1)}$. Schließlich führt der Pfad von a_{32} zum Wert `2`.

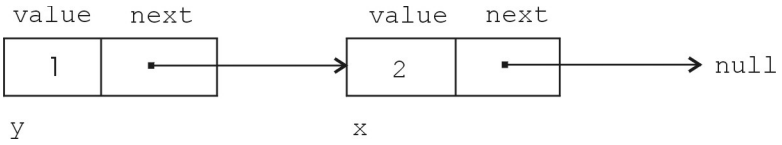

Abbildung 5.8: Liste

In Abbildung 5.8 bedeutet `null`, dass kein drittes Element existiert. Dabei ist *null* $\in$ *Vals* ein ausgezeichneter Wert, der das Fehlen eines Wertes bezeichnet. Wie jeder andere Wert wird `null` in der ersten Schicht gespeichert.

Die Ähnlichkeit bei der Speicherung von Objekten und Arrays wird durch eine Umformung von Objekten zu Arrays ausgenutzt. So entspricht das Attribut f_i eines Objekts `x` einem Element des Arrays `x` mit dem Index $i-1$. Im Quelltext wird jede Qualifikation `x.`f_i durch `x[i-1]` ersetzt.

Definition 11. *Sei* `x` *ein Objekt und* f_i *das i-te Attribut von* `x`*. Bei der Umwandlung eines Objekts zu einem Array wird* `x.`f_i *durch* `x[i-1]` *ersetzt.*

Im Gegensatz zur Definition 10 wird hier nur die Syntax der Anweisungen, nicht aber der Zustand verändert.

5.4.3 Zuweisungen bei zusammengesetzten Variablen

In Abschnitt 5.2.3 werden Zuweisungen für einfache Variablen definiert. Danach wird in Abschnitten 5.4.1 und 5.4.2 auf Zuweisungen für Arrayelemente und Objektattribute eingegangen, soweit es sich dabei wieder um einfache Variablen handelt. Nun wird die Zuweisungssemantik zwischen zusammengesetzten Variablen besprochen.

Die Zuweisungen $_3{=}_3$, $_2{=}_2$ und $_2{=}_3$ funktionieren wie bei einfachen Variablen in Definition 6. Für diese Zuweisungsarten gilt, dass genau ein Wert in genau eine Speicherzelle geschrieben wird. Seien `x` und `y` Objekte aus Tabelle 5.10. So bewirkt z.B.

```
x 2=2 y;
```

dass a_{32} in a_{22} geschrieben wird. Die Analogie zu einfachen Variablen bei diesen Zuweisungsarten stammt daher, dass es keinen Unterschied gibt, solange die zweite Schicht des Pfades nicht verlassen wird.

Die Zuweisung $_1{=}_1$ nimmt eine Sonderrolle ein, da die erste Schicht miteinbezogen ist. Der Pfad zur ersten Schicht unterscheidet sich bei einfachen und zusammengesetzten Variablen. Insbesondere hat eine zusammengesetzte Variable keinen Wert (siehe Abschnitt 5.4.1). Also kann ihr Wert nicht kopiert werden. Die Wertzuweisungsregel aus Definition 6 kann nicht auf zusammengesetzte Variablen angewandt werden, da der Pfad der zusammengesetzten Variablen von der zweiten Schicht in die dritte und nicht in die erste Schicht führt.

Definition 12. *Seien* `x` *und* `y` *Arrays der Länge n:*

$$\texttt{x}\ {}_1{=}_1\ \texttt{y} \Leftrightarrow \forall i \text{ mit } 0 \leq i < n : \texttt{x[i]}\ {}_2{=}_2\ \texttt{y[i]}.$$

Jedes einzelne Element eines Arrays wird kopiert. Analog führt das bei Objekten dazu, dass jedes Attribut einzeln kopiert wird. Die $_1{=}_1$ Zuweisung agiert ähnlich wie Zuweisungen bei Werttypen in anderen Programmiersprachen, z.B. Structs in C# [RNW+03].

Schicht 3:

x			y		
a_{31}	**a_{32}**	**a_{33}**	**a_{34}**	**a_{35}**	**a_{36}**
a_{22}	a_{21}	a_{23}	a_{24}	a_{25}	a_{26}

Schicht 2:

a_{21}	**a_{22}**	**a_{23}**	**a_{24}**	**a_{25}**	**a_{26}**
a_{13}	a_{35}	a_{12}	a_{32}	a_{11}	a_{16}

Schicht 1:

a_{11}	**a_{12}**	**a_{13}**	**a_{14}**	**a_{15}**	**a_{16}**
v_4	v_3	v_1	v_4	v_2	v_5

Tabelle 5.10: Startzustand

Tabelle 5.10 beschreibe den Startzustand des Speichers bei jeder der folgenden Zuweisungen, wobei `x` und `y` Arrays der Länge 2 sind. Bei der Zuweisung

`x` $_3{=}_3$ `y;`

wird wie gewohnt a_{24} in die Speicherzelle a_{31} geschrieben. Bei

`x` $_2{=}_3$ `y;`

wird a_{24} in a_{22} geschrieben. Bei der Zuweisung

`x` $_1{=}_1$ `y;`

werden entsprechend der Definition 12 zwei Werte kopiert. Der erste Speicherzelleninhalt in der zweiten Schicht auf dem Pfad von `y[0]`, nämlich a_{13}, wird in die Speicherzelle der zweiten Schicht von `x[0]`, nämlich a_{25}, geschrieben. Außerdem wird a_{12} in a_{26} kopiert. Nach jeder der durchgeführten Zuweisungen gilt, dass der Wert von `x[0]` dem Wert von `y[0]` gleicht, und der von `x[1]` dem von `y[1]`.

Ein Sonderfall ist eine $_1{=}_1$ Zuweisung, bei der die Variable auf der linken Seite auf `null` zeigt, da bisher kein Platz für die zu speichernden Attribute reserviert ist. In diesem Fall soll der Platz für Attribute mit unbenutzten Speicherzellen angelegt werden.

Bestimmte Objekte dürfen nicht kopiert werden und somit nicht auf der rechten Seite einer $_1{=}_1$ Zuweisung stehen. Das gilt z.B. für Datenbankobjekte. Diese haben einen Primärschlüssel, der einmalig ist [KE99]. Sollte so ein Objekt inklusive des Schlüssels kopiert werden, ginge diese Einmaligkeit verloren; d.h. die $_1{=}_1$ Zuweisung muss entweder bei solchen Objekten verboten sein, oder die Funktionalität der Zuweisung muss für diese Objekte überschrieben werden.

5.4.4 Parameterübergabe

Wie bereits in Abschnitt 2.2.7 und Unterkapitel 4.2 dargestellt, werden bei der Parameterübergabe implizite Zuweisungen zwischen den formalen und den aktuellen Parametern ausgeführt. Es ist eine natürliche Folge der bisherigen Überlegungen, auch die impliziten Zuweisungen zu parametrisieren. Da die Semantik der Zuweisungen bereits definiert ist, wird hier nur die Syntax für Parametrisierung bei der Parameterübergabe eingeführt.

Wieder wird die Syntax für Typen und Methoden aus Java übernommen [AFF99]. Die Änderungen finden nur bei den Parametern statt, nämlich beim Methodenaufruf und der Methodendeklaration. Nehmen wir als Beispiel die folgende Methodendeklaration

```
int method(int x) {...}
```

Diese Methode wird mit dem aktuellen Parameter `y` aufgerufen

```
int z = method(y);
```

Die implizite Zuweisung ist dabei

```
x = y;
```

Diese soll für passende i und j durch

`x` $_i{=}_j$ `y;`

ersetzt werden.

Wie bei der expliziten Zuweisung kommt der Index i nach `x`

`int method(int` x_i`) {...}`

und der Index j vor `y`

`int z = method(`$_j$`y);`

Tabelle 5.11 zeigt die Produktionen für formale und aktuelle Parameter. Die Schreibweise ohne Index bedeutet die Standardeinstellung, nämlich den Index 2.

fpar	::=	*type var*
	|	*type* var_1
	|	*type* var_2
	|	*type* var_3
apar	::=	*var*
	|	$_1var$
	|	$_2var$
	|	$_3var$

Tabelle 5.11: Syntax der formalen und aktuellen Parameter

5.5 Typsicherheit

5.5.1 Anforderungen

Die parametrisierten Zuweisungen können sowohl in einer statisch typisierten als auch in einer dynamisch typisierten Programmiersprache eingebunden werden. Der Schwerpunkt dieser Arbeit liegt auf statisch typisierten Sprachen (siehe Abschnitt 2.1.1). Eine zentrale Anforderung an solche Programmiersprachen ist die Typsicherheit.

Informell bedeutet Typsicherheit, dass keine Typfehler während der Laufzeit an den Stellen auftreten, wo sie nicht erwartet werden [DE99]. Bei der Qualifikation einer Methode bzw. eines Attributs darf es nicht vorkommen, dass diese Methode bzw. dieses Attribut im Objekt nicht vorhanden ist. Die einzigen Stellen in einem Programm, an denen Typfehler erwartet werden, sind bestimmte Typumwandlungen, insbesondere Downcasts. Bei einem Upcast dagegen darf kein Typfehler auftreten. Dabei tritt die Oberklasse anstelle der Unterklasse auf. Die is-a Beziehung der Vererbung besagt, dass ein Objekt der Unterklasse immer ein Objekt der Oberklasse ist (siehe Abschnitt 2.1.1).

Die $_2{=}_2$ Zuweisung ist die Standardzuweisung. Die anderen Zuweisungen werden nur in besonderen Situationen eingesetzt (siehe Kapitel 4 und 6). Um mit den existierenden Programmiersprachen konform zu bleiben, müssen bei der Standardzuweisung sowohl Up- als auch Downcasts erlaubt sein. Man nimmt heute an, dass die gängigen statisch typisierten Programmiersprachen typsicher sind. Für große Teile dieser Programmiersprachen ist dies bewiesen [DE99, Sym99, ON99, DDM07]. Die Anforderung besteht also darin, dass bei der $_2{=}_2$ Zuweisung sowohl Up- als auch Downcasts zugelassen sind, und die Typsicherheit erhalten bleibt.

Es ist wünschenswert, dass auch bei allen anderen Zuweisungen Up- und Downcasts möglich sind. Jedoch zeigt dieses Unterkapitel, dass die Typsicherheit bei vielen Typumwandlungen verletzt wird. Es gilt also, so viele Typumwandlungen wie möglich zuzulassen und dabei die Typsicherheit zu bewahren.

5.5.2 Verbotene und erlaubte Typumwandlungen

Zunächst zeige ich, dass sich bestimmte Typumwandlungen bei verschiedenen Zuweisungsarten nicht vertragen, also die Typsicherheit verletzen.

Lemma 5. *Seien sowohl Up- als auch Downcasts bei der $_2$=$_2$ Zuweisung erlaubt.*

a) Der Downcast bei der $_3$=$_3$ Zuweisung verletzt die Typsicherheit.

b) Der Upcast bei der $_3$=$_3$ Zuweisung verletzt die Typsicherheit.

Beweis. Sei `SuperType` eine Oberklasse von `SubType`. Die Klasse `SubType` enthalte die zusätzliche Methode `method`.

Zuerst zeige ich, dass der Downcast bei der $_3$=$_3$ Zuweisung zur unerwarteten Ausnahme führt. Betrachten wir nun folgenden Codeabschnitt:

```
1 SubType sub 2=2 new SubType();
2 SuperType sup 2=2 new SubType();
3 sub 3=3 (SubType) sup;
4 sup 2=2 new SuperType();
5 sub.method();
```

Dieser Code ist syntaktisch richtig. In Zeile 2 findet ein Upcast und in Zeile 3 ein Downcast statt. Bei den übrigen Zuweisungen entspricht der statische Typ der Variablen auf der linken Seite dem Typ auf der rechten Seite.

Der Downcast in Zeile 3 verläuft erfolgreich, da die Variable `sup` den dynamischen Typ `SubType` hat. Die Zuweisung in Zeile 4 führt dazu, dass nicht nur die Variable `sup` einen neuen Wert erhält, sondern implizit auch `sub`. `sub` hat nach dieser Zeile den dynamischen Typ `SuperType`. Somit führt der Aufruf der Methode `method` zu einer Ausnahme, da es keine Methode mit dem Namen `method` in `SuperType` gibt. Die Ausnahme wird an dieser Stelle nicht erwartet. Die Typisierung erfüllt hier nicht ihren Hauptzweck: Es wird nicht festgestellt, ob eine Methode von einem Objekt aufgerufen werden kann.

Um zu zeigen, dass auch beim Upcast der $_3$=$_3$ Zuweisung eine unerwartete Ausnahme geworfen werden kann, ersetzt man den Downcast in Zeile 3 durch den Upcast:

```
sup 3=3 sub;
```

Dies führt wieder zur Ausnahme in Zeile 5, da in `SuperType` keine Methode mit dem Namen `method` existiert. □

In C++ existiert eine ähnliche Problematik, da in dieser Programmiersprache sowohl c- als auch j-Referenzen zugelassen sind. Während bei j-Referenzen Typumwandlungen

erlaubt sind, ist dies bei c-Referenzen nicht der Fall [Str97]. Das Verbot der Typumwandlungen bei der $_3{=}_3$ Zuweisung kann in Analogie zum Verbot bei c-Referenzen gesehen werden.

Lemma 6. *Seien sowohl Up- als auch Downcasts bei der $_2{=}_2$ Zuweisung erlaubt.*

a) Der Downcast bei der $_2{=}_3$ Zuweisung verletzt die Typsicherheit.

b) Der Upcast bei der $_2{=}_3$ Zuweisung verletzt die Typsicherheit nicht.

Beweis. Der Beweis der Verletzung beim Downcast verläuft analog zum Lemma 5. Wenn im Codeblock aus dem Beweis dieses Lemmas die Zeile 3 durch die folgende Zuweisung ersetzt wird,

```
sub _2=_3 (SubType) sup;
```

gibt es ebenfalls eine unerwartete Ausnahme in Zeile 5.

Um die Typsicherheit beim Upcast zu zeigen, betrachten wir den folgenden Codeblock.

```
1 SubType sub;
2 SuperType sup;
3 ...
4 sup _2=_3 sub;
5 ...
6 sub.method();
```

Nach dem Upcast in Zeile 4 sind die Variablen `sup` und `sub` gleichheitsabhängig. Jede weitere $_2{=}_2$ oder $_2{=}_3$ Zuweisung zu `sup` löst die Abhängigkeit zwischen `sup` und `sub` auf. Wenn `sup` also ein Objekt von einem anderen Typ zugewiesen bekommt, referenziert `sub` weiterhin das Objekt vom dynamischen Typ `SubType`, der die Methode `method` enthält. Es kann also zu keiner Typausnahme in Zeile 6 kommen. □

Lemma 7. *Seien sowohl Up- als auch Downcasts bei der $_2{=}_2$ Zuweisung erlaubt. Wenn bei der $_1{=}_1$ Zuweisung auf der linken und der rechten Seite Objekte verschiedener dynamischer Typen stehen, wird die Typsicherheit verletzt.*

Beweis. Sei `SuperType` eine Oberklasse von `SubType`. Die Klasse `SubType` enthalte die zusätzliche Methode `method`. Betrachten wir nun den folgenden Codeabschnitt:

```
1 SubType sub _2=_2 new SubType();
2 SuperType sup1 _2=_2 new SuperType();
3 SuperType sup2 _2=_2 sub;
4 sup2 _1=_1 sup1;
5 sub.method();
```

Durch die Zuweisung in Zeile 3 sind die Variablen `sup2` und `sub` abhängig. Die Zuweisung ist gültig, da es sich um einen Upcast handelt. Die Zuweisung in Zeile 4 löst diese Abhängigkeit nicht auf, da die $_1{=}_1$ Zuweisung schwächer ist als die $_2{=}_2$ Zuweisung. Also referenzieren nach dieser Zeile sowohl `sup2` als auch `sub` ein Objekt vom Typ `SuperType`, der die Methode `method` nicht hat. Also kommt es zu einer Ausnahme in Zeile 5. □

Im Gegensatz zu anderen Zuweisungsarten ist bei der $_1{=}_1$ Zuweisung nicht der statische, sondern der dynamische Typ entscheidend. Syntaktisch können auf der rechten wie auch auf der linken Seite Variablen sowohl der Oberklasse als auch der Unterklasse stehen. Beim Compilieren ist nur erforderlich, dass der eine Typ in der transitiven Hülle des anderen ist:

$$(LeftType \leq RightType) \vee (RightType \leq LeftType).$$

Somit birgt jede $_1{=}_1$ Zuweisung die Gefahr der Typunverträglichkeit insofern, als der dynamische Typ auf der linken sich von dem Typ auf der rechten Seite unterscheidet. Daher kann bei jeder $_1{=}_1$ Zuweisung ähnlich wie bei jedem Downcast bei der $_2{=}_2$ Zuweisung eine Typverletzung auftreten. Wie in vielen Programmiersprachen üblich, wird bei der Typverletzung eine `ClassCastException` geworfen. Diese Ausnahme kann bei einer Erweiterung um parametrisierte Zuweisungen nur in den genannten beiden Fällen auftreten.

Die Tatsache, dass bei der $_1{=}_1$ Zuweisung die dynamischen Typen der Objekte gleich sind, erleichtert das Kopieren der Objekte, da für das Objekt auf der linken Seite der Zuweisung genauso viel Speicherplatz allokiert ist wie für das Objekt auf der rechten Seite.

Nachdem gezeigt ist, welche Typumwandlungen sich nicht vertragen, beinhaltet das folgende Lemma alle erlaubten Typumwandlungen. Sollten also parametrisierte Zuweisungen in einer statisch typisierten objektorientierten Sprache eingeführt werden, so sollten genau die zunächst genannten Casts zugelassen sein.

Satz 1. *Eine statisch typisierte, objektorientierte Sprache mit parametrischen Zuweisungen ist typsicher, wenn die folgenden Bedingungen für die Zuweisungen gelten:*

a) Bei der $_3{=}_3$ Zuweisung stimmen die statischen Typen auf der linken und rechten Seite der Zuweisung überein.

b) Bei der $_2{=}_3$ Zuweisung sind Upcasts zugelassen, Downcasts dagegen nicht.

c) Bei der $_1{=}_1$ Zuweisung stimmen die dynamischen Typen auf der linken und rechten Seite der Zuweisungen überein. Ist das nicht der Fall, wird eine Ausnahme geworfen.

d) Bei der $_2{=}_2$ Zuweisung sind sowohl Up- als auch Downcasts zugelassen. Beim Downcast wird eine Ausnahme geworfen, wenn der dynamische Typ nicht in der transitiven Vererbungshülle des statischen ist.

Beweis. Nehmen wir an, dass in einem gegebenen Zustand für jede Variable var_i mit dem dynamischen Typ D_{var_i} und dem statischen Typ S_{var_i} gilt:

$$D_{var_i} \leq S_{var_i}$$

Es wird gezeigt, dass nach jeder folgenden Zuweisung entweder ein erwarteter Typfehler erzeugt wird, oder die Invariante

$$D_{var_i} \leq S_{var_i}$$

weiterhin gilt.

a) Betrachten wir zuerst die Zuweisung

```
x 3=3 y;
```

Die statischen Typen der beiden Variablen sind gleich. Da der dynamische Typ von `y` wegen der Annahme keine Oberklasse des statischen Typs ist, gilt dies auch für die Typen von `x` nach der Zuweisung.

b) Vor der Zuweisung

```
x 2=3 y;
```

gilt wegen der Voraussetzung

$$D_x \leq S_x$$

und

$$D_y \leq S_y$$

Dadurch, dass nur Upcasts erlaubt sind, gilt

$$S_y \leq S_x.$$

Nach der Zuweisung hat `x` den dynamischen Typ von `y`. Also gilt

$$D_x \leq S_x.$$

c) Kommen wir nun zu der $_1{=}_1$ Zuweisung. Sollten die dynamischen Typen der Variablen nicht übereinstimmen, gibt es einen erwarteten Typfehler. Ansonsten ändert sich der dynamische Typ nicht. Somit gilt die Invariante auch nach der $_1{=}_1$ Zuweisung.

d) Zum Schluss schauen wir uns die $_2{=}_2$ Zuweisung an. Die Argumentation bei dem Upcast verläuft analog zum Upcast bei der $_2{=}_3$ Zuweisung. Beim Downcast dagegen tritt ein Typfehler auf, falls die Invariante verletzt wird. Ansonsten läuft das Programm weiter, und die Invariante bleibt erhalten.

□

Tabelle 5.12 stellt zusammen, wann Upcasts und wann Downcasts möglich sind. Besondere Bedeutung spielt dabei die $_1{=}_1$ Zuweisung, da es allein um die Gleichheit der dynamischen Typen geht. Die statischen Typen spielen keine Rolle, so dass es nicht wichtig ist, ob ein Downcast oder Upcast stattfindet.

5.6 Weitere Überlegungen

5.6.1 Zyklen

Wegen der $_2{=}_3$ Zuweisung kann ein Pfad von einer Variablen zu ihrem Wert beliebig lang werden. Es ist sogar möglich, dass der Pfad einen Zyklus bildet und somit niemals endet. Beginnend mit einer Speicherbelegung, in der es keine Zyklen gibt (siehe Tabelle 5.13), kann man Zyklen erzeugen. Bereits die Zuweisung

Zuweisung	Upcast	Downcast
$_1{=}_1$	∘	∘
$_2{=}_2$	+	+
$_2{=}_3$	+	−
$_3{=}_3$	−	−

Tabelle 5.12: Up- und Downcasts

`x` $_2{=}_3$ `x;`

bildet einen Zyklus (siehe Tabelle 5.14). Auf der anderen Seite kann man einen Zyklus wieder auflösen, in diesem Fall durch die Zuweisung

`x` $_2{=}_2$ `y;`

Der Endzustand ist in Tabelle 5.15 dargestellt.

Schicht 3:

x	y
a_{31}	**a_{32}**
a_{21}	a_{22}

Schicht 2:

a_{21}	**a_{22}**
a_{11}	a_{12}

Schicht 1:

a_{11}	**a_{12}**
v_1	v_2

Tabelle 5.13: Zyklenfreier Startzustand

Sollte beim Zugriff auf einen Wert, eine Methode oder ein Attribut von einer Variablen festgestellt werden, dass der Zugriff aufgrund eines Zyklus nicht möglich ist, wird eine Exception geworfen. Dieser Sachverhalt ist dadurch feststellbar, dass der Pfad in der zweiten Schicht in eine Zelle der zweiten Schicht führt, die im Anfangspfad bereits enthalten ist. In diesem Fall wird eine `CycleExceptipon` geworfen. Ebenfalls durch eine Ausnahme wird ein Zyklus in SuperGlue [MH06] behandelt (siehe Abschnitt 3.2.3). Solche Vorgehensweise ist auch in anderen Zusammenhängen üblich, wie z.B. im Fall des Zugriffs auf `null`.

5.6.2 n-Schichtenmodell

Das 3-Schichtenmodell ist ein Spezialfall eines n-Schichtenmodells für $n \geq 2$. Dabei können in der ersten Schicht nur Werte gespeichert werden und in Schicht n nur Adressen der Schicht $n-1$. Bei übrigen Speicherzellen können beliebige Adressen als Inhalte fungieren. In einem n-Schichtenmodell sind bestimmte $_i{=}_j$ Zuweisungen für $1 \leq i, j \leq n$ erlaubt.

	x	y
Schicht 3:	**a_{31}**	**a_{32}**
	a_{21}	a_{22}

Schicht 2:	**a_{21}**	**a_{22}**
	a_{21}	a_{12}

Schicht 1:	**a_{11}**	**a_{12}**
	v_1	v_2

Tabelle 5.14: Speicherzustand mit einem Zyklus

	x	y
Schicht 3:	**a_{31}**	**a_{32}**
	a_{21}	a_{22}

Schicht 2:	**a_{21}**	**a_{22}**
	a_{12}	a_{12}

Schicht 1:	**a_{11}**	**a_{12}**
	v_1	v_2

Tabelle 5.15: Zyklenfreier Endzustand

Ein interessanter Extremfall ist $n = 2$. Dabei sind nur zwei Arten von Zuweisungen möglich, nämlich $_1{=}_1$ und $_2{=}_2$. Es handelt sich um zwei symmetrische Zuweisungen. Eine nicht symmetrische Zuweisung ist nicht vorhanden. Somit deckt dieses Modell nicht alle gewünschten Zuweisungsarten ab.

Die Anzahl möglicher Zuweisungsoperatoren wächst quadratisch, also mit $O(n^2)$. Viele verschiedene Zuweisungsoperatoren machen ein Modell kompliziert. Das gewählte 3-Schichtenmodell ist das Modell mit dem kleinsten n, welches sowohl verschieden starke Zuweisungen als auch symmetrische und nicht symmetrische Zuweisungen enthält.

6 Konsequenzen und Evaluation

Dieses Kapitel diskutiert die Auswirkungen einer Erweiterung einer konventionellen, statisch typisierten, objektorientierten Programmiersprache um parametrisierte Zuweisungen. Außerdem werden parametrisierte Zuweisungen mit anderen Ansätzen für Abhängigkeiten verglichen.

Zuerst gebe ich einen Überblick über die vier Zuweisungsarten. Es wird diskutiert, in welchen Situationen welche Zuweisungsart angemessen ist. Dann gehe ich in den Unterkapiteln 6.2 bis 6.4 darauf ein, wie die Probleme in den konkreten Szenarien, die in Kapitel 4 erörtert werden, mit meinem Ansatz gelöst werden. Anschließend untersuche ich weitere weitreichende Konsequenzen des Einsatzes der parametrisierten Zuweisungen. Am Ende des Kapitels wird das Fazit über die Pragmatik, also die Brauchbarkeit der parametrisierten Zuweisungen, gezogen.

6.1 Überblick über die Zuweisungen

Die vier entwickelten Zuweisungsarten sollten im Zusammenspiel miteinander betrachtet werden.

Die $_2{=}_2$ Zuweisung ist die Standardzuweisung, die den Zuweisungen = aus Programmiersprachen wie Java oder C++ weitgehend entspricht. Sie wird im Normalfall, also häufiger als jede andere Zuweisungsart, eingesetzt.

Die $_3{=}_3$ Zuweisung wird gebraucht, wenn eine besonders starke Abhängigkeit gewünscht ist. Diese wird von der Standardzuweisung nicht überschrieben und bleibt oft über lange Zeit erhalten. In der Realität existieren viele Beispiele für Verbindungsobjekte, die von mehreren Entitäten geteilt werden und deren Veränderungen oder deren Austausch für alle Entitäten von Bedeutung sind. Jeder Gegenstand kann so ein Verbindungsobjekt darstellen.

Die $_1{=}_1$ Zuweisung hat den umgekehrten Effekt. Dadurch, dass der Wert einer Variablen in die andere kopiert wird, kann keine weitere Zuweisung an eine Variable Auswirkungen auf die andere haben. Es handelt sich jedoch um eine flache Kopie: Wenn man Attribute der Attribute der Objekte verändert, können dadurch Attribute der Attribute anderer Objekte verändert werden, so dass die beiden Variablen sich indirekt beeinflussen. Diese Zuweisungsart sollte eingesetzt werden, wenn ausdrücklich keine Abhängigkeit zwischen Variablen erwünscht ist. Auch hier gibt es viele Entsprechungen in der realen Welt: Wenn etwas vervielfältigt wird, entsteht zwar eine Kopie, aber diese Kopie wird nach ihrer Erstellung nicht mehr von Änderungen des Originals beeinflusst. Außer gewöhnlichen Gegenständen gehören zu so einem Vorgang auch abstrakte Entitäten. So kann z.B. das Wissen eines Menschen einem anderen Menschen lediglich mitgeteilt, also

kopiert werden. Manchmal kann die $_1{=}_1$ Zuweisung aus Gründen der Sicherheit eingesetzt werden, so dass über eine der Variablen keine sensitiven Informationen der anderen Variablen geändert werden können.

Die $_2{=}_3$ Zuweisung ist insofern besonders, als dass die Gleichheitsabhängigkeit, die durch sie hergestellt wird, von der $_2{=}_2$ Zuweisung sowohl aufgelöst als auch beibehalten werden kann. Die Zuweisung an die linke Variable hebt die Abhängigkeit auf, während die an die rechte nicht. Man kann sagen, dass die linke Variable von der rechten abhängig ist, aber nicht umgekehrt. Insbesondere kann man Ketten von Abhängigkeiten herstellen, so dass der Nachfolger immer von dem Vorgänger abhängig ist. Auch in diesem Fall lassen sich viele Beispiele der realen Welt finden. Wenn z.B. ein Vater und sein Sohn dasselbe Auto fahren, wird die Abhängigkeit nicht aufgelöst, wenn der Vater sein Auto wechselt. Sollte der Sohn sich ein eigenes Auto kaufen, fahren die beiden verschiedene Autos, und die Abhängigkeit besteht nicht mehr (siehe Abschnitt 1.2).

Bei jeder Zuweisung existieren also Parallelen zur realen Welt. Diese Parallelität ermöglicht, dass die Abhängigkeiten in der Software und auch die Software an sich durch die parametrisierten Zuweisungen verständlicher wird.

Abhängigkeiten treten nicht nur bei der Modellierung der realen Welt, sondern auch in technischen Zusammenhängen auf. So kann man beim Model View Controller mit Hilfe der $_3{=}_3$ Zuweisungen Abhängigkeiten zwischen dem Model und dem View herstellen. Mit $_1{=}_1$ Zuweisungen kann man z.B. die Unabhängigkeit zwischen einem Rückgabewert in der Methode und der Variablen außerhalb der Methode erzielen. Dies wird am Beispiel der Methode `getSigners` in Abschnitt 3.2.4 und Unterkapitel 4.4 zwecks Sicherheit motiviert.

6.2 Zugehörigkeiten

In Unterkapitel 4.1 werden Zugehörigkeiten diskutiert. Es handelt sich um Zugriffe auf ein Attribut, das *zu* einem Objekt *gehört*, wobei das Objekt selbst und dessen andere Attribute nicht betrachtet werden. In Unterkapitel 4.1 wird gezeigt, dass Zugehörigkeiten mit vielen vorhandenen sprachlichen Mitteln nur unzureichend umgesetzt werden können.

6.2.1 Lösung

Ein Beispiel aus dem Bankumfeld wird in Abschnitt 4.1.1 ausführlich erläutert. Es geht dabei um verschiedene Arten von Konten und Kunden und um die Interaktion der Komponenten bei der Durchführung von Daueraufträgen.

Das in Abbildung 6.1 vorgestellte Design ist einfach, übersichtlich und genügt allen gestellten Anforderungen, bis auf eine. Die Bank ist am aktuellen Konto einer Person interessiert. Problematisch bei der Umsetzung erweist sich die Benachrichtigung der Bank bei einem Kontowechsel (siehe Unterkapitel 4.1). Es handelt sich dabei um eine Abhängigkeit, da das Konto der Person mit dem Konto, das im Dauerauftrag registriert ist, immer identisch sein muss. Im Abschnitt 4.1.2 wird der Schluss gezogen, dass es nicht erwünscht ist, dieses Design für diese Anforderung zu verändern. Die Anforderung soll

stattdessen am besten mit Hilfe zusätzlicher Sprachkonstrukte erfüllt werden. Mit parametrisierten Zuweisungen kann die geforderte Abhängigkeit umgesetzt werden, ohne dass das Design in Abbildung 6.1 geändert wird.

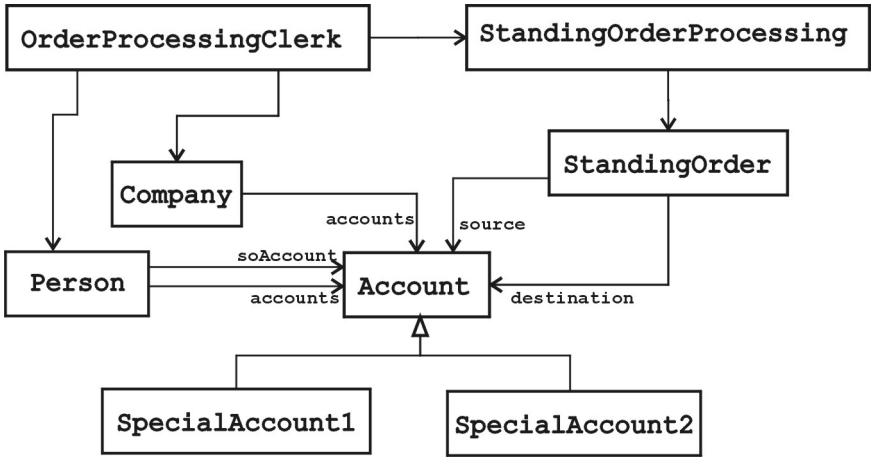

Abbildung 6.1: Klassendiagramm für das Kontobeispiel

Im Quellcode finden lediglich wenige Änderungen statt. Wenn alle Attribute öffentlich wären, würden folgende Zuweisungen ausreichen:

```
1 standingOrder.source 3=3 person1.account;
2 standingOrder.destination 3=3 person2.account;
```

Nun besteht eine Abhängigkeit zwischen dem Attribut `account` in der Klasse `Person` und den Attributen `source` bzw. `destination` in `StandingOrder`. Wenn sich `account` in `Person` ändert, findet die gleiche Änderung in `standingOrder` statt. Das liegt daran, dass die $_3{=}_3$ Zuweisung stärker ist als die sonst übliche $_2{=}_2$ Zuweisung:

```
person1.account 2=2 anotherAccount;
```

Da die Attribute in `Person` und `StandingOrder` geheim sein sollen [FGM99, Kla04], muss die Abhängigkeit durch Methoden hergestellt werden.

```
1  class StandingOrder {
2    private Account source;
3    private Account destination;
4    public void setSource(Account account3) {
5      source 3=3 account;
6    }
7    public void setDestination(Account account3) {
8      destination 3=3 account;
9    }
10   ...
11 }
```

Durch den Aufruf der Methoden

```
1 standingOrder.setSource(3person1.getAccount());
2 standingOrder.setDestination(3person2.getAccount());
```

wird ebenfalls die $_3{=}_3$ Zuweisung ausgeführt, diesmal ist die Zuweisung jedoch implizit.

Bei der Aktualisierung der Konten bei Daueraufträgen kann sowohl die $_3{=}_3$ als auch die $_2{=}_3$ Zuweisung eingesetzt werden. Beide leiten die Änderung des Kontos seitens der Person an den jeweiligen Dauerauftrag weiter.

Die Semantik unterscheidet sich nur, wenn das Konto seitens des Dauerauftrags geändert wird, z.B. durch eine Zuweisung innerhalb der Klasse `StandingOrder`:

```
source 2=2 anotherAccount;
```

Im Fall der $_3{=}_3$ Zuweisung hat die Person, von deren Konto bislang in diesem Auftrag abgebucht wird, ein neues aktuelles Konto. Im Fall der $_2{=}_3$ Zuweisung wird zwar das Konto im Auftrag geändert, aber das aktuelle Konto der Person bleibt gleich. Es gibt Gründe sowohl für den Einsatz der $_3{=}_3$ als auch der $_2{=}_3$ Zuweisung. Da hier angenommen wird, dass die Konten nur seitens Personen geändert werden, sind die beiden Zuweisungsarten gleichwertig.

Abstrahiert von diesem Kontobeispiel kann man sagen, dass der in dieser Arbeit vorgestellte Ansatz Zugehörigkeiten umsetzt. Das geschieht, indem eine Abhängigkeit mittels $_3{=}_3$ oder $_2{=}_3$ zum zugehörigen Attribut eines Objekts hergestellt wird.

6.2.2 Bewertung und Vergleich

Einer der Vorteile der parametrisierten Zuweisungen liegt in ihrer Einfachheit: Es müssen keine neuen Klassen oder Methoden hinzugefügt werden. Eine passende Zuweisungsart an entscheidender Stelle erfüllt bereits die Anforderung der Zugehörigkeit. Die Abhängigkeit kann meistens durch eine kleine Codeänderung und ohne Designmodifikation hergestellt werden. So sind die Probleme bei der Kontinuität und der Verständlichkeit, die in Unterkapitel 4.5 angesprochen werden, gelöst.

Compound References und SuperGlue kommen parametrisierten Zuweisungen am nächsten. Alle drei Ansätze erlauben eine Art Bindung zwischen Variablen bzw. Attributen. Zunächst gehe ich auf die wichtigsten Vorteile der parametrisierten Zuweisungen im Vergleich zu Compound References ein.

- Während Compound References sich vor allem für Zugehörigkeitsprobleme eignen, sind parametrisierte Zuweisungen breiter angelegt, so dass auch andere Anforderungen wie z.B. Parameterübergabevariationen und Rollen gelöst werden.

- Um die Typsicherheit zu gewährleisten, werden bei jeder Zuweisung mit Up- oder Downcast einer Compound Reference der alte und der neue Wert gespeichert (siehe Abschnitt 3.2.2). Einerseits besteht eine große Redundanz, andererseits ist es nicht transparent, welcher der beiden Werte in einem bestimmten Kontext durch die Referenz geliefert wird. Parametrisierte Zuweisungen sind dagegen typsicher, ohne dass Redundanz erzeugt wird.

Die hauptsächlichen Unterschiede zwischen SuperGlue und parametrisierten Zuweisungen in Bezug auf Abhängigkeiten sind die folgenden:

- Die Interaktion zwischen zwei Variablen bei SuperGlue ist vielfältiger als bei parametrisierten Zuweisungen; mein Ansatz berücksichtigt lediglich die Gleichheitsabhängigkeit, die bei Zugehörigkeiten ausreicht.
- Bei SuperGlue werden Abhängigkeiten deklarativ formuliert. Diese gelten für die ganze Lebenszeit der Objekte, so dass eine Bindung zwischen einem Dauerauftrag und dem Konto einer Person nicht aufgelöst werden kann. Parametrisierte Zuweisungen sind in diesem Punkt flexibler: Abhängigkeiten können hergestellt und wieder aufgelöst oder überschrieben werden.
- Bei SuperGlue müssen die Variablen, bei denen Abhängigkeiten möglich sind, mit Schlüsselwörtern gekennzeichnet werden: `export`, `import` oder `port`. Bereits während der Klassendeklaration muss der Entwickler entscheiden, ob ein Attribut exportiert oder importiert wird. Somit kann eine neue Anforderung an Abhängigkeiten bei SuperGlue mehr Änderungen erfordern als bei parametrisierten Zuweisungen, da Konflikte zu den gegebenen Schlüsselwörtern entstehen.
- SuperGlue ermöglicht die Abhängigkeit zwischen zwei Variablen immer nur in eine Richtung, was in diesem Beispiel verlangt wird. Es ist jedoch häufig der Fall, dass das zugehörige Objekt auch von außen verändert werden kann: z.B. das Konto der Person seitens des Dauerauftrages.

	Decorator		Einführung von `AccountOwner`		Observer		Delegates		Compound References		SuperGlue		parametrisierte Zuweisungen	
Hilfsklassen	ja	−	ja	−	ja	−	ja	○	nein	+	nein	+	nein	+
redundante Speicherung	ja	−	nein	+	nein	+	nein	+	ja	−	nein	+	nein	+
Flexibilität	nein	−	nein	−	ja	+	ja	+	ja	+	ja	○	ja	+
mehrfach zusammengesetzte Referenzen	ja	○	nein	−	ja	○	ja	○	ja	+	ja	+	ja	+
Veränderung von außerhalb	ja	+	ja	+	nein	−	nein	−	ja	+	nein	−	ja	+
mehrere Eigentümer	nein	−	nein	−	ja	+	ja	+	ja	+	ja	+	ja	+
Änderung von `SOP`	nein	+	ja	−	ja	−	nein	+	nein	+	nein	+	nein	+
Vielseitigkeit	ja	+	nein	−	ja	+	ja	+	nein	−	ja	+	ja	+

Tabelle 6.1: Vergleich der Lösungen für Zugehörigkeiten

Tabelle 6.1 präsentiert den Vergleich einiger Ansätze. Die Zellen enthalten zwei Werte.

Der erste „ja“ oder „nein“ besagt, ob die zugehörige Eigenschaft erfüllt ist, der zweite „+“, „○“ oder „−“, ob es gut ist, dass diese Eigenschaft erfüllt oder nicht erfüllt ist.

Als erstes gehe ich auf technische Eigenschaften der Ansätze ein. Die erste Zeile drückt aus, ob *Hilfsklassen* für die jeweilige Lösung benötigt werden. Hilfsklassen werden als negativ bewertet, da sie zusätzliche Komplexität mit sich bringen. Deswegen entspricht „ja“ einem „−“ und „nein“ einem „+“. Nur Compound References, SuperGlue und parametrisierte Zuweisungen schneiden hier gut ab. Im Gegensatz dazu fügen sowohl die Design Patterns Decorator und Observer als auch die Erweiterung um `AccountOwner` zusätzliche Klassen hinzu, um Zugehörigkeiten zu bewältigen. Außer zusätzlicher Komplexität entstehen Probleme, die mit Hilfsklassen einhergehen, wie z.B. das Syntactic Fragile Base Class Problem oder die Redundanz wegen der Vererbung des Zustands beim Decorator. Durch die Spracherweiterung Delegates aus C# wird sowohl das Klassendiagramm als auch der Code komplizierter.

Das nächste Kriterium *redundante Speicherung* sagt aus, ob unnötige Werte gespeichert werden. Redundante Speicherung ist nicht erwünscht. Beim Decorator entsteht sie durch die ungenutzten vererbten Attribute, und bei Compound References ist es die Speicherung der Werte bei Up- oder Downcasts. Die anderen Ansätze erfordern keine nennenswerten redundanten Daten.

Nun kommen wir zu allgemeinen Kriterien für Zugehörigkeiten betreffende Ansätze, die hauptsächlich aus Abschnitt 3.2.2 stammen. Ein Kriterium ist die *Flexibilität*. Damit meine ich in diesem Kontext, dass es möglich sein soll, dass Objekte der gleichen Klassen je nach Anforderung abhängig oder unabhängig sind. Manchmal z.B. soll der Kontowechsel dazu führen, dass der Dauerauftrag von dem neuen Konto abgebucht wird, manchmal jedoch weiterhin vom alten. Bei diesem Kriterium entspricht „ja“ „+“ und „nein“ „−“. Einführung einer Assoziation beim Decorator und `AccountOwner` erweist sich in dieser Hinsicht als zu starr. Die übrigen Ansätze sind flexibler.

Mit dem nächsten Kriterium sind *mehrfach zusammengesetzte Referenzen* abgedeckt, also Fälle wie z.B. das Konto der Frau des Kontoinhabers, wobei sowohl das Konto als auch die Frau wechseln können. Die meisten Design Patterns können die Anforderung durch deren doppelte Anwendung umsetzen, wie z.B. Decorator im Decorator. Das Design wird in solchen Fällen so kompliziert, dass die mittlere Bewertung ○ vergeben wird. Nur mit Compound References, SuperGlue und parametrisierten Zuweisungen wird diese Anforderung zufrieden stellend erfüllt.

Das nächste Kriterium behandelt die Veränderung des zugehörigen Attributs sowohl innerhalb als auch außerhalb der Klasse. Dies ist bei Ansätzen, in denen man zwischen Subject und Observer unterscheidet, nicht möglich, da ein Observer üblicherweise nur von Veränderungen des Subjects durch einen Methodenaufruf erfährt, diese Veränderung jedoch nicht beeinflusst. Dazu zählen Observer, Delegate und SuperGlue. Bei den übrigen Ansätzen hat Observer einen direkten Zugriff auf das Subject, so dass Attribute des Subjects vom Observer aus verändert werden können.

Die nächsten zwei Anforderungen beziehen sich direkt auf das Kontobeispiel. Es stellt sich heraus, dass man beim Decorator und bei der Einführung von `AccountOwner` nur auf Umwegen herausfinden kann, ob *mehrere* Personen *Eigentümer* des gleichen Kontos sind (siehe Abschnitt 3.2.2).

Das Kriterium, *Änderung von* `SOP` überprüft, ob bei der Umsetzung von Abhängigkeiten die Klasse `StandingOrderProcessing` verändert werden muss. Das ist nicht erwünscht oder sogar nicht möglich, da sie aus einer Klassenbibliothek oder einem Framework stammen könnte. Nur die Einführung von `AccountOwner` und Observer verletzen diese Anforderung.

Das letzte Kriterium *Vielseitigkeit* ist das einzige, welches unabhängig von Zugehörigkeiten ist. Es wird bewertet, ob die Ansätze nur auf wenige Probleme spezialisiert sind, wie die Einführung von der Klasse `AccountOwner`, oder vielseitig wie Delegates sind. Delegates kann man z.B. für Ereignisse, GUI-Programmierung, Zugehörigkeiten oder Observerfunktionalität verwenden. Außer der Speziallösung `AccountOwner` scheinen nur Compound References in relativ wenigen Fällen einsetzbar.

Tabelle 6.1 zeigt, dass syntaktische Erweiterungen der Programmiersprachen besser abschneiden als Design Patterns. Dies ist ein zu erwartendes Ergebnis, da man mit zusätzlichen sprachlichen Mitteln mehr erreichen kann als ohne. Am besten erweisen sich parametrisierte Zuweisungen, denn sie werden bei jedem Kriterium positiv bewertet.

6.3 Parameterübergabe

Unterkapitel 4.2 diskutiert den Nutzen von starkem und schwachem call-by-reference und call-by-value. Es zeigt, dass alle drei Varianten in einer Programmiersprache vorhanden sein sollten. Die meisten Programmiersprachen erlauben nur eine oder zwei Parameterübergabevariationen. Die Sprachen mit allen drei Variationen wie C++ erkaufen sich diesen Vorteil durch den Einsatz von Zeigern. Heute versuchen Programmiersprachenentwickler auf Zeiger wegen ihrer Komplexität zu verzichten.

6.3.1 Lösung

Da es sich bei der Parameterübergabe um nichts anderes als implizite Zuweisungen handelt, beschreibt das 3-Schichtenmodell mit Hilfe neuer Zuweisungsarten alle Parameterübergabevariationen (siehe Kapitel 5 und insbesondere Abschnitt 5.4.4).

Die Übergabevariation, die auf den Zuweisungen $_3=_3$ und $_2=_3$ basiert, ist starker call-by-reference. Auf der Basis der $_2=_2$ Zuweisung wird schwacher call-by-reference erreicht. Call-by-value entspricht der $_1=_1$ Zuweisung (siehe Tabelle 6.2). Dank dem starken call-by-reference ist es möglich, dass eine Methode mehrere Rückgabewerte hat, die als Methodenparameter auftreten.

Parameterübergabemechanismus	Realisierung durch Zuweisung
starker call-by-reference	$_3=_3$, $_2=_3$
schwacher call-by-reference	$_2=_2$
call-by-value	$_1=_1$

Tabelle 6.2: Realisierung der Parameterübergabevariationen

Nun folgen zur Verdeutlichung die in Unterkapitel 4.2 angesprochenen Methoden. Es

wird gezeigt, wie mit der Syntax parametrisierter Zuweisungen die gewünschte Parameterübergabevariation umgesetzt wird. Dabei wird der schwache call-by-reference übergangen, da in diesem am weitesten verbreiteten Mechanismus bei Objekten keine syntaktischen Änderungen im Vergleich zu Java oder C# erforderlich sind. Man muss dabei weder die aktuellen noch die formalen Parameter parametrisieren, denn das entspricht der impliziten $_2{=}_2$ Zuweisung. Für jede der folgenden Methoden wird mindestens die Signatur und der Aufruf angegeben.

Die Methode `exchangeIntValues` vertauscht mit Hilfe des starken call-by-reference die Werte der formalen Parameter `x` und `y`.

```
1 public void exchangeIntValues(int x3, int y3) {
2   int z = x;
3   x = y;
4   y = z;
5 }
```

Der Aufruf mit aktuellen Parametern `k` und `m` wird folgendermaßen notiert:

```
exchangeIntValues(3k, 3m);
```

Es kommt also zu den impliziten Zuweisungen

```
x 3=3 k;
```

und

```
y 3=3 m;
```

Diese Zuweisungen sind stärker als die in den Zeilen 2 bis 4 (siehe Lemma 3), denn $=$ entspricht $_2{=}_2$. Das bedeutet, dass trotz der Zuweisungen im Rumpf der Methode `k` und `x` bzw. `m` und `y` weiterhin die gleichen Werte haben. In diesem Beispiel kann die $_3{=}_3$ Zuweisung durch die $_2{=}_3$ Zuweisung ersetzt werden, da der formale Parameter immer auf der linken Seite der Zuweisung und der aktuelle auf der rechten steht. Die Änderung des formalen Parameters durch eine $_2{=}_2$ Zuweisung führt in beiden Fällen zur Änderung des aktuellen Parameters.

Mit Hilfe des starken call-by-reference können mehrere Rückgabewerte simuliert werden. In der folgenden Methodendefinition zur Lösung einer quadratischen Gleichung sind `x1` und `x2` diese Rückgabewerte.

```
public void solveQuadraticEquation(int a, int b, int c,
    int x13, int x23) {...}
```

Auf diese Weise bekommt beim folgenden Methodenaufruf `x` den Wert von `x1` und `y` den Wert von `x2`.

```
solveQuadraticEquation(a1, a2, a3, 3x, 3y);
```

Hierbei finden implizite Zuweisungen

```
x1 3=3 x;
```

und

```
x2 3=3 y;
```

statt. Auch hier kann die $_3=_3$ Zuweisung durch die $_2=_3$ Zuweisung ersetzt werden.

Bei der folgenden Methode `print` soll die Übergabevariaton call-by-value sein. Wenn die Variable `o` innerhalb der Methode `print` verändert wird, darf das keine Auswirkungen auf `myObject` außerhalb der Methode haben.

```
public void print(Object o1) {...}
print(1myObject);
```

Also wird die implizite Zuweisung

```
myObject 1=1 o1;
```

ausgeführt.

Das zeigt, dass die Umsetzung der Parameterübergabevariationen mit parametrisierten Zuweisungen einfach ist: Beim schwachen call-by-reference wird nichts parametrisiert, beim starken call-by-reference werden sowohl der aktuelle als auch der formale Parameter mit 3 und beim call-by-value beide Parameter mit 1 parametrisiert.

6.3.2 Bewertung und Vergleich

Wie im letzten Abschnitt demonstriert, ermöglichen parametrisierte Zuweisungen alle Parameterübergabevariationen mit einer einfachen Syntax und ohne den Einsatz der Zeiger. Dies ist ein Vorteil gegenüber Sprachen, die nicht alle Parameterübergabevariationen beherrschen. Dazu gehören u.a. Java, C#, Smalltalk und Python.

In den meisten Programmiersprachen hängt die Übergabevariation von dem Typ der Parameter ab: Handelt es sich z.B. um eine Klasse, ist die Übergabevariation schwacher call-by-reference; bei einem primitiven Datentyp dagegen gilt call-by-value. Die parametrisierten Zuweisungen sind typunabhängig und somit flexibel, da für jeden Spezialfall die Variation frei wählbar ist (siehe Abschnitt 6.5).

In Abschnitt 4.2.2 wird gezeigt, dass es viele Speziallösungen für einzelne Parameterübergabevariationen gibt. Man kann diese in drei Kategorien einteilen.

In das Umfeld der *Design Patterns* gehören sowohl Adapter als auch die Verwendung der Methode `clone`. Bei der Parameterübergabe haben diese Lösungen den Nachteil, komplex zu sein.

Die *Schlüsselwörter* haben den Nachteil, ausschließlich für ein konkretes Problem geschaffen zu sein. Es gibt z.B. in C# sowohl für starken call-by-reference als auch für mehrere Rückgabewerte jeweils ein eigenes Schlüsselwort. Parametrisierte Zuweisungen lösen mehr Aufgaben als nur die Parameterübergabe.

Um call-by-value zu erzwingen, gibt es in den geläufigen Programmiersprachen keine Schlüsselwörter. Jedoch sollten an dieser Stelle Immutable References erwähnt werden (siehe Abschnitt 3.2.4). Sie ermöglichen mittels des Schlüsselworts `readonly`, dass ein Objekt nicht über eine bestimmte Variable verändert werden kann: Sowohl die Variable selbst als auch ihre Attribute, als auch Attribute der Attribute, usw. dürfen nicht modifiziert werden. Da call-by-value oft aus Sicherheitsgründen eingesetzt wird, um bestimmte

Änderungen der Variablen zu vermeiden, ist die Gegenüberstellung zu Immutable References nahe liegend. Im Gegensatz zu Immutable References sind bei parametrisierten Zuweisungen nicht alle Änderungen ausgeschlossen. Dadurch, dass nur eine flache Kopie der Variablen erzeugt wird, ist es möglich, dass Attribute der Attribute sowohl des formalen als auch des aktuellen Parameters auf dasselbe Objekt zeigen. Andererseits wird call-by-value nicht nur zu Zwecken der Sicherheit eingesetzt. Es ist manchmal gewünscht, dass der formale Parameter geändert werden kann, was aber mit Immutable References nicht möglich ist.

Schließlich ermöglichen *Zeiger* alle Parameterübergabevariationen. Sie sind mächtig und vielseitig einsetzbar. Jedoch verschlechtern sie die Lesbarkeit des Codes.

Zusammenfassend lässt sich sagen, dass die parametrisierten Zuweisungen sich nahtlos auf die Parameterübergabe übertragen lassen. Dies betont die Vielseitigkeit dieses Ansatzes.

6.4 Rollen

In Unterkapitel 4.3 werden Rollendesigns diskutiert. Es wird einerseits gezeigt, dass Rollen wichtig und allgegenwärtig sind und andererseits, dass viele Anforderungen an solche Designs bestehen (siehe auch Anhang A). Ein etabliertes Design ist das Role Object Pattern (siehe Abschnitt 3.1.6). Ein Nachteil dieses Patterns liegt in der schlechten Unterstützung für Abhängigkeiten zwischen Rollen, insbesondere für ihre gemeinsamen Teile. Hier wird gezeigt, wie komplexe Abhängigkeit bei Rollen mit parametrisierten Zuweisungen umgesetzt wird.

6.4.1 Lösung

Wir betrachten wieder das abgewandelte Beispiel aus [BD96] in Abbildung 3.6, das in Abschnitt 4.3.1 vorgestellt wird. Es geht um eine Entität, die einen Menschen namens Joe repräsentiert. Joe kann sowohl als Person an sich betrachtet werden als auch als Sportler, Filmenthusiast und Angestellter. Jede seiner Rollen hat eigene Eigenschaften, die sich teilweise mit denen der anderen Rollen überschneiden. Wenn sich eine der Eigenschaften in einer Rolle ändert, muss sie sich auch in anderen Rollen ändern. Dabei können diese Eigenschaften einerseits unterschiedlich heißen und trotzdem den gleichen Wert haben; andererseits gibt es auch Eigenschaften, die zwar gleich heißen, dafür verschiedene Werte haben.

Als Lösung verwenden wir das Objektdiagramm in Abbildung 6.2 und definieren die Abhängigkeiten mit Hilfe neuer Zuweisungsarten. Man erkennt im Diagramm vier Objekte, die Rollen darstellen. Diese Objekte gehören vier verschiedenen Klassen an. Diese Klassen können zueinander in Vererbungshierarchien stehen, um die Wiederverwendbarkeit zu erhöhen. Man kann gleichnamige Eigenschaften, wie `address`, in Oberklassen oder Interfaces auslagern. Es ist jedoch nicht möglich, eine der Rollen als Oberklasse für die anderen zu definieren, da es keine Rolle gibt, deren Attribute gänzlich in anderen Rollen enthalten sind. Also müssen zusätzliche Klassen deklariert werden. Auf das Laufzeitverhalten der Objekte hat die Vererbungshierarchie keine Auswirkungen.

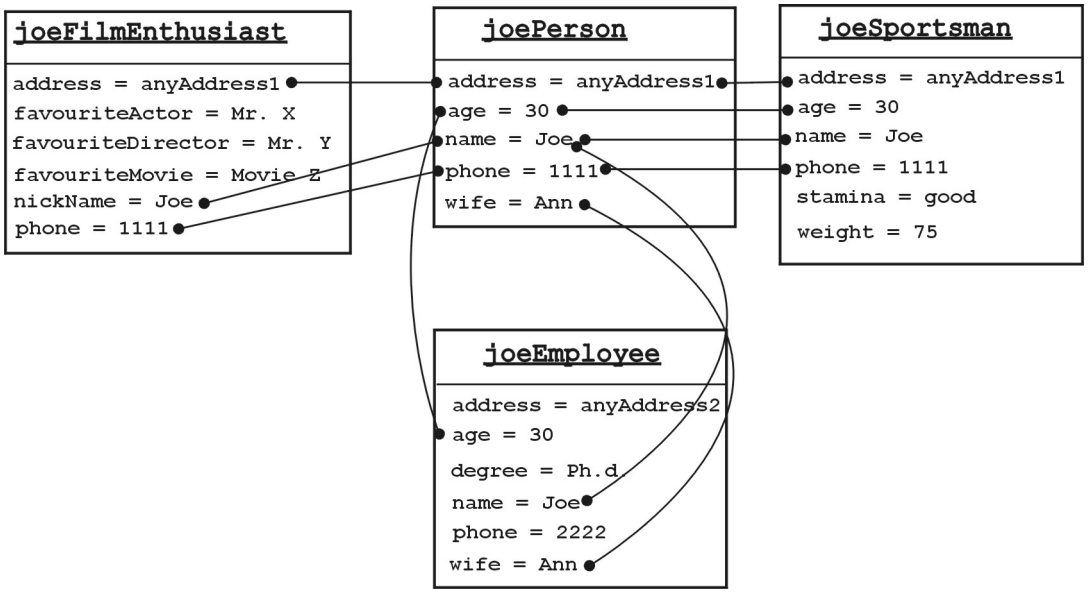

Abbildung 6.2: Joes Rollen

Die Rollenobjekte werden mit Hilfe der parametrisierten Zuweisungen zueinander in Beziehung gebracht. Hier wird davon ausgegangen, dass alle Attribute öffentlich sind. Die Umformung zu geheimen Attributen und öffentlichen Methoden beeinflusst die Grundidee der Abhängigkeiten nicht.

```
1  joeSportsman.name 3=3 joePerson.name;
2  joeSportsman.address 3=3 joePerson.address;
3  joeSportsman.age 3=3 joePerson.age;
4  joeSportsman.phone 3=3 joePerson.phone;
5  joeFilmEnthusiast.address 2=3 joePerson.address;
6  joeFilmEnthusiast.nickName 3=3 joePerson.name;
7  joeFilmEnthusiast.phone 3=3 joePerson.phone;
8  joeEmployee.name 3=3 joePerson.name;
9  joeEmployee.age 3=3 joePerson.age;
10 joeEmployee.wife 3=3 joePerson.wife;
```

Alle aufgeführten Attribute mit Ausnahme von `address` in Zeile 5 sind symmetrisch voneinander abhängig. Wenn sich ein Attribut durch eine $_2{=}_2$ Zuweisung ändert, ändern sich die von ihm abhängigen Attribute ebenfalls, unabhängig davon, in welcher der beiden Rollen die Änderung stattfindet.

Die Anforderung an die Adresse von Joe als Person und die als Filmenthusiast unterscheidet sich von den anderen: Wenn `joePerson` umzieht, soll `address` von `joeFilmEnthusiast` ebenfalls geändert werden; wenn sich die Adresse von `joeFilmEnthusiast` ändert, weil Joe andere Enthusiasten in einem Club treffen möchte, bleibt die Adresse von `joePerson` bestehen. Es handelt sich um eine unsymmetrische

Abhängigkeit: Die Adresse des Filmenthusiasten ist von der Adresse der Person abhängig. Dieser Anforderung wird die $_2{=}_3$ Zuweisung in Zeile 5 gerecht.

Also erlauben parametrisierte Zuweisungen das Erzeugen von Abhängigkeiten zwischen Rollen zur Laufzeit. Diese Vorgehensweise ist sehr flexibel, da man Abhängigkeiten einfach modifizieren kann. Mit parametrisierten Zuweisungen lassen sich komplexe Rollendesigns mit vielen Abhängigkeiten implementieren.

6.4.2 Bewertung und Vergleich

Da Rollen ausgiebig in Anhang A untersucht werden, beschränke ich mich in diesem Abschnitt nur auf die Zusammenfassung der Ergebnisse.

Hier werden sechs Ansätze für einen Vergleich ausgewählt, die ermöglichen, dass nur die Eigenschaften einer Rolle und nicht alle Eigenschaften der Entität zugreifbar sind. Mehrfach- und Interfacevererbung sind häufige Modellierungsansätze, die durch einfache Anwendung der Vererbung umgesetzt sind (siehe Abschnitt 4.3.2 und Unterkapitel A.1). Role Object Pattern ist das allgemein anerkannte Design Pattern für Rollen (siehe Abschnitte 3.1.6 und 4.3.2). Object Teams ist eine Spracherweiterung, die Rollen mittels aspektorientierter Programmierung umsetzt. Dieser Ansatz wird hier als Repräsentant der aspektorientierten Ansätze ausgewählt (siehe Abschnitt 3.2.5 und Unterkapitel A.1). Schließlich handelt es sich bei RCC (siehe Unterkapitel A.2) und parametrisierten Zuweisungen (siehe Abschnitt 6.4.1) um zwei Ansätze, die über Sprachkonstrukte für Abhängigkeiten verfügen.

Die Vergleichskriterien werden in Rollenmerkmale, also Anforderungen an die Rollenansätze, und übergeordnete Eigenschaften, die bei jedem objektorientierten Ansatz wichtig sind, unterteilt.

	MV	IV	ROP	OT	RCC	PZ
Modulkohäsion	+	−	+	+	+	∘
Abhängigkeiten	−	+	−	−	+	+
Dynamik	−	−	+	∘	−	+
gleiche Identität	+	+	−	∘	+	−
mehrfaches Spielen derselben Rolle	−	−	+	+	+	+
Hierarchie	+	+	+	∘	+	∘

Tabelle 6.3: Rollenmerkmale

Die Tabellen 6.3 und 6.4 fassen den Vergleich von sechs Rollenansätzen zusammen. Die Bezeichnungen der Ansätze sind abgekürzt: MV – Mehrfachvererbung, IV – Interfacevererbung, ROP – Role Object Pattern, OT – Object Teams, RCC – Roles as Components of Classes und PZ – parametrisierte Zuweisungen. „+“ bedeutet, dass ein Kriterium erfüllt, „−“ dass es nicht erfüllt und dass es „○“ nicht ganz zufrieden stellend erfüllt ist.

Neuere Verfahren wie parametrisierte Zuweisungen schneiden bei Rollenmerkmalen gut ab, dafür weisen sie beim Entwicklungsstand und bei der Dokumentation aufgrund ihrer Neuheit Schwächen auf.

	MV	IV	ROP	OT	RCC	PZ
Nähe zum Objektmodell	+	+	+	○	○	○
Wartbarkeit und Erweiterbarkeit	○	−	+	+	○	+
Verständlichkeit	+	+	−	−	○	○
Dokumentation	○	+	+	○	−	−
Entwicklungsstand	+	+	+	○	−	−

Tabelle 6.4: Allgemeine Eigenschaften

Parametrisierte Zuweisungen eignen sich vor allem für dynamische Rollendesigns mit komplexer Abhängigkeitsstruktur. Da sie keine Vererbungsstruktur vorschreiben, können sie fast in jedes Design oder in jeden Rollenansatz eingewoben werden und so etablierte Verfahren für Abhängigkeiten rüsten. Vor allem die Kombination mit Role Object Pattern scheint viel versprechend. Dabei werden Eigenschaften, die bei allen Rollen gemeinsam sind, in das Core Object ausgelagert und Eigenschaften, die nur in wenigen Rollen gleich sind, über parametrisierte Zuweisungen umgesetzt.

6.5 Trennung der Abhängigkeiten und der Typen

Die Konsequenzen der parametrisierten Zuweisungen wirken sich nicht nur auf einzelne Szenarien, sondern auch auf so globale Bereiche wie Variablentypen aus. Der übliche Zusammenhang zwischen der Zuweisungssemantik und den Typen der Variablen wird aufgehoben.

6.5.1 Verzicht auf Zeiger

Eine weitreichende Konsequenz parametrisierter Zuweisungen besteht darin, dass auf Zeiger verzichtet werden kann. Fast alle Operationen, die mit Zeigern durchgeführt werden

können, sind ebenfalls mit parametrisierten Zuweisungen möglich.

Nehmen wir an, `x` und `y` sind Variablen vom primitiven Datentyp, z.B. `int`. Die folgende Zuweisung in C++

```
x = y;
```

entspricht bei parametrisierten Zuweisungen

```
x 1=1 y;
```

Auch zur Zuweisung von c-Referenzen

```
int &x = y;
```

gibt es eine Analogie bei parametrisierten Zuweisungen:

```
int x 3=3 y;
```

Wenn `x` und `y` einfache Zeiger sind, z.B. vom Typ `int*`, entspricht

```
x = y;
```

der $_2=_2$ Zuweisung::

```
x 2=2 y;
```

und

```
*x = *y;
```

in C++ entspricht der $_1=_1$ Zuweisung:

```
x 1=1 y;
```

Mehrstufige Zeiger sind mit parametrisierten Zuweisungen nicht uneingeschränkt simulierbar. Wenn `x` den Typ `int***` hat, kann man auf `x` mittels `&x`, `x`, `*x`, `**x` und `***x` zugreifen. Man könnte `x` mit dem Zuweisungsparameter $_3$ simulieren, `*x` mit $_2$ und `**x` mit $_1$. Für `***x` hat das 3-Schichtenmodell keine Entsprechung. Dafür wäre eine Erweiterung auf mehr Schichten erforderlich (siehe Abschnitt 5.6.2).

Der Vorteil des 3-Schichtenmodells gegenüber den Zeigern besteht darin, dass Unterschiede bei Typen `T`, `T*`, `T**`, `T***`, usw. wegfallen. Bei den Typen kommt es allein auf die Werte an, die angenommen werden können. Der Typ einer Variablen sollte nicht von den Zuweisungen abhängen, in denen diese Variable vorkommt, sondern ausschließlich von Werten, die die Variable annehmen kann, bzw. von Methoden, die auf dieser Variablen aufgerufen werden können.

Bei Zeigern muss man sich bereits beim Anlegen der Variablen entscheiden, welche Zuweisungen angewendet werden, und welche Zeigerart geeignet ist. Mit parametrisierten Zuweisungen ist dies nicht nötig, da die Zuweisungen dem Kontext angepasst werden können und nicht vom Typ der Variablen abhängen.

Weiterhin ist es leichter, beim 3-Schichtenmodell den Wert einer einfachen Variablen herauszufinden (siehe Definition 9). In C++ gibt folgender Code nicht den Wert der Variablen aus, sondern eine Adresse.

```
1 int*** x = ...;
2 cout << **x;
```

Sollte sich also der Programmierer in der Anzahl der Sternchen vor der Variablen irren, hätte es fatale Konsequenzen.

Zeigerarithmetik ist eine weitere Ursache vieler Verständnisprobleme beim Programmieren mit Zeigern. Diese ist in parametrisierten Zuweisungen nicht erlaubt. Schließlich ist Garbage Collection bei parametrisierten Zuweisungen möglich, während es bei Zeigern nicht der Fall ist.

6.5.2 Keine Unterscheidung zwischen Werttypen und Referenztypen

Während in C++ die Zuweisungssemantik durch Zeiger gegeben ist, wird in C# und Java zwischen Werttypen und Referenztypen unterschieden (siehe Unterkapitel 2.2). Dies ist der Weg, der in den meisten objektorientierten Programmiersprachen verfolgt wird, um Zeiger zu umgehen.

Bei einer Zuweisung zwischen zwei Variablen eines Referenztyps, die der $_2{=}_2$ Zuweisung entspricht, wird die j-Referenz kopiert, und beide Variablen referenzieren danach das gleiche Objekt. Bei einer Zuweisung zwischen zwei Variablen eines Werttyps, die der $_1{=}_1$ Zuweisung entspricht, wird nicht die j-Referenz, sondern der komplette Inhalt einer Variablen kopiert. Dies liegt an der verschiedenen Speicherung: Instanzen von Referenztypen sind in Heaps gespeichert, und die von Werttypen liegen auf dem Stack.

Das Gleiche gilt für die Parameterübergabe. Instanzen eines Werttyps werden mit call-by-value und die eines Referenztyps mit schwachem call-by-reference übergeben.

Bereits bei der Entscheidung, ob man eine Klasse oder einen Struct in C# erstellt, wird über die Zuweisungssemantik entschieden. Diese Zuweisungssemantik gilt dann für alle Variablen dieses Typs und ist nicht mehr veränderbar. Dagegen ist die Zuweisungssemantik im 3-Schichtenmodell nicht von Variablentypen abhängig. Sie kann bei jeder Zuweisung variiert werden. Parametrisierte Zuweisungen übertreffen also in ihrer Flexibilität alleinige Unterscheidung zwischen Wert- und Referenztypen.

Das 3-Schichtenmodell ermöglicht eine einheitliche Speicherung für die Variablen. Es gilt der Smalltalk-Grundsatz: Alles ist ein Objekt. Instanzen von primitiven Datentypen, wie z.B. ganze Zahlen, sind Objekte. Sie werden genauso gespeichert wie jede andere Instanz. Die einheitliche Speicherung führt in Smalltalk dazu, dass alle Variablen ihre Objekte referenzieren; Zuweisungen kopieren j-Referenzen; nur schwacher call-by-reference ist möglich. Das 3-Schichtenmodell unterliegt nicht dieser Einschränkung. Sogar `null` ist bei parametrisierten Zuweisungen ein Objekt der Klasse `Null` und nicht ein Schlüsselwort (siehe Abschnitt 5.4.2). Es handelt sich dabei um ein Singleton, ähnlich wie in der prototypenbasierten Sprache Self [SU95, US91].

6.6 Fazit

Das 3-Schichtenmodell ermöglicht vier Arten von Zuweisungen. Alle diese Arten erweisen sich als nützlich zur Umsetzung von Abhängigkeiten. Sie leisten eine Verbesserung ob-

jektorientierter Sprachen sowohl bei der Modellierung der realen Welt als auch bei vielen technischen Details wie der Parameterübergabe.

Einzelne der vorgestellten Möglichkeiten der parametrisierten Zuweisungen sind bereits in manchen Programmiersprachen vorhanden, wie z.B. `ref` in C# [Mic05, RNW+03] oder Compound References [OM01]. Es handelt sich jedoch um Speziallösungen. Parametrisierte Zuweisungen stellen einen einheitlichen Ansatz zur Behandlung verschiedener Abhängkeitsszenarien dar.

Besonders nützlich erweist sich, dass man verschieden starke Zuweisungen zur Verfügung hat, die einander überschreiben bzw. nicht überschreiben können. Das Zusammenspiel verschieden starker Zuweisungen ist in gängigen Programmiersprachen nicht möglich.

Die $_2{=}_3$ Zuweisung ist neuartig, da nicht symmetrische Zuweisungen bisher in der Literatur nicht angesprochen wurden.

Der größte Vorteil der parametrisierten Zuweisungen liegt meiner Meinung nach in deren Einfachheit: Sie ermöglichen flexible und komplexe Abhängigkeitsszenarien umzusetzen, ohne die Architektur zu verändern. Die Primärarchitektur der Anwendungen bleibt damit klar und verständlich. So sind die Kriterien Verständlichkeit, Kontinuität, Zerlegbarkeit und Kombinierbarkeit erfüllt (siehe Abschnitt 2.1.2).

Die Ziele, die in Kapitel 4 formuliert werden, sind somit erreicht.

7 Design- und Implementierungsaspekte

In diesem Kapitel wird gezeigt, wie man parametrisierte Zuweisungen objektorientiert entwirft und implementiert. Die komplette Beispielimplementierung ist auf der beiliegenden CD zu finden.

Die erstellte Java Applikation ermöglicht es, die in Kapitel 5 vorgestellten Ideen zu testen. Weiterhin soll den Programmierern, die mit formaler Spezifikation nicht vertraut sind, die Semantik des Modells anhand des Entwurfs und des Codes verständlich gemacht werden. Außerdem stellt dieses Kapitel einen Leitfaden dar, wie man parametrisierte Zuweisungen objektorientiert implementiert. Die in der Applikation erlaubte Syntax stimmt nicht mit der Syntax aus Kapitel 5 überein. Obwohl hier lediglich Java Code erlaubt ist, kann man anhand der Applikation die Konzepte und die Wirkung der Zuweisungen demonstrieren (siehe Unterkapitel 7.2). Die Performance der Programme ist ausdrücklich kein Ziel der gegebenen Implementierung; denn es geht hauptsächlich um Verständlichkeit. Später kann man durch Refactoring die Performance verbessern [Fow99].

Zuerst wird in Unterkapitel 7.1 der Entwurf der Anwendung vorgestellt und diskutiert. Unterkapitel 7.2 zeigt die Syntax, wie parametrisierte Zuweisungen angewendet werden können. Schließlich geht Unterkapitel 7.3 auf die Implementierung wichtiger Methoden ein.

7.1 Entwurf

Das vereinfachte Klassendiagramm der Anwendung ist in Abbildung 7.1 dargestellt.

Die Klasse `Variable` stellt Variablen dar und enthält zwei Attribute: `name` und `slot`. `name` steht für den Namen der Variablen und `slot` für eine Speicherzelle der dritten Schicht. Dies entspricht der Funktion s_4 aus Unterkapitel 5.2.

Eine zentrale Klasse in diesem Design ist `Slot`. Ein Objekt vom Typ `Slot` entspricht einer Speicherzelle aus dem 3-Schichtenmodell. Es kann jede Speicherzelle in jeder Schicht repräsentieren. Durch Attribute `level` und `index` wird die Adresse der Speicherzelle in ihrer Schicht dargestellt. Das Attribut `content` beschreibt den Inhalt der Speicherzelle. Ein weiteres Attribut `numberOfElements` bezeichnet die Anzahl der indirekten Inhalte der Speicherzelle. Der Wert unterscheidet sich von eins nur, wenn es sich um eine Speicherzelle einer zusammengesetzten Variablen handelt, die sich in der zweiten Schicht befindet und das nächste Pfadelement in der dritten Schicht ist. Dann sagt dieses Attribut aus, wie viele Elemente das Array oder wie viele Attribute das Objekt hat.

Die Klassen, die als Inhalt einer Speicherzelle in Frage kommen, müssen das Interface `Content` implementieren. Einerseits kann es sich um Werte, also um Objekte des Typs `Value`, und andererseits um die Speicherzellen selbst, also um Objekte der Klasse `Slot` handeln. Die Klasse `Content` entspricht dem Micro Pattern Taxonomy [GM05].

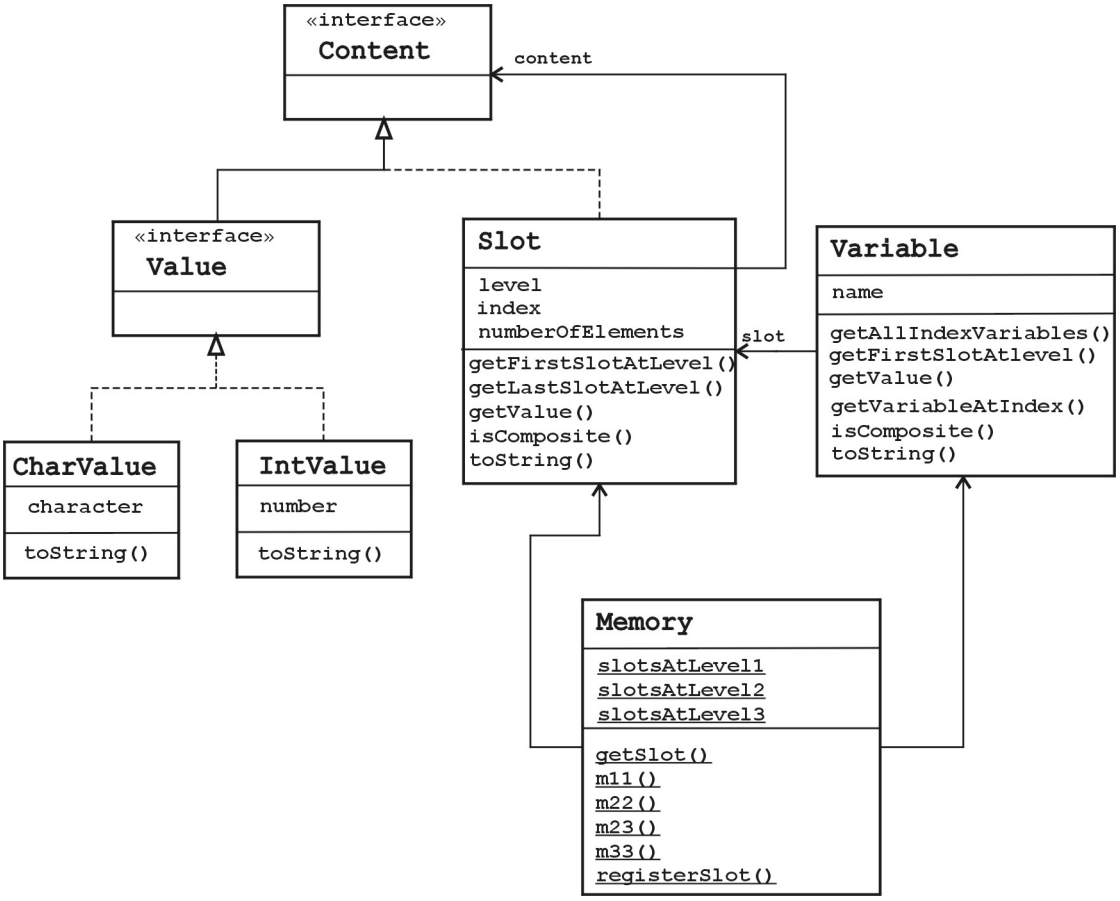

Abbildung 7.1: Klassendiagramm

Pfade von Speicherzellen können konstruiert werden, weil ein Objekt der Klasse `Slot` auf Objekte derselben Klasse verweisen kann. Durch einen Pfad gelangt man von einer Speicherzelle einer einfachen Variablen zu ihrem Wert. Viele Methoden, wie z.B. `getValue` zum Erhalten des Wertes einer Variablen sind rekursiv. Sie rufen sich selbst am nächsten Element des Pfades auf (siehe Unterkapitel 7.3).

Alle Klassen, die in der ersten Schicht als Inhalt fungieren, implementieren das Interface `Value`. Auf der Semantikseite handelt es sich dabei um die Menge *Vals*. Exemplarisch wird `Value` von `IntValue` für ganze Zahlen und von `CharValue` für Buchstaben implementiert. Das Interface `Value` entspricht dem Micro Pattern Designator [GM05].

Die Klasse `Memory` stellt den Speicher dar. Sie enthält drei Attribute des Typs `Slot<Vector>`. Diese Variablen enthalten jeweils eine Liste von Speicherzellen und entsprechen den drei Schichten des Speichers. Weiterhin enthält `Memory` die öffentlichen Methoden `m11`, `m22`, `m23` und `m33`. Diese Methoden haben als Parameter zwei Variablen und führen die Zuweisungen $_1 =_1$ bis $_3 =_3$ durch. Alle Attribute und Methoden von `Memory` sind statisch, so kann man auf sie aus den Klassen `Slot` und `Variable`

zugreifen, ohne dass ein Objekt der Klasse `Memory` zur Verfügung steht.

Die vorgestellte Lösung ist eine Kombination aus zwei fertigen Entwürfen. Um den Sachverhalt möglichst nah zu modellieren, müssen die Typen `Slot` und `Variable` existieren. Der Typ `Value` kommt hinzu, weil es im Gegensatz zu Java keinen Unterschied zwischen Objekten und primitiven Datentypen geben soll. Das hier angewendete Design Pattern wird Adapter oder Wrapper genannt [GHJV97].

Es stellt sich die Frage, wie viel Verantwortung die einzelnen Klassen übernehmen sollen. Dies hängt davon ab, wo der gesamte Speicher modelliert ist. Die endgültige Entscheidung fällt zugunsten der Klasse `Memory`. Alle Methoden und Attribute dieser Klasse sind statisch und somit global zugreifbar. `Memory` entspricht dem Micro Pattern Common State [GM05]. Sowohl `Slot` als auch `Variable` brauchen Zugriff darauf, um z.B. beim Index in einem Array Speicherzellen relativ zu anderen Speicherzellen zu finden. Es wäre ein zusätzlicher Aufwand, wenn diese Klassen eine Referenz auf ein `Memory`-Objekt halten würden.

Eine weitere Fragestellung besteht darin, ob die Zuweisungen in der Klasse `Variable` oder in der Klasse `Memory` definiert werden. In `Variable` hätte eine Zuweisungsmethode nur einen Parameter vom Typ `Variable`. In `Memory` hat so eine Methode beide betroffenen Variablen als Parameter. Ich habe mich für die zweite Version entschieden, weil klassisch betrachtet die Zuweisung keine Methode einer Variablen ist.

7.2 Syntax

Alle Zuweisungen des 3-Schichtenmodells können in der vorgestellten Anwendung simuliert werden. Dies erfordert Umformungen des Codes. So entspricht z.B. die Zuweisung

```
int x = 5;
```

dem Ausdruck

```
IntValue five = new IntValue(5);
Variable x = new Variable("x", five);
```

Dabei ist `IntValue` eine Klasse, die für ganze Zahlen steht, und `Variable` entspricht Variablen.

Weiterhin wird

```
x =3 y;
```

durch den Ausdruck

```
Memory.m33(x, y);
```

simuliert. `m33` ist dabei eine statische Methode der Klasse `Memory`, und `x` und `y` sind Variablen der Klasse `Variable` mit den Namen `"x"` und `"y"`. Genauso verhält es sich mit anderen Zuweisungen.

```
x =1 y;
```

entspricht dem Ausdruck

```
Memory.m11(x, y);
```

7.3 Implementierung

Nachdem die Klassenstruktur beschrieben ist, zeige ich die Funktionalität der wichtigsten Methoden.

Der Konstruktor

```
public Variable(String name, Value value);
```

erzeugt eine einfache Variable mit dem Namen `name` und dem Wert `value`. Es wird der Name der Variablen gesetzt und der Konstruktor der Klasse `Slot` aufgerufen:

```
1 this.name = name;
2 slot = new Slot(value, 3);
```

Dabei wird `slot` eine Speicherzelle der dritten Schicht zugewiesen. Der zum Aufruf passende Konstruktor in `Slot` hat folgenden Aufbau:

```
1  public Slot(Content content, int level) {
2    this.level = level;
3    this.numberOfElements = 1;
4    Memory.registerSlot(this);
5    if (level == 1) {
6      this.content = content;
7    } else {
8      this.content = new Slot(content, level - 1);
9    }
10 }
```

Zuerst werden die Standardattribute gesetzt. Danach wird die neue Speicherzelle in der Klasse `Memory` registriert. Schließlich wird eine weitere Speicherzelle der Schicht `n-1` erstellt, oder, wenn die erste Schicht bereits erreicht ist, wird der Wert der Variablen gespeichert. Ein Konstruktoraufruf einer Speicherzelle in der dritten Schicht erzeugt für jede der drei Schichten eine Speicherzelle. Schließlich wird der Inhalt der Speicherzelle in der ersten Schicht auf den gewünschten Wert gesetzt.

Bei der Registrierung einer Speicherzelle in der Klasse `Memory` wird die Speicherzelle entsprechend dem Wert von `level` einer der drei Schichten `slotsAtLevel1` bis `slotsAtLevel3` hinzugefügt. Dabei wird der Index der Speicherzelle in der Liste bestimmt und gesetzt. Die entscheidende Methode ist dabei

```
1 private static void addSlotToMemory
2       (Vector<Slot> slots, Slot slot) {
3   slots.add(slot);
4   slot.setIndex(slots.indexOf(slot));
5 }
```

Im Gegensatz zu Kapitel 5 startet der Index in einer Schicht nicht bei 1 sondern bei 0. Dieser Unterschied ist jedoch nebensächlich.

Der Konstruktor

```
public Variable(String name, Value[] values);
```

erzeugt ein Array mit Werten `values` und der Konstruktor

```
public Variable(String name, Variable[] fields) {
```

ein Objekt aus bestehenden Variablen, die zu Attributen werden. Die Variablen `fields` können ihrerseits zusammengesetzt sein, so dass komplexe Datenstrukturen erstellt werden können. Ähnlich wie beim ersten vorgestellten Konstruktor rufen diese Konstruktoren entsprechende Konstruktoren der Klasse `Slot` auf. Da wird der Pfad rekursiv von der dritten Schicht aus konstruiert. Entscheidend ist das Vorgehen in der zweiten Schicht. Der folgende Codeabschnitt stammt aus einem Konstruktor in `Slot` bei der Erzeugung eines Arrays.

```
1 assert level == 2;
2 numberOfElements = contents.length;
3 Slot[] slots = new Slot[numberOfElements];
4 for (int i = 0; i < numberOfElements; i++) {
5   slots[i] = new Slot(contents[i], 3);
6 }
7 this.content = slots[0];
```

In Zeile 2 wird `numberOfElements` auf die Anzahl der Elemente im Array gesetzt. In den Zeilen 4 bis 6 wird für jedes Element im Array ein Pfad aus Speicherzellen konstruiert. Schließlich wird der Inhalt des gegebenen Slots auf das nullte Element in der dritten Schicht gesetzt. Dies entspricht der Speicherung der Arrays aus Abschnitt 5.4.1.

Eine weitere interessante Methode in `Variable` ist `getVariableAtIndex`. Dabei wird eine Variable zurückgegeben, die unter `index` in einer zusammengesetzten Variablen, also in einem Objekt oder einem Array, gespeichert ist. Bei Objekten werden in dieser Anwendung die Attribute durch Indizes und nicht mit Namen gespeichert, was in Bezug auf Konzepte keinen Unterschied zum Objektmodell darstellt.

```
1 public Variable getVariableAtIndex(int index) {
2   assert isComposite();
3   Slot slot2 = slot.getLastSlotAtLevel(2);
4   Slot slot30 = (Slot) slot2.getContent();
5   Slot mySlot = Memory.getSlot(3, slot30.getIndex() + index);
6   return new Variable(name + "[" + index + "]", mySlot);
7 }
```

Dabei wird in Zeile 3 eine Speicherzelle des Pfades der Variablen in der zweiten Schicht geholt. Ihr Inhalt ist eine Speicherzelle der dritten Schicht, da es sich um eine zusammengesetzte Variable handelt. Diese Speicherzelle ist der Anfang des Pfades des Elements mit dem Index null bzw. des entsprechenden Attributs eines Objekts. In Zeile 5 wird Gebrauch davon gemacht, dass einige Methoden der Klasse `Memory` global verfügbar sind.

In der dritten Schicht wird eine Speicherzelle gesucht, die sich `index` Stellen entfernt von `slot30` befindet. Dort ist das entsprechende Element gespeichert. Schließlich wird in Zeile 6 eine neue Variable erzeugt, die auf die betreffende Speicherzelle verweist.

Die Methode `getValue` in `Variable` gibt den Wert einer einfachen Variablen zurück. Es wird die gleichnamige Methode der Klasse `Slot` aufgerufen. Dabei handelt es sich um eine rekursive Methode, die jeweils den Inhalt der nächsten Speicherzelle im Pfad ausliest. Ist dieser Inhalt vom Typ `Value`, wird das Ergebnis zurückgegeben, ansonsten wird diese Methode an dem Inhalt der Speicherzelle, also dem nächsten Pfadelement, aufgerufen.

Kommen wir nun zu Methoden der Klasse `Memory`. Diese Klasse ist für den Speicher und insbesondere für Zuweisungen verantwortlich. Die Zuweisungen werden durch Methoden `m11` bis `m33` simuliert. Diese Methoden greifen auf die Methode `mij` zurück, die die ${}_i{=}_j$ Zuweisung darstellt, z.B:

```
1 public static void m23(Variable v1, Variable v2)  {
2   mij(v1, v2, 2, 3);
3 }
```

Alle Methoden außer `m11` enthalten nur eine Zeile, in der der Aufruf von `mij` stattfindet. `mij` hat die folgende Definition:

```
1 private static void mij
2       (Variable v1, Variable v2, int i, int j) {
3   Slot rightSlot = v2.getFirstSlotAtLevel(j);
4   Content right =  rightSlot.getContent();
5   Slot left =  v1.getFirstSlotAtLevel(i);
6   left.setContent(right);
7   left.setNumberOfElements(rightSlot.getNumberOfElements());
8 }
```

Zuerst werden Speicherzellen der gefragten Schichten `i` und `j` in Pfaden von `v1` und `v2` geholt. In Zeile 6 wird der Inhalt der Speicherzelle von `v1` in Schicht `j` in die Speicherzelle von `v2` in Schicht `i` geschrieben. Dies entspricht der Semantik der Zuweisungen in Definition 6.

Die Methode `m11` unterscheidet sich von den anderen, weil die ${}_1{=}_1$ Zuweisung für einfache und zusammengesetzte Variablen unterschiedlich verläuft.

```
1  public static void m11(Variable v1, Variable v2) {
2    if (!v1.isComposite()) {
3      assert !v2.isComposite();
4      mij(v1, v2, 1, 1);
5    } else {
6      assert v2.isComposite();
7      Variable[] variables1 = v1.getAllIndexVariables();
8      Variable[] variables2 = v2.getAllIndexVariables();
9      assert variables1.length == variables2.length;
10     for (int i = 0; i < variables1.length; i ++ ) {
```

```
11          m22(variables1[i], variables2[i]);
12        }
13      }
14    }
```

Für den Fall, dass wenn beide Variablen einfach sind, wird wie bei üblichen $_i{=}_j$ Zuweisungen vorgegangen (siehe Zeilen 2 von bis 4). Wenn `v1` und `v2` zusammengesetzt sind, verläuft die Zuweisung entsprechend Definition 12. Dabei wird jedes Element eines Arrays bzw. jedes Attribut eines Objekts mit der $_2{=}_2$ Zuweisung kopiert (siehe Zeile 11).

Die vorgestellte Anwendung wurde ausgiebig getestet. Für fast jede Methode sind automatisierte JUnit Tests erstellt [JUn06, RS05]. Einerseits kann man anhand dieser Tests die Nebeneffekte von neuen oder veränderten Teilen erkennen (siehe Abschnitt 2.1.3). Andererseits sind sie zum Verständnis der Interaktion der einzelnen Module wichtig. Die Anwendung selbst wurde mit Eclipse erstellt [Ecl06, GB03].

8 Zusammenfassung und Ausblick

8.1 Zusammenfassung

Das Ziel dieser Arbeit besteht darin, ein Sprachkonstrukt zu entwickeln, das sich zur Umsetzung von Gleicheitsabhängigkeiten eignet.

Bei der Modellierung der realen Welt stellt man fest, dass Objekte häufig gemeinsame Eigenschaften haben oder, anders ausgedrückt, voneinander abhängig sind. Um diese Behauptung zu untermauern, führe ich eine Studie durch, die verschiedene Szenarien in der Software untersucht (siehe Kapitel 4). Es stellt sich nicht nur heraus, dass Abhängigkeiten allgegenwärtig sind, sondern auch, dass es bisher keinen einheitlichen und, meines Erachtens nach, zufrieden stellenden Weg gibt, diese in objektorientierten Sprachen zu behandeln. In der vorhandenen Software wird häufig für jede Abhängigkeit eine Speziallösung konstruiert.

In dieser Arbeit werden parametrisierte Zuweisungen als Lösung für Gleichheitsabhängigkeiten präsentiert. Eine wesentliche Eigenschaft der parametrisierten Zuweisungen liegt darin, dass die Länge des Pfades von einer Variablen zu ihrem Wert nicht vom Typ der Variablen bestimmt wird, sondern von der Art der Zuweisung, wie der Wert an die Variable gebunden wird. Das steht im Gegensatz zur üblichen Referenzierung in Java oder Zeigern in C++. So sind z.B. Variablen der Typen `int*` und `int**` in C++ verschieden gespeichert und können nicht den gleichen Wert annehmen. Mit parametrisierten Zuweisungen haben beide Variablen den Typ `int`, die Werte der Variablen sind jedoch durch verschiedene Zuweisungsarten an die Variablen gebunden. Die Abhängigkeit zwischen Variablen wird durch den gemeinsamen Teilpfad von der Variablen zu ihrem Wert ausgedrückt. Insbesondere ist es von Bedeutung, dass einige Zuweisungen *stärker* sind als andere und dass es sowohl *symmetrische* als auch *unsymmetrische* Zuweisungen gibt. Abhängig von der Situation wird eine geeignete Zuweisungsart gewählt. Die parametrisierten Zuweisungen werden auf der Basis eines 3-Schichtenmodells entwickelt, welches einfach aufgebaut ist, denn eine wesentliche Anforderung an die vorgestellte Spracherweiterung ist deren Verständlichkeit.

Das 3-Schichtenmodell wird mit Hilfe der operationalen Semantik spezifiziert, und wichtige Eigenschaften werden bewiesen. Insbesondere wird gezeigt, dass eine statisch typisierte, objektorientierte Sprache mit parametrisierten Zuweisungen typsicher ist (siehe Kapitel 5). Außerdem wird ein Beispielentwurf und eine Beispielimplementierung entwickelt (siehe Kapitel 7).

Die Evaluation zeigt, dass parametrisierte Zuweisungen sich sehr gut eignen, Gleichheitsabhängigkeiten zu bewältigen (siehe Kapitel 6).

Die parametrisierten Zuweisungen bieten sich nicht nur für objektorientierte Sprachen an, stellen aber besonders in der Objektorientierung einen eleganten Weg von der Mo-

dellierung bis zur Implementierung vieler Szenarien dar.

8.2 Ausblick

Trotz der gemachten Fortschritte sind weitere Forschungen zu parametrisierten Zuweisungen und Abhängigkeiten an sich wünschenswert.

Die wichtigste Herausforderung besteht darin, dass parametrisierte Zuweisungen den Eingang in Programmiersprachen finden. Dazu wird eine Implementierung auf einem hohen Entwicklungsstand benötigt. Ein Eclipse-Plugin auf der Basis der entwickelten Java Anwendung (siehe Kapitel 7 und die Anwendung auf der beiliegenden CD) scheint ein geeigneter Anfang zu sein.

Weiterhin gilt es zu erforschen, ob parametrisierte Zuweisungen für andere Zwecke außerhalb von Abhängigkeiten hilfreich sind. Einige mögliche Einsatzgebiete des neuen Konstrukts sind in Kapitel 6 angesprochen.

Möglicherweise ist es sinnvoll, UML [OMG05] um Abhängigkeitsarten, die in dieser Arbeit vorgestellt werden, zu erweitern. Es ist jedoch fraglich, ob UML die Semantik so exakt wiedergeben soll, schließlich handelt es sich um eine Abstraktion und nicht eine exakt definierte Programmiersprache [Coo06].

Interessant wäre es zu untersuchen, ob parametrisierte Zuweisungen außerhalb von objektorientierten Sprachen sinnvoll eingesetzt werden können. Der Schritt zu z.B. prozeduralen Programmiersprachen scheint mir überschaubar zu sein. Bei z.B. funktionalen Sprachen sehe ich weder einen sinnvollen Einsatz noch eine machbare Umsetzung.

Ein weiterer Punkt sind *allgemeine* Abhängigkeiten, also nicht nur Gleichheitsabhängigkeiten, in objektorientierten Programmiersprachen. Durch das 3-Schichtenmodell oder seine Erweiterung können sie nicht verwirklicht werden. Dennoch sind allgemeine Abhängigkeiten eine häufig gestellte Anforderung in Softwarearchitekturen.

Schließlich stellen Abhängigkeiten, die nicht zwischen Objekten, sondern zwischen Modulen [SJSJ05], Klassenbibliotheken, Datenbankeinträgen [Bee80], Datenbanken, usw. auftreten, ein großes Forschungsgebiet dar.

A Abhängigkeiten bei Rollen

Dieser Anhang behandelt ein großes Einsatzgebiet für Abhängigkeiten, nämlich Rollen. An vielen Beispielen ist bereits gezeigt, wie allgegenwärtig Abhängigkeiten im Zusammenhang mit Rollen sind: Abschnitt 4.3.1 beschreibt Rollen einer Person; in [CLD05, Laz05] wird ein Krankenhausinformationssystem untersucht; in [CD05a, CD05b] geht es um ein Telekommunikationsunternehmen; in [Rie98] wird ein Bürokratieszenario eingeführt.

Das allgegenwärtige Paradigma der Rolle gewinnt in vielen Forschungsgebieten an Anerkennung. In objektorientierten Programmiersprachen versteht man unter einer Rolle die sichtbaren Eigenschaften eines Objekts, also eine Untermenge seiner Eigenschaften [Rie00]. Ein Objekt ist in der Lage, mehrere Rollen gleichzeitig zu spielen. Es kann von verschiedenen Perspektiven aus betrachtet werden, so dass verschiedene Eigenschaften zum Vorschein kommen, denn Attribute und Methoden können durch Rollen überschrieben werden. Ein Client hat nur auf ausgewählte Rollen Zugriff und sieht nur relevante Informationen.

Eine wichtige Eigenschaft in Rollenszenarien ist die Abhängigkeit, denn Rollen haben Einfluss aufeinander. Insbesondere können Rollen einer Entität gemeinsame, abhängige und unabhängige Teile besitzen, so dass, wenn ein Attribut einer Rolle geändert wird, möglicherweise Attribute anderer Rollen angepasst werden müssen. Trotz der bisherigen umfassenden Forschungen im Bereich der Rollen ist dieses Problem nicht adäquat gelöst [CD05a].

Um Entitäten und Rollen auszudrücken, unterscheidet man zwischen natürlichen Typen und Rollentypen [Sow84]. Natürliche Typen stehen für Entitäten, und Rollentypen stellen Rollen dar. In statisch typisierten Programmiersprachen entsprechen Rollentypen oft den statischen Typen und natürliche Typen den dynamischen. Wenn `o` z.B. ein Objekt des dynamischen Typs `N` ist, spielt eine Referenz `r` des statischen Typs `R` die Rolle `R` dieses Objekts:

```
R r = o;
```

Die Rolle `R` sollte aussagekräftig genug sein, ohne dass `N` nach außen bekannt gegeben wird [Utt92].

Unterkapitel A.1 präsentiert Rollenansätze aus verschiedenen Kontexten. Danach wird in A.2 der Ansatz *Roles as Components of Classes*, kurz *RCC*, vorgestellt und untersucht. Schließlich werden in Unterkapitel A.3 Vergleichskriterien ausgearbeitet, und sechs Rollenansätze inklusive RCC und parametrisierten Zuweisungen verglichen.

A.1 Verwandte Arbeiten

Der Rollenbegriff wird in der Forschung vielseitig verwendet. Er variiert von beobachteten Eigenschaften [Ste01], perspektivabhängigem Verhalten der Objekte [Mez98] bis zu subjektivem Verhalten [Kri01].

Wie bereits in Unterkapitel 4.3 erörtert werden Rollenszenarien meist durch Modellierung umgesetzt. Am verbreitetsten ist der Role Object Design Pattern (siehe Abschnitte 3.1.6 und 4.3.2) [BRSW00, Fow97]. Weitere Modellierungsansätze sind Mehrfachvererbung [Sny86, Mez98, CLD05] und Interfacevererbung [Ste01].

Bei der Mehrfachvererbung kann eine Klasse mehrere Oberklassen haben. Jede Rolle wird als eigene Klasse deklariert. Der natürliche Typ erbt dabei von allen Rollentypen. Wenn man das Szenario aus Abschnitt 4.3.1 mit Mehrfachvererbung modelliert, ergeben sich die Rollentypen `Person`, `Sportsman`, `FilmEnthusiast` und `Employee` und der natürliche Typ `Human` (siehe Abbildung A.1). Das folgende C++-Codefragment

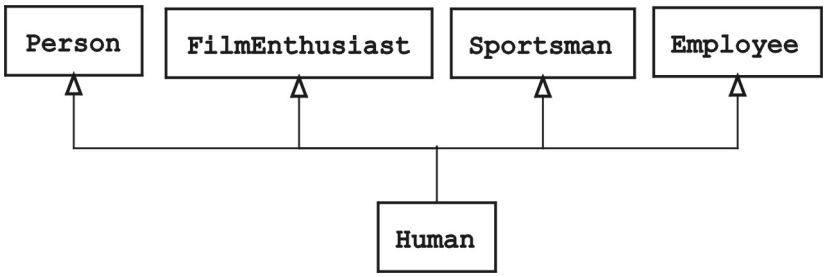

Abbildung A.1: Mehrfachvererbung

zeigt einen Zugriff auf eine Entität `Human` mit Hilfe zweier Referenzen der Rollentypen `Employee` und `Person`. Die Änderung des Alters in einer Rolle ruft die Änderung des Alters in der anderen Rolle hervor, da beide Variablen das gleiche Objekt referenzieren (siehe Zeilen 3 und 4).

```
1 Employee joeEmployee = new Human;
2 Person joePerson = (Person) joeEmployee;
3 joePerson.incrementAge();
4 System.out.println(joeEmployer.getAge());
```

Jede Variable des Rollentyps stellt eine Referenz auf das eigentliche Objekt dar. Mit Hilfe solcher Variablen des Typs der Oberklasse wird die Sicht auf das Objekt eingeschränkt. Alle Eigenschaften des natürlichen Typs, die nicht zur Rolle gehören, werden ausgeblendet [CLD05, Laz05].

Interfacevererbung ist ähnlich zur Mehrfachvererbung. Der Unterschied besteht darin, dass eine Klasse höchstens von einer anderen Klasse erben kann, dafür kann sie mehrere Interfaces implementieren. Interfaces können im Gegensatz zu Klassen lediglich Methoden und Konstanten deklarieren. Diese Methoden müssen in den Unterklassen implementiert werden. Nichtsdestotrotz sind Interfaces Typen und können insbesondere als Rollentypen fungieren [Ste01]. Um das gegebene Beispiel mit Rollen umzusetzen, ändert man in

Abbildung A.1 drei oder vier Oberklassen zu Interfaces. In diesem Fall wird die Klasse `Human` komplizierter als bei der Mehrfachvererbung, da die Funktionalität mehrerer Rollen nun in dieser Klasse implementiert wird.

Ein Client, der über eine Variable auf das Objekt zugreift, merkt keinen Unterschied zwischen Mehrfach- und Interfacevererbung. Er sieht ausschließlich einen Rollentyp.

In [VN96, Van97] wird ein weiterer Ansatz beschrieben, der Rollen mit vorhandenen objektorientierten Kompositionsmechanismen zusammensetzt. In diesem Fall handelt es sich dabei um Vererbung und Templates.

Weiterhin existieren mehrere Klassenbibliotheken [GSR96, ST02] und Frameworks [Rie00] für Rollen.

Es gibt viele Spracherweiterungen, in deren Mittelpunkt Rollenprogrammierung steht. Aspektorientierte Rollenansätze erlauben eine feinere Modularisierung. Aspektorientierte Programmierung wird in diesem Kontext kontrovers diskutiert. Auf der einen Seite wird sie als viel versprechend bezeichnet [Ken99], auf der anderen Seite wird der Schluss gezogen, dass Rollen und Aspekte zu unterschiedlich sind [HU02, Ste06]. In [HSU05] wird sogar die Ansicht vertreten, dass Rollenprogrammierung ein Spezialfall der aspektorientierten Programmierung ist. Object Teams [Her02a, Her02b, Her05], Chameleon [GB02] and EpsilonJ [Tam02, Tam05] sind aspektorientierte Ansätze, die Rollen syntaktisch definieren.

Häufig ist ein Ziel der Einführung von Rollen, die Arbeit mehrerer unabhängiger Entwicklerteams an einem Softwareprojekt zu ermöglichen. Diese Idee ist ursprünglich als subjektorientierte Programmierung bekannt [OKH+95, OKK+96]. Heute spricht man von subjektorientierter Programmierung, wenn bei einem Methodenaufruf für ein gegebenes Objekt die Interpretation unterschiedlich sein kann [Kri01]. Diese Interpretation kann z.B. von der Rolle des Objekts abhängen. Die Hauptprojekte der subjektorientierten Community setzen aspektorientierte Programmierung ein. Dazu gehören Hyper/J [TK02] und Concern Manipulation Environment [HOT04].

Ich wähle Object Teams als Repräsentanten für aspektorientierte Ansätze und behandle diesen etwas ausführlicher. Object Teams führt neue Programmkonstrukte `team` und `role` ein. Ein Team ist eine Zusammenlegung von Klassen, die in einem Kontext interagieren. Klassen innerhalb eines Teams nennen sich Rollen und werden von ihren *Basisklassen* gespielt. Solch eine Basisklasse ist ein natürlicher Typ. Eine Rolle und eine Basisklasse sind durch das Schlüsselwort `playedBy` verbunden.

Bei Methodenbindungen wird zwischen *callout* und *callin* unterschieden. Callout bedeutet, dass die Basisklasse Methoden an ihre Rollen weiterleitet. Ein callin webt eine Rollenmethode in die Basisklasse ein (siehe Unterkapitel 3.2.5). Dabei gibt es drei Alternativen: Die Rollenmethode kann vor, nach oder anstatt der Methode der Basisklasse ausgeführt werden.

```
1 team class ContextWork {
2   class Employee playedBy Human {
3     int getAge() -> int getAge();  // callout
4     void work() {...};
5   }
```

```
6     class Employer playedBy ...{
7       ...
8     }
9   }
```

Dieses Codefragment demonstriert eine vereinfachte Version des Teams `ContextWork`. In diesem Kontext gibt es zwei Rollen: den Arbeitnehmer und den Arbeitgeber. Die Rolle `Employee` wird von der Basisklasse `Human` gespielt. Die Methode `getAge` wird mittels eines callout aus der Basisklasse importiert. Außerdem enthält die Rolle `Employee` rollenspezifisches Verhalten, in diesem Fall die Methode `work`.

Eine weitere nicht aspektorientierte Spracherweiterung für Rollen ist variationsorientierte Programmierung [Mez98]. Dabei geht es um zusätzliches Rollenverhalten in dynamisch typisierten Programmiersprachen. Prototypenbasierte Sprachen bieten ein weiteres Einsatzgebiet für Rollen [VPDMD04].

In [KØ96] wird ein konzeptionelles Modell für Rollen beschrieben. Die Autoren unterscheiden zwischen innewohnenden und äußerlichen Eigenschaften der Objekte (siehe Abschnitt 3.1.6). Ein anderer Ansatz ist Object-Oriented Role Analysis and Modelling [And97], bei dem nicht Objekte, sondern Rollen miteinander [RWL95, Ree97] interagieren. Das Modell Objects with Roles beschäftigt sich vor allem mit Rollen langlebiger Objekte [Per90].

Rollen und Perspektiven sind nicht nur in der Programmierung, sondern auch in Datenbanken, insbesondere in objektorientierten Datenbanken entscheidend. Zu den Datenbankansätzen gehören die Ansätze Object-Role-Model [PK97], Fibonacci [AGO95] und Door [WCL97].

Eine gute Übersicht über verwandte Arbeiten findet man in [Bau03, Ste00].

A.2 Roles as Components of Classes

Roles as Components of Classes, kurz RCC, erweitert das objektorientierte Modell um eine neue Zwischenebene, nämlich um Rollen. Dadurch können Abhängigkeiten zwischen den Rollen elegant ausgedrückt werden. Dabei soll das Design der Programmiersprache dem konzeptuellen Modell entsprechen. Deswegen wird zusätzlich zum Modell die Syntax entwickelt. Ein prototypisches Eclipse Plugin für RCC ist bereits entwickelt [Ake06].

RCC unterscheidet zwischen *Klassen*, die natürlichen Typen entsprechen, und *Rollen*, die Rollentypen entsprechen. Dabei entsprechen Rollen den konventionellen Klassen im Objektmodell. Eine Klasse in RCC dagegen kann aus mehreren Rollen bestehen. Ein Objekt ist Instanz einer Klasse und enthält alle ihre Rollen. Klassen sind instanziierbar, Rollen dagegen nicht.

A.2.1 Abhängigkeiten zwischen Attributen und Methoden

Betrachten wir zunächst die RCC-Implementierung des Beispiels aus Abschnitt 4.3.1, bevor die darin eingeführten Begriffe formal definiert werden. Es geht um einen Menschen,

der die Rollen Person, Filmenthusiast, Sportler und Arbeitnehmer annehmen kann. Die verschiedenen Rollen teilen miteinander manche ihrer Eigenschaften.

```
1  role Person {
2    Address address;
3    int age;
4    String name;
5    String phone;
6    Person wife;
7  }
8  role FilmEnthusiast {
9    Address address;
10   Actor favouriteActor;
11   String nickName;
12   ...
13 }
14 role Sportsman {
15   Address address;
16   int age;
17   int weight;
18   ...
19 }
20 role Employee {
21   ...
22 }
23 class Human {
24   include Person;
25   include FilmEnthusiast;
26   include Sportsman;
27   include Employee;
28   equal(Sportsman:address, Person:address);
29   equal(FilmEnthusiast:nickName, Person:name);
30   ...
31 }
```

Vier Rollen werden in den Zeilen 1 bis 22 definiert. Ihre Syntax entspricht der der gewöhnlichen Klassen mit dem Unterschied, dass das Schlüsselwort `class` durch `role` ersetzt wird. Die Klasse in den Zeilen 23 bis 31 umfasst alle diese Rollen und definiert Gleichheitsabhängigkeiten zwischen Attributen mit dem Schlüsselwort `equal`. So bewirkt z.B. Zeile 29, dass das Attribut `nickName` in der Rolle `FilmEnthusiast` immer den gleichen Wert hat wie das Attribut `name` in `Person`. Diese Abhängigkeit unterscheidet sich von der Abhängigkeit, die bei parametrisierten Zuweisungen durch $_3{=}_3$ erzeugt wird. Denn sie existiert für alle Objekte dieser Klasse und kann nicht aufgelöst oder überschrieben werden.

Rollen können nur als statischer Typ fungieren, während Klassen ausschließlich dynamische Typen darstellen. Eine Rolle einer Klasse kann auf eine andere Rolle derselben Klasse gecastet werden:

```
1 Sportsman joeSportsman = new Human();
2 Person joePerson = (Person) joeSportsman;
```

`joeSportsman` und `joePerson` sind zwei Referenzen auf das gleiche Objekt vom dynamischen Typ `Human`. Während in `joeSportsman` nur Eigenschaften des Sportlers zugreifbar sind, verfügt `joePerson` nur über die Eigenschaften von Joe als Person. Dadurch dass es sich um Referenzen auf das gleiche Objekt handelt, haben Änderungen einer Variablen Auswirkungen auf die andere.

Nun sollen die intuitiv eingeführten Begriffe formal definiert werden.

Definition 13. *Eine* Rolle *ist ein Tupel $R = (N_R, A_R, M_R)$, wobei N_R den Rollennamen, A_R eine Menge der Attribute und M_R eine Menge der Methoden darstellt.*

Mit dem Unterschied, dass Rollen nicht instanziiert werden können, entsprechen sie den Klassen im Standardobjektmodell. Ein Attribut der Rolle R wird in dieser Arbeit üblicherweise mit a_R und eine Methode mit m_R bezeichnet.

Zwei Rollen sind *abhängig*, wenn mindestens eine Abhängigkeit zwischen deren Attributen oder Methoden existiert.

Definition 14. *Zwei Attribute verschiedener Rollen sind genau dann* gleichheitsabhängig*, wenn diese Attribute immer den gleichen Wert haben.*

In [BK00] werden gleichheitsabhängige Attribute als gemeinsame Eigenschaften bezeichnet. Seien a und b Attribute der Rollen R und S. Die Gleichheitsabhängigkeit zwischen diesen Attributen wird durch

$$a_R \sim b_S$$

gekennzeichnet.

Definition 15. *Zwei Methoden verschiedener Rollen sind genau dann* gleichheitsabhängig*, wenn sie semantisch äquivalent sind.*

Seien m und n Methoden der Rollen R und S. Die Gleichheitsabhängigkeit zwischen m und n wird durch

$$m_R \sim n_S$$

gekennzeichnet.

Zwei semantisch äquivalente Methoden sind nicht zwangsläufig syntaktisch äquivalent. Nehmen wir an, die Methode m_R nutzt das Attribut a_R. Die Methode n_S wird dadurch erzeugt, dass m_R kopiert und a_R durch ein gleichheitsabhängiges Attribut b_S ersetzt wird. m_R und n_S sind dann zwar semantisch, aber nicht syntaktisch äquivalent. Eigentlich ist man an der Untermenge der semantisch äquivalenten Methoden interessiert, deren Äquivalenz nach Änderungen automatisch angepasst werden kann, wenn eine der gleichheitsabhängigen Methoden geändert wird.

Solche gleichheitsabhängigen Methoden könnten `getNickName` in der Rolle `FilmEnthusiast` und `getName` in `Person` sein. Attribute bzw. Methoden verschiedener Rollen können verschiedene Namen, aber gleichen Wert bzw. gleiche Bedeutung haben. Umgekehrt können sie gleiche Namen haben, aber voneinander nicht abhängig sein. Im Code wird die Gleichheitsabhängigkeit zwischen Methoden mit dem Schlüsselwort `equal` deklariert:

```
equal(FilmEnthusiast:getNickName(), Person:getName());
```

Die Abhängigkeit der Attribute unterscheidet sich grundlegend von der Abhängigkeit der Methoden. Bei den Attributen handelt es sich um eine Laufzeitabhängigkeit: Wenn sich während der Laufzeit der Wert eines Attributs ändert, wird der Wert des gleichheitsabhängigen Attributs ebenfalls geändert. Die Abhängigkeit zwischen Methoden spielt dagegen zur Entwicklungszeit eine Rolle, da Methoden zur Laufzeit für gewöhnlich nicht verändert werden. Sollte ein Programmierer während der Entwicklung eine Methode modifizieren, so wird eine gleichheitsabhängige Methode automatisch angepasst.

Definition 16. *Eine* Klasse *ist ein Tupel* $C = (N_C, R_C, D_C)$. *Dabei steht* N_C *für den Klassennamen,* R_C *für eine geordnete Menge von Rollen und* D_C *für Abhängigkeiten zwischen diesen Rollen.*

$$R >_{pr} S$$

bedeutet, dass die Rolle R eine größere Priorität hat als die Rolle S.

Rollen bilden also eine Ebene zwischen Klassen auf der einen Seite und Attributen und Methoden auf der anderen. Ein Objekt besitzt die volle Komplexität seiner Klasse und enthält alle Rollen dieser Klasse und deren Attribute und Methoden.

In [CD05a, CD05b] wird funktionale Abhängigkeit eingeführt. Der Zweck besteht darin, dass Attribute durch eine Funktion abhängig sind, z.B.

$$a_R = f(b_S)$$

So kann z.B. eine Größe in einer Rolle in Metern und dieselbe Größe in einer anderen Rolle in Zentimetern gespeichert sein. Die Implementierung und die exakte Semantik der funktionalen Abhängigkeit sind nicht zufrieden stellend ausgearbeitet, so dass ich an dieser Stelle nicht weiter darauf eingehe.

A.2.2 Abhängigkeiten beim Einsatz der Vererbung

Mit

$$R < S$$

wird bezeichnet, dass die Rolle R von der Rolle S erbt.

$$R \leq S$$

bedeutet, dass R von S erbt oder dass es sich bei R und S um die gleiche Rolle handelt.

Ein Spezialfall der Vererbung liegt vor, wenn verschiedene Rollen einer Klasse von der gleichen Rolle erben. Wenn man keine Abhängigkeiten zwischen den erbenden Rollen deklariert, sind diese unabhängig. Oft ist es jedoch sinnvoll, dass alle geerbten Merkmale gleichheitsabhängig sind.

Definition 17. *Zwei Rollen sind genau dann* vererbungsabhängig, *wenn ein gemeinsamer Vorfahre beider Rollen existiert und alle dessen Attribute und nicht überschriebene Methoden in den erbenden Rollen gleichheitsabhängig sind.*

Seien R, S und T Rollen. Dabei ist T ein Vorfahre von R und S. Formal kann der Sachverhalt der Definition folgendermaßen beschrieben werden:
$R \sim_T S \Leftrightarrow$

1. Für jedes Attribut $a_T \in A_T$ und seine vererbten Entsprechungen $a_R \in A_R$ und $a_S \in A_S$ gilt $\quad a_R \sim a_S$.
2. Für jede Methode $m_T \in M_T$ und ihre vererbten nicht überschriebenen Entsprechungen $m_R \in M_R$ und $m_S \in M_S$ gilt $\quad m_R \sim m_S$.

In Abbildung A.2 ist eine Klasse `C` mit Rollen `R` und `S` dargestellt. `U` und `V` sind Klassen, die eine einzige gleichnamige Rolle enthalten. Sowohl `R` als auch `S` erben von der Rolle `V`. Die Vererbungsabhängigkeit ist durch einen waagerechten Strich gekennzeichnet. Die Rollen `R` und `S` haben zwei gemeinsame Vorfahren, nämlich `U` und `V`. Man muss syntaktisch den Vorfahren spezifizieren, über den die Rollen abhängig sind. Die folgende Deklaration in der Klasse `C`

```
inheritanceDependent(R,S,V);
```

bedeutet, dass `R` und `S` über `V` vererbungsabhängig sind. Dies wird als

$$R \sim_V S$$

bezeichnet.

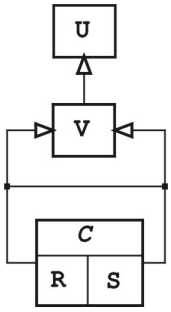

Abbildung A.2: Gleiche Oberrolle

Es ist ebenfalls möglich, dass die Rollen nicht über den direkten Vorfahren abhängig sind (siehe Abbildung A.3). In diesem Fall sind `R` und `S` über `U` vererbungsabhängig, und `U` ist nicht der direkte Vorfahre von `R`.

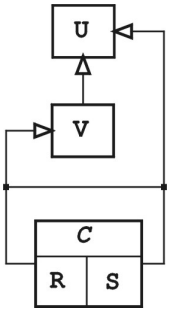

Abbildung A.3: Verschiedene Oberrollen

Lemma 8. *Seien* R, S, T *und* U *Rollen. Außerdem seien* R *und* S *vererbungsabhängig über* T. *Wenn* U *ein Vorfahre von* T *ist, dann sind* R *und* S *vererbungsabhängig über* U

$$R \sim_T S \wedge T \leq U \Rightarrow R \sim_U S.$$

Beweis. Da alle Attribute von T gleichheitsabhängig in R und S sind, ist jede Teilmenge dieser Attribute gleichheitsabhängig. Wegen der Vererbung sind Attribute von U eine Teilmenge der Attribute von T

$$T < U \Rightarrow A_U \subseteq A_T.$$

Daher sind alle Attribute von U in R und S gleichheitsabhängig. Analog zeigt man, dass nicht überschriebene Methoden von U gleichheitsabhängig in R und S sind. Somit sind R und S vererbungsabhängig über U. □

Ein weiterer Spezialfall der Vererbung besteht darin, dass mehrere Rollen einer Klasse von mehreren Rollen einer anderen Klasse erben. Ohne Klassenabhängigkeit (siehe Definition 18) müsste jede Abhängigkeit zwischen den Rollen zweifach deklariert werden, nämlich bei den Ober- und bei den Unterrollen. Mit Hilfe der Klassenabhängigkeit wird diese Redundanz beseitigt.

Definition 18. *Sei die Rolle* R1 *ein Vorfahre der Rolle* R2 *und* S1 *ein Vorfahre der Rolle* S2. *Die Rollen* R1 *und* S1 *gehören zur Klasse* C1 *und die Rollen* R2 *und* S2 *zur Klasse* C2. *Die Rollen* R2 *und* S2 *sind* klassenabhängig *über* R1, S1 *und* C1 *genau dann, wenn alle Abhängigkeiten der geerbten Attribute und der nicht überschriebenen Methoden der Rollen* R1 *und* S1 *in* C2 *von den Rollen* R2 *und* S2 *übernommen sind.*

Dieser Sachverhalt wird syntaktisch durch

```
classDependent(R2, S2, C1, R1, S1);
```

deklariert. Ich notiere das als

$$R2 \sim_{C1,R1,S1} S2$$

Das folgende Lemma zeigt, dass Klassenabhängigkeit transitiv ist. Es ist möglich, Abhängigkeiten durch mehrere Ebenen zu vererben.

Lemma 9. *Seien* R1 *und* S1 *Rollen der Klasse* C1*,* R2 *und* S2 *Rollen der Klasse* C2 *und* R3 *und* S3 *Rollen der Klasse* C3*. Sei* R1 *Vorfahre von* R2 *und* R2 *Vorfahre von* R3*. Sei* S1 *Vorfahre von* S2 *und* S2 *Vorfahre von* S3*. Weiterhin seien* R2 *und* S2 *klassenabhängig über* C1*,* R1 *und* S1*;* R3 *und* S3 *seien klassenabhängig über* C2*,* R2 *und* S2*. Dann sind* R3 *und* S3 *klassenabhängig über* C1*,* R1 *und* S1*:*

$$\begin{aligned} R1, S1 \in R_{C1} \wedge R2, S2 \in R_{C2} \wedge R3, S3 \in R_{C3} \\ \wedge\, R3 \leq R2 \leq R1 \wedge S3 \leq S2 \leq S1 \\ \wedge\, R2 \sim_{C1,R1,S1} S2 \wedge R3 \sim_{C2,R2,S2} S3 \\ \Rightarrow R3 \sim_{C1,R1,S1} S3. \end{aligned}$$

Beweis. Abbildung A.4 zeigt diesen Sachverhalt. Da die Vererbung transitiv ist, ist R1 ein Vorfahre von R3 und S1 ein Vorfahre von S3:

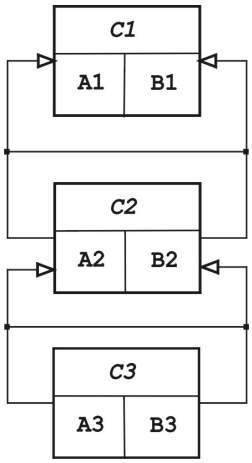

Abbildung A.4: Transitivität bei der Klassenabhängigkeit

$$\begin{aligned} R3 \leq R2 \leq R1 \Rightarrow R3 \leq R1, \\ S3 \leq S2 \leq S1 \Rightarrow S3 \leq S1. \end{aligned}$$

Wegen der Klassenabhängigkeit ist die Menge der abhängigen Attribute von R2 und S2 eine Untermenge der abhängigen Attribute von R3 und S3. Analog ist die Menge der abhängigen Attribute von R1 und S1 eine Untermenge der abhängigen Attribute von R2 und S2. Daher sind alle abhängigen Attribute von R1 und S1 in R3 und S3 ebenfalls abhängig.

Die Argumentation über abhängige nicht überschriebene Methoden verläuft analog. Somit sind R3 und S3 klassenabhängig über C1, R1 und S1. □

A.2.3 Weitere Aspekte der Vererbung

Zusätzlich zu den Sichtbarkeitsstufen `public`, `protected` und `private` wird in RCC eine neue Stufe `inner` eingeführt. Sie ist zwischen `protected` und `public` angesiedelt.

Private Eigenschaften sind nur in einer Rolle zugreifbar, protected in der Vererbungshierarchie einer Rolle. Mit inner kann man zusätzlich auf Eigenschaften anderer Rollen innerhalb derselben Klasse zugreifen und mit public auf Eigenschaften aller Rollen in allen Klassen. Die Standardsichtbarkeitsstufe ist inner.

Nun wird die Vererbung von Klassen definiert. Dies ist sinnvoll, denn auch in der realen Welt gibt es Hierarchien von natürlichen Typen.

Definition 19. *Seien*
$C = (N_C, \{R_1, \ldots, R_n, S_1, \ldots, S_m\}, D_C)$ *und*
$D = (N_D, \{R_1, \ldots, R_n, T_1, \ldots, T_m, U_1, \ldots, U_k\}, D_D)$
Klassen. `D` *erbt von* `C` *genau dann, wenn folgende Bedingungen erfüllt sind:*

$$T_1 < S_1, \ldots, T_m < S_m$$
$$\forall i, j \in \{1, \ldots, m\} \; mit \; i \neq j : T_i \sim_{C,S_i,S_j} T_j$$
$$\forall i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \; : R_i \sim_{C,R_i,S_j} T_j$$

Das bedeutet erstens, dass Rollen von der Oberklasse in die Unterklasse übernommen werden. Zweitens werden die Abhängigkeiten zwischen den Rollen von der Oberklasse übernommen und können erweitert werden. Drittens können neue Rollen zu der Unterklasse hinzugefügt werden. Ein Objekt der Unterklasse kann zu jedem Zeitpunkt als Objekt der Oberklasse betrachtet werden [Lis88]. Um das Substitutionsprinzip einzuhalten, dürfen Vorbedingungen nur erhalten oder abgeschwächt, nie jedoch verstärkt werden und Nachbedingungen nur erhalten oder verstärkt, nie jedoch abgeschwächt werden [Hoa86]. Dies ist in Definition 19 der Fall.

Im Folgenden zeige ich, warum eine *Menge* der Rollen in einer Klasse nicht ausreicht und eine *geordnete Menge* bzw. Prioritäten nötig sind (siehe Definition 16). Als statischer Typ kann entweder die Rolle selbst oder ihre Oberrolle angegeben werden. Nehmen wir an, die Rolle `T` hat ein Attribut `a`. Die Rollen `R` und `S` der Klasse `C` erben beide von `T` und sind nicht vererbungsabhängig. Somit kann das Attribut `a` in `R` und `S` verschiedene Werte annehmen. Weiterhin habe die Variable `c` den dynamischen Typ `C`.

Die Zuweisung

```
1 R r = (R) c;
2 System.out.println(r.getA());
```

spezifiziert exakt die Rolle und somit den Wert von `a`. Aber wenn nun die Oberrolle `T` eingesetzt wird,

```
1 T t = (T) c;
2 System.out.println(t.getA());
```

kann der Compiler nicht unterscheiden, ob es sich um den Wert von `a` in `R` oder in `S` handelt, denn beide Rollen können durch `T` abstrahiert werden. In diesem Fall wird das Attribut `a` der Rolle mit der höheren Priorität zurückgegeben. Damit ist die Rolle gemeint, die als erste in der Klassendefinition angegeben ist.

Bei der Vererbung von Klassen ist die Änderung der Rollenprioritäten möglich. Dies verletzt nicht die is-Eigenschaft der Vererbung. Die Situation soll nun an einem Beispiel

veranschaulicht werden. Nehmen wir an, die Klasse `C` hat die Rollen `R` und `S`. `R` habe eine höhere Priorität als `S`: $R >_{pr} S$. Die Klasse `D` erbt von `C` und hat ebenfalls die Rollen `R` und `S`, diesmal in der umgekehrten Reihenfolge $R <_{pr} S$. Betrachten wir zwei Zuweisungen

```
1 R r = (R) value;
2 S s = (S) value;
```

Nehmen wir an, dass `value` den dynamischen Typ `C` hat. Wenn `S` von `R` erbt, hat die Variable `r` die aktuelle Rolle `R`, weil `R` eine höhere Priorität hat als `S`. Bei der zweiten Anweisung hat `s` die aktuelle Rolle `S`, weil `R` mit höherer Priorität nicht spezifisch genug ist. Nehmen wir nun an, dass `value` den dynamischen Typ `D` hat. Bei der ersten Zuweisung hat `r` die Rolle `S`, weil `S` die höhere Priorität hat und jede Instanz von `S` aufgrund der Vererbung auch eine Instanz von `R` ist. Bei der zweiten Zuweisung hat `s` die Rolle `S`.

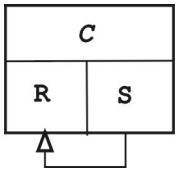

Abbildung A.5: Vererbung zwischen den Rollen der gleichen Klasse

Das Beispiel in Abbildung A.5 zeigt das Problem der Rollen in einer Klasse, die voneinander erben. Wenn $R < S$ und $R >_{pr} S$, dann gibt es keine Möglichkeit, auf die Rolle `S` zuzugreifen.

Trotz der beschriebenen Einschränkungen lösen die Prioritäten das Problem der Nichteindeutigkeit. Diese Lösung ist nicht konform mit der realen Welt, denn es existiert keine Möglichkeit, abhängig vom Kontext die Prioritätsordnung zu ändern. Ein alternativer RCC-Ansatz arbeitet mit Kontexten [CD05a]. Ein weiterer Ansatz spezifiziert die Rolle durch einen Cast, ähnlich der Notation in [CG90] z.B.:

```
((S) value).getField();
```

A.3 Vergleich

Da sehr viele Rollenansätze existieren, sprengt ein Vergleich aller dieser Ansätze den Rahmen der Arbeit. Hier werden die sechs von ihnen ausgewählt, die ermöglichen, dass nur die Eigenschaften einer Rolle und nicht alle Eigenschaften der Entität zugreifbar sind. Mehrfach- und Interfacevererbung sind häufige Modellierungsansätze, die durch einfache Anwendung der Vererbung umgesetzt sind (siehe Abschnitt 4.3.2 und Unterkapitel A.1). Role Object Pattern ist das allgemein anerkannte Design Pattern für Rollen (siehe Abschnitte 3.1.6 und 4.3.2). Object Teams ist eine Spracherweiterung, die Rollen mittels aspektorientierter Programmierung umsetzt. Dieser Ansatz wird hier als Repräsentant der aspektorientierten Ansätze ausgewählt (siehe Abschnitt 3.2.5 und Unterkapitel

A.1). Schließlich handelt es sich bei RCC (siehe Unterkapitel A.2) und parametrisierten Zuweisungen (siehe Abschnitt 6.4.1) um zwei Ansätze, die über Sprachkonstrukte für Abhängigkeiten verfügen.

A.3.1 Vergleichskriterien

Ich unterscheide zwischen Rollenmerkmalen, also Anforderungen an die Rollenansätze, und übergeordneten Eigenschaften, die bei jedem objektorientierten Ansatz wichtig sind.

Die Rollenmerkmale werden aus [Kri96, Ste00, Laz05, CLD05] entnommen:

- *Modulkohäsion* besagt, dass logisch zusammenhängende Komponenten zusammen auftreten [SMC74, Par72, CD04].
- Das Kriterium *Abhängigkeiten* ist erfüllt, wenn Rollen gemeinsame und unabhängige Eigenschaften haben können.
- *Dynamik* ist gegeben, wenn bei einer Entität Rollen zur Laufzeit hinzugefügt und entfernt werden können.
- Ein Objekt und seine Rollen sollten die *gleiche Identität* haben.
- *Mehrfache Rollenverwendung* bedeutet, dass ein Objekt die gleiche Rolle mehrfach verwenden kann.
- *Geheimnisprinzip* wird befolgt, wenn beim Zugriff auf eine Rolle die Eigenschaften anderer Rollen verborgen bleiben [DD02].
- Sowohl die *Hierarchie* von Rollentypen als auch von natürlichen Typen soll gebildet werden können.

Zu den übergeordneten Eigenschaften [Laz05, CLD05] gehören

- Nähe zum Objektmodell,
- Wartung und Erweiterbarkeit,
- Verständlichkeit,
- Dokumentation und
- Entwicklungsstadium.

Zwischen den einzelnen Kriterien existieren Abhängigkeiten. So hängt z.B. die Verständlichkeit eines Ansatzes von seiner Nähe zum Objektmodell ab. Dies liegt daran, dass viele Entwickler mit dem Objektmodell bereits vertraut sind.

A.3.2 Gegenüberstellung der Ansätze

Die Befolgung des Geheimnisprinzips ist so wichtig, dass die zu vergleichenden Ansätze so ausgewählt sind, dass sie alle dieses Kriterium erfüllen. Bei jedem Ansatz ist es möglich, ausschließlich die Rolle einer Entität anzusprechen. Bei Mehrfach-, Interfacevererbung und RCC geschieht das durch die Spezifikation des statischen Typs. Bei Role Objekt, Object Teams und parametrisierten Zuweisungen haben die Clients den Zugriff ausschließlich auf das relevante Rollenobjekt.

In den Tabellen A.1 und A.2 ist der Vergleich der sechs Rollenansätze zusammengefasst. Die Bezeichnungen der Ansätze sind abgekürzt: MV – Mehrfachvererbung, IV – Interfacevererbung, ROP – Role Object Pattern, OT – Object Teams, RCC – Roles as Components of Classes und PZ – parametrisierte Zuweisungen. „+“ bedeutet, dass ein Kriterium erfüllt, „−“ dass es nicht erfüllt und „○“ dass es nicht ganz zufrieden stellend erfüllt ist.

	MV	IV	ROP	OT	RCC	PZ
Modulkohäsion	+	−	+	+	+	○
Abhängigkeiten	−	+	−	−	+	+
Dynamik	−	−	+	○	−	+
gleiche Identität	+	+	−	○	+	−
mehrfaches Spielen derselben Rolle	−	−	+	+	+	+
Hierarchie	+	+	+	○	+	○

Tabelle A.1: Rollenmerkmale

Betrachten wir zunächst die Mehrfachvererbung. Ein wichtiger Nachteil der Mehrfachvererbung im Bezug auf Rollen besteht darin, dass Rollen zur Laufzeit nicht hinzugefügt oder entfernt werden können. Das liegt daran, dass Oberklassen bereits zur Compilezeit feststehen und üblicherweise nicht zur Laufzeit geändert werden können. Ein weiterer Nachteil besteht darin, dass es nicht möglich ist, auf eine einfache Art und Weise Abhängigkeiten zwischen Attributen zu deklarieren. In [Laz05] wird eine Möglichkeit vorgestellt, dies durch Modifizierung der setter-Methoden umzusetzen. Weiterhin kann eine Rolle nicht mehrfach gespielt werden, da diese durch den statischen Typ definiert wird. Eine Unterscheidung zwischen zwei gleichen statischen Typen ist nicht möglich: Eine Person, die zwei Jobs hat, muss man anders modellieren als eine mit einem Job. Auf der anderen Seite ist die Modulkohäsion gut erfüllt, da jede Rolle ihre Eigenschaften in einer eigenen Klasse verwalten kann. Weiterhin haben die Rollen und die Entität dieselbe

Identität aufgrund der is-Eigenschaft der Vererbung. Schließlich sind Hierarchien sowohl von Rollentypen als auch von natürlichen Typen möglich, da es sich in beiden Fällen um Klassen handelt.

Vor- und Nachteile der Mehrfachvererbung werden bereits seit langer Zeit diskutiert [Mez98]. Mittlerweile ist die Mehrheit der Programmiersprachenentwickler davon überzeugt, dass die Nachteile die Vorteile überwiegen. Daher verfügen die meisten modernen Programmiersprachen nur über Einfach- und Interfacevererbung. Die bekanntesten Nachteile der Mehrfachvererbung sind Namenskonflikte und das Diamond Problem [Mez98]. Namenskonflikte treten auf, wenn verschiedene Oberklassen einer Unterklasse gleichnamige Attribute bzw. Methoden mit der gleichen Signatur aufweisen. Sobald die Unterklasse ein solches Attribut oder eine solche Methode verwendet, kommt es zu einem Fehler, da nicht klar ist, welches Attribut oder welche Methode angesprochen wird. Mit Hilfe geeigneter Upcasts kann man dieses Problem in Einzelfällen lösen. Diamond Inheritance tritt auf, wenn eine Unterklasse von einer Oberklasse auf mehreren Pfaden erbt. Alternative Ansätze zur Vermeidung von Namenskonflikten und dem Diamond Problem findet man in [Mez98].

Mehrfachvererbung wird nicht von allen objektorientierten Sprachen unterstützt. Insbesondere verfügen moderne Programmiersprachen wie Java und C# nur über Interfacevererbung. Nichtsdestotrotz ist Mehrfachvererbung ein verständlicher Ansatz, da der Zugriff auf Rollen durch einen einfachen Upcast geschieht. Außerdem entspricht dieser Ansatz dem objektorientierten Paradigma. Mehrfachvererbung existiert bereits seit langer Zeit, so dass der Entwicklungsstand weit fortgeschritten und die Dokumentation umfangreich ist. Jedoch gibt es fast keine Dokumentation bezüglich Rollen. Zur graphischen Modellierung verwendet man UML.

Mit Hilfe der Interfaces wird das Diamond Problem der Mehrfachvererbung beseitigt. Die Namenskonflikte zwischen den Oberklassen werden dadurch gelöst, dass die Methoden keinen Inhalt haben, und dieser sich somit nicht in verschiedenen Interfaces unterscheiden kann. Dafür wird der Nachteil in Kauf genommen, dass die Rollenfunktionalität nicht in Oberklassen ausgelagert werden kann, so dass Unterklassen komplexer werden, und Module nicht sauber getrennt sind. Dies erschwert die Wartbarkeit der Anwendungen. Abhängigkeiten können umgesetzt werden, da die ganze Funktionalität innerhalb einer Klasse enthalten ist: Methoden nutzen die gleichen Attribute oder rufen einander auf.

Das Entwicklungsstadium der Interfacevererbung ist weit fortgeschritten, und sie wird von vielen objektorientierten Sprachen unterstützt. Die Dokumentation bezüglich Rollen ist ausführlich [Ste01].

Vereinfacht gesagt, wird die Vererbung in den ersten beiden besprochenen Ansätzen in Role Object Pattern durch Delegation ersetzt. Dadurch ist dieser Ansatz dynamisch: Rollen können sowohl hinzugefügt als auch entfernt werden. Auf der anderen Seite haben die Rollen und die Entität verschiedene Identitäten aufgrund der has-Eigenschaft der Delegation. Alle weiteren Anforderungen an die Rollen sind bis auf Abhängigkeiten erfüllt. Wenn bei den Rollen viele Abhängigkeiten erfordert sind, stößt Role Object Pattern an seine Grenzen, denn die Schnittmenge der einzelnen Rollen wird in das Core Object ausgelagert. Da die einzelnen Schnittmengen unterschiedlich sein können, braucht man

mehrere Core Objekte, was zur Verkomplizierung des Designs führt. Im ungünstigsten Fall werden bei einem Rollenszenario $\frac{n(n-1)}{2}$ Core Objekte für n Rollen gebraucht, um die Schnittmengen zwischen je zwei Rollen auszulagern.

Die allgemeinen Eigenschaften von Role Object Pattern sind zufrieden stellend erfüllt, da es sich um einen flexiblen und gut erforschten Ansatz handelt. Der einzige Nachteil besteht in der empfundenen Komplexität der entstehenden Designs, da die angesprochene Flexibilität durch Hilfsklassen erreicht wird.

Der aspektorientierte Ansatz Object Teams versucht die Vorteile der besprochenen Ansätze zu vereinen. Diesmal wird die Bindung zwischen Rollentypen und natürlichen Typen weder durch Vererbung noch durch Delegation, sondern durch aspektorientierte Verwebung erreicht. Dies geschieht durch den Einsatz von callins und callouts.

Auf der einen Seite ist etwas Dynamik gegeben, da Methodeninhalte der Rollen abhängig von aktivierten Kontexten variiert werden können. Hinzufügen und Entfernen von Rollen ist leider nicht möglich. Auf der anderen Seite sind Rollen und Basisklassen so eng verbunden, dass man von einem zusammengesetzten Objekt sprechen könnte. Abhängigkeiten zwischen den Rollen sind leider nicht möglich. Hierarchien sind sowohl bei den Basisklassen als auch bei Rollen möglich. Die Vererbung von Rollen ist jedoch dadurch eingeschränkt, dass diese nur von Rollen im Oberteam und von Interfaces erben können. Das mehrfache Spielen einer Rolle ist ebenfalls möglich, weil mehrere Teams des gleichen Typs aktiviert sein können.

Der größte Vorteil von Object Teams besteht darin, dass aspektorientierte Programmierung eine einfache Erweiterung existierender Anwendungen erlaubt. Der größte Nachteil besteht darin, dass die Programmierer viele neue Programmkonstrukte lernen und umdenken müssen, denn Object Teams hat die Nähe zum Objektmodell verloren. Außerdem erfordern mehrere Aktionen, wie z.B. Rollenerzeugung oder Dynamikumsetzung, eine komplexe Implementierung.

Object Teams ist ziemlich ausgereift. Ein Eclipse Plugin mit der Version 1.0 ist bereits entwickelt. Eine ausführliche Sprachdefinition, die viele hilfreiche Beispiele enthält, ist vorhanden. Da dieses Tool nicht sehr verbreitet ist, ist der Dokumentationsumfang nicht so groß wie bei etablierten Ansätzen. In [Her02a] wird eine Erweiterung von UML entwickelt, um Anwendungen mit Object Teams graphisch zu modellieren. Jedoch ist die Toolunterstützung bisher unzureichend.

RCC verfügt über alle gewünschten Rollenmerkmale bis auf Dynamik. Das Kriterium Abhängigkeiten ist besser erfüllt als bei den vorherigen Ansätzen. Nicht nur Gleichheitsabhängigkeiten von Attributen, sondern auch weiterführende Abhängigkeiten sind möglich: Gleichheitsabhängigkeit von Methoden, Vererbungs- und Klassenabhängigkeit.

Obwohl RCC nicht im Objektmodell enthalten ist, handelt es sich dabei um dessen kanonische Erweiterung. Dies trägt zur Verständlichkeit des Ansatzes bei. Der größte Nachteil von RCC besteht darin, dass die Entwicklungsumgebung zwar vorhanden, aber noch nicht ausgereift ist [Ake06]. Die Dokumentation ist bisher ebenfalls nicht ausführlich.

Mit parametrisierten Zuweisungen kann man Rollen mittels Abhängigkeiten zu einer abstrakten Entität verbinden. Der natürliche Typ ist in diesem Fall nicht vorhanden. Die Modulkohäsion ist nicht zufrieden stellend erfüllt, weil die internen Merkmale der Entität über mehrere Rollen zerstreut sind. Dafür ist dieser Ansatz sehr dynamisch und

bewältigt hervorragend Abhängigkeitsanforderungen. Es ist lediglich eine Hierarchie von Rollentypen möglich, nicht die der natürlichen Typen, da es keine expliziten natürlichen Typen gibt. Dafür ist dieser Ansatz so leichtgewichtig, dass er mit anderen Ansätzen kombiniert werden kann, um deren Abhängigkeitspotential zu erweitern.

Auch dieser Ansatz ist eine kanonische Erweiterung des Objektmodells. Ich denke, dass parametrisierte Zuweisungen gut verständlich sind, da sie lediglich eine neue Art von Delegation einführen. Außerdem ist das Kriterium Wartbarkeit und Erweiterbarkeit erfüllt, weil keine starren Konstrukte, wie z.B. durch Hilfsklassen, hinzugefügt werden. Aufgrund der Neuheit der parametrisierten Zuweisungen stehen sowohl die Entwicklung als auch die Dokumentation im Vergleich zu etablierten Ansätzen erst am Anfang.

	MV	IV	ROP	OT	RCC	PZ
Nähe zum Objektmodell	+	+	+	○	○	○
Wartbarkeit und Erweiterbarkeit	○	−	+	+	○	+
Verständlichkeit	+	+	−	−	○	○
Dokumentation	○	+	+	○	−	−
Entwicklungsstand	+	+	+	○	−	−

Tabelle A.2: Allgemeine Eigenschaften

A.4 Schlussfolgerung

Der durchgeführte Vergleich zeigt, dass es erwartungsgemäß keinen Ansatz gibt, der bzgl. jedes Kriteriums den anderen überlegen ist. Dabei sind zwei Eigenschaften besonders hervorzuheben: Mächtigkeit und Verständlichkeit. Es zeigt sich, dass je mächtiger ein Ansatz ist, desto komplexer und somit unverständlicher wird er empfunden. Neuere Ansätze schneiden bei Rollenmerkmalen besser ab, dafür weisen sie aufgrund ihrer Neuheit Schwächen bei der Dokumentation und beim Entwicklungsstand auf.

Für einfache Designs, die nur wenige Rollen umfassen, die weder hinzugefügt noch entfernt werden können, eignet sich aufgrund ihrer Einfachheit Interfacevererbung am besten. Wenn dagegen Dynamik erforderlich ist, ist Role Object das gesuchte Pattern. Sind die Abhängigkeiten zwischen den Rollen vielfältig, die Dynamik jedoch zweitrangig, empfiehlt sich RCC. Parametrisierte Zuweisungen sind dann einzusetzen, wenn sowohl die Anforderungen an die Dynamik als auch an die Abhängigkeiten sehr groß sind.

Viel versprechend ist die Kombination aus dem Role Object Pattern und den parametrisierten Zuweisungen. Dabei werden Eigenschaften, die bei allen Rollen gemeinsam sind

in das Core Object ausgelagert und Eigenschaften, die nur in wenigen Rollen gleich sind über parametrisierte Zuweisungen umgesetzt. Diese Kombination verfügt sowohl über die Unterscheidung zwischen Rollen- und natürlichen Typen als auch über die Dynamik als auch über die Möglichkeit, komplexe Abhängigkeit zwischen den Rollen abzubilden.

Literaturverzeichnis

[AFF99] Alves-Foss, J. und D. Frincke: *Formal Grammar for Java.* In: Alves-Foss, J. (Herausgeber): *Formal Syntax and Semantics of Java*, Band 1523 der Reihe *LNCS*, Seiten 1–40. Springer, 1999.

[AGO95] Albano, A., G. Ghelli und R. Orsini: *Fibonacci: A Programming Language for Object Databases.* The VLDB Journal, 4(3):403–439, 1995.

[Ake06] Aker, A.: *Prototypische Entwicklung eines Eclipse-Plugins für Rollen in OOP.* Diplomarbeit, University of Dortmund, 2006.

[And97] Andersen, E. P.: *Conceptual Modeling of Objects: A Role Modeling Approach.* Doktorarbeit, Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo, 1997.

[APS+05] Allan, Ch., Avgustinov P., Christensen A. S., L. Hendren, Kuzins S., O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam und J. Tibble: *Adding Trace Matching with Free Variables to AspectJ.* ACM SIGPLAN Notices, 40(10):345–364, 2005.

[Ast96] Astudillo, H.: *Reorganizing Split Objects.* In: *OOPSLA*, Seiten 138–149, 1996.

[Bal98] Balzert, H.: *Lehrbuch der Software-Technik.* Spektrum Akademischer Verlag, 1998.

[Bar06] Barnes, J.: *Programming in Ada 2005.* Addison-Wesley, 2006.

[Bau03] Baumgart, J.: *Analyse, Entwurf und Generierung von Rollen- und Variantenmodellen.* Doktorarbeit, University of Darmstadt, 2003.

[Bau04] Baustert, T.: *Aspektorientierte Programmierung - Beispiele mit AspectJ.* Resco GmbH, 2004.

[Bau05] Baustert, T.: *Aspektorientierte Programmierung mit AspectJ.* Eclipse Magazin, (3), 2005.

[BD96] Bardou, D. und Ch. Dony: *Split Objects: a Disciplined Use of Delegation within Objects.* In: *OOPSLA*, Seiten 122–137, 1996.

[Bec03] Beck, K.: *Extreme Programming.* Addison-Wesley, München, Dezember 2003.

[Bee80] Beeri, C.: *On the Membership Problem for Functional and Multivalued Dependencies in Relational Databases.* ACM Transactions on Database Systems, 5(3):241–259, 1980.

[BK00] Bækdal, L. K. und B. B. Kristensen: *Perspectives and Complex Aggregates.* In: *Proceedings of the 6th International Conference on Object-Oriented Information Systems*, England, 2000. Springer.

[Boo95] Booch, G.: *Objektorientierte Analyse und Design.* Addison-Wesley, Bonn, 1995.

[BRSW00] Bäumer, D., D. Riehle, W. Siberski und M. Wulf: *Role Object.* In: *Pattern Languages of Program Design*, Seiten 15–32. Addison-Wesley, 2000.

[BS99] Börger, E. und W. Schulte: *A Programmer Friendly Modular Definition of the Semantics of Java.* In: Alves-Foss, J. (Herausgeber): *Formal Syntax and Semantics of Java*, Band 1523 der Reihe *LNCS*, Seiten 353–404. Springer, 1999.

[CD04] Chernuchin, D. und G. Dittrich: *Aspekte der Objektorientierten Modellierung am Beispiel Evolutionärer Algorithmen.* Forschungsbericht 791, University of Dortmund, 2004.

[CD05a] Chernuchin, D. und G. Dittrich: *Dependencies of Roles.* In: *Views, Aspects and Roles at ECOOP '05*, 2005.

[CD05b] Chernuchin, D. und G. Dittrich: *Role Types and their Dependencies as Components of Natural Types.* In: *2005 AAAI Fall Symposium: Roles, an Interdisciplinary Perspective*, Seiten 39–46, 2005.

[CG90] Carré, B. und J.-M. Geib: *The Point of View Notion for Multiple Inheritance.* ACM SIGPLAN Notices, 25(10):312–321, 1990.

[Che03] Chernuchin, D.: *Zur objektorientierten Modellierung Evolutionärer Algorithmen.* Diplomarbeit, University of Dortmund, 2003.

[Cho59] Chomsky, N.: *On Certain Formal Properties of Grammars.* Information and Control, 2:137–167, 1959.

[CLD05] Chernuchin, D., O. Lazar und G. Dittrich: *Comparison of Object-Oriented Approaches for Roles in Programming Languages.* In: *2005 AAAI Fall Symposium: Roles, an Interdisciplinary Perspective*, Seiten 31–38, 2005.

[Coo06] Cook, S.: *Object Technology - A Grand Narrative?* In: Thomas, D. (Herausgeber): *ECOOP '06*, Band 4067 der Reihe *LNCS*, Seiten 174–179. Springer, 2006.

[DD02] Doberkat, E.-E. und S. Dissmann: *Einführung in die objektorientierte Programmierung mit Java.* Oldenburg, 2002.

[DDM07] Dietl, W., S. Drossopoulou und P. Müller: *Generic Universe Types.* In: *ECOOP*, 2007.

[DE99] Drossopoulou, S. und S. Eisenbach: *Describing the Semantics of Java and Proving Type Soundness.* LNCS, 1523:41–82, 1999.

[Dij68] Dijkstra, E.W.: *Go To Statement Considered Harmful.* CACM, 11(3):147–148, 1968.

[Dij76] Dijkstra, E.W.: *Discipline of Programming.* Prentice Hall, 1976.

[DRP01] Dustin, E., J. Rashka und J. Paul: *Software automatisch testen.* Springer, Berlin, 2001.

[DW02] Denecke, K. und S.L. Wismath: *Universal Algebra and Applications in Theoretical Computer Science.* Chapman and Hall/CRC Press, 2002.

[Ecl06] Eclipse Foundation: *Eclipse*, 2006. www.eclipse.org.

[FGM99] Floyd, C., G. Gryczan und J. Mack: *Softwaretechnik.* Vorlesungsskript, University of Hamburg, 1999.

[Fow97] Fowler, M.: *Dealing with Roles*, 1997. www.martinfowler.com.

[Fow99] Fowler, M.: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Fow04] Fowler, M.: *Inversion of Control Containers and the Dependency Injection pattern*, 2004. martinfowler.com/articles/injection.html.

[GB02] Graversen, K. B. und J. Beyer: *Chameleon.* Diplomarbeit, IT-University of Copenhagen, 2002.

[GB03] Gamma, E. und K. Beck: *Contributing to Eclipse.* Addison Wesley, 2003.

[GHJV97] Gamma, E., R. Helm, R. Johnson und J. Vlissides: *Design Patterns.* Addison-Wesley, 1997.

[GM05] Gil, J. und I. Maman: *Micro Patterns in Java Code.* ACM SIGPLAN Notices, 40(10):97–116, 2005.

[Gra89] Graver, J: *Type-Checking and Type-Inference for Object-Oriented Programming Languages.* Doktorarbeit, University of Illinois, 1989.

[Gru05] Grude, U.: *Grammatiken und Parser*, 2005. www.tfh-berlin.de/∼grude/.

[GS02] Grant, M. und S.F. Smith: *Programming Languages.* 2002. www.cs.jhu.edu/∼scott/plbook/.

[GSR96] Gottlob, G., M. Schrefl und B. Roeck: *Extending Object-Oriented Systems with Roles.* ACM Press, 14(3):268–296, 1996.

[Has80] Hasse, H.: *Number Theory*, Band 229. Springer, 1980.

[HDSM05] Hauswirth, M., A. Diwan, P.F. Sweeney und M.C. Mozer: *Automating Vertical Profiling.* ACM SIGPLAN Notices, 40(10):281–296, 2005.

[Her02a] Herrmann, S.: *Composable Designs with UFA.* In: *Workshop on Aspect-Oriented Modeling with UML at 1st Intl. Conference on Aspect Oriented Software Development*, 2002.

[Her02b] Herrmann, S.: *ObjectTeams: Improving Modularity for Crosscutting Collaborations*, 2002. Technical University of Berlin.

[Her05] Herrmann, S.: *Programming with Roles in ObjectTeams/Java.* In: *2005 AAAI Fall Symposium: Roles, an Interdisciplinary Perspective*, 2005.

[HLR+99] Hakonen, H., V. Leppänen, T. Raita, T. Salakoski und Teuhola J.: *Improving Object Integrity and Preventing Side Effects via Deeply Immutable References.* In: *FUSST*, Seiten 139–150, 1999.

[Hoa86] Hoare, C. A. R.: *Mathematics of Programming.* BYTE, Seiten 115–126, August 1986.

[HOT97] Harrison, D., H. Ossher und P. Tarr: *Using Delegation for Software and Subject Composition.* Technischer Bericht RC 20946, IBM Thomas J. Watson Research Center, 1997. www.research.ibm.com/sop/abstracts/delegation.htm.

[HOT04] Harrison, D., H. Ossher und P. Tarr: *Concepts for Describing Composition of Software Artifacts.* IBM Research Report, IBM, 2004.

[HR03] Heller, P. und S. Roberts: *Complete Java 2 Certification Study Guide.* Sybex, 2003.

[HSU05] Hanenberg, S., D. Stein und R. Unland: *Roles From an Aspect-Oriented Perspective.* In: *Views, Aspects and Roles at ECOOP*, 2005.

[HU02] Hanenberg, S. und R. Unland: *Roles and Aspects: Similarities, Differences, and Synergetic Potential.* In: *8th International Conference on Object-Oriented Information Systems*, France, 2002.

[Jac02] Jackson, D.: *Module Dependences in Software Design.* In: Wirsing, M., A. Knapp und S. Balsamo (Herausgeber): *RISSEF*, Band 941 der Reihe *LNCS*, Seiten 198–203. Springer, 2002.

[JDAO04] Jolly, P., S. Drossopoulou, C. Anderson und K. Ostermann: *Simple Dependent Types: Concord*, 2004. citeseer.ist.psu.edu/708077.html.

[Jec02] Jech, Th.: *Set Theory.* Springer, 2002.

[JP04] Jacobs, B. und E. Poll: *Java Program Verification at Nijmegen: Developments and Perspective*. In: Futatsugi, K., F. Mizoguchi und N. Yonezaki (Herausgeber): *Software Security – Theories and Systems*, Band 3233 der Reihe *LNCS*, Seiten 134–153. Springer, 2004.

[JUn06] *JUnit*, 2006. www.junit.org.

[KE99] Kemper, A. und A. Eickler: *Datenbanksysteme*. Oldenburg, 1999.

[Ken99] Kendall, A.: *Role Model Designs and Implementations with Aspect-Oriented Programming*. In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seiten 353–369, USA, 1999. ACM Press.

[KK96] Kassera, W. und V. Kassera: *Turbo Pascal 7.0 Kompendium*. Markt & Technik, 1996.

[Kla04] Klaeren, H.: *Softwaretechnik*. Vorlesungsskript, University of Tübingen, 2004.

[KLM+97] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier und J. Irwin: *Aspect-Oriented Programming*. In: *ECOOP*, Band 1241, Seiten 220–242. Springer-Verlag, 1997.

[KM05a] Kiczales, G. und M. Mezini: *Aspect-Oriented Programming and Modular Reasoning*. In: *ICSE'05*, Seiten 49–58, 2005.

[KM05b] Kiczales, G. und M. Mezini: *Separation of Concerns with Procedures, Annotations, Advice and Pointcuts*. In: Black, A.P. (Herausgeber): *ECOOP*, Band 3586 der Reihe *LNCS*, Seiten 195–213. Springer, 2005.

[KMP95] Kaasbøll, J. und R. Motschnig-Pitrik: *Lifetime Dependency: An Abstraction Relation for Modelling Roles, Symbolic Substance, and Relations with Attributes*, 1995. citeseer.ist.psu.edu/524229.html.

[Kni00] Kniesel, G.: *Darwin – Dynamic Object-Based Inheritance with Subtyping*. Doktorarbeit, University of Bonn, 2000.

[Knu97] Knuth, D.E.: *Fundamental Algorithms*, Band 1. Addison-Wesley, 1997.

[KØ96] Kristensen, B. B. und K. Østerbye: *Roles: Conceptual Abstraction Theory and Practical Language Issues*. TAPOS, Seiten 143–160, 1996.

[KP88] Krasner, G.E. und S.T. Pope: *A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80*. Journal of Object Oriented Programming, 1(3):26–49, 1988.

[Kri96] Kristensen, B. B.: *Object-Oriented Modelling with Roles*. In: *Proceedings of the 2nd International Conference on Object-oriented Information Systems*, Ireland, 1996.

[Kri01] KRISTENSEN, B. B.: *Subjective Behavior.* International Journal of Computer Systems Science and Engineering, 16(1):13–24, 2001.

[Kru04] KRUCHTEN, PH.: *The Rational Unified Process.* Addison-Wesley, 2004.

[KT01] KNIESEL, G. und D. THEISEN: *JAC – Access Right Based Encapsulation for Java.* Software – Practice and Experience, 31(6):555–576, 2001.

[Laz05] LAZAR, O.S.: *Vergleich objektorientierter Ansätze für Rollen.* Diplomarbeit, University of Dortmund, 2005.

[Lep92] LEPAGE, F.: *Partial Functions in Type Theory.* Notre Dame Journal of Formal Logic, 33(4):493–516, 1992.

[Lew99] LEWIS, S.: *The Art and Science of Smalltalk.* Hewlett-Packard, 1999.

[Lie86] LIEBERMAN, H.: *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.* Seiten 214–223, 1986.

[Lis88] LISKOV, B.: *Data Abstraction and Hierarchy.* ACM SIGPLAN Notices, 23(5):17–34, 1988.

[MB98] MATTHEY, TH. und H. BIERI: *An Object-Oriented Approach to Model Scenes of Buildings.* In: *Computer Graphics International*, Seiten 14–23. IEEE, 1998.

[Mey97] MEYER, B.: *Object-Oriented Software Construction.* Sams, 1997.

[Mez98] MEZINI, M.: *Variational Object-Oriented Programming Beyond Classes and Inheritance.* Kluwer, 1998.

[MH06] MCDIRMID, S. und W.C. HSIEH: *SuperGlue: Component Programming with Object-Oriented Signals.* In: THOMAS, D. (Herausgeber): *ECOOP*, Seiten 206–229. Springer-Verlag, 2006.

[Mic05] MICROSOFT: *C# Language Specification*, 2005. msdn.microsoft.com/vcsharp/programming/language/default.aspx.

[Nau60] NAUR, P.: *Revised Report on the Algorithmic Language ALGOL 60.* Communications of the ACM, 3(5):299–314, 1960.

[OKH+95] OSSHER, H., M. KAPLAN, W. HARRISON, A. KATZ und V. KRUSKAL: *Subject-Oriented Composition Rules.* ACM SIGPLAN Notices, 30(10):235–250, 1995.

[OKK+96] OSSHER, H., M. KAPLAN, A KATZ, W. HARRISON und K. VINCENT: *Specifying Subject-Oriented Composition.* In: *Theory and Practice of Object Systems*, Band 2, Seiten 179–202. Wiley & Sons, 1996.

[OM01] Ostermann, K. und M. Mezini: *Object-Oriented Composition Untangled.* In: *OOPSLA*, Seiten 283–299, 2001.

[OMG05] OMG: *Unified Modeling Language Specification 2.0*, 2005. www.uml.org/.

[ON99] Oheimb, von D. und T. Nipkow: *Machine-Checking the Java Specification: Proving Type-Safety.* In: Alves-Foss, J. (Herausgeber): *Formal Syntax and Semantics of Java*, Band 1523 der Reihe *LNCS*. Springer, 1999.

[Par72] Parnas, D. L.: *On the Criteria To Be Used in Decomposing Systems into Modules.* Communications of the Association of Computing Machinery, 15(12):1053–1058, 1972.

[Par78] Parnas, D. L.: *Designing Software for Ease of Extension and Contraction.* In: *Proceedings of the Third International Conference on Software Engineering*, Seiten 264–277. IEEE, 1978.

[Per90] Pernici, B.: *Objects with Roles.* In: *Proceedings of the ACM-IEEE Conference on Office Information Systems*, Seiten 205–215, 1990.

[Pil06] Pillai, P.: *Static Typing versus Dynamic Typing*, 2006. premshree.seacrow.com/writings/typing.

[PJ99] Page-Jones, M.: *Fundamentals of Object-Oriented Design in UML.* Addison-Wesley, 1999.

[PK97] Papazoglou, M. P. und B. J. Krämer: *A Database Model for Object Dynamics.* The VLDB Journal, 6(2):73–96, Mai 1997.

[Qui61] Quine, W.V.O.: *From a Logical Point of View: Logico-Philosophical Essays.* Harper Torchbook, 1961.

[Ree97] Reenskaug, T.: *Role Modeling Enters the Main Stream.* Object EXPERT, Januar 1997.

[Reg91] Reggio, G.: *Entities: An Institution for Dynamic Systems.* In: Ehrig, H., K.P. Jantke, F. Orejas und H. Reichel (Herausgeber): *Proceedings of Recent Trends in Data Type Specification*, Band 534 der Reihe *LNCS*, Seiten 246–265. Springer, 1991.

[Rie98] Riehle, D.: *Bureaucracy.* In: *Pattern Languages of Program Design 3*, Seiten 163–186. Addison-Wesley, 1998.

[Rie00] Riehle, D.: *Framework Design: A Role Modeling Approach.* Doktorarbeit, ETH Zürich, 2000.

[RNW+03] Robinson, S., C. Nagel, K. Watson, M. Skinner, Glynn J., Z. Greenvoss, S. Allen, B. Harvey und O. Cornex: *C#.* WROX, 2003.

[RRH00] Reilly, E. D., A. Ralston und D. Hemmendinger: *Encyclopedia of Computer Science*. Nature Pub. Group, 2000.

[RS05] Rainsberger, J.B. und S. Stirling: *JUnit Recipes: Practical Methods for Programmer Testing*. Manning Publications, 2005.

[RWL95] Reenskaug, Trygve, P. Wold und O.A. Lehne: *Working with Objects: The OOram Software Engineering Method*. Prentice-Hall, 1995.

[SA97] Schulte, W. und K. Achatz: *Functional Object-Oriented Programming with Object-Gofer*. In: *GI Jahrestagung*, Seiten 552–561, 1997.

[Sch97] Schmidt, D.A.: *Denotational Semantics, A Methology for Language Development*. 1997. www.cis.ksu.edu/ schmidt/text/densem.html.

[Sip97] Sipser, M.: *Introduction to the Theory of Computation*. PWS Publishing, 1997.

[SJSJ05] Sangal, N., Ev. Jordan, V. Sinha und D. Jackson: *Using Dependency Models to Manage Complex Software Architecture*. ACM SIGPLAN Notices, 40(10):167–176, 2005.

[SMC74] Stevens, W. P., G. J. Myers und L. L. Constantine: *Structured Design*. IBM Systems Journal, 2:115–139, 1974.

[Sny86] Snyder, A.: *Encapsulation and Inheritance in Object-Oriented Programming Languages*. ACM SIGPLAN Notices, 21(11):38–45, 1986.

[Sow84] Sowa, J. F.: *Conceptual Structures: Information Processing in Mind and Machine*. In: *Addison-Wesley Systems Programming Series*. Addison-Wesley, 1984.

[SS05] Simmons, H. und A. Schlak: *λ-Calculi and Arithmetic*. 2005. www.cs.man.ac.uk/∼hsimmons/BOOKS/books.html.

[ST02] Schrefl, M. und Th. Thalhammer: *Using Roles in Java*. Journal Software – Practice and Experience, 34(5):449–464, 2002.

[Ste00] Steimann, F.: *On the Representation of Roles in Object-Oriented and Conceptual Modelling*. Data & Knowledge Engineering, 35(1):83–106, 2000.

[Ste01] Steimann, F.: *Role = Interface: A Merger of Concepts*. Journal of Object-Oriented Programming, 14(4):23–32, 2001.

[Ste06] Steimann, F.: *The Paradoxical Success of Aspect-Oriented Programming*. In: *OOPSLA*, Seiten 481–497. ACM Press, 2006.

[Str97] Stroustrup, B.: *The C++ Programming Language*. Addison-Wesley, 1997.

[SU95] SMITH, R. B. und D. UNGAR: *Programming as an Experience: The Inspiration for Self.* In: OLTHOFF, W. (Herausgeber): *ECOOP*, Band 952 der Reihe *LNCS*, Seiten 303–330. Springer, 1995.

[Sun07a] SUN MICROSYSTEMS: *Java API Specification*, 2007. java.sun.com/j2se/1.5.0/docs/api/.

[Sun07b] SUN MICROSYSTEMS: *The Java Language Specification*, 2007. java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

[SW02] SNEED, H.-M. und M. WINTER: *Testen objektorientierter Software.* Hanser, München, 2002.

[Sym99] SYME, D.: *Proving Java Type Soundness.* In: ALVES-FOSS, J. (Herausgeber): *Formal Syntax and Semantics of Java*, Band 1523 der Reihe *LNCS*, Seiten 83–118. Springer, 1999.

[Szy98] SZYPERSKI, C.: *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[Tam02] TAMAI, T.: *Evolvable Programming Based on Collaboration-Field and Role Model.* In: *IWPSE '02: Proceedings of the International Workshop on Principles of Software Evolution*, Seiten 1–5, USA, 2002. ACM Press.

[Tam05] TAMAI, T.: *Conquering the Eight-Tailed Dragon - An Attempt to Deal with Structural and Behavioral Complexities.* In: *ICECCS*, China, 2005.

[TE05] TSCHANTZ, M.S. und M.D. ERNST: *Javari: Adding Reference Immutability to Java.* ACM SIGPLAN Notices, 40(10):211–230, 2005.

[TK02] THANG, N. und T. KATAYAMA: *Collaboration-Based Evolvable Software Implementations: Java and Hyper/J vs. C++-templates Composition.* In: *Proceedings of the International Workshop on Principles of Software Evolution*, Seiten 29–33, USA, 2002.

[TOHS99] TARR, P., H. OSSHER, W. HARRISON und M.P. SUTTON: *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* In: *ICSE*, Seiten 107–119, 1999.

[US91] UNGAR, D. und R.B. SMITH: *SELF: The Power of Simplicity.* Lisp and Symbolic Computation, 4(3):187–205, 1991.

[Utt92] UTTING, M.: *An Object-Oriented Refinement Calculus with Modular Reasoning.* Doktorarbeit, University of New South Wales, 1992.

[Van97] VANHILST, M.: *Role-Oriented Programming for Software Evolution.* Doktorarbeit, University of Washington, 1997.

[Vin97] VINEK, G.: *Objektorientierte Softwareentwicklung mit Smalltalk*. Springer, April 1997.

[VN96] VANHILST, M. und D. NOTKIN: *Using C++ Templates to Implement Role Based Designs*. In: *Proceedings of the International Symposium on Object Technologies for Advanced Software*, Seiten 22–37, 1996.

[Voß05] VOSS, R.: *Untersuchung von Möglichkeiten für die AOP in Java durch deren Erweiterung mit Annotations*. Diplomarbeit, University of Dortmund, 2005.

[VPDMD04] VAN PAESSCHEN, E., W. DE MEUTER und T. D'HONDT: *Domain Modeling in Self Yields Warped Hierarchies*. In: *The MASPEGHI Workshop at ECOOP '04*, 2004.

[WCL97] WONG, R. K., H. L. CHAU und F. H. LOCHOVSKY: *A Data Model and Semantics of Objects with Dynamic Roles*. In: *Proceedings of the 13th International Conference on Data Engineering*, Seiten 402–411. IEEE, 1997.

[Wei03] WEIGEND, M.: *Python GE-PACKT*. Mitp, April 2003.

[Wir90] WIRSING, M.: *Algebraic Specification*. In: LEEUWEN, J. VAN (Herausgeber): *Handbook of Theoretical Computer Science*, Band B der Reihe *Formal Models and Semantics*, Kapitel Algebraic Specification, Seiten 675–788. Elsevier, 1990.

[YC78] YOURDON, E. und L. L. CONSTANTINE: *Structured Design*. Yourdon, 1978.

[Zuc95] ZUCCA, E.: *Implementation of Data Structures in an Imperative Framework*. In: *Recent Trends in Data Type Specifications*, Band 906 der Reihe *LNCS*, 1995.

Symbolverzeichnis

$_1{=}_1$	eins-eins-Zuweisung
$_2{=}_2$	zwei-zwei-Zuweisung
$_2{=}_3$	zwei-drei-Zuweisung
$_3{=}_3$	drei-drei-Zuweisung
$=_f$	Zuweisungsoperator: Die Schreibweise `x` $=_f$ `y` ist äquivalent zu $f(x, y, s)$ beim gegebenen Zustand s.
$\twoheadrightarrow$	partielle Funktion: $f : A \twoheadrightarrow B$ bedeutet, dass f eine partielle Funktion mit dem Definitionsbereich A und Bildbereich B ist.
$* = *$	Zuweisung der Art `*x = *y`, wobei `x` und `y` Zeiger sind
$+$	positive Bewertung
$\circ$	mittlere Bewertung
$-$	negative Bewertung
$::=$	Überführung bei einer Produktion in Backus-Naur-Fom: $v_1 ::= v_2 \mid t$ bedeutet, dass ein Nichtterminalzeichen v_1 sowohl in v_2 als auch in t überführt werden kann.
$<$	Vererbungsrelation: $C < D$ bedeutet, dass die Klasse `C` von der Klasse D erbt.
$>_{pr}$	Prioritätsrelation zwischen Rollen
$\mid$	Vereinigung bei einer Produktion in Backus-Naur-Fom: $v_1 ::= v_2 \mid t$ bedeutet, dass ein Nichtterminalzeichen v_1 sowohl in v_2 als auch in t überführt werden kann.
$\sim$	Abhängigkeitsrelation: $a_R \sim b_S$ – das Attribut `a` in der Rolle `R` ist gleichheitsabhängig zum Attribut `b` in `S`; $m_R \sim n_S$ – die Methode `m` in `R` ist gleichheitsabhängig zur Methode `n` in `S`; $R \sim_T S$ – die Rollen `R` und `S` sind vererbungsabhängig über `T`; $R2 \sim_{C1,R1,S1} S2$ die Rollen `R2` und `S2` sind klassenabhängig über die Rollen `R1`, `S1` und die Klasse `C1`.

$\xrightarrow[path]{}$	Pfad: $x \xrightarrow[a_{ij},a_{kl},a_{mn}]{} v$ bedeutet, dass der Pfad von von `x` zu v durch die Speicherzellen a_{ij}, a_{kl}, a_{mn} verläuft.
$\xrightarrow{a_i}$	Zustandsübergang: $s \xrightarrow{a_i} s'$ bedeutet, dass der Zustand s in den Zustand s' durch die Anweisung a_i übergeht.
`*`	Kennzeichnung von Zeigern
`->`	Qualifikation von Methoden und Attributen bei einem Zeiger in C++
`.`	Qualifikation von Methoden und Attributen
`:`	Qualifikation von Attributen und Methoden von Rollen bei RCC
`<-`	Compound Reference
`&`	Kennzeichnung einer Referenz in C++
a_1, a_2, a_3, ...	Aktionen
a_{ij}	Speicheradresse in der i-ten Schicht
a_R	Attribut `a` der Rolle `R`
D	dynamischer Typ: D_x ist der dynamische Typ der Variablen `x`.
f, g	Zuweisungen
f^i	i-fache Komposition der Funktion f
ker	Kern einer Funktion: $ker(f) = \{(x, y) \mid f(x) = f(y)\}$
m_R	Methode `m` der Rolle `R`
S	statischer Typ: S_x ist der statische Typ der Variablen `x`.
s	Zustand
$s = (s_1, s_2, s_3, s_4)$	Zustand im 3-Schichtenmodell
v_1, v_2, v_3, ...	Werte
`x`, `y`, `z`, ...	Variablennamen

Index