

Werttypen in objektorientierten Programmiersprachen

Dissertation

zur Erlangung des akademischen Grades

Dr. rer. nat.

an der Fakultät für Mathematik, Informatik und Naturwissenschaften
der Universität Hamburg

eingereicht beim Fach-Promotionsausschuss Informatik von

Beate Ritterbach

aus Nievenheim

Hamburg, Dezember 2014

Gutachter:

Prof. Dr.-Ing. Heinz Züllighoven, Universität Hamburg

Prof. Dr. Axel Schmoltzky, HAW Hamburg

Prof. Dr. Matthias Riebisch, Universität Hamburg

Tag der Disputation: 11. Juni 2015

Für Inge und Wilhelm Ritterbach

Es sind zwei komplementäre Bewegungen, die unsere Weltaneignung bestimmen: Reduktion und Konstruktion. Erst lösen wir die Komplexität des großen Ganzen auf, um anschließend einzelne Elemente auf subjektiv sinnvolle Weise wieder zusammenzusetzen.

Christian Ankwitsch: Mach's falsch, und du machst es richtig

Zusammenfassung

Objektorientierung als Unterparadigma der imperativen Programmierung ist inhärent zustandsorientiert: Jedes Objekt verfügt über einen – gekapselten – Zustand. Das prädestiniert objektorientierte Programmiersprachen zur Abbildung von „Gegenständen der realen Welt“ und ihren Änderungen im Laufe der Zeit. Dagegen sind sie schwach bei der Umsetzung von Werten und zustandsloser Verarbeitung.

Diese Arbeit präsentiert ein Sprachmodell, das Zustand und Zustandslosigkeit gleichermaßen unterstützt und am Typsystem verankert: Es unterscheidet zustandsbehaftete Objekttypen und zustandslose Werttypen.

Die Arbeit legt einen an MacLennan angelehnten Wertbegriff zugrunde, der Werte anhand ihres Verhaltens definiert. Sie zeigt, dass die kennzeichnenden Merkmale von Werten – kein Verändern, kein Erzeugen, Seiteneffektfreiheit und referentielle Transparenz – einander ergänzen und zusammen genommen Zustandslosigkeit charakterisieren. Diese implementierungsunabhängige Definition beschreibt Werte als abstraktes Konzept, grenzt sie gegenüber Objekten ab und bildet das Fundament der Arbeit.

Werttypen erfordern eine dedizierte Sprachunterstützung. Die Arbeit schlägt vor, Werttypen und Objekttypen auf programmiersprachlicher Ebene zu trennen und verschiedene Typkonstruktoren für sie vorzusehen. Sie nimmt das objektorientierte Paradigma als Ausgangsbasis und fokussiert auf die Gestaltung eines Typkonstruktors für Werttypen. Dieser kann – in einem prinzipiell zustandsbehafteten Umfeld – Werte klar abbilden, ihre Zustandslosigkeit sicherstellen und ihre Programmierung vereinfachen. Die Forschungssprache VJ, eine Erweiterung von Java, belegt die Machbarkeit des Ansatzes und veranschaulicht die Wirkungsweise eines Typkonstruktors für Werttypen innerhalb einer objektorientierten Sprache.

Ein wesentlicher Nutzen der Wert-/ Objekt-Trennung liegt in ihrer Auswirkung auf andere Sprachkonstrukte: sie führt zu Umgestaltungen, die potentielle Fehlerquellen heutiger objektorientierter Sprachen beseitigen. Beispielsweise ermöglicht sie die Verschmelzung von Wertgleichheit und Objektidentität zu einem einheitlichen Konzept und vermeidet damit viele Schwierigkeiten objektorientierter Sprachen im Umgang mit Gleichheit. Sie erlaubt es auch, Werttypen und Objekttypen mit unterschiedlichen Mechanismen für das Subtyping auszustatten und liefert damit neue Lösungsansätze für etliche bekannte Probleme der Objektorientierung, z. B. das Kreis-Ellipsen-Dilemma oder die Typisierung binärer Methoden.

Der Grundgedanke, Werttypen und Objekttypen voneinander zu trennen, liefert einen Impuls für den Programmiersprachenentwurf. Seine konsequente Umsetzung mündet in ein grundlegend neues Sprachmodell.

Abstract

The object-oriented paradigm, being a sub-paradigm of imperative programming, implicitly assumes state: every object has an – encapsulated – state. This makes object-oriented programming well-suited for modeling „real-world objects“ and their changes in the course of time. However, they are weak when it comes to modeling values and stateless processing.

This thesis presents a language model that equally supports both state and statelessness and links them to the type system: It distinguishes stateful object types and stateless value types.

The thesis is grounded on the work of MacLennan who defined values based on their behavior. It shows that the defining properties of values – no change, no creation, no side effects, referential transparency – complement each other and that jointly they characterize statelessness. The definition is implementation-independent. It discerns values and objects as different kinds of abstractions and it constitutes the foundation of the thesis.

Value types require dedicated language support. The thesis proposes distinguishing value types and object types on the language level by providing a separate type constructor for each. Presupposing the object-oriented paradigm and its related language elements, the thesis focuses on the design of a type constructor for value types. This type constructor models value types clearly; it facilitates easy programming and it guarantees their statelessness in a language setting that, in general, allows for mutable state. The research language VJ proves the feasibility of the approach. VJ adds a value type constructor to Java; it illustrates the look-and-feel of value types within an object-oriented programming language.

The distinction of values and objects brings about the redesign of other language elements, thereby eliminating pitfalls known from object-oriented languages and enhancing safety and expressiveness. For example, it enables merging value equality and object identity in a unified concept, thus preventing many difficulties with equality that object-oriented languages usually bring forth. It also allows for providing value types and object types each with a distinct subtyping mechanism and by that delivering new solutions to some well-known problems of object-oriented programming, such as the circle-ellipse dilemma or the typing of binary methods.

The separation of value types and object types gives an impetus to programming language design. Its consequent realization yields a fundamentally new language model.

Danksagung

Eine Promotion erfordert nicht nur eine Idee und die Bereitschaft, hart zu arbeiten, sie braucht vor allem Menschen, die an sie glauben und sie begleiten. Ich möchte allen danken, die zum guten Ende der vorliegenden Arbeit beigetragen haben.

Danke meinem Betreuer Prof. Dr. Axel Schmolitzky für die Chance, die er mir gab, als ich am 17. September 2003 unvermittelt vor seiner Bürotür auftauchte und erklärte, ich wolle mich mit Werttypen auseinandersetzen. Danke für die intensive Betreuung, für das Engagement und für zahlreiche spannende Diskussionen. Danke meinem Betreuer Prof. Dr. Heinz Züllighoven dafür, dass er dieses Promotionsvorhaben befürwortet, gefördert und mit Rat und Tat begleitet hat. Ohne die Aufgeschlossenheit meiner beiden Betreuer gegenüber diesem akademischen Quereinstieg wäre diese Arbeit nicht möglich gewesen. Danke an Prof. Dr. Matthias Riebisch für viele wertvolle Tipps und für die Bereitschaft, die Arbeit zu begutachten.

Danke allen Mitstreitern am Arbeitsbereich Softwaretechnik der Universität Hamburg, die meine Ideen angehört und leidenschaftlich diskutiert haben, u. a. auf diversen SWT-Tagen. Fred Winkler hat den Werdegang dieser Arbeit von Anfang an mitverfolgt; unsere gemeinsamen Diskussionen über alles und jedes waren ein Quell der Inspiration. Freds Begeisterung für die Informatik reisst jeden mit, der ihn erlebt. Zusammen mit Jörg Rathlev wurden im Zuge des Entwurfs der Forschungssprache VJ die grundlegenden Konzepte der hier vorliegenden Arbeit einem ersten Test auf Tragfähigkeit unterzogen. Mit Petra Becker-Pechau habe ich im Endspurt unserer jeweiligen Promotionen viel Zeit beim gemeinsamen „Dissen“ im Elbgold verbracht. Danke für das hilfreiche Feedback zu meinen ersten Entwürfen, für viele unschätzbare Tipps und für die gemeinsame Zeit.

Den Anstoß zu dieser Arbeit haben praktische Fragestellungen gegeben, die Machart heutiger Programmiersprachen und die Umwege, die sie manchmal verlangen. Über lange Zeit habe ich sowohl in der kommerziellen Softwareentwicklung als auch an der Dissertation gearbeitet und so in zwei Welten gelebt. Danke an meine Kollegen aus unzähligen Projekten in verschiedenen Firmen für viele spannende Debatten und für die Anregungen, die ich daraus in die akademische Welt mitnehmen konnte. Der Kontakt zur Praxis hat mir die notwendige Erdung gegeben.

Die Arbeit an einer Dissertation ist wahrscheinlich immer ein langwieriger Prozess mit vielen Wendungen, Unsicherheiten und unerwarteten Schwierigkeiten. Ich danke meinen Freunden und meiner Familie, dass sie mich in dieser Zeit mit so viel Geduld unterstützt und ermutigt, aufgemuntert und ertragen haben.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ausgangspunkt	1
1.2	Motivation	2
1.3	Andere Arbeiten	4
1.4	Zielsetzung	4
1.5	Forschungsfragen	5
1.6	Einordnung der Arbeit	6
1.7	Methodik	7
1.8	Aufbau der Arbeit	8
I	Werte und Werttypen: Einordnung und Definition	11
2	Programmiersprachen und Programmierparadigmen	13
2.1	Programmierparadigmen	13
2.2	Imperative Programmierung	15
2.2.1	Prozedurale Programmierung	16
2.2.2	Objektorientierte Programmierung	17
2.3	Funktionale Programmierung	22
2.4	Gegenüberstellung imperativ versus funktional	27
2.5	Objekt-funktionale Sprachen	29
2.6	Vorgehen in dieser Arbeit	32
2.7	Zusammenfassung	33
3	Werte und Werttypen: Begriffsbestimmung	35
3.1	Der Begriff „Wert“ in der Informatik	35
3.2	Programmiersprachliche Grundbegriffe	37
3.3	Werttypen: Definition und Abgrenzung	40
3.3.1	Kein Verändern, kein Erzeugen	40
3.3.2	Seiteneffektfreiheit und referentielle Transparenz	40
3.3.3	Definition Werttyp	41
3.3.4	Werttypen versus Objekttypen	42

3.3.5	Der Wertbegriff von MacLennan	43
3.3.6	Wertbegriffe anderer Arbeiten	44
3.3.7	Zustandslosigkeit und Unveränderlichkeit	44
3.3.8	Elemente und Variablen	46
3.3.9	Zusammenhänge und Abhängigkeiten	46
3.3.10	Werte und Gleichheit	47
3.3.11	Zustandslosigkeit als Eigenschaft von Typen	48
3.4	Beispiele für Werttypen	49
3.4.1	Primitive Datentypen	49
3.4.2	Zeichenketten	50
3.4.3	Wochentag	50
3.4.4	Scheinbare Grenzfälle: Sammlungen	50
3.5	Asymmetrische Abhängigkeit zwischen Werttypen und Objekttypen .	51
3.6	Zusammenfassung	53

II Sprachunterstützung für Werttypen 55

4 Softwaretechnische Verfahren zur Abbildung von Werttypen 57

4.1	Untersuchungskriterien	58
4.2	Wertartige Klassen	59
4.2.1	Value Object	59
4.2.2	Fliegengewicht	63
4.2.3	Typesafe Enum Pattern	66
4.3	Klientenseitige Programmierkonventionen	69
4.3.1	Defensives Kopieren	69
4.3.2	Copy-on-write	71
4.3.3	Body-Handle-Idiom	73
4.3.4	Mutable Companion	76
4.4	Vergleich der Verfahren	80
4.5	Zusammenfassung	82

5 Unterstützung für Werte in bestehenden Programmiersprachen 83

5.1	Untersuchte Sprachen	83
5.2	Untersuchungskriterien	84
5.3	Sprachseitig vorgegebene Werttypen	86
5.3.1	Basisdatentypen	86
5.3.2	Aufzählungstypen	88
5.3.3	Ergebnis	90
5.4	Partielle Unterstützung von Werteigenschaften	90
5.4.1	Unveränderlichkeit	92

5.4.2	Keine Erzeugung	93
5.4.3	Gleichheit	94
5.4.4	Seiteneffektfreiheit und referentielle Transparenz	96
5.4.5	Ergebnis	97
5.5	Abgrenzung: Wert-ähnliche Sprachkonstrukte	99
5.5.1	Wertsemantik für programmierer-definierte Typen	99
5.5.2	Wert-Parameter (Call-by-value)	104
5.5.3	Ergebnis	107
5.6	Zusammenfassung	107
6	Ein Typkonstruktor für Werttypen	109
6.1	Anforderungen	110
6.2	Sprachliche Grundelemente und Syntax	110
6.3	Entwurf einer Sprachunterstützung für Werttypen	112
6.3.1	Dedizierter Typkonstruktor	112
6.3.2	Datenfelder	113
6.3.3	Selektoren	114
6.3.4	Kein Zugriff auf Objekte	121
6.3.5	Gleichheit für Wertklassen	122
6.3.6	Aufzählungs-Wertklassen	125
6.3.7	Benutzung von Wertklassen	129
6.3.8	Überblick: Sprachunterstützung für Werttypen	131
6.4	Abgleich mit den Anforderungen	132
6.5	Zusammenfassung	133
7	Eine exemplarische Implementation: Die Forschungssprache VJ	135
7.1	Ziele von VJ	135
7.1.1	Nachweis der Machbarkeit	135
7.1.2	Einfache Programmierung	136
7.1.3	Kompatibilität zur Java	136
7.2	Sprachentwurf	136
7.2.1	Sprachkonstrukte für Werttypen	137
7.2.2	Sprachregeln	141
7.3	Der VJ-Compiler	143
7.3.1	Ansatz und Werkzeuge	143
7.3.2	Aufbau des VJ-Compilers	144
7.3.3	Prüfungen für Wertklassen	145
7.3.4	Umwandlung in Java-Code	146
7.3.5	Umsetzung und Test	148
7.4	Bewertung und Folgerungen	149

7.4.1	Zielerreichung	149
7.4.2	Einfluss von VJ auf den Sprachentwurf dieser Arbeit	150
7.5	Zusammenfassung	153

III Auswirkungen der Wert-/ Objekttrennung 155

8 Gleichheit/ Identität bei Wert- und Objekttypen 157

8.1	Bestandsaufnahme: Schwierigkeiten im Umgang mit Gleichheit	157
8.1.1	Schwierigkeit 1: Gleichheit ist keine Äquivalenzrelation	158
8.1.2	Schwierigkeit 2: Gleichheit abhängig von änderbarem Zustand	159
8.1.3	Schwierigkeit 3: Hashcode nicht passend zur Gleichheit	159
8.1.4	Schwierigkeit 4: Unterscheidbarkeit gleicher Elemente	160
8.1.5	Schwierigkeit 5: Benutzung eines „falschen“ Vergleichs	160
8.1.6	Schwierigkeit 6: Vergleich typ-inkompatibler Ausdrücke	161
8.1.7	Schwierigkeit 7: Asymmetrie bei null-Referenzen	161
8.1.8	Zusammenfassung der Schwierigkeiten	162
8.2	Ursachenanalyse: Fehlende Trennung von Werten und Objekten . . .	162
8.2.1	Simulation von Werttypen mit Objektklassen	163
8.2.2	Synonyme und Homonyme	164
8.2.3	Vergleiche in Programmiersprachen	166
8.2.4	Konzept und Implementierung	168
8.2.5	Gleichheitsvertrag	169
8.3	Ein neues Sprachmodell für Gleichheit/ Identität	169
8.3.1	Modifizierter Gleichheitsvertrag	169
8.3.2	Gleichheit und Identität als Operationen auf Ausdrücken . . .	173
8.3.3	Verschmelzung von Gleichheit und Identität	175
8.3.4	Sprachunterstützung für Gleichheit/ Identität	178
8.3.5	Vergleiche zu bestehenden Programmiersprachen	182
8.4	Beseitigte Probleme	183
8.5	Zusammenfassung	185

9 Subtyping und Subclassing bei Werttypen 187

9.1	Bestandsaufnahme: Werte und Vererbung	187
9.1.1	Befürwortung von Vererbung bei Werttypen	188
9.1.2	Ablehnung von Vererbung bei Werttypen	189
9.2	Unterscheidung von Subtyping und Subclassing	189
9.2.1	Subtyping	190
9.2.2	Subclassing	191
9.2.3	Subclassing als ein Subtyping-Mechanismus	192
9.2.4	Subclassing bei Wertklassen	193

9.2.5	Subtyp-Beziehungen bei Werttypen	199
9.3	Subtyping-Mechanismus für Wertklassen: Inklusionsfunktion	201
9.3.1	Definition und Wirkungsweise der Inklusionsfunktion	201
9.3.2	Regeln für die Inklusionsfunktion	203
9.3.3	Unterschiede zwischen Inklusionsfunktion und Coercion	206
9.3.4	Zuordnung der Inklusionsfunktion im Code	207
9.3.5	Typäquivalenz zwischen zwei Wertklassen	208
9.3.6	Gegenüberstellung: Inklusionsfunktion versus Subclassing	209
9.3.7	Vergleiche zu ähnlichen Sprachkonstrukten	210
9.4	Anwendung auf bekannte Probleme	213
9.4.1	Kreis-Ellipsen-Dilemma	213
9.4.2	Typisierung binärer Methoden	215
9.5	Zusammenfassung	220
10	Zusammenfassung und Ausblick	223
10.1	Zusammenfassung	223
10.2	Offene Fragen	226
10.3	Ausblick	229
	Anhänge	231
A	Code-Beispiele und Spezifikationen zu Kapitel 6	233
A.1	Grammatik der Sprachunterstützung für Wertklassen	234
A.2	Beispiele für Wert-Aufzählungen und Objekt-Aufzählungen	238
B	Code-Beispiele und Spezifikationen zu Kapitel 7	241
B.1	Beispiel für eine mit VJ programmierte Wertklasse: Date	242
B.2	Beispiel: VJ-Wertklasse/ vom VJ-Compiler erzeugter Java-Code	243
B.3	Beispiel: Klientencode/ vom VJ-Compiler erzeugter Java-Code	246
B.4	Testfälle für den VJ-Compiler	248
C	Code-Beispiele und Spezifikationen zu Kapitel 8	267
C.1	Vertrag für Gleichheit in Java	268
C.2	Beispiel: Implementierung von equals in Vererbungshierarchien	269
C.3	Beispiel: Gleichheit abhängig von änderbarem Zustand	273
C.4	Beispiel: Unterscheidbarkeit gleicher Objekte	275
D	Code-Beispiele und Spezifikationen zu Kapitel 9	277
D.1	Beispiel Inklusionsfunktion: Character/ String	278
D.2	Herleitung der Verhaltenskonformität	279

D.3 Beispiel Inklusionsfunktion: Workday/ Weekday	280
D.4 Beispiel Typäquivalenz: Date und Fecha	283
D.5 Beispiel Fortress: extends- und coerce-Beziehungen	287
D.6 Beispiel Werte versus Objekte: Rechteck/ Quadrat	288
Literaturverzeichnis	291
Eigene Veröffentlichungen	307

Abbildungsverzeichnis

1.1	Ziele der Arbeit	5
2.1	Funktionale versus imperative Programmierung	14
2.2	Programmierparadigmen	27
3.1	Eigenschaften von Werttypen und Objekttypen	42
3.2	Kategorien von Operationen	45
3.3	Abhängigkeiten zwischen den Werteigenschaften	47
3.4	Asymmetrische Abhängigkeit zwischen Werttypen und Objekttypen .	52
8.1	Gleichheit und Identität: Synonyme und Homonyme	164
9.1	Sprachmechanismen für Subtyping	209

Tabellenverzeichnis

2.1	Imperative und funktionale Sprachen: fundamentale Merkmale	28
4.1	Body-Handle-Idiom: Verarbeitung abhängig von der Art der Operation	74
4.2	Softwaretechnische Verfahren zur Abbildung von Werten	81
5.1	Sprachseitig unterstützte Werttypen in objektorientierten Sprachen .	91
5.2	Sprachkonstrukte mit partieller Unterstützung von Werteigenschaften	98
6.1	Schlüsselworte für die Definition einer Wertklasse	131
6.2	Sprach-generierte Funktionalität für Wertklassen	131
6.3	Sprachregeln für Wertklassen	132
8.1	Vergleichsoperatoren und -methoden in Programmiersprachen	160
8.2	Schwierigkeiten im Umgang mit Gleichheit	162
8.3	Verhalten von == und equals in Java	167
8.4	Extensionale und intensionale Gleichheit nach Grogono und Chalin .	177
8.5	Sprachgenerierte versus programmierer-definierte Gleichheit/ Identität	180
8.6	Gegenüberstellung der Sprachmodelle	183
8.7	Beseitigte Schwierigkeiten	184
9.1	Subklassenbeziehungen zwischen Wertklassen und Objektklassen . . .	201
9.2	Subtypbeziehungen zwischen Werttypen und Objekttypen	201
9.3	Wirkungsweise von Subclassing und Inklusionsfunktion	210
9.4	Beurteilungskriterien für Lösungen zu binären Methoden	217

Kapitel 1

Einleitung

Kernthese

Werte und Objekte sind fundamental verschieden. Die programmiersprachliche Trennung von Werten und Objekten macht die Programmierung klarer, sicherer und verhindert eine Reihe potentieller Fehler.

1.1 Ausgangspunkt

Programmiersprachen sind das zentrale Arbeitsmittel der Softwareentwicklung. Ludewig und Lichter bezeichnen eine Programmiersprache als den „Werkstoff, aus dem das Programm gebaut wird“ [LL13, p. 456]. Eine Sprache legt, gleich einem Baukasten, das Material fest, aus dem der Softwareentwickler das Produkt – den Programmcode – zusammensetzt. Die Bausteine, d. h. die von der Sprache unterstützten Konstrukte, bestimmen, welche Gegenstände oder Abläufe wie leicht abgebildet werden können. Sie formen auf diese Weise die Weltsicht des Softwareentwicklers.

Code wird nicht nur geschrieben. Er wird in noch größerem Ausmaß gelesen¹ – bei der Einarbeitung, der Fehlersuche, der Erweiterung, und oft auch, um eine zuverlässige Aussage über die aktuell in Kraft befindlichen fachlichen Regeln zu erhalten. Die Bausteine der Sprache beeinflussen nicht nur den Aufwand für die Programmierung, sondern auch, wie gut der Code lesbar ist.

Viele Sprachen streben danach, ihre Bausteine an einer zentralen Abstraktion auszurichten. LISP baut beispielsweise auf der grundlegenden Struktur der Liste

¹Z. B. konstatiert Hoare „the readability of programs is immeasurably more important than their writeability“ [Hoa73]. Dennoch wird in der Informatik Lesbarkeit oft vernachlässigt und der Fokus auf das Schreiben von Software gerichtet: „The software business is one of the few places we teach people to write before we teach them to read. That’s really a mistake.“ Tom Love in [BW09, p. 250].

auf, funktionale Sprachen stellen Funktionen (im mathematischen Sinne) in den Mittelpunkt ihres Sprachmodells.

In der heutigen Praxis sind objektorientierte Sprachen am weitesten verbreitet; ihre zentrale Abstraktion ist das *Objekt*. Hinter dieser Entwurfsentscheidung steht die Auffassung, dass Objekte als universell einsetzbare Sprachbausteine jedes beliebige Konzept abbilden können. Diese Arbeit stellt die universelle Tragfähigkeit von Objekten in Frage. Sie richtet den Fokus auf *Werte* und stellt damit den Objekten eine zweite, fundamental verschiedene zentrale Abstraktion an die Seite.

1.2 Motivation

Die Beschäftigung mit Werten in einer vom Leitbild des Objektes geprägten Welt ist durch mehrere Faktoren motiviert. Z. B. erscheinen Abstraktionen wie Zahlen bereits „intuitiv anders“ als typische Objekte. Auch die Art und Weise, wie viele Programmiersprachen mit einigen Typen umgehen, deutet darauf hin, dass es neben Objekten noch andere Abstraktionen gibt. Nicht zuletzt führt die unzureichende Unterstützung von Werten in einem objektorientierten Umfeld zu einer Reihe von Problemen. Das wird im Folgenden näher ausgeführt.

„Zahlen“ versus „reale Welt“

Programmierung war in ihren Anfängen automatisiertes Rechnen. Z. B. diente Fortran – eine der ersten höheren Programmiersprachen – in erster Linie dazu, Berechnungen abzubilden. Dementsprechend waren die von der Sprache unterstützten Typen im Wesentlichen Zahlentypen: natürliche und ganze Zahlen, Festkomma- und Gleitkommazahlen.

Mit dem Aufkommen der Objektorientierung wurde es möglich, beliebige Typen zu programmieren und mit ihnen beliebige „Gegenstände der realen Welt“ 1:1 im Programm abzubilden: Kunden, Transportcontainer oder Immobilien, Bankkonten, Versicherungsverträge oder Urlaubsreisen. Das erhöhte das Abstraktionsniveau der Sprache und verringerte die semantische Lücke zwischen dem Softwaresystem und dem abgebildeten Gegenstandsbereich. Der plakative Leitsatz des objektorientierten Paradigmas „Alles ist ein Objekt“ (zurückgehend auf Kay [Kay93]) drückt die Überzeugung aus, dass Objekte *jede* beliebige Abstraktion abbilden können.

Intuitiv betrachtet passen Zahlen aber nicht in das objektorientierte Schema. Z. B. mutet es seltsam an, die Addition zweier Zahlen „ $a + b$ “ zu interpretieren als „Sende an a die Botschaft 'addiere' mit dem Parameter b “.² Auch Abstraktionen

²Die Interpretation der Addition als einer Botschaft an den ersten Summanden wurde insbesondere im Smalltalk-Umfeld häufig herangezogen, um die Prinzipien der Objektorientierung zu erklären, s. z. B. [GR89].

wie Datum oder Zeichenkette, Temperatur oder Gewicht, Geldbetrag oder Wertpapierrating haben mehr Gemeinsamkeiten mit Zahlen als z. B. mit Personen oder Versicherungsverträgen - es sind *Werte*.

Die Grundüberzeugung, dass „alles ein Objekt“ sei, presst Werte in ein Schema, das ihnen nicht gerecht wird. Objektorientierung hat Werte aus dem Blickfeld verloren und sie kurzerhand zu speziellen Objekten erklärt.

Beobachtungen in Programmiersprachen

Viele objektorientierte Sprachen unterstützen nicht nur Objekte, sondern auch Werte, und zwar in Form von vordefinierten, primitiven Datentypen: Typen für Zahlen, Zeichen oder bool'sche Wahrheitswerte. Diese unterscheiden sich in vielfältiger Hinsicht von Objekttypen. Primitive Datentypen stehen oft außerhalb der Typhierarchie, sie weisen mit Operatoren und Literalen oft eine spezielle Syntax auf, und sie unterliegen etlichen Sonderregeln, die für Objekttypen nicht gelten, z. B. hinsichtlich der möglichen Konvertierungen. Selbst in einer als „rein objektorientiert“ geltenden Sprache wie Smalltalk existieren für Zahlen, Zeichen oder Symbole spezielle Regeln. Ihre Andersartigkeit beruht nicht allein auf ihrer – häufig als Rechtfertigung für die Existenz vordefinierter Typen angeführten – technischen Effizienz. Sie ist auch konzeptionell begründet und kann als Indiz dafür betrachtet werden, dass Werte und Werttypen einen unverzichtbaren Bestandteil der Programmierung darstellen.

Durch das Fehlen von Werten verursachte Probleme

Die Notwendigkeit von Werten erweist sich besonders deutlich anhand der Probleme, die ihr Fehlen hervorruft:

Die Implementierung von Werten durch ein verbreitetes Schema, nämlich durch eine (Objekt-)Klasse mit Datenfeldern und zugehörigen get- und set-Methoden macht ihre Benutzung schwierig und kann für Klienten zu unerwünschten Seiteneffekten führen. Ein Beispiel dafür ist die Klasse `java.util.Date` für einen Datums-Typ.³

Ein anderes Beispiel sind physikalische Größen wie Abmessungen, Gewichte oder Temperaturen. Aus fachlicher Perspektive stellen sie jeweils eigenständige Typen dar: 3 Meter sind etwas anderes als 3 Zoll, 3 Gramm oder 3 Grad Celsius. Es gibt eine Neigung, derartige Größen durch (vordefinierte) Zahlentypen wie `float` oder `double` abzubilden (s. dazu Allen et al. [All+04]). Das verringert die dokumentarische Aussagekraft des Quellcodes und macht ihn unsicher. Der Absturz der Sonde *Mars Climate Orbiter* im September 1999 ist auf fehlende Einheiten und eine daraus folgende Verwechslung von amerikanischen und metrischen Größen zurückzuführen: Einige für die Flugbahnberechnung notwendige Daten lagen im amerikanischen Sys-

³Etliche Autoren halten die Änderbarkeit von `Date` rückblickend für einen Entwurfsfehler, z. B. Brian Goetz [Goe+06, p. 53]. Die (als Verbesserung gedachte) Klasse `java.util.Calendar` weist die gleichen Probleme auf wie `Date`.

tem vor, wurden aber als Größen im metrischen System interpretiert und weiterverarbeitet [IS99].

Die häufig zu beobachtende fälschliche Benutzung des Operators „==“ anstelle der Methode `equals` bei String-Ausdrücken in Java ist ebenfalls ein Symptom für die lückenhafte Unterstützung von Werten.

In vielen Fällen ist erst auf den zweiten Blick erkennbar, dass komplizierte, unklare oder fehleranfällige Programme mit einer mangelnden Unterstützung von Werten zusammenhängen. Das in der Informatik eingehend diskutierte „Kreis-Ellipsen-Dilemma“ (s. dazu Majorinc [Maj98], Martin [Mar96] oder Henney [Hen01]) beruht auf der impliziten Annahme, dass „alles ein Objekt“ sei und dem daraus resultierenden Versuch, die bei Objekten hinlänglich funktionierende Vererbung auf Werte zu übertragen. Das gilt auch für die von binären Methoden aufgeworfenen Probleme (Bruce et al. [Bru+95] geben eine Übersicht über diese Problematik).

1.3 Andere Arbeiten

Es gibt eine Reihe von Publikationen, die auf die Verschiedenheit von Werten und Objekten eingehen. Beispielsweise beschreibt die grundlegende Arbeit von MacLennan die verschiedenen Charakteristika von Werten und Objekten und betont, dass beide Abstraktionen für die Programmierung gebraucht werden [Mac82]. Eckert und Kempe diskutieren die Notwendigkeit von Werten und ihre Unterschiede zu Objekten in Form eines sokratischen Dialoges [EK95].

Etliche softwaretechnische Ansätze, Vorgehensmodelle und Rahmenwerke verweisen auf die Erfordernis von Werten in der Modellierung, z. B. der Werkzeug- und Materialansatz von Züllighoven et al. mit den darin enthaltenen *Fachwerten* [Zül98; Zül05], oder die *value objects* in den Ansätzen von Evans [Eva04] oder von Fowler [Fow09]. Auch in der Praxis sind Werte von Bedeutung, z. B. spielen sie in der Unternehmensarchitektur einer norddeutschen Krankenversicherung eine wesentliche Rolle [Deg11].

1.4 Zielsetzung

Ein Ziel dieser Arbeit ist, mit einer belastbaren *Definition* der Begriffe „Wert“ und „Werttyp“ eine Grundlage zu schaffen und zu zeigen, dass es sich bei Werten und Objekten um zwei verschiedene, zueinander komplementäre Konzepte handelt.

Die Arbeit wird nachweisen, dass die Unterscheidung von Werten und Objekten auf der Ebene der Programmiersprache die Programmierung klarer und sicherer macht. Zu diesem Zweck wird ein *Typkonstruktor* entworfen, der Werttypen ähn-

lich flexibel und sicher unterstützt, wie heutige objektorientierte Sprachen es für Objekttypen tun.

Die Arbeit wird zeigen, dass die resultierende Zweiteilung des Typsystems weitreichende *Auswirkungen* auf andere Bestandteile der Sprache nach sich zieht und viele in objektorientierten Sprachen auftretende Schwierigkeiten verringert oder sogar vermeidet.

Die genannten Ziele bauen aufeinander auf wie die Stockwerke eines Hauses (s. Abbildung 1.1). Die Definition von Werttypen bildet das Fundament für den Entwurf einer Sprachunterstützung. Die Sprachunterstützung für Werte ist Grundlage für die Untersuchung ihrer Auswirkungen auf andere Sprachkonstrukte.

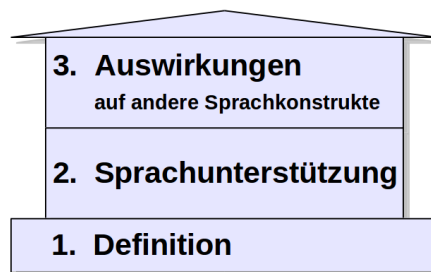


Abbildung 1.1: Ziele der Arbeit

Es ist nicht das Ziel dieser Arbeit, eine vollständige Programmiersprache zu entwerfen. Im Zentrum der Arbeit steht das Konzept von Werten und infolgedessen *ein* einzelnes Sprachkonstrukt. Weitere Sprachkonstrukte werden insoweit betrachtet, als sie durch das Vorhandensein von Werten beeinflusst werden. Eine komplette, neue Programmiersprache dagegen würde etliche Details enthalten, die zu Werten keine Berührungspunkte aufweisen, die aber gleichwohl im Sprachentwurf berücksichtigt werden müssten. Die Auseinandersetzung mit diesen Details würde nicht zur Zielsetzung beitragen.

1.5 Forschungsfragen

Die vorliegende Arbeit wird folgende Forschungsfragen beantworten:

1. Was sind Werte?

- Wie wird der Begriff „Wert“ in der Informatik verwendet? Ist diese Verwendung einheitlich oder gibt es Unterschiede? Ist ein gemeinsamer Nenner erkennbar?
- Welche Eigenschaften können zur Charakterisierung von Werten herangezogen werden?

- Was unterscheidet Werte von Objekten? Besteht eine klare Trennung, oder gibt es einen fließenden Übergang?
2. **Wie können Werte auf programmiersprachlicher Ebene unterstützt werden?**
- Welche Probleme wirft die Abbildung von Werten in objektorientierten Programmiersprachen auf?
 - Welchen Anforderungen muss eine programmiersprachliche Unterstützung für Werte genügen?
 - Wie kann eine Sprachunterstützung für Werte in einer objektorientierten Sprache im einzelnen aussehen?
3. **Welche Auswirkungen hat die Unterscheidung von Werten und Objekten auf die Programmiersprache?**
- Können andere Sprachkonstrukte mit Werten und Objekten orthogonal kombiniert werden?
 - Gibt es Sprachkonstrukte, die nur für Werte oder nur für Objekte sinnvoll einsetzbar sind?
 - An welchen Stellen führt die Unterscheidung von Werten und Objekten zu Anpassungen im Sprachentwurf? Wie sehen sie im einzelnen aus?

1.6 Einordnung der Arbeit

Die Arbeit bewegt sich im Umfeld der *Softwaretechnik*, also dem Teilgebiet der Informatik, das sich mit Prinzipien, Methoden und Werkzeugen für die systematische, wirtschaftliche, arbeitsteilige Entwicklung von komplexen Softwaresystemen befasst (vergl. dazu Ludewig und Lichter [LL13, p. 46], Balzert [Bal01, p. 35] und Pomberger [PB96, p. 3]). Die Unterscheidung von Werten und Objekten ist aus einem softwaretechnischen Blickwinkel motiviert: Mit der sprachlichen Trennung von Werten und Objekten kann ein Softwaresystem die Sachverhalte und Abläufe eines Gegenstandsbereiches direkter und besser lesbar abbilden als mit Objekten allein.

Ein Teil der Arbeit befasst sich mit der Gestaltung einer Sprachunterstützung für Werte und der Erforschung ihrer Auswirkungen auf andere Sprachbausteine. Sie bewegt sich damit in den Bereich des *Programmiersprachenentwurfs*. Auch die den Sprachentwurf behandelnden Teile der Arbeit orientieren sich an softwaretechnischen Leitlinien. Sie betrachten die Sprachkonstrukte aus der Perspektive des Softwareentwicklers und stellen Qualitätskriterien wie eine einfache Benutzbarkeit der Sprache sowie Lesbarkeit und Wartbarkeit des Quellcodes in den Vordergrund. Effizienzbetrachtungen (wie Laufzeitverhalten oder Speicherverbrauch) oder Details der technischen Umsetzung spielen in dieser Arbeit nur eine untergeordnete Rolle.

1.7 Methodik

Wissenschaftliche Arbeiten im Bereich der Informatik können verschiedenen methodischen Kategorien zugeordnet werden. Idealtypisch werden drei Arten von Arbeiten unterschieden: konzeptionelle, empirische und konstruktive.⁴

Bei konzeptionellen Arbeiten beruht der Erkenntnisgewinn auf „der Diskussion und neuartigen Verknüpfung bereits bekannter Daten“ [MG]. Aus diesen Gründen wird dieser Typus auch als „diskursiv“, als „theoretisch“ oder als „Literaturarbeit“ bezeichnet. Empirische Arbeiten nähern sich der Fragestellung experimentell oder durch eigene Beobachtungen. Bei konstruktiven Arbeiten wiederum trägt die Gestaltung eines Artefaktes, z. B. einer Software oder eines Werkstücks, den wesentlichen Anteil zur Beantwortung der Fragestellung bei.

Die vorliegende Arbeit ist anhand ihrer Zielsetzung und ihres Fokus schwerpunktmäßig als *konzeptionelle Arbeit* einzuordnen: Sie sichtet, analysiert und vergleicht bestehende Konzepte, Sprachkonstrukte und Programmierverfahren. Sie ordnet vorhandene Bestandteile neu und fügt sie auf eine Weise zusammen, die erkannte Schwächen behebt und neue Sicht- und Herangehensweisen mit sich bringt. In geringerem Umfang enthält die Arbeit, wo möglich und angebracht, auch konstruktive Elemente.

Die Methodik der Arbeit orientiert sich an ihrem vorwiegend konzeptionellen Charakter. Sie bedient sich einer argumentativen Vorgehensweise, einer iterierten Abfolge der drei Schritte „Begriffsbildung - Aussage - Begründung“. (Damit lehnt sie sich strukturell an das in der Mathematik gebräuchliche Schema „Definition - Satz - Beweis“ an.)

- Die *Begriffsbildung* legt fest, was unter dem jeweils im Fokus der Betrachtung stehenden programmiersprachlichen Konzept verstanden werden *soll*, indem sie es anhand seiner charakteristischen Merkmale definiert. Ausgangspunkt dafür sind bestehende Konzepte vorangehender Arbeiten – sie werden systematisiert und, soweit erforderlich, konkretisiert und präzisiert – sowie die in den vorhergehenden Schritten entwickelten Konzepte. Auf Basis des so gewonnenen Begriffsgefüges werden neue Sprachkonstrukte definiert, die die zuvor definierten Konzepte unterstützen und dem Anwendungsprogrammierer ihre direkte Benutzung ermöglichen.
- *Aussagen* schreiben diesen Konzepten und Sprachkonstrukten Eigenschaften zu – Eigenschaften, die zuvor als Zielvorgabe festgelegt wurden (z. B. Eindeu-

⁴Die Bezeichnungen, teils auch die Einteilung in verschiedene Typen variieren von Autor zu Autor erheblich, vergl. z. B. Friedrich [Fri88], Voss [Vos14] oder Karmasin und Ribing [KR12]. Es lassen sich jedoch deutlich die drei genannten Grundtypen erkennen.

tigkeit oder softwaretechnische Qualitäten) und die Schwächen des zugrunde gelegten objektorientierten Sprachmodells beheben.

- Die *Begründung* der Aussagen geschieht, abhängig von der Art der Aussage, deduktiv oder argumentativ. Konzeptionelle Eigenschaften werden logisch-deduktiv aus den definierenden Merkmalen der Konzepte oder den Bestandteilen der Sprachkonstrukte hergeleitet, was einer mathematischen Beweisführung entspricht. Andere, insbesondere softwaretechnische Eigenschaften werden argumentativ hergeleitet. Beispielsweise kann die erhöhte Sicherheit eines vorgeschlagenen Sprachbausteins aus seiner per Konstruktion eingeschränkten Verwendungsmöglichkeit begründet werden und aus der damit einhergehenden Ausschaltung einer spezifischen Art von fehlerhaften Programmen.

Zur Veranschaulichung werden alle Aussagen mit prototypischen Code-Beispielen unterlegt, welche die behaupteten Eigenschaften und die Funktionsweise der Sprachkonstrukte illustrieren. Zusätzlich vermitteln die Code-Beispiele einen Eindruck von den potentiellen Einsatzmöglichkeiten des jeweiligen Sprachkonstruktes in einem praktischen Nutzungskontext. Das zentrale Konzept der Arbeit (Werttypen, integriert in ein objektorientiertes Umfeld) wird zusätzlich konstruktiv durch eine Machbarkeitsstudie evaluiert.

Ein empirischer Nachweis der softwaretechnischen Qualitätskriterien oder der praktischen Nützlichkeit ist aufgrund des konzeptionellen Charakters der Arbeit nicht möglich, grundsätzlich aber wünschenswert und als Gegenstand von empirisch ausgerichteten nachfolgenden Arbeiten vorstellbar.

1.8 Aufbau der Arbeit

Die weitere Arbeit ist entsprechend der oben beschriebenen Zielsetzung in drei Teile gegliedert:

Teil I legt mit der Begriffsbildung das Fundament der Arbeit:

Kapitel 2 zeigt, dass die Ausrichtung auf Werte dem funktionalen, die Ausrichtung auf Objekte dem imperativen Paradigma entspricht und gibt einen Abriss über die wesentlichen Merkmale dieser beiden Sprachparadigmen.

Kapitel 3 legt ausgehend von den in Kapitel 2 herausgearbeiteten Charakteristika fest, was im Kontext dieser Arbeit unter Werten und Werttypen verstanden werden soll und worin sie sich von Objekten und Objekttypen unterscheiden.

Teil II befasst sich mit der programmiersprachlichen Unterstützung von Werten:

Kapitel 4 weist nach, dass Werttypen mit Hilfe von Mustern oder Programmierkonventionen nicht hinreichend klar und sicher abgebildet werden können.

Kapitel 5 zeigt, dass bestehende objektorientierte Programmiersprachen Werttypen nur rudimentär unterstützen.

Kapitel 6 schlägt vor, dass Werttypen und Objekttypen auf der Ebene der Programmiersprache durch getrennte Typkonstruktoren unterstützt werden und beschreibt eine mögliche Ausgestaltung für einen Werttypkonstruktor.

Kapitel 7 stellt die im Rahmen dieser Arbeit entwickelte Forschungssprache VJ vor. Sie erweitert Java um einen eigenen Typkonstruktor für Werttypen und evaluiert und illustriert die praktische Umsetzbarkeit der in Kapitel 6 vorgeschlagenen Werttyp-Unterstützung.

Teil III untersucht Auswirkungen der Wert-/ Objekttrennung:

Kapitel 8 präsentiert, aufbauend auf der Trennung von Werttypen und Objekttypen, ein alternatives Sprachmodell für die Konzepte Gleichheit und Identität, das viele in bestehenden Sprachen auftretende Fehler im Umgang mit Gleichheit vermeidet.

Kapitel 9 zeigt, dass Vererbung nicht mit Werttypen vereinbar ist. Bei Werttypen kann die Anwendung eines anderen Sprachmechanismus, einer Inklusionsfunktion, einen ähnlichen Effekt erzielen wie Vererbung bei Objekttypen und gleichzeitig die Sprache sicherer machen.

Kapitel 10 fasst die Arbeit zusammen und gibt einen Ausblick auf weitere Fragestellungen, deren Untersuchung in Zusammenhang mit der Wert-/ Objekt-Trennung aussichtsreich erscheint.

Der Anhang der Arbeit enthält Code-Beispiele, Herleitungen und Spezifikationen.

Teil I

Werte und Werttypen: Einordnung und Definition

Kapitel 2

Programmiersprachen und Programmierparadigmen

Objekte oder Werte in den Mittelpunkt des Sprachmodells zu stellen, entspricht zwei eigenständigen Programmierparadigmen: Die Ausrichtung auf Objekte entspricht dem objektorientierten Paradigma, die Ausrichtung auf Werte dem funktionalen Paradigma.

Dieses Kapitel arbeitet die für die beiden Paradigmen wesentlichen Sprachkonstrukte heraus, stellt sie einander gegenüber und führt Arbeiten auf, welche auf die Vereinigung der beiden Paradigmen abzielen. Es beschreibt das Vorgehen der hier vorliegenden Arbeit und die Gründe für den gewählten Ansatz.

2.1 Programmierparadigmen

Die Bezeichnung von fundamental verschiedenen Sicht- und Herangehensweisen als „Paradigmen“ geht auf Kuhn und auf Floyd zurück. Kuhn versteht unter einem Paradigma ein Leitbild, an dem sich ein Zweig der Wissenschaft ausrichtet, „... die Menge der Theorien, Standards und Methoden, die zusammen genommen bestimmen, auf welche Weise Wissen repräsentiert wird und damit den Blick auf die Welt festlegt“ [Kuh70]. Floyd übertrug den Begriff des Paradigmas auf den Bereich der Programmiersprachen. Hier bezeichnet es „... a way of conceptualizing what it means to perform computation, and how tasks that are to be carried out on a computer should be structured and organized“ [Flo79].

In der Informatik sind im Wesentlichen zwei Programmierparadigmen vorherrschend:

Objektorientierte Programmierung, ein Zweig der imperativen Programmierung, geht davon aus, dass „alles ein Objekt“ ist. Sie bildet Zustand und Veränderungen, Gegenstände der realen Welt und ihre Lebenszyklen ab.

Funktionale Programmierung – auch als wert-orientiert bezeichnet – nimmt eine mathematisch orientierte Perspektive auf die Welt ein. Sie stellt Werte und Funktionen in den Mittelpunkt und ist gekennzeichnet durch die Abwesenheit von änderbarem Zustand.

Weil das objektorientierte Paradigma die Welt der Objekte und das funktionale Paradigma die Welt der Werte repräsentiert (s. Ritterbach und Schmolitzky [RS14]), sind ihre historische Entwicklung und ihre jeweiligen charakteristischen Merkmale (s. Abbildung 2.1) relevant für diese Arbeit. (Andere Paradigmen – z. B. Logikprogrammierung mit Prolog als ihrem prominentesten Vertreter – spielen im Rahmen dieser Arbeit keine Rolle und werden im Folgenden nicht betrachtet.)

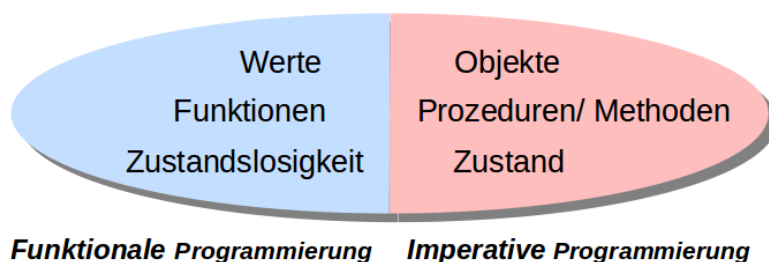


Abbildung 2.1: Funktionale versus imperative Programmierung

Dieses Kapitel untersucht die prägenden Sprachkonstrukte der jeweiligen Paradigmen. Dabei wird unter einem *Sprachkonstrukt* ein „Sprachbaustein“, ein „Bestandteil“ der Sprache verstanden, der für einen Entwickler, der in der jeweiligen Sprache programmiert, sichtbar oder benutzbar ist. Der Begriff „Sprachkonstrukt“ fasst alle Konzepte und Mechanismen zusammen, die eine Sprache aus Sicht des Entwicklers zur Verfügung stellt. Dabei kann es sich um vergleichsweise einfache Bausteine handeln (wie Variablen oder Zuweisungen) oder komplexe Sprachmittel (wie Typisierung, Vererbung oder verschiedene Übergabeverfahren für Parameter). Eine Unterscheidung von Konzepten und Mechanismen wird dann vorgenommen, wenn sie im jeweiligen Kontext relevant ist.

Details der zugrunde liegenden technischen Implementierung, z. B. die Art der Übersetzung des Quellcodes, der Zeitpunkt der Bindung oder die Speicherverwaltung, sind keine Sprachkonstrukte und werden nicht betrachtet. Auch syntaktische Besonderheiten einzelner Sprachen sollen im Folgenden unbeachtet bleiben.

2.2 Imperative Programmierung

Zustand

Das kennzeichnende Merkmal imperativer Sprachen ist *änderbarer Zustand*. Er beruht auf Variablen und Zuweisungen, auf Anweisungen und ihren Seiteneffekten.

Variablen

Zustand wird in imperativen Sprachen durch *Variablen* repräsentiert. Der Zustand eines Programms besteht aus dem aktuellen Inhalt aller seiner Variablen; umgekehrt bildet eine Variable die kleinste ansprechbare Einheit des Zustandes.

Variablen bilden den Dreh- und Angelpunkt der imperativen Programmierung – Dijkstra brachte das auf den Punkt durch die Aussage:

„Once a person has understood the way in which variables are used in programming he has understood the quintessence of programming.“[Dij72]

Anweisungen und Seiteneffekte

Ein Programm in einer imperativen Sprache besteht typischerweise aus einer Abfolge von *Anweisungen* (auch bezeichnet als Statements, Commands oder Befehle, lat. impera = befehlen), die Änderungen des Zustandes bewirken. Zustandsänderungen werden auch als *Seiteneffekte*¹ bezeichnet. Z. B. definiert Louden „... we define a side effect of a statement to be any change in the value of a variable that persists beyond the execution of the statement“ [Lou93, p. 134].

Zuweisungen

Die elementare Anweisung zur Änderung von Zustand ist die *Zuweisung*. Sie verknüpft zur Laufzeit eine Variable mit einem neuen Inhalt. Zuweisungen bilden ein universelles Mittel, um eine Zustandsänderung der realen Welt abzubilden, z. B. den Anstieg eines Aktienkurses oder die Gewichtsabnahme eines Patienten.

Variablen in imperativen Sprachen sind Abstraktionen von Speicherplätzen. Die Zuweisung zu und der Zugriff auf den Inhalt einer Variablen bilden auf höherer Ebene die Gegenstücke zu maschinennahen Befehlen zum Beschreiben und Auslesen von Speicherzellen (*store* und *fetch*). Mit diesem auf den Speicher ausgerichteten

¹Der Begriff „Seiteneffekt“ ist als neutraler Fachterminus zu verstehen. Er drückt aus, dass eine Anweisung eine bleibende, nach ihrer Ausführung fortbestehende Wirkung hat. Beispielsweise besteht die beabsichtigte (und einzige) Wirkung einer Zuweisung in der Änderung der zugehörigen Variablen und damit der Erzielung eines Seiteneffektes. Einige Entwickler verwenden das Wort „Seiteneffekt“ (fälschlich), um ausschließlich *unerwünschte* Zustandsänderungen zu beschreiben – also für eine spezifische Sorte von Programmierfehlern.

Verarbeitungsmodell lehnen sich imperative Sprachen eng an die zugrunde liegende von-Neumann-Architektur des Rechners an:

„...because of the von Neumann architecture, the central features of imperative languages are variables, which model the memory cells; assignment statements, which are based on the piping operation ...“[Seb10, Kap. 1.4.1, p. 18]

Im Unterschied zu einer maschinennahen Sprache muss sich der Programmierer in einer imperativen Hochsprache nicht um technische Details wie die Länge der Worte oder die Codierung des Inhalts kümmern.

Ausdrücke

Neben Anweisungen verfügen imperative Sprachen auch über Ausdrücke. Ein Ausdruck ist ein Stück Code, das ein Ergebnis bezeichnet und somit „ausgewertet“ werden kann, z. B. eine Variable oder der Aufruf einer Funktionsprozedur. Ausdrücke werden u. a. als rechte Seite einer Zuweisung benötigt oder als aktuelle Parameter eines Prozeduraufrufs. Um nicht mit zwei Begriffen, Anweisung *und* Ausdruck, hantieren zu müssen, vereinheitlichen einige Sprachen diese beiden – im Grunde verschiedenen – Konzepte und erklären Anweisungen zu speziellen Ausdrücken („expression languages“, s. Louden [Lou93, p. 138]).

Imperative Steuerung

Mit Zustand als Grundlage des Sprachmodells gehen typisch imperative Steuerungsanweisungen einher. Z. B. werden Wiederholungen in der Regel durch Iteration abgebildet (while- oder for-Schleifen). Sie sind erst durch Zustand möglich: Die Steuerung selbst – die Prüfbedingung oder die Variable zum Hochzählen der Schleifendurchläufe – muss änderbar sein. Auch ist eine mehrfache Ausführung desselben Schleifenrumpfes nur deshalb sinnvoll, weil sie aufgrund fortschreitender Zustandsänderungen bei jedem Durchlauf etwas anderes bewirkt. Auch bedingte Anweisungen (z. B. if-then-else-Konstruktionen) erfordern änderbaren Zustand. Sie entfalten ihre Wirkung dadurch, dass die zugehörigen Zweige unterschiedliche Anweisungen enthalten und so abhängig von der Bedingung verschiedene Zustandsänderungen hervorrufen.

2.2.1 Prozedurale Programmierung

Viele imperative Sprachen sind prozedural. Eine Prozedur – auch bezeichnet als Routine, Subroutine oder Unterprogramm – ermöglicht dem Programmierer, einen zusammengehörenden Block von Anweisungen unter einem Namen zusammenzufassen. Unter diesem Namen ist sie dann aufrufbar, ähnlich einer in der Sprache vordefinierten Anweisung. Durch die Möglichkeit, eigene Prozeduren zu definieren, kann der Programmierer den Befehlsumfang der Sprache nahezu beliebig erweitern.

Prozeduren als Module

Prozeduren bilden ein zentrales, in einigen imperativen Sprachen das einzige Mittel zur Modularisierung (s. Horowitz [Hor83, p. 203]): Eine Prozedur verfügt über eine Schnittstelle (ihren Namen und ggf. ihre Parameter), und sie verbirgt die Details ihrer Implementation vor Aufrufern („information hiding“).

Eine Prozedur kann ein Ergebnis liefern. Sie wird dann als Funktionsprozedur (*functional procedure*) bezeichnet, manchmal auch einfach als Funktion². Eine Prozedur ohne Ergebnis heißt in Abgrenzung dazu auch Routine oder Subroutine. Zustandsänderungen sind in imperativen Sprachen im Prinzip überall möglich, auch in Funktionsprozeduren ([Hor83, p. 205]).

Die Aufstellung auf Seite 18 gibt einen Überblick über einige typische oder historisch bedeutsame prozedurale Sprachen.

2.2.2 Objektorientierte Programmierung

Objekte

Zentrales Leitbild und Namensgeber der objektorientierten Programmierung ist das Objekt: eine Einheit mit einem spezifischen Verhalten. Dieses Verhalten wird mit Hilfe von zum Objekt gehörenden Operationen festgelegt, in der objektorientierten Terminologie als *Methoden* bezeichnet.

Die enge Verknüpfung von Objekten mit ihren zugehörigen Methoden führt zu einem anderen Ausführungsmodell: Während in einer prozeduralen Sprache ein Programm aus einer zentralen Routine (dem „Hauptprogramm“) besteht, die weitere Prozeduren aufruft, wird in einer objektorientierten Sprache ein Programm als eine Menge von Objekten aufgefasst, die gegenseitig ihre Methoden aufrufen, oder – im objektorientierten Jargon – einander „Botschaften senden“.

Kapselung/ Partitionierung des Zustandes

Auch objektorientierte Sprachen sind im Kern imperativ: Jedes Objekt verfügt über einen prinzipiell änderbaren Zustand. Er ist gekapselt, d. h. Klienten können nicht direkt auf die Repräsentation eines Objektes zugreifen, sondern nur mit Hilfe der dafür vorgesehenen Methoden. Information hiding erstreckt sich somit in objektorientierten Sprachen auch auf die Ebene der Daten.

Diese „Partitionierung des Zustandes“, seine „Aufteilung in autonome, gekapselte Einheiten“ – die Objekte – ist nach Wegner das wesentliche Merkmal der objektorientierten Programmierung:

²Diese Bezeichnung birgt Verwechslungsgefahr mit den Funktionen der funktionalen Programmierung, s. Abschnitt 2.3.

Einige prozedurale Sprachen

FORTRAN, entwickelt ab 1954 unter der Führung von John Backus [Bac+57], gilt als erste produktiv eingesetzte höhere Programmiersprache. Sie ist auf mathematische Berechnungen zugeschnitten (FORTRAN = FORMula TRANslation) und war über Jahrzehnte die führende Sprache im technisch-wissenschaftlichen Bereich.

Algol (ALgorithmic LANguage) entstand in den 1950er und 1960er Jahren als Produkt eines Expertenkomitees, dem u. a. Backus, Naur und Perlis angehörten. Primäres Ziel von Algol ist, bekannte und erprobte Sprachkonzepte orthogonal in einem klar beschriebenen Sprachentwurf miteinander zu verbinden. Die Sprachen der Algol-Familie umfassen u. a. Algol58, Algol60 [Bac+63], Algol-W [WH66] und Algol68 [Wij+75].

COBOL wurde ab 1959 von einem Komitee von Forschern und industriellen Anwendern entwickelt [Sam81]. Für den Sprachentwurf zeichnet Grace Hopper verantwortlich. COBOL ist in erster Linie für den Einsatz in kaufmännischen/ administrativen Anwendungen konzipiert (COBOL = COmmon Business Oriented Language) und wird bis heute eingesetzt.

PL/I [Abr78] wurde 1964 von IBM entwickelt. Die Sprache enthält eine große Menge verschiedener Konstrukte. Sie ist für einen universellen Einsatz, sowohl im wissenschaftlichen wie im kommerziellen Bereich, konzipiert worden.

Pascal wurde Ende der 1960er von N. Wirth entworfen [Wir71; JW91]. Die Sprache war ursprünglich für Lehrzwecke gedacht, strebt Einfachheit und gute Lesbarkeit an und beschränkt sich bewusst auf einen reduzierten Sprachumfang. Pascal stellt einen Gegenentwurf zu dem – von Wirth als überfrachtet angesehenen – Algol68 dar.

Die Programmiersprache **C**, 1972 von Dennis Ritchie für die Programmierung des Betriebssystems UNIX entwickelt [KR88], ist bis heute eine der verbreitetsten Sprachen. C unterstützt auch maschinennahe Befehle und wird vorwiegend für die systemnahe Programmierung eingesetzt.

Ada [Taf06] wurde Anfang der 1980er Jahre vom Department of Defense von einem Team unter der Leitung von Jean Ichbiah entwickelt. Ada sollte möglichst viele der zu jener Zeit bekannten Sprachkonstrukte in sich vereinigen. Die Sprache war dazu gedacht, die Vielfalt an verfügbaren Sprachen abzulösen.

„In contrast to the shared-memory model of procedure-oriented programming, object-oriented programming partitions the state into encapsulated chunks each associated with an autonomous, potentially concurrent, virtual machine.“[Weg90, p. 23]

Klassen und Abstrakte Datentypen

Viele objektorientierte Sprachen sind klassenbasiert. Eine Klasse bildet eine Schablone für die Beschreibung mehrerer gleichartiger Objekte. Sie definiert die auf die Objekte anwendbaren Methoden und legt eine Struktur für ihre Repräsentation fest. In der Regel stellt eine Klasse auch einen Konstruktionsmechanismus zum Erzeugen von Objekten zur Verfügung.

Eine Klasse definiert einen neuen *Typ*. Er kann auf die gleiche Weise benutzt werden wie die vordefinierten Typen der Sprache. Nach Meyer ist diese Möglichkeit zur flexiblen Erweiterung des Typsystems das wesentliche Merkmal objektorientierter Sprachen [Mey97, p. 23].

Durch die Kapselung der internen Repräsentation und das Verbergen der Methoden-Implementationen bildet eine Klasse auch ein *Modul*: Sie verfügt über eine definierte Schnittstelle (bestehend aus den Signaturen der öffentlich aufrufbaren Methoden) und über eine Implementation, die ohne Beeinträchtigung von Klienten variiert werden kann.

Damit weisen Klassen Parallelen zu den – unabhängig von der Objektorientierung entstandenen – abstrakten Datentypen [Gut77] auf. Wie abstrakte Datentypen beschreiben auch Klassen einen Typ über seine verfügbaren Operationen und abstrahieren – aus Klientensicht – von der konkreten Implementation. Im Unterschied zu abstrakten Datentypen verfügen Klassen im allgemeinen nicht über eine algebraische Spezifikation, bestehend aus Vorbedingungen, Nachbedingungen und Axiomen (Regeln, die Zusammenhänge zwischen den Operationen angeben)³. Klassen legen ihr Verhalten statt dessen durch die Implementation ihrer Methoden fest. Es ist damit aus der Perspektive abstrakter Datentypen bei konkreten Methoden „überspezifiziert“, bei abstrakten Methoden dagegen „unterspezifiziert“.

Objektorientierung als Unterparadigma

Zu den Eckpfeilern der Objektorientierung zählen, neben der bereits genannten Kapselung und Datenabstraktion, auch Identität, Vererbung und die damit einhergehende Subtyp-Polymorphie.

Objektorientierung ist ein Unterparadigma der imperativen Programmierung (s. Watt [Wat90, p. 189], Moseley und Marks [MM06, p. 12], Sebesta [Seb10, p. 21]).

³Eiffel ist eine der wenigen objektorientierten Programmiersprachen, die ein Vertragsmodell mit Vor- und Nachbedingungen unterstützt. Eine algebraische Spezifikation mit Hilfe von Axiomen gibt es aber auch hier nicht.

Das zeigt sich auch darin, dass die Grundbausteine imperativer Sprachen – Variablen, Zuweisungen, Schleifen, etc. – ebenfalls in objektorientierten Sprachen zu finden sind. Die zusätzlichen Sprachmittel der Objektorientierung – die Einteilung des Quelltextes in Klassen, die Kapselung der internen Repräsentation und die Trennung von Schnittstelle und Implementation – erhöhen das Abstraktionsniveau der Programmiersprache und erleichtern die Modularisierung der Software. Sie tragen dazu bei, die Programmierung von Zustand besser beherrschbar zu machen.

Aktueller Stand und Historie

Objektorientierung kann als „Stand der Technik“ innerhalb der imperativen Programmierung angesehen werden (s. Moseley und Marks [MM06, p. 12]). Sie prägt seit den 1990er Jahren die kommerzielle Softwareentwicklung.

Die Aufstellung auf Seite 21 führt einige Sprachen auf, die einen Meilenstein in der Entwicklung der objektorientierten Programmierung markieren oder die aufgrund ihrer großen Verbreitung de facto einen Industriestandard darstellen.

Soweit nicht anders vermerkt, geht diese Arbeit bei imperativer, d. h. auf Zustand ausgerichteter Programmierung ab hier vom objektorientierten Ansatz aus. Prozedurale Sprachen werden nachfolgend keine weitere Rolle spielen.

Stärken der objektorientierten Programmierung

Programmierung dient dazu, eine reale Welt nachzubilden. Diese beinhaltet in der Regel Gegenstände mit einem Zustand und Prozesse, die Änderungen bewirken. S. z. B. Watt:

„... Programs are written to model real-world processes affecting real-world objects, and such objects often possess a state that varies with time.“[Wat90, p. 189]

Objektorientierte Programmierung eignet sich passgenau für die Simulation einer solchen Welt, weil ihre Sprachkonstrukte diese Elemente widerspiegeln: Objekte bilden auf einfache und natürliche Weise Gegenstände der realen Welt ab. Methoden beschreiben das Verhalten der Objekte, Variablen modellieren ihre Eigenschaften und Zuweisungen deren Änderungen im Laufe der Zeit. Objektorientierung kann darum viele unterschiedliche Gegenstandsbereiche geradlinig auf ein programmier-technisches Modell abbilden – egal, ob es um Kunden, Containerschiffe oder Hypothekenkredite geht. Es gibt keine semantische Lücke („impedance mismatch“) zwischen Wirklichkeit und Programm. Das erhöht das Abstraktionsniveau der Sprache und die Lesbarkeit der darin geschriebenen Quelltexte.

Einige objektorientierte Sprachen

Simula und ihr Nachfolger Simula67 wurden in den 1960er Jahren von Dahl und Nygaard für die Simulation physikalischer Systeme entwickelt [DN66]. Simula baut auf Algol auf, führt Klassen und Vererbung ein und wird als erste objektorientierte Sprache angesehen.

Smalltalk, 1972 von Kay, Ingalls und Goldberg entwickelt [GR89], prägte den Begriff „objektorientiert“ [Kay93, p. 513] und verfolgt den puren Ansatz „Alles ist ein Objekt“. Auch Elemente einfacher Typen wie Ganzzahlen, sogar Codeblöcke und die Klassen selbst werden als Objekte behandelt. Smalltalk machte Objektorientierung populär und beeinflusste etliche nachfolgende Sprachen.

C++, 1983 von Stroustrup entworfen, erweitert C um objektorientierte Konstrukte wie Klassen und Vererbung und ist aufwärtskompatibel dazu [Str86]. C++ ist auf Effizienz ausgerichtet. Die Sprache ermöglicht systemnahe Programmierung und wird auch verbreitet dafür eingesetzt.

Eiffel, 1986 von Meyer entworfen [Mey92], kann als Gegenentwurf zu C++ angesehen werden. Die Sprache verfügt über wenige, aber ausdrucksstarke Konstrukte, u. a. Mehrfachvererbung, parametrische Polymorphie und über ein Vertragsmodell („design by contract“) mit Vor- und Nachbedingungen.

Java, 1995 von SUN vorgestellt, ist syntaktisch eng an die Sprache C++ angelehnt, verzichtet aber bewusst auf viele ihrer als fehlerträchtig eingestuften Konstrukte, u. a. auf Zeigerarithmetik und manuelle Speicherverwaltung [Gos+11]. Die mit der Übersetzung in eine Zwischensprache (JVM) einhergehende Flexibilität sowie das zeitliche Zusammenfallen der ersten Java-Versionen mit dem Boom des Internets haben die Sprache populär gemacht und zu ihrer weiten Verbreitung beigetragen.

C# wurde 2000 von Microsoft unter der Führung von Anders Hejlsberg entwickelt. C# weist Ähnlichkeiten mit Java auf, es unterstützt darüber hinaus etliche in Java nicht vorhandene Konstrukte, z. B. properties, extension methods, structs, oder die Möglichkeit, kontrolliert Code aus Sprachen wie C („unsafe Code“) einzubinden [Mic10].

Schwächen der objektorientierten Programmierung

Objektorientierung geht immer und überall von Zustand aus. Es gibt aber in fast allen Gegenstandsbereichen auch zustandslose Abstraktionen, Dinge wie Punkte, komplexe Zahlen oder Geldbeträge. Solche Abstraktionen mit Objektklassen und imperativen Sprachmitteln abzubilden, ist umständlich, unklar und fehlerträchtig. Weil Sprache die Weltsicht der Menschen beeinflusst, die sie benutzen, führt der Glaubensgrundsatz „Alles ist ein Objekt“ auch dazu, dass Werte, Funktionen und Zustandslosigkeit in der Gedankenwelt vieler Programmierer gar nicht vorkommen.

2.3 Funktionale Programmierung

Zustandslosigkeit

Das charakteristische Merkmal funktionaler Programmierung ist *Zustandslosigkeit*: die Abwesenheit von änderbaren Variablen, Zuweisungen oder sonstigen imperativen Sprachkonstrukten [Lou93, p. 12].⁴

Mathematische Funktionen

Namensgeber und zentrales Konzept der funktionalen Programmierung ist der mathematische Begriff der Funktion: eine eindeutige Zuordnungsvorschrift von einer Definitionsmenge auf eine Zielmenge. Beispiele für Funktionen sind Sinus, Quadratwurzel, Determinante (einer Matrix) oder die Addition (die zwei Zahlen auf eine dritte, nämlich ihre Summe, abbildet). Eine Funktion verändert nicht ihre Parameter, sondern sie ermittelt ausgehend von den Parametern ein Ergebnis. Die Berechnung einer Quadratwurzel wandelt z. B. nicht die Eingabe 9 in die 3 um, sondern sie ordnet der 9 das Ergebnis 3 zu. Funktionale Programmierung besteht im Wesentlichen aus der Anwendung („Applikation“) von Funktionen auf ihre Parameter [Seb10, p. 20], weswegen sie auch als „applikativ“ bezeichnet wird.

Ausdrücke, keine Anweisungen

Funktionale Programme enthalten ausschließlich Ausdrücke, und die Verarbeitung besteht aus der Auswertung von Ausdrücken. Es gibt keine Anweisungen, insbesondere keine Zuweisungen. Variablen in funktionalen Sprachen stehen nicht für veränderbare Speicherplätze, sondern sind Namen für feste Größen, was sie näher an den Variablenbegriff der Mathematik rückt. Weil es keine Seiteneffekte gibt, hängt das Ergebnis eines Ausdrucks nicht davon ab, in welcher Reihenfolge seine Teilausdrücke (z. B. die aktuellen Parameter eines Funktionsaufrufs) ausgewertet werden.

⁴Die Zustandslosigkeit funktionaler Abstraktionen kann auch durch die Aussage ausgedrückt werden, dass sie sich immer „in demselben Zustand“ befinden. In dieser Arbeit wird unter Zustand ausschließlich ein grundsätzlich änderbarer Zustand verstanden.

Werte

Funktionale Programmierung ist per se wert-orientiert; die Abstraktionen, mit denen die Programme hantieren, sind Werte. Dazu zählen mathematische Gebilde wie Zahlen, Vektoren oder Matrizen, auch viele nicht-mathematische Abstraktionen wie Zeichen und Zeichenketten, Längen- und Zeitangaben. Werte zeichnen sich durch Unveränderlichkeit und durch eine feste Wertemenge aus: Da jede Verarbeitung zustandslos ist, können Werte nicht verändert werden. Das Hinzufügen oder Löschen eines Wertes würde ebenfalls eine Zustandsänderung darstellen und ist damit ebenfalls nicht möglich.

Funktionale Steuerung

Ausgehend von Zustandslosigkeit haben sich in funktionalen Sprachen spezifische Sprachkonstrukte und Idiome etabliert:

Wiederholungen werden typischerweise mit Hilfe von *Rekursion* ausgedrückt. Schleifen – kennzeichnend für imperative Sprachen – wären wirkungslos, weil es keinen Zustand gibt, der sich mit jedem Schleifendurchlauf ändern ließe. Auch die Kontrolle über die Anzahl der Wiederholungen würde änderbaren Zustand erfordern, z. B. eine Variable zum Zählen der Durchläufe oder eine änderbare Abbruchbedingung.

Funktionale Sprachen kennen *bedingte Ausdrücke*. Diese liefern abhängig von der Bedingung eines von zwei oder mehr möglichen Ergebnissen (ähnlich dem ternären Operator von Java). Es gibt aber keine *bedingten Anweisungen* (wie if-then-else in imperativen Sprachen).

Weitere funktionale Sprachkonstrukte

Im Umfeld funktionaler Sprachen wurde eine Vielzahl weiterer Sprachkonstrukte entwickelt; sie werden deshalb als „typisch funktional“ und als essentiell für funktionale Programmierung angesehen (s. Hudak [Hud89] und Hughes [Hug84]).

Weil Funktionen im Mittelpunkt des Ansatzes stehen, unterstützen viele funktionale Sprachen sie als „Bürger erster Klasse“ und erlauben, sie auf die gleiche Weise zu verarbeiten wie einfachere Werte. Insbesondere können Funktionen als Parameter oder als Ergebnis einer anderen Funktion dienen. Das ermöglicht die Bildung von **Funktionen höherer Ordnung** („higher order functions“).

Etliche funktionale Sprachen unterstützen **Currying**. Dabei wird eine Funktion, die (ohne Currying) m Parameter hat, dargestellt als eine Funktion (höherer Ordnung) mit einem Parameter und einer Ergebnisfunktion mit $m-1$ Parametern. Dieses Verfahren ermöglicht, alle Funktionen als Funktion mit genau einem Parameter zu notieren.

Viele funktionale Sprachen unterstützen **Typinferenz** („type inference“, auch als Typableitung bezeichnet): Sie ermitteln den Typ eines Ausdruck aus seinen Be-

standteilen oder seinem Kontext. Auf diese Weise ist die Sprache statisch getypt, trotzdem kann der Anwendungsprogrammierer weitgehend auf explizite Typannotationen verzichten.

In funktionalen Sprachen sind **algebraische Datentypen** ein verbreitetes Mittel, um aus bestehenden Typen neue Typen zu bilden, als Produkttypen, Summentypen oder als Kombination davon. Musterabgleich (**Pattern Matching**) dient dazu, auf einfache und sichere Weise auf ihre Bestandteile zuzugreifen.

Die in vielen funktionalen Sprachen unterstützte **verzögerte Auswertung** („lazy evaluation“, „non-strictness“) ermittelt das Ergebnis eines Ausdrucks erst dann, wenn es für die weitere Verarbeitung gebraucht wird (und damit ggf. gar nicht). Bezogen auf Funktionsparameter entspricht verzögerte Auswertung dem Übergabemodus „call-by-name“ (oder, in einer optimierten Variante davon, „call-by-need“). Verzögerte Auswertung kann sich auch auf Datenstrukturen beziehen. Sie macht z. B. Streams („unendliche Listen“) möglich – von ihnen wird immer nur der benötigte Teil berechnet.

Pure und impure funktionale Sprachen

Funktionale Sprachen lassen sich in zwei Gruppen unterteilen:

Impure funktionale Sprachen unterstützen Werte, Funktionen und viele der oben genannten, „typisch funktionalen“ Sprachkonstrukte. Sie fördern einen funktionalen Programmierstil, erlauben aber daneben auch Zustand, z. B. in Form von änderbaren Variablen und Zuweisungen.

Rein funktionale („pure“) Sprachen unterbinden änderbaren Zustand und Seiteneffekte vollständig und verzichten konsequent auf alle zustandsbezogenen Sprachkonstrukte. Sie erzwingen zustandslose Verarbeitung und lassen keinen anderen Programmierstil zu.

Historie und Beispiele

Funktionale Programmierung hat sich zeitlich parallel zur imperativen Programmierung entwickelt. Sie ist überwiegend Gegenstand akademischer Forschung; der Einsatz funktionaler Sprachen im kommerziellen Umfeld beschränkt sich auf wenige Nischen (z. B. setzen T-Mobile, Motorola und Ericsson die funktionale Sprache Erlang in einigen produktiven Systemen ein [CT09, p. 2]).

Als Initialzündung für funktionale Programmierung gilt ein Vortrag, den John Backus 1977 anlässlich der Verleihung des Turing-Awards hielt [Bac78], für seine Arbeit an der (imperativen!) Sprache Fortran. Darin weist er auf die Unzulänglichkeiten hin, welche die implizite Ausrichtung imperativer Programmiersprachen auf die vorherrschende Rechnerarchitektur und die damit einhergehende Fokussierung auf Zustand mit sich bringt („Can Programming Be Liberated from the von Neumann Style?“). Als Gegenentwurf schlägt er ein radikal anderes, nämlich das

funktionale Verarbeitungsmodell vor und illustriert es am Beispiel der von ihm entwickelten Sprache FP.

In der Folge entstanden etliche Programmiersprachen, die diesen Kerngedanken aufgreifen und ihn zur Grundlage ihres Sprachentwurfs machen (u. a. Miranda und Haskell, beide rein funktional).

Bereits vor Backus' Vortrag gab es Sprachen, die Funktionen und Ausdrücke in den Vordergrund stellten und die heute dem funktionalen Paradigma zugeordnet werden (z. B. die impure Sprache LISP und APL). Die Aufstellung auf Seite 26 gibt einen Abriss über einige bedeutsame Vertreter des funktionalen Paradigmas.

Stärken der funktionalen Programmierung

Der konsequente Verzicht auf änderbaren Zustand und Seiteneffekte macht funktionale Programmierung sicher und verlässlich: Funktionale Programme sind ausdrucksstärker, leichter nachzuvollziehen, zu analysieren und zu testen als imperative. Programm-Verifikation und statische Programmanalyse sind in der funktionalen Welt leichter. Es gibt Ansätze, die Korrektheit funktionaler Programme durch formale Verfahren nachzuweisen. Weil das Ergebnis einer Funktion nicht von einem Zustand und nicht von der Reihenfolge der Auswertung der Teilausdrücke abhängt, sind funktionale Programme inhärent „thread-safe“. Sie eignen sich damit ideal für nebenläufige Programmierung und parallele Verarbeitung.

Schwächen der funktionalen Programmierung

Gleichzeitig ist die fehlende Unterstützung für änderbaren Zustand die größte Schwäche funktionaler Programmierung. Gegenstände der realen Welt und ihre Veränderungen im Laufe der Zeit passen nicht ins funktionale Schema. Um sie dennoch abzubilden, muss der Programmierer Umwege gehen. Ein mögliches Verfahren, um zustandsabhängige Verarbeitung in einem rein funktionalen Kontext nachzubilden, besteht z. B. darin, an eine Funktion den gesamten potentiell benötigten „Zustand“ in Form von expliziten Parametern zu übergeben („state passing“). Ein weiteres sind Monaden (s. z. B. Wadler [Wad95; Wad97]), ihr Einsatz hat sich insbesondere im Haskell-Umfeld etabliert. Solche Verfahren bilden in einem rein funktionalen Umfeld zustandsabhängige Verarbeitung durch komplexes Ineinandergreifen verschiedener Sprachmittel nach. Sie können trotzdem nicht von den Stärken funktionaler Programmierung profitieren: Beispielsweise wird mit Einsatz von Zustands-Monaden die Reihenfolge der Auswertung von Teilausdrücken relevant. Es spielt damit keine Rolle, ob die Programmiersprache Zustand direkt unterstützt, d. h. durch imperative Konstrukte, oder ob Zustand in einem funktionalen Umfeld durch aufwändige Techniken emuliert wird (s. dazu Kühne [Küh99, p. 268]).

Einige funktionale Sprachen

LISP wurde 1958 von John McCarthy entwickelt [McC78] und ist damit nach Fortran eine der ersten höheren Programmiersprachen. Neben den namensgebenden Listen (LISP = LISt Processing) zählen Funktionsapplikation und Rekursion zu den Eckpfeilern der Sprache. LISP ist bis heute in verschiedenen Dialekten in Gebrauch, dazu zählen u. a. **CommonLisp** [Gra97], **Scheme** [AS96] sowie die 2007 entstandene, u. a. auf der JVM lauffähige Sprache **Clojure** [FH14].

APL wurde 1964 von Kenneth E. Iverson entworfen [Ive62] und ist auf mathematisch-naturwissenschaftliche Programmierung ausgerichtet; z. B. können damit etliche Vektor- oder Matrix-Operationen in einem einzigen Befehl notiert werden. APL abstrahiert vollständig von der zugrunde liegenden Maschine und unterscheidet sich dadurch von den meisten anderen in den 1960er Jahren entstandenen Sprachen.

ML, 1973 von Robin Milner an der Universität von Edinburgh konzipiert, basiert ursprünglich auf einem Theorem-Beweis-System. Dessen zugehörige Sprache wurde zu einer vollwertigen Programmiersprache weiterentwickelt [MTH90]. ML unterstützt als erste Sprache Typinferenz. Es gibt mehrere Dialekte, am weitesten verbreitet sind SML (Standard ML) und Caml.

Hope, entwickelt von Burstall et al. [BMS80], entstand zeitgleich zu ML, aber unabhängig davon, an der Universität von Edinburgh. Hope führte algebraische Datentypen und das zugehörige Pattern Matching ein.

Miranda, entworfen 1985 von David Turner [Tur85], ist die erste funktionale Sprache, deren Ausführungsmodell standardmäßig auf verzögerter Auswertung beruht.

Haskell, von einem Team von Forschern entworfen (u. a. Peyton-Jones, Hudak, Hughes, Meijer, Wadler) wurde 1990 vorgestellt [Hud+92]. Haskell wurde entwickelt, um die zu diesem Zeitpunkt bekannten und bewährten Konzepte auf dem Gebiet der funktionalen Programmierung in einer Sprache zu vereinen. Die Sprache greift etliche Konstrukte ihrer Vorgänger auf (u. a. Typinferenz, algebraische Datentypen, verzögerte Ausführung), fügt einige neue hinzu (z. B. Typklassen) und kann heute als „Stand der Technik“ im Bereich der funktionalen Programmierung angesehen werden.

2.4 Gegenüberstellung imperativ versus funktional

Die Abschnitte 2.2 und 2.3 haben gezeigt, dass objektorientierte (oder allgemeiner: imperative) Programmierung und funktionale Programmierung als verschiedene Zweige in der Entwicklung der Programmiersprachen angesehen werden können, s. Abbildung 2.2.

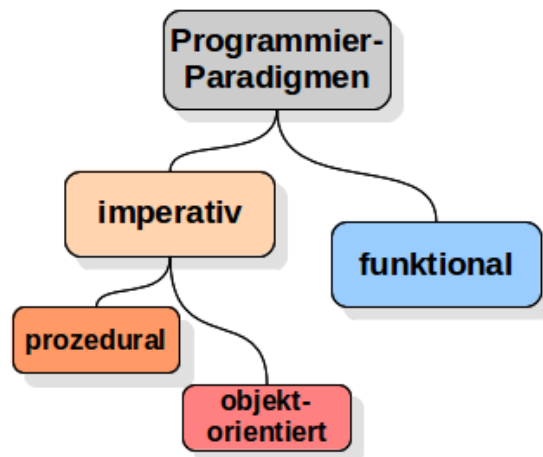


Abbildung 2.2: Programmierparadigmen

Es wurde ebenfalls deutlich, dass die beiden Programmierparadigmen als typisierende Leitbilder zu verstehen sind und dass eine konkrete Sprache nur selten ein Paradigma in Reinkultur verkörpert. Die meisten Sprachen enthalten zumindest Spurenelemente des „entgegengesetzten“ Paradigmas, etwa impure funktionale Sprachen, die in begrenztem Umfang auch über imperative Sprachkonstrukte verfügen, oder objektorientierte Sprachen, deren primitive Datentypen in der Regel rein funktionale Abstraktionen darstellen.

Zustand versus Zustandslosigkeit

Die fundamentalen Unterschiede der beiden Programmierparadigmen beruhen auf ihrem jeweiligen Umgang mit *Zustand*: Objektorientierte Programmierung basiert auf – prinzipiell änderbarem – Zustand, funktionale Programmierung dagegen auf Zustandslosigkeit. Das macht die beiden Paradigmen zu Gegensätzen:

„Indeed, the lack of local state in functional programming makes it in a sense the opposite of object-oriented programming, where computation proceeds by changing the local state of objects.“[Lou93, p. 362].

Tabelle 2.1 fasst die in den Abschnitten 2.2 und 2.3 beschriebenen, auf Zustand bzw. auf Zustandslosigkeit ausgerichteten Sprachkonstrukte zusammen und stellt sie einander gegenüber.

imperative Programmierung Unterparadigmen: prozedural, objektorientiert		funktionale Programmierung auch bezeichnet als: applikativ, wertorientiert	
Zustand, prinzipiell änderbar	Variable $\approx$ änderbarer Speicherplatz	Variable $\approx$ Name für eine feste Größe	Zustandslosigkeit
	Zuweisungen	keine Zuweisungen	
	Seiteneffekte	keine Seiteneffekte	
	Anweisungen und Ausdrücke	nur Ausdrücke, keine Anweisungen	
	Prozeduren oder Methoden	Funktionen	
	Objekte	Werte	
	Programm $\approx$ Sequenz von Anweisungen	Programm $\approx$ Aufruf von Funktionen	
	Verarbeitung $\approx$ schrittweise Änderung des Zustandes	Verarbeitung $\approx$ Auswertung von Ausdrücken	
	Reihenfolge der Ausführung ist entscheidend	Reihenfolge der Auswertung ist irrelevant	

Tabelle 2.1: Imperative und funktionale Sprachen: fundamentale Merkmale

Funktionale Programmierung als Einschränkung

Tabelle 2.1 verdeutlicht, dass die Grundbausteine objektorientierter und funktionaler Sprachen *nicht disjunkt* sind. Vielmehr bilden die elementaren Konstrukte funktionaler Sprachen eine Teilmenge oder Spezialfälle von denen objektorientierter Sprachen. Z. B. enthalten sowohl funktionale als auch objektorientierte Programme Ausdrücke. Z. B. sind Funktionen spezielle Prozeduren (nämlich solche, die ein Ergebnis ermitteln und weder Zustand ändern noch davon abhängen). Funktionale Programmierung zeichnet sich gegenüber objektorientierter Programmierung also nicht durch andersartige Sprachkonstrukte aus, sondern durch *Einschränkungen*.

Grundlegende Berechnungsmodelle

Theoretische Grundlage imperativer Sprachen ist die Turing-Maschine [Tur36]. Funktionale Sprachen basieren auf dem (ungetypten) Lambda-Kalkül [Chu36]. Es wurde bereits früh nachgewiesen, dass diese beiden formalen Berechnungsmodelle gleich mächtig sind: Jede von einer Turing-Maschine durchgeführte Berechnung kann auch im Lambda-Kalkül ausgedrückt werden und umgekehrt [Tur37]. Auf einer grundlegenden Ebene sind objektorientierte und funktionale Programmierung also äquivalent, keine von beiden ist prinzipiell mächtiger oder ausdrucksstärker.

Aus softwaretechnischer Sicht aber führen die beiden Ansätze zu grundlegend verschiedenen Programmiermodellen. Im zustandsbehafteten/ objektorientierten und im zustandslosen/ funktionalen Umfeld haben sich jeweils unterschiedliche Programmierstile, Idiome, Sicht- und Herangehensweisen etabliert.

Zustandsunabhängige Sprachkonstrukte

In beiden Sprachparadigmen existieren etliche als „typisch“ angesehenen Sprachkonstrukte (s. Abschnitt 2.3), die nicht auf Zustand bzw. nicht auf Zustandslosigkeit aufbauen.

Beispielsweise gelten Funktionen höherer Ordnung als wesentliches Merkmal („distinguishing feature“) funktionaler Programmierung (s. Hudak [Hud89, p. 382]). Prinzipiell ist es aber auch in einem zustandsbehafteten Umfeld denkbar, Funktionen (Codeblöcke, Closures, ...) als Parameter zu übergeben. Tatsächlich finden sich in etlichen objektorientierten Sprachen derartige Sprachkonstrukte: Smalltalk unterstützt Blöcke, C++ function pointer, Eiffel (in der Fassung von 2006) Agents [ECM06b], Java führte mit Version 8 Lambdas ein [Goe14].

Ein weiteres Beispiel ist Kapselung, sie gilt als wesentliches Merkmal objektorientierter Programmierung. Sie ist aber nicht zwingend an Zustand gebunden. Auch zustandslose Typen können so programmiert werden, dass sie ihre interne Repräsentationen verbergen und mit Klienten über eine definierte Schnittstelle kommunizieren.

2.5 Objekt-funktionale Sprachen

Historisch haben sich imperative Sprachen und funktionale Sprachen parallel zueinander entwickelt und nur wenig gegenseitig beeinflusst.

Es hat einige Ansätze gegeben, objektorientierte und funktionale Programmierung miteinander zu kombinieren und in einer hybriden („objekt-funktionalen“) Sprache zu vereinen.

Beispiele/ Historie

Die Aufstellung auf Seite 30 nennt einige Beispiele für objekt-funktionale Sprachen, die – gemessen an ihrer historischen Bedeutung, wegen eines Alleinstellungsmerkmals oder aufgrund ihrer Aktualität – als repräsentativ für diese Gruppe von Sprachen anzusehen sind.

Der Ansatz objekt-funktionaler Sprachen

Ungeachtet vieler spezifischer Besonderheiten liegt diesen objekt-funktionalen Programmiersprachen ein gemeinsamer Ansatz zugrunde: Sie vereinigen möglichst viele

Einige objekt-funktionale Sprachen

CLOS (Common Lisp Object System [BGW91]) war eine der ersten Sprachen, die objektorientierte und funktionale Programmierung vereinte. CLOS erweitert LISP um objektorientierte Konstrukte wie Klassen und (mehrfache) Vererbung. Anders als viele objektorientierte Sprachen ordnet CLOS Methoden nicht einer Klasse zu, sondern gruppiert sie zu „generischen Operationen“.

UFO [Sar93] wurde in erster Linie für parallele Verarbeitung entworfen: Es enthält eine rein funktionale Teilsprache („functional UFO“) mit zustandslosen Klassen. Zustand wird durch „stateful classes“ abgebildet, aber als Ausnahme betrachtet und – wegen der impliziten Parallelität – starken Einschränkungen unterworfen.

Clover [CB96] geht von einem rein funktionalen Sprachkern aus und ergänzt objektorientierte Konstrukte wie Klassen, Vererbung, Überschreiben und Überladen von Methoden. Die Sprache lässt keine Seiteneffekte zu und erhält so den rein funktionalen Ansatz.

O’Caml („Objective Caml“ [Ler+13]) wurde 1996 am INRIA (Institut National de Recherche en Informatique et en Automatique) von Leroy et al. entwickelt. O’Caml baut auf der impuren funktionalen Sprache ML auf und erweitert sie um objektorientierte Konstrukte wie Klassen und mehrfache Vererbung.

O’Haskell [Nor01] ist eine Erweiterung von Haskell um objektorientierte Konstrukte wie Klassen und Vererbung. Die Sprache lässt keine Seiteneffekte zu und bleibt damit (wie Clover) „pur“. Die als ihr Nachfolger geltende Sprache **Timber** [Bla+02] verfügt dagegen über Zuweisungen und Seiteneffekte.

F# [SGC07], hervorgegangen aus einem Forschungsprojekt von Microsoft Research, ist an ML und O’Caml angelehnt. F# unterstützt auf der objektorientierten Seite Klassen (sowie structs, enums und interfaces) und Vererbung, auf der funktionalen Seite Typinferenz, Pattern matching, Funktionen höherer Ordnung und Currying.

Scala [OSV11] wurde 2003 von M. Odersky an der EPFL entwickelt und läuft auf der Java Virtual Machine. Scalas objektorientierte Bestandteile sind größtenteils an die von Java angelehnt. Auf der funktionalen Seite unterstützt Scala Typinferenz, Pattern matching (in Form von Case Classes), verzögerte Auswertung und Funktionen höherer Ordnung.

als „typisch objektorientiert“ und „typisch funktional“ geltende Konstrukte innerhalb einer Sprache.

Die genannten Sprachen verfügen auf der funktionalen Seite über:

- Funktionen höherer Ordnung, anonyme Funktionen oder Closures
- Algebraische Datentypen und Pattern Matching
- Currying
- Typinferenz (mit Ausnahme der vor 1996 entstandenen Sprachen)
- verzögerte Ausführung („Laziness“)

und auf der objektorientierten Seite über:

- vom Programmierer definierbare Klassen
- Kapselung der internen Repräsentation (Ausnahme: CLOS)
- Vererbung und die daraus resultierende Subtyppolymorphie
- Abstrakte Methoden, Überschreiben von Methoden in Subklassen
- Überladen von Methoden und/ oder Operatoren

Dilemma: Unvereinbarkeit

Das Verbinden von objektorientierter und funktionaler Programmierung zielt explizit oder implizit darauf ab, ihre jeweiligen Vorteile zu kombinieren. Doch das stößt auf ein Dilemma: Die beiden Paradigmen sind in ihren Grundzügen nicht nur verschieden – sie schließen einander aus. Zustandslosigkeit ist das logische Gegenteil von änderbarem Zustand. Deshalb ist es nicht so leicht möglich, durch Zusammenfügen typisch objektorientierter und typisch funktionaler Sprachkonstrukte auch die Stärken beider Paradigmen zu vereinen. Das zeigt sich auch bei den oben aufgeführten hybriden Sprachen:

Viele der genannten Sprachen, z. B. O’Caml, F# oder Scala, lassen änderbaren Zustand grundsätzlich zu, sie sind „impure“. Dadurch verlieren sie die wesentlichen Vorteile funktionaler Programmierung: die mit der garantierten Abwesenheit von Zustand einhergehende Sicherheit und alle daraus resultierenden Eigenschaften, von Nachvollziehbarkeit bis zur Eignung für parallele Verarbeitung (s. Abschnitt 2.3).

„Pure hybride“ Sprachen (wie Clover) gehen den umgekehrten Weg. Sie schließen jegliche Zustandsverarbeitung aus und können damit die Stärken funktionaler Programmierung ausspielen. Sie eignen sich aber gerade wegen ihrer fehlenden Unterstützung von Zustand nicht für die Abbildung von Gegenständen der realen Welt und damit kaum für den Einsatz in der Praxis.

2.6 Vorgehen in dieser Arbeit

Die Erweiterung objektorientierter Sprachen um Werttypen stellt ebenfalls einen Ansatz zur Verbindung objektorientierter und wert-orientierter (=funktionaler) Programmierung dar. Das in dieser Arbeit gewählte Vorgehen unterscheidet sich in mehreren Hinsichten von den in Abschnitt 2.5 aufgeführten hybriden Sprachen: in der Fokussierung auf Zustand und Zustandslosigkeit, in der Verankerung am Typsystem und in der gewählten Ausgangsbasis.

Fokus auf Zustand versus Zustandslosigkeit

Diese Arbeit fokussiert zunächst auf die grundlegenden Charakteristika objektorientierter und funktionaler Programmierung: Zustand und Zustandslosigkeit. Sie grenzt sie klar gegeneinander ab und erkundet ihr mögliches Zusammenspiel.

Es ist nicht das Ziel der Arbeit, möglichst viele Sprachkonstrukte aus beiden Paradigmen zusammenzubringen. Das gilt insbesondere im Hinblick darauf, dass etliche als „typisch funktional“ eingeordnete Sprachkonstrukte nicht mit Zustandslosigkeit zusammenhängen, und viele „typisch objektorientierte“ Konstrukte nicht mit Zustandsverarbeitung (s. Abschnitt 2.4).

Erst nach eingehender Untersuchung, Abgrenzung und Verknüpfung der zustandsbehafteten und der zustandslosen Welt befasst sich die Arbeit mit einigen Auswirkungen des resultierenden Sprachmodells auf andere Sprachkonstrukte und erhärtet und konkretisiert dadurch den vorgeschlagenen Ansatz.

Verankerung am Typsystem

Durch die Unterscheidung von zustandsbehafteten Objekttypen und zustandslosen Werttypen verankert die Arbeit die grundlegenden Charakteristika der beiden Paradigmen am Typsystem. Zustand ist dadurch, ebenso wie Zustandslosigkeit, eine Eigenschaft eines Typs, nicht ein Merkmal einer einzelnen Variablen oder einer gesamten Sprache.

Ausgangsbasis Objektorientierung

Die Arbeit setzt auf objektorientierter Programmierung als Ausgangsbasis auf. Sie ergänzt eine zustandsbehaftete Welt um zustandslose Abstraktionen. Damit geht sie einen anderen Weg als die in Abschnitt 2.5 beschriebenen Ansätze, die entweder von funktionaler Programmierung ausgehen und objektorientierte Sprachkonstrukte hinzufügen (wie CLOS, UFO, O’Caml oder O’Haskell) oder die Sprache als Mischung beider Paradigmen von Grund auf neu entwerfen (wie F# oder Scala).

Für Objektorientierung als Startpunkt sprechen mehrere Faktoren:

- Änderbarer Zustand ist unverzichtbar für die Abbildung von Gegenständen der realen Welt und eine wesentliche Stärke objektorientierter Programmierung

(siehe dazu Meyer [Mey97, p. 750], Kühne [Küh99, p. 38, p. 68], van Roy und Haridi [RH04, p. 315]).

- Die elementaren Bausteine der funktionalen Programmierung sind in denen der objektorientierten Programmierung bereits enthalten (s. Abschnitt 2.4). Darum liegt es näher, Werte und Zustandslosigkeit in die objektorientierte Welt zu integrieren als umgekehrt die funktionale Welt um Objekte und Zustand zu erweitern.⁵
- Objektorientierung unterstützt die Modularisierung von Softwaresystemen und zugehörige softwaretechnische Prinzipien – Kapselung, Trennung von Schnittstelle und Implementation – direkt durch passende Sprachmechanismen.
- Objektorientierung ist in der Praxis erprobt und bewährt.

Voraussetzungen

Weil Typen flexibel programmierbar und im Quellcode erkennbar sein sollen, ist ein Sprachkonstrukt zur Definition eines Typs – entsprechend dem Konstrukt der *Klasse* in objektorientierten Sprachen – erforderlich.

Im Hinblick auf die angestrebte Unterscheidung von Werten und Objekten spielt es eine wesentliche Rolle, dass mit *statischer Typisierung* jedem Ausdruck im Quelltext ein Typ zugeordnet werden kann.

Die Arbeit geht darum von *objektorientierten*, *klassenbasierten*, *statisch getypten* Programmiersprachen aus.

Fokus der Arbeit

Weil Objektklassen und Zustand als vorhanden postuliert werden, befasst sich die Arbeit schwerpunktmäßig mit der Gestaltung des „zustandslosen Teils“: mit Werttypen, mit ihren Eigenschaften und mit ihrer Integration in ein sprachliches Umfeld, das änderbaren Zustand prinzipiell zulässt.

2.7 Zusammenfassung

Kapitel 2 befasste sich mit den beiden für diese Arbeit relevanten Programmierparadigmen. Objekte und objektorientierte Programmierung verkörpern das imperative Paradigma, Werte und wert-orientierte Programmierung gehören dem funktionalen Paradigma an.

Es wurde untersucht, welche Sprachmerkmale die jeweiligen Paradigmen auszeichnen und worin ihre Stärken und Schwächen liegen. Eine Gegenüberstellung

⁵Kühne begründet mit dem gleichen Argument, warum es aussichtsreicher ist, funktionale Verarbeitung in objektorientierten Sprachen mit Hilfe von Mustern abzubilden als umgekehrt [Küh99, p. 45].

arbeitete das unterscheidende Wesensmerkmal heraus: imperative Programmierung ist geprägt von änderbarem Zustand, funktionale Programmierung von Zustandslosigkeit.

Es gibt eine Reihe objekt-funktionaler Sprachen. Sie vereinen eine Vielzahl von Sprachkonstrukten, die als „typisch objektorientiert“ und als „typisch funktional“ gelten. Etliche davon weisen aber keinen Zusammenhang zu Zustand oder Zustandslosigkeit auf. Objekt-funktionale Sprachen stehen vor einem Dilemma: Zustand und Zustandslosigkeit sind Gegensätze. Das erschwert es, die Vorteile der beiden Paradigmen innerhalb einer Sprache miteinander zu verbinden.

Kapitel 2 skizzierte das Vorgehen der vorliegenden Arbeit: Sie fokussiert auf Zustand und Zustandslosigkeit, und damit auf die grundlegenden Charakteristika der beiden Paradigmen. Sie verankert diese beiden Merkmale am Typsystem der Sprache. Sie nimmt das objektorientierte Paradigma als Ausgangspunkt und konzentriert sich darauf, das „fehlende“, funktionale Paradigma in Form von Werttypen zu ergänzen und in eine grundsätzlich zustandsbehaftete Umgebung einzubinden.

Kapitel 3

Werte und Werttypen: Begriffsbestimmung

Der Begriff „Wert“ ist nicht einheitlich definiert und wird in vielen Arbeiten nur unscharf verwendet. Als Basis für die nachfolgende Arbeit wird eine Definition für Werte und Werttypen angegeben.

Die Definition bringt zum Ausdruck, dass Werte zustandslose Abstraktionen sind und legt dafür klare, überprüfbare Kriterien fest. Werttypen werden anhand ihrer charakteristischen Eigenschaften definiert, und diese Eigenschaften grenzen Werttypen gegenüber Objekttypen ab. Bereits aus der Definition lassen sich Schlussfolgerungen für das resultierende Sprachmodell ziehen.

3.1 Der Begriff „Wert“ in der Informatik

Der Begriff „Wert“ ist in der Informatik allgegenwärtig und wird oft in einem allgemeinen Sinne gebraucht. Man spricht vom Wert einer Variablen oder vom Rückgabewert einer Operation. Ein Typ wird oft beschrieben als eine „Menge von Werten“. Ausdrücke werden zur Laufzeit *ausgewertet* („e-*valu*-iert“). Es gibt Wertparameter („*call-by-value*“) und Wertsemantik. Diese vielfältige Verwendung macht den Begriff „Wert“ unscharf und mehrdeutig.

Es gibt bereits eine Reihe von Arbeiten, die Werte als spezielle Abstraktionen ansehen und gegenüber Objekten abgrenzen. Der Fokus dieser Arbeiten liegt häufig auf der Beschreibung geeigneter Modellierungs- oder Programmier Techniken zur Abbildung von Werten, d. h. sie befassen sich vorrangig damit, *wie* mit Werten umzugehen ist. Dabei tritt eine präzise Definition, *was* unter Werten zu verstehen ist, in den Hintergrund. Diese Arbeiten stützen sich auf unterschiedliche, oft schwer messbare und teils vage Umschreibungen. Ihr Wert-Begriff ist nicht einheitlich, und teils widersprechen sie sich:

Eckert & Kempe [EK95] setzen Werte mit unveränderlichen Objekten gleich. Gemäß van der Werf [Wer98] haben Werte „keine Identität“ und „keinen Zustand“. Fowler [Fow03; Fow09] beschreibt Werte zunächst als „kleine, einfache Objekte“ – und merkt an, diese Definition des Wertbegriffs sei „annoyingly informal“. Er charakterisiert Werte an anderer Stelle als „Objekte, deren Gleichheit nicht auf Identität basiert“. Evans [Eva04] führt aus, dass Werte „keine konzeptuelle Identität“ haben und einen „beschreibenden Aspekt“ darstellen. Henney [Hen00] versteht unter Werten „fein-granulare Objekte“, mit „signifikantem Zustand“ und „transparenter Identität“. Gemäß Hoogendorn [Hoo07] sind Werte „klein“ und „prüfen Gleichheit ausschließlich anhand der Werte ihrer Eigenschaften“. Die Unterscheidung von Werten und Objekten hat (in jüngerer Zeit) auch Eingang in Lehrbücher zur Objektorientierung gefunden: Nach Poetzsch-Heffter [Poe09] sind Werte „abstrakt, unveränderlich und lassen sich nicht lokalisieren“. Lahres & Raýman [LR09] charakterisieren Werte durch ihre „fehlende Identität“.

Beispiele können den Begriff „Wert“ veranschaulichen. Prototypische Beispiele für Werte sind Zahlen, z. B. ganze, rationale oder komplexe Zahlen. Das gilt ebenfalls für andere mathematische Abstraktionen wie Vektoren, Matrizen oder bool'sche Werte. Die oben genannten Arbeiten führen daneben zahlreiche weitere, nicht-mathematische Beispiele für Werte auf: Datum, Zeichenkette, Geldbetrag, Punkt, Wochentag, ISBN oder URL. Beispiele können einen intuitiven Eindruck davon vermitteln, was Werte ausmacht. Sie ersetzen aber keine klare Definition.

Eine präzise Begriffsbestimmung ist als Ausgangspunkt für die weitere Arbeit unerlässlich. Eine klare Definition hat auch einen praktischen Nutzen: Sie dient der Modellierung und Programmierung als Richtschnur, welche Abstraktionen eines abzubildenden Gegenstandsbereichs als Werte und welche als Objekte einzuordnen sind.

Dieses Kapitel legt fest, was in der hier vorliegenden Arbeit unter Werten und Werttypen verstanden werden soll. Die Definition des Wertbegriffs soll

- klar überprüfbare Kriterien für Werte nennen,
- die wesentlichen („essentiellen“) Eigenschaften von Werten herausarbeiten und von unwesentlichen („akzidentellen“) abstrahieren,
- Werte gegenüber Objekten abgrenzen, und
- sich nah an den Wertbegriffen der genannten Arbeiten bewegen, insbesondere die aufgeführten Beispiele einschließen.

3.2 Programmiersprachliche Grundbegriffe

Bevor die zentralen Begriffe „Wert“ und „Werttyp“ definiert werden können, ist es erforderlich, als Ausgangsbasis die Begriffe, auf denen sie aufbauen, zu klären. Kapitel 2 hat erkennen lassen, dass viele fundamentale programmiersprachliche Begriffe nicht explizit, nicht präzise oder nicht einheitlich definiert sind. Dieser Umstand wurde bereits von Strachey kritisiert:

„Any discussion on the foundations of computing runs into severe problems right at the start. The difficulty is that although we all use words such as 'name', 'value', 'program', 'expression' or 'command' which we think we understand, it often turns out on closer investigation that in point of fact we all mean different things by these words, so that communication is at best precarious.“[Str00]

Die folgende Aufstellung nennt die grundlegenden Sprachkonstrukte, deren Vorhandensein vorausgesetzt wird und erläutert, in welcher Bedeutung sie in dieser Arbeit verwendet werden.

Variable: *ein vom Programmierer definierter Bezeichner im Programmtext, der stellvertretend für ein Element steht und Zugriff auf dieses Element ermöglicht.*

Dieser Variablenbegriff umfasst z. B. lokale Variablen, Exemplarvariablen und formale Parameter von Operationen. Er legt – mit Absicht – nicht fest, ob die Variable durch Zuweisungen änderbar oder fest mit dem zugehörigen Element verknüpft ist. Er beinhaltet auch keine Aussage darüber, auf welche Weise eine Variable speichertechnisch umgesetzt ist. Er bildet den kleinsten gemeinsamen Nenner der Variablenbegriffe imperativer und funktionaler Sprachen.

Zustand: *der aktuelle Inhalt aller Variablen eines Systems oder eines Gegenstandes.*

Im engeren Sinne wird unter Zustand nur ein prinzipiell änderbarer Zustand verstanden. Bei Gegenständen, die nur genau „einen Zustand“ annehmen, diesen aber nicht ändern können, wird nicht von Zustand gesprochen.

Seiteneffekt: *eine beobachtbare Zustandsänderung.*

Seiteneffekte können z. B. durch Zuweisungen bewirkt werden, durch Aufrufe von Operationen oder durch Ausgaben auf die Konsole oder in eine Datei.

Ausdruck: *ein Stück Quelltext, das ein Element bezeichnet.*

Als Ausdrücke zählen u. a. Literale und Konstanten (bezeichnen immer dasselbe Element), Variablen (bezeichnen das aktuell an sie gebundene Element) oder Aufrufe von Operationen, die ein Ergebnis liefern (bezeichnen das durch den Operationsaufruf ermittelte Element).

Anweisung: *eine Stück Quelltext, das ausgeführt werden und dabei einen Seiteneffekt bewirken kann.*

Eine Zuweisung ist eine spezielle Anweisung (sie verändert genau eine Variable). Weitere Anweisungen sind z. B. Prozeduraufrufe.

Operation: *eine aufrufbare, benannte Codeeinheit.*

In den meisten Sprachen können Operationen vom Programmierer definiert werden, oft enthält die Sprache auch einige vordefinierte Operationen. Operationen können (müssen aber nicht): Ergebnisse liefern, formale Parameter haben oder Seiteneffekte bewirken. Der Begriff der Operation subsumiert die im letzten Kapitel angesprochenen Begriffe Funktion, Prozedur, Routine, Methode, Konstruktor, etc. Der Operationsbegriff abstrahiert von syntaktischen Details und lässt offen, mit welcher Notation der Aufruf erfolgt (Klammern, Punktnotation, Juxtaposition, Infix-Operator, ...).

Referentielle Transparenz: *die Eigenschaft einer Operation, für die gleichen Eingaben immer das gleiche Ergebnis zu liefern.*

Das Ergebnis einer referentiell transparenten Operation ist durch ihre Parameter festgelegt. Es hängt insbesondere nicht von einem Zustand ab, auch nicht von anderen Faktoren, etwa der Reihenfolge der Auswertung der Eingabeparameter. Ist eine Operation nicht referentiell transparent, wird sie als *referentiell opak* bezeichnet.¹

Funktion: *eine Operation, die ein Ergebnis ausschließlich auf Basis ihrer Eingabeparameter ermittelt und die weder Zustand ändert (d. h. sie ist seiteneffektfrei) noch Zustand in das Ergebnis einfließen lässt (d. h. sie ist referentiell transparent).*

Dieser Funktionsbegriff entspricht dem der funktionalen Programmierung. Er erleichtert im weiteren Verlauf der Arbeit die sprachliche Trennung von zustandslosen und zustandsbehafteten Operationen. Der Begriff der Funktion wird hier nicht in einem umfassenderen Sinne für alle Operationen verwendet, die ein Ergebnis liefern. Diese werden als Funktionsprozeduren bezeichnet.

Typ: *eine Menge von Elementen zusammen mit den für sie verfügbaren Operationen.* Dieser Typbegriff geht auf Hoare zurück [Hoa72]. Hier wurde eine (gegenüber Hoare)

¹Der Begriff der referentiellen Transparenz ist zentral für diese Arbeit. Er stammt aus der Linguistik und Sprachphilosophie (s. dazu Quine [Qui60]). Referentielle Transparenz beschreibt dort die Eigenschaft eines Kontextes, z. B. eines Satzes oder eines Ausdrucks, dass ein Teilausdruck durch einen gleichwertigen ersetzt werden kann, ohne den Sinn zu verändern („Ersetzung von Gleichem durch Gleiches“). In dieser Bedeutung wird er vielfach auch in der Informatik benutzt (s. dazu u. a. Strachey [Str00] oder Søndergaard und Sestoft [SS90]). Diese Ersetzbarkeit folgt aus der in dieser Arbeit verwendeten Definition, aber nur in einem Kontext ohne Zustandsänderungen. Enthält ein zusammengesetzter Ausdruck Operationen mit Seiteneffekten – allgegenwärtig in imperativen, objektorientierten Sprachen – so zerstört das i. A. die Ersetzbarkeit. Weil es in dieser Arbeit um das Einbinden von Werten in ein objektorientiertes Umfeld geht, wurde hier bewusst die auf eine Operation bezogene, in einem imperativen Kontext tragfähige Definition gewählt. In dieser Bedeutung ist der Begriff der referentiellen Transparenz u. a. auch bei Horowitz [Hor83, p. 357] oder bei Louden [Lou93, p. 361] zu finden. Weitere Details siehe Abschnitte 3.3.2 und 3.3.9 bis 3.3.11.

abgewandelte Formulierung verwendet, nämlich „eine Menge von Elementen“ statt „eine Menge von Werten“, weil der Begriff „Wert“ in dieser Arbeit nur in einem engeren Sinne für die Elemente von Werttypen benutzt wird.

Element: *ein Element eines Typs.*

Die – weiter unten folgende – Trennung von Werttypen und Objekttypen impliziert die Unterscheidung von Werten und Objekten. Der Begriff „Element“ bildet einen Oberbegriff für beide. Er wird verwendet, wenn sich eine Aussage auf die Elemente eines Typs bezieht und nicht bekannt oder nicht relevant ist, ob es sich um Werte oder Objekte handelt.

Typkonstruktor: *ein programmiersprachlicher Mechanismus, mit dem der Programmierer einen Typ definieren kann.*

Beispielsweise verfügt C# über die Typkonstruktoren `class`, `enum`, `struct` und `interface`.

Klasse: *ein Stück Code (in vielen Sprachen durch das Schlüsselwort `class` eingeleitet), welches einen (Objekt-)Typ und eine zugehörige Implementation beschreibt.*

Eine Klasse enthält Datenfelder und Methoden. Datenfelder legen die strukturelle Repräsentation der zur Klasse gehörenden Objekte fest, Methoden die für die Objekte der Klasse aufrufbaren Operationen. Methoden und Datenfelder einer Klasse werden unter dem Oberbegriff „Member“ zusammengefasst.

Datenfeld: *eine in einer Klasse definierte Variable.*

Jedes Objekt der Klasse verfügt über ein eigenes Datenfeld dieses Namens, sein Inhalt ist unabhängig vom Inhalt des gleichen Datenfeldes anderer Objekte. In etlichen objektorientierten Sprachen werden Datenfelder auch als Exemplarvariablen, Instanzvariablen oder Membervariablen bezeichnet.²

Methode: *eine in einer Klasse definierte Operation.*

Der Aufruf einer Methode involviert ein Objekt der Klasse, als *Receiver* bezeichnet. Der Receiver nimmt in der Aufrufsyntax häufig eine besondere Position ein (z. B. bei der in Java verwendeten „Punktnotation“ vor dem Punkt statt in der Parameterliste). Der Methodenaufruf wird auch als Aufruf „an diesem Objekt“ oder als „Botschaft an den Receiver“ bezeichnet.³

Konstruktor: *eine spezielle Operation einer Klasse, deren Aufruf ein neues Objekt dieser Klasse erzeugt.*

Die Definition und der Aufruf von Konstruktoren unterliegen in den meisten Sprachen einer speziellen Syntax, spezifischen Schlüsselworten und Regeln, die sie von

²Klassenvariablen (z. B. in Java definiert durch das Schlüsselwort `static`) spielen in dieser Arbeit keine Rolle.

³Die von vielen objektorientierten Sprachen unterstützten Klassenmethoden werden anders aufgerufen, sie spielen in dieser Arbeit keine Rolle.

anderen Methoden unterscheiden. Ein Konstruktor „initialisiert“ in der Regel das zu erstellende Objekt, d. h. er legt die anfänglichen Inhalte seiner Datenfelder fest.

Klient: *ein Stück Code, das ein anderes Stück Code benutzt.*

Z. B. benutzen Klienten eine Klasse, indem sie Variablen dieses Typs definieren, Objekte dieser Klasse erzeugen und/ oder Methoden dieser Klasse aufrufen.

3.3 Werttypen: Definition und Abgrenzung

Prototypische Beispiele für zustandslose Abstraktionen und damit für Werte sind Zahlen (s. auch Abschnitt 1.2 und Abschnitt 3.1). An ihnen orientiert sich der Wertbegriff dieser Arbeit. Ihre essentiellen Eigenschaften dienen als Vorlage für die Definition und werden im Folgenden herausgearbeitet.

3.3.1 Kein Verändern, kein Erzeugen

Zahlen sind *unveränderlich*: ihre Operationen verändern nicht die Zahlen, auf die sie angewendet werden, sondern sie liefern als Ergebnis eine andere Zahl. Addiert man z. B. die beiden rationalen Zahlen $1/2$ und $1/3$, so erhält man als Ergebnis $5/6$, aber weder $1/2$ noch $1/3$ werden verändert.

Zahlen *werden nicht erzeugt* und nicht vernichtet. Sie existieren per se. Im Konzept der ganzen Zahlen sind alle zugehörigen Elemente bereits enthalten. Sie bilden eine feste Menge, der keine weiteren Elemente hinzugefügt werden können. Man spricht zwar z. B. davon, dass eine „neue“ Primzahl entdeckt wurde, aber das ist eine sprachliche Ungenauigkeit. In diesem Fall geht es vielmehr darum, dass man an einer (bereits existierenden) Zahl eine bisher noch nicht bekannte Eigenschaft (hier: die Primzahl-Eigenschaft) nachweisen konnte.

Unerzeugbarkeit kann als eine spezielle Form von Unveränderlichkeit interpretiert werden, nämlich als Unveränderlichkeit der Wertemenge. Nicht nur ist jede rationale Zahl unveränderlich, sondern auch die Menge aller rationalen Zahlen. Es können weder rationale Zahlen gelöscht noch neue rationale Zahlen erzeugt werden.

3.3.2 Seiteneffektfreiheit und referentielle Transparenz

Um die Zustandslosigkeit von Zahlen adäquat zu beschreiben, sind Unveränderlichkeit und Unerzeugbarkeit nicht hinreichend. Operationen auf Zahlen bewirken keinerlei Zustandsänderungen, sie sind also seiteneffektfrei. Operationen auf Zahlen hängen auch nicht von Zustand ab, sie sind damit referentiell transparent.

3.3.3 Definition Werttyp

Der Werttypbegriff dieser Arbeit orientiert sich an den Eigenschaften, die an Zahlentypen zu beobachten sind und verallgemeinert sie zu einem universellen Konzept.

Definition:

Werte sind die Elemente eines Werttyps. Der Werttyp beschreibt die Menge der zu ihm gehörenden Werte und die darauf anwendbaren Operationen.

Werte sind *unveränderlich*, sie werden *nicht erzeugt* und nicht vernichtet.

Ihre Operationen sind *frei von Seiteneffekten* und *referentiell transparent*.

Der Wertbegriff beschreibt Werte anhand ihrer *konzeptionellen* Eigenschaften, unabhängig von einer zugrunde liegenden technischen Implementation. Unveränderlichkeit bedeutet, dass es keine Möglichkeit gibt, eine Veränderung eines Wertes zu beobachten. Interne, aber für Klienten nicht sichtbare Veränderungen, z. B. das Caching einer abgeleiteten Eigenschaft zwecks Vermeidung einer erneuten Berechnung oder der Wechsel zu einer anderen internen Repräsentation, stehen nicht in Widerspruch zur Unveränderlichkeit. Unerzeugbarkeit bedeutet, dass Werte aus Sicht von Klienten so wirken, als existierten sie schon immer. Trotzdem kann z. B. in der zugrunde liegenden Implementation Speicher für die Repräsentation dieses Wertes allokiert werden.

Die Definition legt sowohl Eigenschaften der Elemente eines Werttyps (Unveränderlichkeit und Unerzeugbarkeit) als auch Eigenschaften seiner Operationen (Seiteneffektfreiheit und referentielle Transparenz) fest. Wertoperationen sind damit per Definition Funktionen im Sinne der funktionalen Programmierung.

Unveränderlichkeit und Unerzeugbarkeit können auch als Aussagen über die Operationen eines Werttyps aufgefasst werden. Unveränderlichkeit bedeutet: es gibt keine Operation, die eine sichtbare Änderung eines Wertes bewirkt. Eine Änderung würde dadurch sichtbar, dass das Ergebnis einer anderen Operation vor und nach dem Aufruf der ersten Operation unterschiedlich wäre. Unerzeugbarkeit bedeutet: es gibt keine Operation, die einen neuen, zuvor noch nicht existierenden, Wert liefert. Die Elementemenge des Werttyps ist damit festgelegt, sie besteht aus der Menge der Ergebnisse aller Operationen, die ein Element dieses Typs liefern. Der Begriff des Werttyps steht dadurch in Einklang mit dem Konzept der – ausschließlich über ihre Operationen definierten – abstrakten Datentypen.

3.3.4 Werttypen versus Objekttypen

Die vier in der Definition festgelegten Eigenschaften sind Unterscheidungskriterien zwischen Werttypen und Objekttypen: Werte sind unveränderlich und existieren per se, Objekte können verändert, erzeugt und vernichtet werden. Wertoperationen sind seiteneffektfrei und referentiell transparent, Objektoperationen können Seiteneffekte bewirken und referentiell opak sein. Abbildung 3.1 stellt die charakteristischen Merkmale von Werttypen und Objekttypen einander gegenüber.

Werte:	Objekte:
- unveränderlich	- änderbar
- unerzeugbar	- erzeugen/ vernichten
Wert-Operationen:	Objekt-Operationen:
- seiteneffektfrei	- Seiteneffekte
- referentiell transparent	- referentiell opak

Zustandslos

Zustand

Abbildung 3.1: Eigenschaften von Werttypen und Objekttypen

Die Wert-Eigenschaften beschreiben Einschränkungen, d. h. Verhaltensweisen, die für Werttypen *nicht* zulässig sind. Es handelt sich bei ihnen um „Muss-Bedingungen“, für einen Werttyp müssen *alle* aufgeführten Bedingungen erfüllt sein. Objekttypen unterliegen dagegen „Kann-Bedingungen“. Objekte können veränderlich sein, prinzipiell gibt es aber auch unveränderliche Objekte. Objektoperationen können Seiteneffekte hervorrufen, müssen das aber nicht, etc.⁴

⁴Das von Meyer propagierte Prinzip der Trennung von Abfragen und Anweisungen („Command-Query-Separation“ [Mey97, p. 751]) fordert sogar ausdrücklich, dass eine Operation nicht gleichzeitig Seiteneffekte bewirkt *und* referentiell opak ist. Abfragen (Queries) geben Auskunft über den aktuellen Zustand, sie sind frei von Seiteneffekten, aber i. A. referentiell opak. Anweisungen (Commands) haben Seiteneffekte, liefern aber kein Ergebnis und sind damit referentiell transparent.

Die meisten Objektoperationen sind entweder referentiell opak oder bewirken Seiteneffekte, sie können im Prinzip – bei Verstoß gegen die Command-Query-Separation – auch beides sein. Da es sich bei den Charakteristika von Objekttypen um Kann-Bedingungen handelt, sind aber auch Operationen, die beides sind, seiteneffektfrei *und* referentiell transparent, nicht ausgeschlossen. Ein Beispiel dafür ist eine lesende Operation eines eindeutigen, identifizierenden Merkmals eines Objektes, z. B. einer Kundennummer, Artikelnummer oder Versicherungsnummer.

3.3.5 Der Wertbegriff von MacLennan

Der Wertbegriff dieser Arbeit stimmt weitgehend überein mit dem von MacLennan [Mac82]. MacLennan beschreibt Werte als zeitlos, abstrakt und unveränderlich. Er weist darauf hin, dass sie nicht erzeugt, nicht zerstört und nicht kopiert werden können und stellt fest, dass eine gemeinsame Nutzung von Werten keinen Sinn ergibt. Nach MacLennan ist wert-orientierte Programmierung durch referentielle Transparenz und Abwesenheit von Seiteneffekten gekennzeichnet.

Werte gemäß MacLennan sind Bestandteil des Werkzeug- und Material-Ansatzes von Züllighoven [Zül98; Zül05], sie werden dort als „Fachwerte“ bezeichnet. Ihre Umsetzung in einem objektorientierten Rahmenwerk wird ausführlich von Bäumer et al. [Bäu+98], sowie von Bleek, Lilienthal & Züllighoven [BLZ99] diskutiert.

MacLennan und die eben genannten, darauf aufbauenden Arbeiten unterscheiden nicht, welche der aufgeführten Eigenschaften Werte definieren („was soll unter Werten verstanden werden?“), und welche Aussagen über Werte darstellen („welche weiteren Eigenschaften ergeben sich daraus?“).

Die vorliegende Arbeit definiert Werte anhand weniger, klar entscheidbarer Kriterien. Die obige Definition führt einige der von MacLennan genannten Kriterien nicht auf, weil die vier o. g. Eigenschaften sie bereits implizieren:

Zeitlosigkeit ergibt sich aus Unveränderlichkeit und Unerzeugbarkeit. Verändern, Erzeugen und Vernichten sind zeitgebundene Vorgänge, sie setzen das Vorhandensein eines Vorher und eines Nachher voraus. Umgekehrt ist der Begriff der Veränderung implizit im Begriff der Zeit enthalten: Zeit ist nur anhand von Veränderungen wahrnehmbar. Zeitlosigkeit und Zustandslosigkeit sind de facto Synonyme.

Auch dass für Werte *kein Kopieren* möglich ist, ergibt sich unmittelbar aus der Definition. Kopieren beinhaltet implizit das Erzeugen eines neuen Elementes, dessen Eigenschaften mit denen der Vorlage übereinstimmen. Das ist für Werte aufgrund ihrer Unerzeugbarkeit nicht möglich.

Eine *gemeinsame Nutzung* („sharing“) gibt es zwar auch bei Werten, sie ist aber von anderer Qualität als bei Objekten. Z. B. können zwei Konten den gleichen Saldo aufweisen. Sie „nutzen“ damit ein- und dasselbe Element eines Werttyps „Geldbetrag“. Aufgrund der Unveränderlichkeit von Werten ermöglicht die gemeinsame Nutzung aber, anders als bei Objekten, keine Kommunikation zwischen den beiden Nutzern.

3.3.6 Wertbegriffe anderer Arbeiten

Viele andere Arbeiten, die sich mit Werten befassen – s. Abschnitt 3.1 – charakterisieren Werte mit Hilfe von Merkmalen wie „klein“ (Fowler), „deskriptiv“ (Evans), „einfach“ (Poetzsch-Heffter), „keine Identität“ (van der Werf, Lahres & Rayman) oder „keine konzeptuelle Identität“ (Evans).

Solche Umschreibungen sind schwer abgrenzbar, oder es handelt sich um akzidentelle Eigenschaften, die nur auf einige, aber nicht auf alle Werttypen zutreffen. Sie sind aus diesen Gründen nicht in den Wertbegriff dieser Arbeit eingeflossen.

Beispielsweise unterliegt es subjektiver Beurteilung, ob Elemente einfach oder komplex, klein oder groß sind; es gibt hier fließende Übergänge. Auch ist die Größe eines Elementes eine Eigenschaft der gewählten Implementation, nicht eines konzeptionellen Wertes. Zeichenketten („Strings“) sind Beispiele für Werte, die sowohl leicht- als auch schwergewichtige Elemente umfassen und damit die Größe als Kriterium für Werte in Frage stellen.

„Fehlende Identität“, häufig als Kriterium für Werte angeführt, ist schwer fassbar und kann missverstanden werden. Da Werte durchaus voneinander unterschieden werden – 4 ist ein anderer Wert als 5 – kann man in diesem Sinne auch bei Werten von Identität sprechen. Soweit dagegen gemeint ist, dass Werte keine von ihrem (unveränderlichen) Zustand unabhängige Identität besitzen – es gibt z. B. keine zwei verschiedenen komplexen Zahlen mit dem Realteil 2 und dem Imaginärteil 3 – ist die fehlende Identität bereits in der Unveränderlichkeit und Unerzeugbarkeit von Werten enthalten.⁵

3.3.7 Zustandslosigkeit und Unveränderlichkeit

Die in Abschnitt 3.1 genannten Arbeiten fokussieren ausschließlich auf die Unveränderlichkeit von Werten. Ihr Wertbegriff berücksichtigt nicht das in dieser Arbeit als essentiell angesehene Kriterium der Unerzeugbarkeit. Einige Autoren sprechen sogar ausdrücklich von der Erzeugung von Werten. Sie bezeichnen sie als „Wertobjekte“ und suggerieren dadurch, dass es sich um spezielle Objekte handelt (z. B. van der Werf, Fowler, Evans, Henney, Hoogendorn, Poetzsch-Heffter sowie Lahres & Rayman).

Die durch die obige Definition beschriebene Zustandslosigkeit ist mehr als nur Unveränderlichkeit. Werte sind etwas anderes als unveränderliche Objekte. Z. B. sind Buchungssätze in einem revisionssicheren System gemäß Anforderung unveränderlich. Sie sind aber nicht zustandslos. Sie werden erzeugt, damit handelt es sich um Objekte. Die Erzeugung eines solchen Buchungssatzes verändert die Gesamtmenge

⁵Abschnitt 3.3.10 und Kapitel 8 befassen sich ausführlicher mit Gleichheit, Identität und ihrem Zusammenhang zur Wert-/Objekt-Trennung.

der vorhandenen Buchungen. Sie kann sich auf das Ergebnis von Operationen (z. B. Kontensalden) auswirken und ist damit eine Zustandsänderung.

Wertoperationen (Funktionen) sind zu unterscheiden von seiteneffektfreien, aber zustandsabhängigen Objektoperationen, die hier als Queries bezeichnet werden. Beispielsweise bewirkt der Aufruf einer Operation wie „Summe der Monatsgehälter der Mitarbeiter der Marketingabteilung“ keine Seiteneffekte und ist damit eine Query. Dennoch ist diese Operation zustandsabhängig. Sie macht eine Aussage über den aktuellen Zustand der Welt (hier: der Marketingabteilung), und das Ergebnis kann zu verschiedenen Zeitpunkten unterschiedlich ausfallen, abhängig davon, ob Mitarbeiter kündigen, neu eingestellt werden oder Gehaltserhöhungen erhalten. Eine Funktion, z. B. „Anzahl der Tage zwischen dem 4.2.2016 und dem 13.3.2017“ ist dagegen zeit- und zustandslos. Sie liefert bei gleichen Eingaben immer das gleiche Ergebnis, unabhängig von einem Zustand, und ist damit eine Funktion im Sinne der funktionalen Programmierung.

Das Gegenstück zu Queries sind Objektoperationen mit Seiteneffekten. Sie werden als Mutator oder Modifier bezeichnet. Mutator umfassen nicht nur seiteneffekt-behaftete Operationen ohne Ergebnis (d. h. Commands im Sinne der Command-Query-Separation, s. Abschnitt 3.3.4). Sie können auch ein Ergebnis liefern, referenziell opak sein *und* Seiteneffekte bewirken – solche Fälle passen nicht in das Schema der Command-Query-Separation.

Abbildung 3.2 illustriert die eben beschriebene Systematik von Wert- und Objektoperationen.

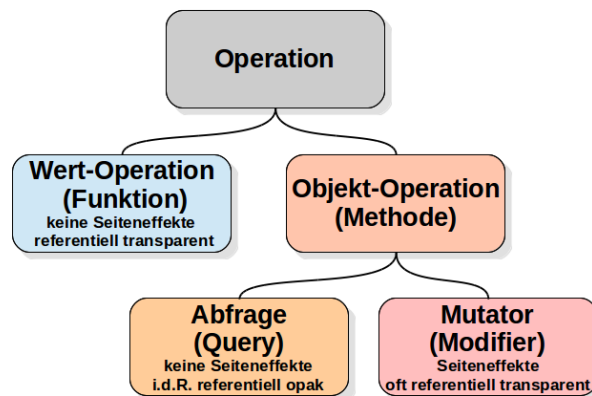


Abbildung 3.2: Kategorien von Operationen

3.3.8 Elemente und Variablen

Die *Elemente* eines Werttyps sind zu unterscheiden von *Variablen*, deren Typ ein Werttyp ist. Darum ist es auch kein Widerspruch zur Unveränderlichkeit von Werten, dass Variablen von Werttypen änderbar sind. Bereits MacLennan weist darauf hin, dass die Änderung einer Variablen nicht mit der Änderung eines Wertes verwechselt werden darf:

„We can give names to values, and we can change the names that we give to values, but this doesn’t change the values.“[Mac82, p. 71].

Eine Variable x wird geändert durch eine Zuweisung⁶, z. B. $x := e$. Die Änderung eines – über die Variable x angesprochenen – Elementes geschieht dagegen durch Aufruf einer ändernden Operation, z. B. $x.\text{modify}(\dots)$ – und ist bei Werttypen wegen des Fehlens ändernder Operationen nicht möglich.

Die Möglichkeit, eine Variable x per Zuweisung zu verändern, und die Möglichkeit, das Element zu verändern, sind orthogonal zueinander. Umgangssprachlich werden beide Vorgänge häufig als „Änderung von x “ bezeichnet, was die Gefahr birgt, sie miteinander zu vermengen.

Isoliert betrachtet hat die Änderung einer Variablen den gleichen Effekt wie die Änderung des Elementes. Der Unterschied zwischen beiden Vorgängen kommt zum Tragen, wenn das Element in mehreren Kontexten verwendet wird, z. B. weil es als Übergabeparameter an eine andere Operation weitergereicht wurde. Die Zuweisung wirkt sich nur auf die Variable und damit auf genau einen Kontext aus. Die Änderung des Elementes würde sich in allen Kontexten auswirken, die dieses Element benutzen.

3.3.9 Zusammenhänge und Abhängigkeiten

Referentielle Transparenz und Seiteneffektfreiheit sind *unabhängig* voneinander. Keine dieser beiden Eigenschaften ist für die andere notwendig oder hinreichend. Louden führt ein Beispiel für eine referentiell transparente Operation mit Seiteneffekten auf und eines für eine referentiell opake Operation ohne Seiteneffekte [Lou93, p. 607].

Referentielle Transparenz und Seiteneffektfreiheit können als *komplementär* zueinander angesehen werden. Eine Operation ohne Seiteneffekte *bewirkt* keine Zustandsänderungen. Eine referentiell transparente Operation macht keine Zustandsänderungen *sichtbar*. Beide Bedingungen zusammen beschreiben Zustandslosigkeit.

In einem Umfeld, das Zustand und Zustandsänderungen grundsätzlich zulässt, sind zur Charakterisierung von Zustandslosigkeit beide Bedingungen erforderlich.

⁶In diesem Zusammenhang zählen auch syntaktische Sonderformen als Zuweisung, z. B. der in Java oder C++ gebräuchliche Operator „++“, da eine Anweisung wie „i++“ eine Abkürzung für die Zuweisung $i=i+1$ darstellt.

In einem rein funktionalen Umfeld dagegen würde (globale) Seiteneffektfreiheit referentielle Transparenz implizieren: wenn es keine Operation gibt, die Seiteneffekte bewirkt, kann keine andere Operation sie sichtbar machen.

Seiteneffektfreiheit und referentielle Transparenz sind stärkere Bedingungen als Unveränderlichkeit und Unerzeugbarkeit:

Wenn eine Operation keine Zustandsänderung bewirkt, dann verändert sie insbesondere nicht ihre Eingabeparameter. Seiteneffektfreiheit impliziert also Unveränderlichkeit.

Dass Werte nicht erzeugt werden können, kann aus der referentiellen Transparenz hergeleitet werden. Alle Operationen eines Werttyps sind Funktionen und damit referentiell transparent, das gilt auch für solche Funktionen, deren Ergebnis ein Wert des eigenen Typs ist. Wird eine derartige Funktion mehrfach mit den gleichen Parametern aufgerufen, liefert sie definitionsgemäß immer den gleichen Wert (anders als z. B. Konstruktoraufrufe bei Objekttypen). Dieser Wert erscheint deshalb aus Sicht von Klienten so, als sei er „schon immer da“.

Aufgrund dieser Implikationen wäre es möglich gewesen, die obige Definition noch stärker zu reduzieren und auf die beiden Eigenschaften „Unveränderlichkeit“ und „Unerzeugbarkeit“ zu verzichten. Sie sind aus Gründen der Anschaulichkeit in den Werttypbegriff mit aufgenommen worden und um Unterschiede zu vorhergehenden Arbeiten aufzuzeigen.

Abbildung 3.3 zeigt die Zusammenhänge und Abhängigkeiten zwischen den vier Eigenschaften von Werttypen.

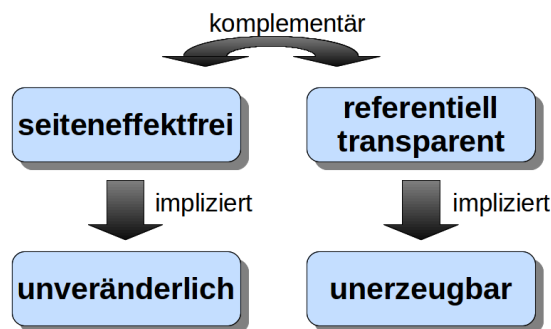


Abbildung 3.3: Abhängigkeiten zwischen den Werteigenschaften

3.3.10 Werte und Gleichheit

Die in der Definition postulierte referentielle Transparenz beinhaltet implizit, dass jeder Werttyp über genau eine Gleichheit verfügt: „gleiche Eingaben“ und „das gleiche Ergebnis“ (s. o.) setzen voraus, dass ein Vergleich zweier Wertausdrücke verfügbar ist.

Das Konzept der Unerzeugbarkeit hängt ebenfalls eng mit Gleichheit zusammen. Ein Wert wirkt für Klienten so, als sei er „schon immer vorhanden“, *weil*, wie weiter oben ausgeführt, mehrere Aufrufe einer Funktion mit den gleichen Parametern immer den gleichen Wert liefern. Diese Funktionsweise der Gleichheit ist das entscheidende Kriterium für die Nicht-Erzeugung. Das Vorhandensein oder das Fehlen eines Schlüsselwortes wie „new“ oder „create“ kann den Erzeugungsvorgang bzw. sein Fehlen lediglich dokumentierend unterstreichen.

3.3.11 Zustandslosigkeit als Eigenschaft von Typen

Der in dieser Arbeit gewählte Ansatz koppelt Zustandslosigkeit an Werttypen und Zustand an Objekttypen. Er trennt damit die funktionale und die imperative Welt auf *Typ-Ebene* und unterscheidet sich in dieser Hinsicht von vielen anderen Arbeiten.

Etliche Sprachen nehmen keine Differenzierung vor. Sie gehen prinzipiell überall von änderbarem Zustand aus (imperative/ objektorientierte Sprachen) oder ausschließlich von Zustandslosigkeit (rein funktionale Sprachen).

Andere Sprachen kennen Differenzierungen, setzen aber auf anderen Ebenen an. Z. B. trennt Scala auf der Ebene von Variablen, sie werden mit `val` als unveränderlich und mit `var` als änderbar deklariert. Ufo unterscheidet auf der Ebene der Operationen, nämlich `functions` (ohne Seiteneffekte) und `procedures` (mit Seiteneffekten). Ähnliche Ansätze gibt es in Fortran und in D, dort ist es möglich, eine Operation als „pure function“ zu definieren.

Wie in Abschnitt 3.3.7 und 3.3.8 ausgeführt, kann dadurch Zustandslosigkeit nicht adäquat erfasst werden. Jede einzelne der Bedingungen – Unveränderlichkeit, Seiteneffektfreiheit, etc. – erfasst jeweils nur einen Teilaspekt von Zustandslosigkeit. Eine einzelne Variable kann unveränderlich sein, aber nicht zustandslos. Eine einzelne Operation kann seiteneffektfrei sein, oder referentiell transparent, trotzdem ist sie dadurch nicht zustandslos. Die Ebene der Variablen oder der Operationen ist zu feinkörnig, um Zustandslosigkeit zu erfassen. Zustandslosigkeit (und ihr Gegenstück, Zustand) bezieht sich auf einen Typ; sie beinhaltet eine Aussage über *alle* zu diesem Typ gehörenden Operationen.

Typen erfüllen in Programmiersprachen mehrere Aufgaben: U. a. bilden sie ein Mittel zur Abstraktion, sie gruppieren gleichartige Elemente, sie ermöglichen Korrektheitsprüfungen, und sie dokumentieren den Quelltext (s. dazu [Weg90; Pie02]). Mit der Trennung von Werttypen und Objekttypen kommt ihnen eine weitere Aufgabe zu: Typen kennzeichnen damit das *Verhalten* ihrer zugehörigen Elemente, entweder als zustandslos oder als zustandsgebunden.

3.4 Beispiele für Werttypen

Mehrere Beispiele veranschaulichen den Wertbegriff. Sie zeigen, dass Typen, deren Elemente intuitiv als Werte eingestuft werden, auch Werttypen im Sinne der obigen Definition sind. Sie verdeutlichen, dass Werte und Objekte Gegensätze bilden, zwischen denen es keine Grauzone gibt, dass es aber von den Anforderungen abhängen kann, ob ein Gegenstand als Wert oder als Objekt abgebildet wird.

3.4.1 Primitive Datentypen

Die Eigenschaften von Zahlentypen dienten als Vorlage bei der Bildung des Wertbegriffs. Weil die meisten primitiven Datentypen Zahlentypen sind, (`int`, `long`, `float`, etc.), handelt es sich bei ihnen um Werttypen. Auch andere primitive Typen wie bool'sche Werte (`boolean`) und Zeichentypen (`char`) sind Werttypen, wie durch Überprüfen der Werteigenschaften nachgewiesen werden kann:

Die Werte von `boolean` werden nicht erzeugt, es gibt nur die beiden Elemente `true` und `false`. Ein Literal wie `true` bezeichnet einen – bereits existierenden – Wert. Durch mehrfaches Verwenden des Literals entstehen nicht mehrere Kopien des Wertes, alle Vorkommen des Literals bezeichnen den gleichen Wert. Die bool'schen Operationen (logische Konjunktion, Disjunktion, etc.) – die syntaktisch in vielen Sprachen über Infix- oder Prefix-Operatoren aufgerufen werden – ändern nicht die Werte, auf die sie angewendet werden, sondern sie liefern einen Ergebniswert. Sie sind frei von Seiteneffekten und liefern für gleiche Eingaben immer das gleiche Ergebnis.

Das kann analog für die anderen primitiven Typen, ihre Literale und ihre Operationen nachgewiesen werden.

Primitive Datentypen sind in etlichen Sprachen (wie Java, C#, etc.) in vielfältiger Hinsicht anders als Objekttypen. Sie verfügen über eine andere Syntax, sie sind nicht in die Klassenhierarchie eingebunden, etc. Oft werden sie als notwendiges Übel betrachtet, das im Sprachentwurf aus Performance-Gründen erforderlich ist (s. dazu [KR99]), aber nicht in das Gesamtkonzept passt und dem rein objektorientierten Ansatz zuwider läuft. Mit der hier vorgeschlagenen Zweiteilung des Typsystems relativiert sich die Sonderstellung primitiver Datentypen: sie bilden aus diesem Blickwinkel keine Ausnahmen im Typsystem, sondern sie sind lediglich durch die Sprache vordefinierte Werttypen.

3.4.2 Zeichenketten

Zeichenketten („Strings“) weisen alle konzeptionellen Eigenschaften von Werttypen auf. Ihre Operationen (verketteten, extrahieren von Bestandteilen, etc.) ändern nicht die Elemente, auf die sie angewendet werden, sie bewirken auch keine anderen Seiteneffekte, und sie liefern bei gleichen Eingaben immer das gleiche Ergebnis. Zeichenketten verfügen in vielen Programmiersprachen über Literale, was bereits auf ihre Ähnlichkeit mit primitiven Datentypen hinweist. Die Literale ermöglichen Zugriff auf die jeweilige Zeichenkette, ohne einen Erzeugungsvorgang zu involvieren.

Zeichenketten können beliebig lang sein. Sie liefern damit ein Beispiel für einen Werttyp, der auch schwergewichtige Element umfasst (s. auch Abschnitt 3.3).

3.4.3 Wochentag

Wochentag – der Typ, der die sieben Tage der Woche von Montag bis Sonntag beschreibt – ist ein Werttyp. Die Elemente dieses Typs werden in seiner Definition explizit aufgezählt, wodurch die Wertemenge feststeht und die Unerzeugbarkeit erfüllt ist. Der Typ verfügt über nur wenige Operationen, wie Gleichheit und ggf. eine Nachfolger-Funktion. Weil diese Operationen die Elemente des Typs unverändert lassen, referentiell transparent sind und keine Seiteneffekte bewirken, sind die vier Kriterien für Werttypen erfüllt.

Das lässt sich auf gleiche Weise für viele weitere Aufzählungstypen zeigen, z. B. Himmelsrichtung, Spielkartenfarbe oder Wertpapierrating.⁷

3.4.4 Scheinbare Grenzfälle: Sammlungen

Es gibt Abstraktionen, die scheinbar auf der Grenze zwischen Werten und Objekten liegen und Eigenschaften von beiden aufweisen. Dazu zählen Sammlungen („Collections“): Mengen, Listen, Bäume, Dictionaries, etc.

Mengen werden beispielsweise anhand der darin enthaltenen Elemente verglichen, in dieser Hinsicht verhalten sie sich wie Werte. Andererseits können Mengen verändert werden, z. B. durch Hinzufügen oder Entfernen von Elementen, was dafür spricht, dass es sich um Objekte handelt.

Der Eindruck, dass sich Mengen damit in einer Grauzone zwischen Werten und Objekten bewegen würden, ist aber falsch und beruht auf einer sprachlichen Mehrdeutigkeit. Kent zeigt, dass in solchen Fällen dasselbe Wort (hier: „Menge“) für zwei verschiedene Dinge verwendet wird, nämlich für „extensionale“ und „intensionale

⁷Nicht alle Aufzählungen sind notwendigerweise Werttypen. Anhang A.2 führt ein Beispiel für einen mittels Aufzählung definierten Objekttyp auf. Kapitel 6 geht näher auf Aufzählungen und ihre mögliche sprachliche Unterstützung ein.

Mengen“ [Ken91]. Intensionale Mengen (wie „die Namen der Mitarbeiter der Marketingabteilung“) und extensionale Mengen (wie {"Alina", "Britta", "Charlie"}) sind zwei verschiedene Konzepte. Sie können miteinander zusammenhängen, z. B. besteht zu einem festen Zeitpunkt der aktuelle Zustand einer intensionalen Menge aus einer extensionalen Menge – Kent spricht in dem Zusammenhang von „Carrier“ und „Cargo“. In der Begriffswelt des hier zugrunde gelegten Sprachmodells sind extensionale Mengen Werte und intensionale Mengen Objekte. Das zweigeteilte Typsystem trägt dazu bei, sie gedanklich klar voneinander zu trennen.

Eine Grauzone zwischen Werten und Objekten kann es nicht geben, denn die charakteristischen Merkmale von Werttypen und von Objekttypen sind einander logisch entgegengesetzt. Wie das Beispiel zeigt, kann es eine Frage der Sichtweise oder der Anforderungen sein, ob eine Klasse von Gegenständen als Werttyp oder als Objekttyp abgebildet wird. Das gilt nicht nur für Sammlungen, sondern auch für andere Abstraktionen. Z. B. weist Evans darauf hin, dass eine Adresse als Wert aufgefasst werden kann (als unveränderliche Struktur, bestehend aus Straße, Hausnummer, Postleitzahl, Ort, etc.) oder als Objekt (das z. B. bei der Umbenennung einer Straße oder der Umstellung der Postleitzahlen angepasst wird, aber trotzdem dieselbe Adresse bleibt)[Eva04, p. 98].

3.5 Asymmetrische Abhängigkeit zwischen Werttypen und Objekttypen

Aus der obigen Definition lässt sich eine universelle Gesetzmäßigkeit ableiten:

Werttypen haben keine Typabhängigkeiten zu Objekttypen.

D. h. in der Schnittstelle von Werttypen kann es keine Objekttypen geben, Funktionen haben als Eingaben und Ausgaben nur Werte.

Das kann folgendermaßen begründet werden:

Wären für eine Funktion Objekte als Parameter zulässig, so könnte sie deren Operationen aufrufen. Weil Objektoperationen aber i. A. Seiteneffekte hervorrufen oder referentiell opak sind, könnten sich diese Eigenschaft auf die Funktion übertragen, was im Widerspruch zur Definition steht.

Wären Objekte als Ausgabe einer Funktion zulässig, so müsste – weil Objekte als Eingabeparameter bereits ausgeschlossen wurden – das jeweilige Rückgabe-Objekt von der Funktion erzeugt werden. Dann aber würde jeder Aufruf der Funktion ein neues und damit anderes Objekt liefern, auch bei gleichen Eingabeparametern, im Widerspruch zur referentiellen Transparenz der Funktion.

In umgekehrter Richtung kann es Abhängigkeiten geben: Objekttypen können Werttypen benutzen. Die Eigenschaften von Objekten werden häufig durch Werttypen beschrieben (Länge, Breite, Farbe, Preis, ...). Viele Objektoperationen haben Werttypen als Eingabeparameter oder als Ergebnis (Kurs eines Wertpapiers an einem bestimmten Datum, Alter einer Person, etc.)

Daraus ergibt sich:

Zwischen Werten und Objekten besteht ein asymmetrisches Abhängigkeitsverhältnis: Objekte können Werte benutzen, Werte aber nicht Objekte.

Die Werttypen eines Systems bilden ein geschlossenes Teilsystem. Es wird in dieser Arbeit als „funktionaler Kern“ bezeichnet. Im funktionalen Kern gibt es nur Werte und Funktionen. Er ist geprägt von Zustandslosigkeit und hat keine Verbindung zu Objekten.

Die Menge aller Objekttypen wird als „objektorientierte Schale“ bezeichnet. Sie umschließt den funktionalen Kern, Objekte benutzen und beziehen sich auf Werte. Die objektorientierte Schale ist geprägt von Seiteneffekten, Zeitbezogenheit und fortlaufenden Zustandsänderungen.

Abbildung 3.4 veranschaulicht das asymmetrische Abhängigkeitsverhältnis.

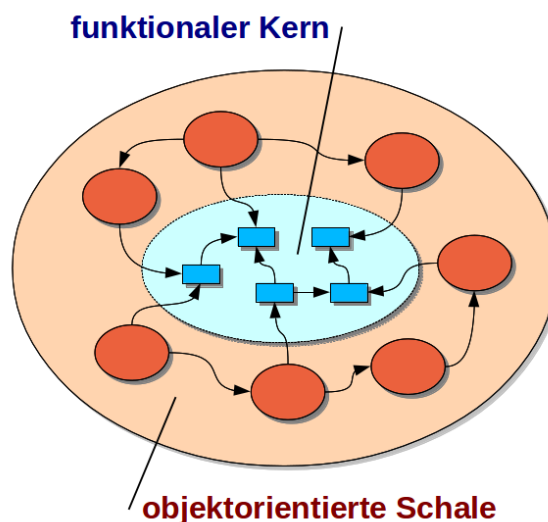


Abbildung 3.4: Asymmetrische Abhängigkeit zwischen Werttypen und Objekttypen

3.6 Zusammenfassung

Kapitel 3 legte fest, was in der vorliegenden Arbeit unter Werten und Werttypen verstanden werden soll und legte damit das Fundament der Arbeit.

Unter einem Werttyp wird ein Typ verstanden, dessen Elemente unveränderlich sind, nicht erzeugt werden und dessen Operationen seiteneffektfrei und referentiell transparent sind. Die genannten Eigenschaften beschreiben eine konzeptionelle Sichtweise, d. h. das von Klienten beobachtbare Verhalten, unabhängig von einer technischen Implementation.

Die genannten Eigenschaften grenzen Werttypen von Objekttypen ab. Die Elemente von Objekttypen können verändert sowie erzeugt und vernichtet werden, und ihre Operationen können Seiteneffekte bewirken oder referentiell opak sein.

Der Wertbegriff dieser Arbeit stimmt mit dem Wertbegriff von MacLennan überein. Die in dieser Arbeit gegebene Definition verzichtet jedoch bewusst auf einige von MacLennan als kennzeichnend für Werte genannte Eigenschaften, weil sie diese bereits impliziert.

Viele Arbeiten zu Werten fokussieren ausschließlich auf deren Unveränderlichkeit und führen Unerzeugbarkeit nicht als wesentliches Merkmal von Werten auf. Die durch die Definition beschriebene Zustandslosigkeit ist mehr als nur Unveränderlichkeit. Kapitel 3 hat die Unterschiede anhand mehrerer Beispiele verdeutlicht.

Seiteneffektfreiheit und referentielle Transparenz bilden zwei voneinander unabhängige, komplementäre Facetten von Zustandslosigkeit. Sie implizieren Unveränderlichkeit und Unerzeugbarkeit. Jeder Werttyp verfügt über eine Gleichheit, denn referentielle Transparenz setzt implizit das Vorhandensein von Gleichheit voraus. Auch Unerzeugbarkeit ist eng an die Funktionsweise – und damit das Vorhandensein – von Gleichheit gekoppelt. Der in dieser Arbeit gewählte Ansatz trennt Zustandslosigkeit und Zustand auf Typebene und unterscheidet sich in dieser Hinsicht von vielen anderen Arbeiten.

Zur Veranschaulichung des Wertbegriffs wurden verschiedene Beispiele für Werttypen aufgeführt. Sie haben gezeigt, dass viele Typen, die intuitiv als Werttypen eingestuft werden, auch Werttypen im Sinne der Definition sind. Weitere Beispiele machten deutlich, dass Werttypen und Objekttypen klar definierte Gegensätze bilden und es zwischen ihnen keine Grauzone gibt. Es kann aber von den Anforderungen abhängen, ob ein Gegenstand als Wert oder als Objekt abzubilden ist.

Aus der Definition wurde abgeleitet, dass Objekte Werte benutzen können, aber nicht umgekehrt. Werttypen und Objekttypen stehen damit in einem asymmetrischen Abhängigkeitsverhältnis. Es ergibt sich ein Sprachmodell, das jedes System in einen zustandslosen Bereich (den „funktionalen Kern“) und einen zustandsbezogenen Bereich (die „objektorientierte Schale“) unterteilt.

Teil II

Sprachunterstützung für Werttypen

Kapitel 4

Softwaretechnische Verfahren zur Abbildung von Werttypen

Es gibt verschiedene Verfahren, um Werttypen in Programmcode umzusetzen. Dieses Kapitel untersucht eine Reihe von softwaretechnischen Verfahren, die in objektorientierten Sprachen verbreitet eingesetzt werden, um Werttypen und wertartiges Verhalten abzubilden. Es überprüft, ob und wie diese Verfahren die Werteigenschaften herstellen und es analysiert ihre softwaretechnische Qualität.

Die untersuchten Verfahren umfassen Entwurfsmuster¹ (diese sind unabhängig von der Programmiersprache), Programmiermuster² (auch als Idiome bezeichnet, in der Regel auf eine spezifische Sprache zugeschnitten) und Programmierkonventionen (Übereinkünfte, Code nur in einer speziellen Weise zu programmieren oder aufzurufen). Soweit im Folgenden nicht relevant ist, ob es sich um ein Entwurfsmuster, ein Idiom oder eine Programmierkonvention handelt, wird von „softwaretechnischen Verfahren“, oder kurz, von „Verfahren“, die Rede sein.

Der gemeinsamer Nenner aller softwaretechnischen Verfahren besteht darin, dass sie die Werteigenschaften durch eine Kombination geeigneter Sprachmittel oder durch selbst-auferlegte Beschränkungen herstellen. Dadurch ist der Anwendungsprogrammierer für das wert-gemäße Verhalten zuständig.

¹Zu einem Muster gehören im allgemeinen ein Kontext, eine wiederkehrende Problemstellung und ein allgemeingültiger, wiederverwendbarer Lösungsansatz (s. Alexander [Ale79]). Im Hinblick darauf, dass etliche der betrachteten Muster für das Abbilden von Werten geeignet, aber nicht ausdrücklich für diesen Zweck zugeschnitten sind, wird sich dieses Kapitel vorrangig mit den Lösungsansätzen befassen. Die ursprüngliche Problemstellung und der Kontext der jeweiligen Muster rücken dabei in den Hintergrund.

²Zur Klassifizierung der Muster siehe auch Buschmann et al. [Bus+96]. Andere Arten von Mustern, z. B. Analyse- oder Architekturmuster, spielen in Zusammenhang mit Werten keine Rolle.

Die hier untersuchten Verfahren lassen sich in zwei Gruppen einteilen:

Wertartige Klassen sind Entwurfs- oder Programmiermuster, bei denen die Werteigenschaften ganz oder überwiegend in der Klasse umgesetzt sind, die den Werttyp abbildet.

Klientenseitige Programmierkonventionen legen eine spezifische Art und Weise fest, wie Klienten eine Klasse benutzen dürfen. In der Klasse, die den Werttyp abbildet, werden keine oder nur wenige Vorkehrungen für ein wertgemäßes Verhalten getroffen.

4.1 Untersuchungskriterien

Zur Beurteilung, wie gut sich die nachfolgend beschriebenen Verfahren zur Abbildung von Werttypen eignen, wird evaluiert, ob und welche Werteigenschaften sie umsetzen:

Werteigenschaften: Wie viele und welche Werteigenschaften setzt das Verfahren um? Wird das Ziel – ein Verhalten entsprechendes Verhalten – erreicht? In welchem Umfang ist dafür die Klasse zuständig, welche den Werttyp verkörpert, in welchem der Klient, der die Klasse benutzt?

Es wird die softwaretechnische Qualität der Verfahren beurteilt:

Einfachheit: Wie aufwändig ist es, das Verfahren anzuwenden? Wie viele und welche Sprachmittel sind beteiligt? Wie umfangreich ist der Code, der dadurch entsteht?

Klarheit: Ist der Code lesbar? Sind Werttypen als solche erkennbar?

Sicherheit: Wie stabil oder fehleranfällig ist das Verfahren? Besteht die Gefahr, die Klasse in einer nicht intendierten Weise zu benutzen? Ist es leicht möglich, die Werteigenschaften bei Codeänderungen zu zerstören?

Weil einige Verfahren ausdrücklich der Effizienzsteigerung dienen, werden in den Vergleich auch Auswirkungen auf technische Eigenschaften des Codes einbezogen:

Effizienz: Wie wirkt sich das Verfahren auf den Speicherverbrauch aus? Wie auf das Laufzeitverhalten?

Welche Werteigenschaften erfüllt sind, und ob sie in der Zuständigkeit der jeweiligen Klasse oder ihrer Klienten liegen, ist eindeutig zu beurteilen. Die anderen Kriterien sind weniger leicht quantifizierbar. Die softwaretechnische Qualität der Verfahren und ihre Effizienz werden hier in Relation zu denen der anderen Verfahren eingeordnet.

4.2 Wertartige Klassen

Die in diesem Abschnitt untersuchten Verfahren setzen die Werteigenschaften ganz oder überwiegend in der Klasse um, welche den Werttyp darstellt, und damit an zentraler Stelle. Dazu zählen die Muster *Value Object*, *Fliegengewicht* und das *Typesafe Enum Pattern*.

4.2.1 Value Object

Riehle empfiehlt, Werte mit Hilfe einer Klasse zu implementieren, deren Methoden den Zustand ihrer Elemente nicht verändern. Er bezeichnet diesen Ansatz als „Value Object Pattern“ [Rie06]. Kühne beschreibt dieses Verfahren unter dem Namen „Value Object“ [Küh99]. Goetz bezeichnet es als „Immutable Object“ [Goe03], Bloch als „Immutable Class“ [Blo08].

Gemäß der in Abschnitt 3.3.7 getroffenen Einteilung verfügt eine als Value Object programmierte Klasse ausschließlich über Queries und Konstruktoren, sie enthält keine Mutator-Methoden. Eine Methode, die bei einer änderbaren Klasse als Mutator implementiert wäre, stellt sich bei ihr als eine Methode dar, die als Ergebnis ein anderes, die „Veränderung“ widerspiegelndes Element dieser Klasse liefert. Kühne bezeichnet eine derartige Methode als „generator operation“ [Küh99, p. 154].

Die Java-Klasse `java.lang.BigInteger` ist ein Beispiel für eine als Value Object implementierte Klasse. Z. B. dient ihre Methode `negate` dazu, das Vorzeichen der Zahl umzukehren. Sie lässt das Element, auf dem sie operiert, unverändert, und liefert statt dessen eine Zahl mit gleichem Betrag und umgekehrtem Vorzeichen. `negate` hat damit die Signatur

```
BigInteger negate() {...}
```

Das gilt entsprechend für alle Methoden von `BigInteger`. Für ein Value Object wäre es nicht zulässig, das Element selbst zu verändern, z. B. im Fall von `negate` durch Umkehrung des Vorzeichens, wodurch die Methode den Rückgabotyp `void` erhielte:

```
void negate() {...} // nicht zulässig bei value objects!
```

Weil eine Value Object-Klasse nur Queries anbietet, ähnelt ihre Benutzung der von Basisdatentypen. Z. B. muss jede Methode einer solchen Klasse in Zusammenhang mit einer Zuweisung aufgerufen werden, also

```
myBigInteger = myBigInteger.negate();
```

Weil mehrere Elemente einer Value Object-Klasse denselben Wert repräsentieren können, ist die Wertgleichheit nicht durch die von der Sprache vorgegebene Objektidentität überprüfbar. Der Entwickler der Value Object-Klasse muss die Wertgleichheit programmieren, auf Basis der unveränderlichen Membervariablen. Diese Wertgleichheit ist häufig in einer Methode implementiert, in Java z. B. wird für diesen Zweck die Methode `equals` überschrieben. In Sprachen mit Unterstützung für „operator overloading“ kann die Wertgleichheit auch das Symbol für die Identitätsprüfung (in C++ z. B. den Operator `==`) überladen.

Das Value Object-Muster wird verbreitet eingesetzt. Z. B. sind in den Java-Bibliotheken neben `BigInteger` u. a. die Klassen `BigDecimal` und `String` sowie die Wrapper-Klassen für die primitiven Typen (`Boolean`, `Integer`, `Float`, etc.) nach diesem Muster implementiert. Das Framework `JValue` [RS03] liefert weitere Beispiele für als Value Object implementierte Klassen.

Werteigenschaften:

Das Value Object-Muster bildet nur Unveränderlichkeit und damit nur eine Werteigenschaft ab.

Unerzeugbarkeit setzt dieses Muster nicht um. Eine als Value Object programmierte Klasse hat Konstruktoren, Klienten können – und müssen – diese aufrufen. Die Erzeugung von Elementen ist Bestandteil des Musters, z. B. ist bei Riehle [Rie06, p. 3] und bei Kühne [Küh99, p. 155] ausdrücklich davon die Rede, dass value objects erzeugt werden.

Auch die Bezeichnung des Musters – *Value Object* – deutet darauf hin, dass damit Werte gedanklich als Spezialfälle von Objekten eingeordnet werden, nicht als Gegensatz. Auch die Verwendung einer objektorientierten Sprache macht die Umsetzung von Unerzeugbarkeit schwierig; (Objekt-)Klassen haben immer einen Konstruktor.

Seiteneffektfreiheit und referentielle Transparenz der Methoden werden durch das Muster weder gefordert noch erleichtert. Value Objects unterbinden nur Änderungen in der eigenen Klasse. Die Typen der Datenfelder oder der Methodenparameter können änderbare Klassen sein, und ein Zugriff auf deren Methoden zerstört im Allgemeinen die referentielle Transparenz oder die Seiteneffektfreiheit.

Einfachheit:

Der Verzicht auf ändernde Methoden ist leicht und mit geringem Aufwand umsetzbar. Die Sicherstellung von Unveränderlichkeit durch softwaretechnische Verfahren hält allerdings einige Fallstricke bereit.

Beispielsweise kann Unveränderlichkeit durch Vererbung unterlaufen werden; eine Subklasse kann änderbare Felder und ändernde Methoden hinzufügen. Aus diesem Grund wird empfohlen, das Erben von einer unveränderlichen Klasse zu verhindern

in Java z. B. durch den Klassen-Modifier `final`. Dann aber ist es auch nicht möglich, die Klasse um regelkonforme, d. h. nicht-ändernde Methoden, zu erweitern. Bloch schlägt ein umfassendes Regelwerk vor, welches die Unveränderlichkeit einer Klasse sicherstellt [Blo08, Item 15].

Die Programmierung der Wertgleichheit ist aufwändig und schwierig, unabhängig davon, ob sie durch Überschreiben einer Methode wie `equals` oder durch Überladen eines Vergleichsoperators umgesetzt wird.³

Klarheit:

Weil das Value Object-Muster nur eine einzige charakteristische Eigenschaft von Werten umsetzt – Unveränderlichkeit – ist die Klasse nicht als Werttyp erkennbar.

Das Muster kann die intendierte Unveränderlichkeit nur unzureichend zum Ausdruck bringen. Es ist nicht allein auf Basis des Codes erkennbar, ob das Fehlen von ändernden Methoden eine Momentaufnahme darstellt – d. h. ob ändernde Methoden im Prinzip zulässig sind und bei der weiteren Programmierung ergänzt werden dürfen – oder ob die Unveränderlichkeit ein Wesensmerkmal des abgebildeten Gegenstandes ist.

Ein gravierendes Problem des Value Object-Musters besteht in der semantischen Lücke zwischen dem konzeptionellen Werttyp und der Klasse, die ihn abbildet: Es gibt keine 1:1-Entsprechung zwischen den Elementen des Typs und den Exemplaren der Klasse; *ein* Wert wird potentiell durch *mehrere* Elemente der Klasse repräsentiert. Die oben bereits angesprochene Wertgleichheit dient dazu, diese Lücke zu schliessen:

„Two value objects that are distinct objects may still mean the same value, and equality checking needs to realize this.“[Rie06, p. 4]

Es besteht die Gefahr, dass Benutzer einer Value Object-Klasse sich nicht darüber im klaren sind, dass sie – auf konzeptioneller Ebene – mit nur einem einzigen Wert hantieren, nicht mit „mehreren gleichen“ Objekten.

Sicherheit:

Unveränderlichkeit verhindert Alias-Effekte, das macht die Programmierung sicherer. Unveränderliche Objekte können in mehreren Kontexten gemeinsam genutzt werden. Sie erfordern keinen Programmieraufwand für Synchronisation, sie sind „thread-safe“.

Aus Klientensicht gibt es bei einer Value Object-Klasse mehrere Möglichkeiten einer falschen Benutzung:

³Kapitel 8 geht näher auf die Implementierung einer korrekten Gleichheit und die dabei zu beachtenden Regeln ein.

Das Vorhandensein von zwei Vergleichen ist eine Fehlerquelle, es birgt die Gefahr der Verwechslung. Beispielsweise ist es in Java ein bekannter Programmierfehler, für String-Vergleiche statt der `equals`-Methode den Operator „`==`“ zu verwenden.

Es ist möglich, Methoden einer Value Object-Klasse isoliert aufzurufen, d. h. ohne einen Kontext, nicht als rechte Seite einer Zuweisung und nicht als Parameter einer anderen Methode. Z. B. wird in Java eine allein stehende Anweisung wie

```
myBigInteger.negate();
```

vom Compiler nicht beanstandet. Sie bewirkt aber nichts, weil die Operation `negate` keinen Seiteneffekt hat und ihr Resultat ungenutzt bleibt. Es kann sich darum nur um einen Programmierfehler handeln. Ein Value Object verhält sich in dieser Hinsicht weniger sicher als ein primitiver Datentyp, bei dem ein isolierter Aufruf einer Operation zu einem Übersetzungsfehler führt. Die Unveränderlichkeit von Value Objects dagegen beruht nur auf einer Programmierkonvention, die Programmierungsumgebung kann sie darum nicht sicherstellen.

Aus dem gleichen Grund sind Value Objects auch im Hinblick auf Codewartung fehleranfällig. Weil ihre Unveränderlichkeit nur eine Programmierkonvention darstellt, kann sie bei Code-Überarbeitungen leicht zerstört werden, z. B. durch Hinzufügen von ändernden Operationen. Zwar können die Membervariablen einer Value Object-Klasse als unveränderlich markiert werden – in Java z. B. durch den Modifier `final` – was die Unveränderlichkeit stärker absichert und außerdem klarer dokumentiert. Doch auch das ist leicht nachträglich änderbar, und in einigen Fällen ist es zu rigide. Es verhindert z. B. Caching-Verfahren, bei denen die Zuweisung zu einer Membervariablen lediglich dem Vermeiden mehrfacher Berechnungen dient.

Effizienz:

Value objects haben unter Effizienzgesichtspunkten sowohl Vor- als auch Nachteile. Unveränderliche Elemente können kontextübergreifend gemeinsam genutzt werden. Das trägt dazu bei, Speicherplatz einsparen.

Die Änderung (=Neu-Zuweisung) einer Variablen erfordert in der Regel, Speicher für das neue Element bereitzustellen und den Speicher des alten Elementes freizugeben. Berechnungsvorgänge mit einer großen Anzahl von Zwischenergebnissen erhöhen deshalb den Speicherverbrauch. Auch belastet die häufige Speicherfreigabe den Garbage Collector, was das Laufzeitverhalten negativ beeinflusst (s. auch Kühne [Küh99, p. 157]). Ein Beispiel dafür ist ein String – der mehrere tausend Zeichen umfassen kann – an den ein weiteres Zeichen angehängt wird, wie in folgendem Java-Code:

```
String myString = "very long ..... String";  
myString = myString + "!";
```

Wird das von der Variablen `myString` bezeichnete String-Element nur in diesem Kontext verwendet, so bedeutet die Erstellung eines neuen und die Freigabe des alten Strings unnötigen Aufwand. Im Hinblick auf Speicherverbrauch und Laufzeitverhalten wäre es effizienter, das vorhandene String-Objekt zu ändern. Das Value Object-Muster eignet sich darum primär für leichtgewichtige Werte.

4.2.2 Fliegengewicht

Das Fliegengewicht (*Flyweight*) ist eines der 23 Entwurfsmuster, die von Gamma et al. publiziert wurden [GHJV98, Kap. 4]. Es dient primär der Ersparnis von Speicherplatz, Gamma klassifiziert es als „technisches Muster“. Das Fliegengewicht wird häufig in Zusammenhang mit der Abbildung von Werttypen eingesetzt. Es kann die gemeinsame Nutzung von Werten unterstützen und ihren Speicherverbrauch verringern, s. Züllighoven [Zül05, p. 240] und Riehle [Rie06, p. 4].

Zur Anwendung des Fliegengewichtes müssen die folgenden Bedingungen erfüllt sein [GHJV98, p. 197]:

- Ein Klient verwaltet viele ähnliche Objekte (z. B. die Buchstaben eines Dokumentes in einer Textverarbeitung), und bereits die große Anzahl dieser Objekte würde zu einem erheblichen Speicherverbrauch führen, und
- ein Teil des Objekt-Zustandes ist kontextabhängig (z. B. aufeinanderfolgende Buchstaben mit gleicher Schriftgröße und Schriftart), so dass es möglich ist, den gemeinsamen Zustand in den Kontext (z. B. die Zeile, den Absatz) auszulagern und von diesem verwalten zu lassen.

Das Muster trennt den kontextunabhängigen (*intrinsischen*) vom kontextabhängigen (*extrinsischen*) Zustand und fasst mehrere Objekte mit gleichem intrinsischen Zustand zu einem einzigen zusammen. Die Fliegengewicht-Klasse bildet den intrinsischen Zustand ab – im Beispiel der Textverarbeitung die einzelnen Zeichen eines Zeichensatzes. Der extrinsische Zustand, z. B. die Schriftgröße, ist in den Klassen abgelegt, welche die Fliegengewichte benutzen – Zeilen, Absätze, etc. Benötigt eine Methode der Fliegengewicht-Klasse – z. B. drucken – Information über den extrinsischen Zustand, so erhält sie ihn als Parameter. Das spart Speicherplatz,

- weil es die Anzahl der Fliegengewicht-Elemente reduziert. Im Beispiel genügt ein Element der Klasse „Zeichen“ für jeden Buchstaben, es muss nicht jeder im Text vorkommende Buchstabe durch ein eigenes Objekt abgebildet werden.
- weil der extrinsische Zustand oft für eine größere Zusammenfassung von Fliegengewichten (z. B. alle Zeichen eines Absatzes) verwaltet werden kann. Im Beispiel ist es nicht erforderlich, diese Information jedem einzelnen Vorkommen eines Zeichens zuzuordnen.

Das Muster sieht vor, dass Klienten Fliegengewichte nicht selbst erzeugen, sondern ausschließlich über eine Fabrik („Factory“) auf sie zugreifen. Die Fabrik verwaltet alle Elemente der Fliegengewicht-Klasse an zentraler Stelle (z. B. in einer Hashmap) und sorgt dafür, dass es jedes Element nur einmal gibt. Kühne bezeichnet dieses Verfahren als „lookup in repository“ [Küh99, p. 160].

Die Fabrik kann die Elemente verzögert („lazy“) erzeugen, d. h. erst beim ersten Zugriff. Ab dem zweiten Zugriff erhält ein Klient dann das bereits in der Hashmap vorhandene Element. Alternativ können auch alle Elemente bei Systemstart erzeugt werden. Das bietet sich z. B. dann an, wenn die Fliegengewichtklasse nur wenige Elemente umfasst.

Fliegengewichte sind unveränderlich. Gamma et al. verlangen in ihrer Beschreibung die Unveränderlichkeit nicht explizit. Die Voraussetzungen und der Zweck des Musters („A flyweight is a shared object that can be used in multiple contexts simultaneously“ [GHJV98, p. 196]) legen aber nahe, dass es auf unveränderliche Abstraktionen zugeschnitten ist. Dafür spricht auch, dass alle aufgeführten Beispiele unveränderlich sind. Nachfolgende Arbeiten, z. B. die von Goetz, fordern Unveränderlichkeit ausdrücklich als Voraussetzung für die Anwendung des Fliegengewicht-Musters[Goe03].

Das Fliegengewicht kommt in verschiedenen Frameworks zum Einsatz, z. B. im JWAM-Framework [BLZ99]. Es enthält die Klassen `DomainValue` und `DomainValueImpl`, letztere enthält einen großen Teil der Fabrik-Handhabung. Konkrete Wertklassen können von `DomainValueImpl` erben und müssen diese Verarbeitung damit nicht selbst programmieren.

Werteigenschaften:

Das Fliegengewicht setzt sowohl Unveränderlichkeit als auch Unerzeugbarkeit um (als einziges in der Reihe der hier untersuchten Verfahren).

Die Unerzeugbarkeit beruht darauf, dass die Erzeugung vollständig in der Implementation der Klasse verborgen ist. Im Klientencode taucht kein Aufruf eines Konstruktors (`new ...`) auf. Aus Sicht von Klienten erscheinen die Elemente der Fliegengewicht-Klasse so, als seien sie „schon immer da“ und würden nicht erzeugt.

Seiteneffektfreiheit und referentielle Transparenz werden durch das Fliegengewicht nicht nur *nicht* umgesetzt; das Muster sieht ausdrücklich Seiteneffekt-behaftete und referentiell opake Operationen vor: Methoden der Fliegengewicht-Klasse können (s. o.) den extrinsischen Zustand als Parameter übergeben bekommen. Dieser ist in der Regel änderbar, und so können diese Operationen Seiteneffekte bewirken oder sichtbar machen. Das widerspricht dem Konzept von Werttypen, welche keine Objekte in ihrer Schnittstelle akzeptieren und keine Seiteneffekte hervorrufen dürfen. Damit Fliegengewichte Werttypen abbilden, müssten derartige Operationen

in die Kontextklassen verlagert werden und dann umgekehrt die Fliegengewichtsklasse als Parameter erhalten.

Einfachheit:

In Bezug auf Unveränderlichkeit gelten die gleichen Aussagen wie für das Value Object. Die Unerzeugbarkeit erfordert erhebliche Umwege. Das Fliegengewicht emuliert Unerzeugbarkeit in einem programmiersprachlichen Umfeld, das grundsätzlich davon ausgeht, dass alle Elemente eines Typs erzeugt werden müssen. Der Aufwand für die Umsetzung des Musters und seine Komplexität sind darum hoch.

Für Klienten dagegen ist die Benutzung von Fliegengewichten einfach und die Fehleranfälligkeit gering. Unveränderlichkeit, Unerzeugbarkeit und korrekte Funktionsweise des Vergleichsoperators werden von der Fliegengewichtsklasse sichergestellt.

Obwohl Klienten für die Wertgleichheit den von der Sprache vorgegebenen Identitätsoperator verwenden können, ist eine Methode wie `equals` aus technischen Gründen erforderlich. Um festzustellen, ob ein angefordertes Element bereits existiert, bildet die Fabrik aus den Parametern ein temporäres Element und vergleicht es anhand von `equals` mit den vorhandenen Elementen.

Werden die Elemente der Fliegengewicht-Klasse verzögert erzeugt, so ist die Möglichkeit eines konkurrierenden Zugriffs zu berücksichtigen, s. Riehle [Rie06, p. 5]. Wird ein Element von zwei Klienten gleichzeitig angefordert, muss vermieden werden, es intern zweimal zu erzeugen. Die Synchronisation verursacht zusätzlichen Programmieraufwand und macht die Umsetzung eines Fliegengewichts komplexer und schwieriger als z. B. die eines Value Objects.

Klarheit:

Weil das Fliegengewicht einen Spezialfall des Value Objects darstellt, gelten dafür auch die dort getroffenen Aussagen in Bezug auf das Erkennen der Unveränderlichkeit im Code (s. o.).

Auch die Nicht-Erzeugung der Fliegengewichte ist aufgrund der beschrittenen Umwege im Code nicht deutlich ablesbar. Die Dokumentation des Musters trägt ebenfalls nicht dazu bei, die Unerzeugbarkeit klar zu kommunizieren. In der Beschreibung des Musters ist bei Gamma et al. mehrfach die Rede von einer „*Erzeugung*“ der Fliegengewichte [GHJV98, p. 199]. Auch die Bezeichnung des Zugriffspunktes für die Fliegengewichte als „*Fabrik*“ deutet auf einen Erzeugungsvorgang hin und ist dadurch sprachlich verwirrend. Die „FliegengewichtFabrik“ ist so programmiert, dass sie die Nichterzeugung der Fliegengewichte (aus Klientensicht) korrekt widerspiegelt. Es wäre klarer, sie nicht als Fabrik, sondern z. B. als „Fliegengewicht-Pool“ zu bezeichnen. (Tatsächlich findet sich bei Gamma et al. diese Bezeichnung auf einem der UML-Diagramme, die das Muster illustrieren [GHJV98, p. 198], die nachfolgenden Ausführungen greifen diesen Begriff aber nicht auf.)

Sicherheit:

Für Klienten ist die Benutzung von Fliegengewichten einfach und sicher benutzbar. Sie müssen zur Gewährleistung der Werteigenschaften keine zusätzlichen Konventionen einhalten. Unveränderlichkeit, Unerzeugbarkeit und korrekte Funktionsweise des Vergleichsoperators werden von der Fliegengewichtsklasse sichergestellt. Die Wertgleichheit kann mit dem Operator für Objekt-Identität (z. B. in Java mit „==“) geprüft werden. Anders als beim Value Object ist kein Umweg über eine Methode wie `equals` erforderlich. Das ist – neben der ursprünglichen Intention des Musters, Speicherplatz einzusparen – ein weiterer Vorteil der zentralen Verwaltung der Elemente.

Aufgrund der hohen Komplexität des Fliegengewicht-Musters können aber die Werteigenschaften bei einer Codeüberarbeitung leicht zerstört werden.

Effizienz:

In Bezug auf Speicherverbrauch ist das Fliegengewicht ähnlich problematisch wie das Value Object. Einerseits ermöglicht die Unveränderlichkeit der Elemente eine gemeinsame Nutzung, auch über mehrere Kontexte, und spart damit Speicherplatz ein. Andererseits erfordert jede Variation gegenüber einem vorhergehenden Element ein anderes Element und damit auf technischer Ebene potentiell die Allokation von weiterem Speicher. Beim Fliegengewicht besteht die Chance, ein bereits vorhandenes Element zu verwenden, was die Gefahr von hohem Speicherverbrauch verringert.

Das Fliegengewicht eignet sich vorwiegend für Werttypen mit nur wenigen Elementen oder für Werttypen, deren Klienten nur einen Bruchteil der konzeptionell verfügbaren Werte nutzen und keine aufwändigen Verarbeitungen mit vielen Zwischenergebnissen vornehmen.

4.2.3 Typesafe Enum Pattern

In der ersten Ausgabe von „Effective Java“ beschreibt Bloch das „Typesafe Enum Pattern“, ein Verfahren zur typsicheren Abbildung von Aufzählungstypen („Replace enum constants with classes“) [Blo01, Item 21]. Die Grundidee des Typesafe Enum Pattern besteht darin,

- für die Aufzählung eine eigene Klasse zu definieren,
- für jedes Element der Aufzählung eine öffentlich zugängliche Konstante zu definieren (z. B. in Java durch `public static final`), und
- die Konstruktoren vor Klienten zu verbergen (in Java durch `private`), so dass Klienten keine weiteren Exemplare der Klasse erzeugen können.

Hier ein Java-Beispiel zur Abbildung eines Aufzählungstyps für Jahreszeiten, Season:

```
class Season {  
    private final String name;  
    private Season(String name) { this.name = name; }  
    public String toString() { return name; }  
    public static final Season SPRING = new Season("spring");  
    public static final Season SUMMER = new Season("summer");  
    public static final Season AUTUMN = new Season("autumn");  
    public static final Season WINTER = new Season("winter");  
}
```

Das Beispiel greift für die Umsetzung des Typesafe Enum Patterns auf Klassenvariablen (`static`) zurück. In einer Sprache, die keine Klassenvariablen unterstützt, können die Elemente der Aufzählung alternativ in einer Hashmap gespeichert werden. Das Muster entspricht dann einem Fliegengewicht (s. Abschnitt 4.2.2), bei dem für jedes Element eine eigene, parameterlose Zugriffsmethode programmiert wird.

Vor der Verbreitung des Typesafe Enum Pattern wurden (insbesondere in Java-Bibliotheken) Aufzählungen häufig mit Hilfe von Ganzzahlkonstanten abgebildet, von Bloch auch als „int enum pattern“ bezeichnet [Blo01, p. 80][Blo08, p. 147]. D. h. die Elemente der Aufzählung wurden durch Definitionen wie

```
public static final int SPRING = 0;  
public static final int SUMMER = 1;  
...
```

festgelegt. Gegenüber Ganzzahlkonstanten besteht der Vorteil des Typesafe Enum Pattern in seiner (namensgebenden) Typsicherheit. Zuweisungen von Konstanten mit einer anderen fachlichen Bedeutung, wie

```
int mySeason = Shirtsize.XXL
```

sind damit nicht möglich, auch keine fachlich unsinnigen Berechnungen, wie

```
(Season.WINTER - Season.SPRING) / Season.SUMMER
```

Sie führen mit dem Typesafe Enum Pattern zu einem Übersetzungsfehler.

Ein Nachteil des Typesafe Enum Pattern besteht darin, dass die Elemente der Aufzählung in den meisten Sprachen nicht in einer Mehrfachauswahl (Java: in einem `switch`-Statement) verwendet werden können.

Aufgrund seiner Ausrichtung auf Aufzählungen eignet sich das Typesafe Enum Pattern nicht für alle Werttypen. Für Typen mit einer großen Anzahl von Elementen, wie Datum oder String, ist es nicht praktikabel einsetzbar.

In einer Sprache mit dedizierter Unterstützung für Aufzählungen (s. auch Abschnitt 5.3.2) hat das Typesafe Enum Pattern nur wenig Relevanz. So hat das Muster

z. B. in Java ab Version 1.5, die den Typkonstruktor `enum` einführt, seine Bedeutung verloren. Es wird in der zweiten Auflage von Effective Java [Blo08] nicht mehr erwähnt, statt dessen enthält das Buch ein Kapitel mit Hinweisen für eine korrekte und effiziente Benutzung von `enums`.

Werteigenschaften:

Das Typesafe Enum Pattern gewährleistet nur eine charakteristische Eigenschaft von Werttypen, die Unerzeugbarkeit. Aus Klientensicht sind die Elemente einer Aufzählung „schon immer da“, sie können weder erzeugt noch zerstört werden. Die Umsetzung der weiteren Werteigenschaften sind nicht Muster-inhärent, sondern erfordern entsprechende Programmierdisziplin.

Einfachheit:

Für Klienten ist eine Typesafe Enum einfach zu benutzen. Es müssen keine klientenseitigen Konventionen eingehalten werden. Insbesondere kann die Wertgleichheit (wie auch beim Fliegengewicht) mit dem spracheigenen Operator für Objektidentität geprüft werden.

Gemessen daran, dass Aufzählungen einfache Abstraktionen darstellen, und in Vergleich mit dem „int enum pattern“, erfordert die Programmierung einer Typesafe Enum hohen Aufwand und führt zu umfangreichen Code.

Klarheit:

Das Typesafe Enum Pattern bringt die Kerneigenschaft von Aufzählungen – nämlich, dass es sich um eine diskrete Aufzählung einzelner Elemente handelt – klar zum Ausdruck. Die Werteigenschaften dagegen sind nicht deutlich erkennbar, auch dann nicht, wenn eine Typesafe Enum sie korrekt umsetzt. Hier gilt das Gleiche, was bereits analog für das Value Object festgestellt wurde: Dem Code ist nicht zu entnehmen, ob das Fehlen von ändernden und erzeugenden Operationen eine zufällige Momentaufnahme darstellt oder einen Wesenszug des abgebildeten Typs.

Sicherheit:

Das Typesafe Enum Pattern gewährleistet Unerzeugbarkeit, aber keine andere der charakteristischen Werteigenschaften. Es ist aufschlussreich, dass in Java auch die in Version 1.5 hinzugefügte Sprachunterstützung für Aufzählungen (d. h. der Typkonstruktor `enum`, s. Abschnitt 5.3.2) Unveränderlichkeit nicht sicherstellt. Bloch beschreibt (in der 2. Auflage von Effective Java) Aufzählungen als grundsätzlich unveränderliche Abstraktionen und empfiehlt, diese Eigenschaft bei ihrer Programmierung zu berücksichtigen [Blo08, p. 150]. Das lässt die Frage offen, warum nicht bereits der Typkonstruktor `enum` die Unveränderlichkeit gewährleistet.

Abhängig von der verwendeten Sprache kann die Implementierung einer Typesafe Enum subtile Fallstricke bereithalten. In Java muss z. B. im Fall einer serialisierbaren Aufzählungs-Klasse die Deserialisierung (Methode: `readResolve`) dafür sorgen, dass diese Operation ein bereits vorhandenes Exemplar liefert anstatt standardmäßig ein neues zu erzeugen (siehe Bloch [Blo01, p. 107]).

Effizienz:

Aufzählungstypen verfügen im Allgemeinen nur über wenige Elemente. Ihre Programmierung in Form einer Typesafe Enum sorgt in der Regel dafür, dass diese bereits bei Systemstart im Speicher angelegt werden und dort bleiben. Eine Speicherbereinigung ist aufgrund des statischen Charakters der Elemente nicht erforderlich. Die beim Value Object beschriebene Problematik – hohe Speichieranforderung und Belastung des Garbage Collectors bei Verarbeitungen mit vielen Zwischenschritten – tritt damit beim Typesafe Enum nicht auf.

4.3 Klientenseitige Programmierkonventionen

Ein technisches Problem unveränderlicher Klassen ist das permanente Bereitstellen und Freigeben von Speicher bei Verarbeitungen mit vielen Zwischenergebnissen (s. Abschnitte 4.2.1 und 4.2.2). Es belastet den Garbage Collector und macht die Verarbeitung ineffizient, insbesondere bei schwergewichtigen Klassen.

Um diese Nachteile zu vermeiden, werden Werttypen teils auch mit Hilfe von *änderbaren* Klassen abgebildet. Z. B. empfiehlt Riehle:

„For heavyweight value objects, you sometimes give up on immutability. The problem here is that depending on how they are used, a lot of heavyweight value objects may be created only to be dropped quickly, because they are only an intermediate computation result.“[Rie06, p. 4]

Bei solchen Ansätzen liegt die Verantwortung für ein wert-gemäßes Verhalten vollständig bei den Klienten. Sie sind dafür zuständig, unerwünschte Alias-Effekte zu vermeiden. Es gibt verschiedene Verfahren, das zu erreichen: *Defensives Kopieren*, das *Copy-on-write*-Verfahren, das *Body-handle*-Idiom, oder die Verwendung eines *Mutable Companion*.

4.3.1 Defensives Kopieren

Defensives Kopieren vermeidet unerwünschte Seiteneffekte, indem es von jedem Element, das in einen Kontext – z. B. eine andere Operation – hinein gelangt oder ihn verlässt, eine Kopie anfertigt (s. dazu Bloch: „Make defensive copies when nee-

ded“ [Blo08, Item 39]). Kühne bezeichnet dieses Verfahren als „copy-on-passing“ [Küh99, p. 154].

Das folgende Java-Beispiel illustriert die Funktionsweise von defensivem Kopieren. Es wird vorausgesetzt, dass es eine Klasse `Date` gibt, die einen Typ „Datum“ implementiert und die (wie auch die Klasse `java.util.Date`) über ändernde Operationen verfügt, u. a. die Methode `addDays`. Den Kontext bildet im Beispiel eine Klienten-Klasse namens `Project`. Sie verfügt über ein Datenfeld `beginDate` vom Type `Date` und zugehörige lesende und schreibende Methoden `getBeginDate` und `setBeginDate`:

```
class Project {
    Date beginDate;
    Date getBeginDate() { return beginDate.clone(); }
    void setBeginDate(Date inDate) { beginDate = inDate.clone(); }
    void postponeBegin(int days) { beginDate.addDays(days); }
    ...
}
```

Die Methode `setBeginDate` erstellt eine Kopie des übergebenen Elementes, und `getBeginDate` eine Kopie des auszugebenden Elementes. Beides zusammen gewährleistet, dass der Kontext `Project` mit einem eigenen Element der Klasse `Date` arbeitet. Änderungen, die innerhalb der Klasse `Project` auf diesem Element ausgeführt werden, z. B. in der Methode `postponeBegin`, können sich außerhalb der Klasse `Project` nicht auswirken.

Voraussetzung für das defensive Kopieren ist, dass Klienten eine Möglichkeit erhalten, eine Kopie anzufertigen, z. B. in Form einer Methode `clone`⁴ oder mit einem Konstruktor, dem als Parameter die erforderlichen Eigenschaften des zu kopierenden Elementes übergeben werden.

Werteigenschaften:

Beim defensiven Kopieren unterliegt die Klasse, die den Werttyp abbildet, keinerlei Restriktionen. Ihre Elemente können erzeugt und verändert werden, ihre Operationen können Seiteneffekte bewirken und referentiell opak sein. Sie erfüllt keine der charakteristischen Werteigenschaften.

Das Verhalten der Klienten bewirkt ebenfalls keine Unveränderlichkeit. Es grenzt lediglich die Änderungen auf einen einzelnen Kontext ein, z. B. auf eine Klasse oder eine Methode.

⁴Im Beispiel wurde aus Vereinfachungsgründen die Methode `clone` verwendet. Das ist nicht immer ein geeignetes Verfahren. `clone` sollte nur dann eingesetzt werden, wenn sichergestellt ist, dass diese Methode keine Seiteneffekte bewirken kann. Diese wären z. B. dann möglich, wenn `clone` das kopierte Element zwischenspeichert oder an einen anderen Kontext weitergibt. Bloch rät dazu, die Klasse als `final` zu programmieren, so kann ausgeschlossen werden, dass eine Subklasse ein solches Verhalten implementiert [Blo08, p. 186].

Einfachheit:

Defensives Kopieren ist leicht umzusetzen. Es verursacht aber hohen Aufwand, weil es von jedem Klienten und bei jedem Übergang eines Elementes von einem Kontext in einen anderen anzuwenden ist.

Klarheit:

Das Verfahren beruht darauf, in der Klasse, die den Werttyp darstellt, ändernde Methoden (aus Effizienzgründen) zuzulassen und ihre Elemente unter festgelegten Bedingungen zu kopieren. Weil Werte konzeptionell aber nicht geändert und nicht erzeugt, und damit auch nicht kopiert werden, steht der Code in eklatantem Widerspruch zu der Abstraktion, die er abbildet. Die Werte sind nicht als Werte erkennbar.

Sicherheit:

Wird in einem Kontext defensives Kopieren korrekt und durchgängig angewendet, so sind dort unerwünschte Alias-Effekte ausgeschlossen. Damit das überall gilt, müssen *alle* Klienten defensives Kopieren einsetzen – eine Forderung, auf die in der Dokumentation der Klasse hingewiesen, die aber nur schwer sichergestellt werden kann.

Da es sich beim defensiven Kopieren um eine reine Programmierkonvention handelt, ist die Gefahr hoch, bei Codeüberarbeitungen das wert-ähnliche Verhalten durch Unkenntnis oder durch Flüchtigkeitsfehler zu zerstören.

Effizienz:

Das Verfahren vermeidet den oben beschriebenen hohen Speicherverbrauch unveränderlicher Klassen. Durch defensives Kopieren können aber mehr Kopien erstellt werden als erforderlich. Wird z. B. ein Element an einen Kontext weitergegeben, der nur lesend darauf zugreift, ist Kopieren unnötig (s. dazu Riehle [Rie06, p. 2]).

4.3.2 Copy-on-write

Beim Copy-on-write-Verfahren gehen Klienten umgekehrt vor wie beim defensiven Kopieren: ein Element wird ohne Einschränkungen weitergegeben, kopiert wird es unmittelbar vor einer Änderung. Das Verfahren wird z. B. beschrieben von Koenig und Moo [KM96, Kap. 6.7], von Sutter unter der Bezeichnung „lazy copy“ [Sut02, p. 90], und von Kühne unter dem Namen „copy-on-change“ [Küh99, p. 158].

Übertragen auf das obige Beispiel ergibt sich mit dem Copy-on-write-Verfahren folgender Klienten-Code:

```
class Project {  
    Date beginDate;  
    Date getBeginDate() { return beginDate; }  
    void setBeginDate(Date inDate) { beginDate = inDate; }  
    void postponeBegin(int days) {  
        beginDate = beginDate.clone();  
        beginDate.addDays(days);  
    }  
}
```

Copy-on-write kann als komplementär zum defensiven Kopieren angesehen werden. Das copy-on-write-Verfahren benutzt ein Element so lange wie möglich gemeinsam, auch kontextübergreifend. Defensives Kopieren dagegen kopiert prophylaktisch, d. h. bereits dann, wenn ein Element in einen anderen Kontext übergeben wird.

Werteigenschaften:

Copy-on-write setzt ebenso wie defensives Kopieren keine der charakteristischen Werteigenschaften um. Die Elemente der Klasse werden erzeugt, kopiert und geändert.

Einfachheit:

Copy-on-write ist wie defensives Kopieren einfach anzuwenden. Klienten müssen lediglich wissen, welche Methoden der Klasse Änderungen bewirken, um vor ihrem Aufruf eine Kopie anfertigen zu können.

Klarheit:

Weil Copy-on-write keine der charakteristischen Werteigenschaften umsetzt, sind Werte im Quelltext nicht als Werte erkennbar. Das beeinträchtigt die Lesbarkeit des Codes und macht den Ansatz fragil.

Sicherheit:

Die Anwendung von Copy-on-write kann, anders als defensives Kopieren, nicht lokal auf einen Kontext begrenzt werden. Copy-on-write vermeidet unerwünschte Seiteneffekte nur dann, wenn *alle* beteiligten Klienten es einhalten. Das erhöht seine Fehleranfälligkeit.

Wie bei den anderen Programmierkonventionen besteht auch bei Copy-on-write die Gefahr, das gewünschte Verhalten im Rahmen einer Codeänderung zu zerstören.

Effizienz:

Wenn Elemente häufig von Kontext zu Kontext weitergereicht werden, aber selten Zwischenergebnisse komplexer, mehrschrittiger Berechnungen sind, dann trägt ihre kontextübergreifende Nutzung dazu bei, unnötige Kopien einzusparen. Copy-on-write verbraucht dann – verglichen mit defensivem Kopieren – weniger Speicherplatz. (Umgekehrt ist defensives Kopieren speichertechnisch effizienter, wenn Elemente selten ihren Kontext verlassen und dort häufig Verarbeitungen mit vielen Zwischenschritten anfallen.)

Auch mit Copy-on-write können überflüssige Kopien entstehen. Für jedes Element, das in nur einem Kontext verwendet wird, ist es unnötig, erst eine Kopie anzufertigen und diese dann zu ändern. In dem Fall wäre es effizienter, das vorhandene Element zu ändern.

4.3.3 Body-Handle-Idiom

Coplien beschreibt ein im C++-Umfeld gebräuchliches Verfahren, welches die eben bemängelte Erstellung unnötiger Kopien vermeidet, indem es den Code auf zwei Klassen verteilt: Body und Handle. Er bezeichnet es als „handle/body class idiom“, auch als „representation sharing and reference counting“ [Cop92, Kap. 3.5]. Auch die oben aufgeführten Copy-on-write-Verfahren von Koenig und Moo [KM96] und von Sutter [Sut02] legen eine derartige Zweiteilung zugrunde. Züllighoven et al. bezeichnen das Verfahren als Body-Handle-Muster; sie verwenden es, um den Speicherbedarf von Werttypen zu optimieren [BLZ99].

Bei diesem Verfahren enthält der Body die interne Repräsentation, die fachliche Verarbeitungslogik und einen Referenzzähler. Das Handle enthält einen Verweis auf den Body und ist zuständig für die Speicherverwaltung. Ein Body kann von mehreren Handles referenziert werden, der Referenzzähler protokolliert ihre Anzahl. Klienten greifen ausschließlich auf das Handle zu, der Body ist nur für das Handle sichtbar.

Das Handle entscheidet bei allen Verarbeitung auf Basis des Referenzzählers, ob der referenzierte Body weiter verwendet werden kann oder ob eine Kopie davon erstellt werden muss. Beispielsweise sorgt das Handle dafür, dass bei Aufruf einer ändernden Operation im Fall „Referenzzähler = 1“ die Änderung auf dem Body ausgeführt wird. Im Fall „Referenzzähler > 1“ dagegen wird erst der Referenzzähler verringert, dann eine Kopie des Body (mit Referenzzähler = 1) angefertigt und schließlich die Änderung auf der Kopie durchgeführt. Auf diese Weise verhindert das Verfahren unnötiges Kopieren des Body.

Tabelle 4.1 gibt das Verarbeitungsschema für das Body-Handle-Idiom wieder, gegliedert nach verschiedenen Arten von Operationen (Erzeugen, Kopieren, Ändern, Lesen, Löschen). Löschende Operationen („Destruktoren“) spielen nur in Sprachen mit manueller Speicherverwaltung, wie C++, eine Rolle.

Art der Operation	Verarbeitungsschema
Erzeugen	Handle erstellen, Body erstellen mit Referenzzähler = 1
Kopieren	Handle erstellen, vorhandenen Body referenzieren, Referenzzähler erhöhen
Ändern	falls Referenzzähler = 1: Änderung auf dem Body ausführen
	falls Referenzzähler > 1: Referenzzähler verringern, Kopie des Body mit Referenzzähler = 1 erstellen, Änderung auf der Kopie durchführen
Lesen	Aufruf an Body weiterleiten, Ergebnis an Klient zurückgeben
Löschen („Destruktor“)	Handle löschen, Referenzzähler des Body verringern, wenn Referenzzähler = 0: Body löschen

Tabelle 4.1: Body-Handle-Idiom: Verarbeitung abhängig von der Art der Operation

Funktionsweise und Handhabung des Body-Handle-Idioms unterscheiden sich in Programmiersprachen mit automatischer oder solchen mit manueller Speicherverwaltung erheblich voneinander:

In Sprachen mit automatischer Speicherverwaltung (wie Java, C# oder Eiffel) enthält der Referenzzähler die Anzahl der Handles, die einen Body benutzen. Das ist *nicht* notwendigerweise die Anzahl der Kontexte, in denen ein Body – indirekt über die Handles – verwendet wird. Grund dafür ist die Funktionsweise der Parameterübergabe: Das Weitergeben eines Handle als Parameter an eine andere Operation erhöht *nicht* den Referenzzähler. Das gilt analog für die Rückgabe aus einer Operation und für Zuweisungen. Der Body befindet sich in solchen Fällen trotz eines Referenzzählers von 1 in mehreren Kontexten, eine Änderung ist dann in allen diesen Kontexten sichtbar. Klienten müssen selbst dafür sorgen, dass Änderungen auf einen Kontext begrenzt bleiben, z. B. durch defensives Kopieren oder durch Einsatz des Copy-on-write-Verfahrens. Das Body-Handle-Idiom ersetzt diese Verfahren hier nicht, sondern ergänzt sie.

In Sprachen mit manueller Speicherverwaltung, z. B. C++, funktioniert das Body-Handle-Idiom anders. In C++ ist auch die Zuweisung ein „normaler“ Ope-

rator, der vom Programmierer der zugehörigen Klasse implementiert werden kann. Beim Body-Handle-Idiom ist die Zuweisung in der Handle-Klasse so zu programmieren, dass sie bei jedem „Aufruf“ den Referenzzähler des Body erhöht. Das gilt gleichermaßen für die Parameterübergabe (welche implizit einen Aufruf des Copy-Konstruktors auslöst). Zuweisung und Parameterübergabe zählen hier als kopierende Operationen. Umgekehrt ist bei einem Absinken des Referenzzählers auf 0 der Speicher des Body manuell wieder freizugeben. Auf diese Weise ist der Speicher vollständig unter der Kontrolle des Handle, und der Referenzzähler gibt die Anzahl der Kontexte wieder, welche indirekt den Body verwenden.

Das Body-Handle-Idiom ist deshalb besonders im C++-Umfeld verbreitet. Koenig und Moo kombinieren das Body-Handle-Verfahren mit Copy-on-write, um dem Handle „Wertsemantik zu verleihen“ [KM96, p. 62]. Sutter verwendet die Begriffe „body-handle“ und „copy-on-write“ synonym [Sut02, Kap. 14-16].

Werteigenschaften:

Das Body-Handle-Idiom setzt – wie die beiden zuvor untersuchten klientenseitigen Verfahren – keine der charakteristischen Werteigenschaften um. Sowohl das Handle als auch der (für Klienten nicht sichtbare) Body verfügen über ändernde und erzeugende/ kopierende Operationen.

Einfachheit:

Die Aufteilung der Zuständigkeiten zwischen Body- und Handle-Klasse und die Verwaltung des Referenzzählers machen das Verfahren komplex und seine Programmierung aufwändig. Die Handle-Klasse muss in *jeder* ihrer Methoden die passende Speicherverwaltung implementieren. Zwar kann die Handhabung des Referenzzählers in Hilfsmethoden verlagert werden, was eine Code-Vervielfachung teilweise vermeidet. (Sutter benutzt für Änderungen z. B. eine Methode namens `aboutToModify` [Sut02, p. 93]). Die Aufruf der passenden Hilfsmethode kann aber leicht vergessen werden.

Wertgleichheit muss die Übereinstimmung des Body, nicht die des Handle überprüfen. Das Body-Handle-Idiom gewährleistet nicht, dass es zu einem konzeptionellen Wert nur ein Exemplar der Body-Klasse gibt – anders als z. B. das Fliegengewicht. Der Programmierer der Body-Klasse muss das berücksichtigen und (wie bei einem Value Object) eine passende Wertgleichheit programmieren.

Klarheit:

Das Body-Handle-Idiom setzt, wie die beiden zuvor beschriebenen klientenseitigen Verfahren, änderbare, erzeugbare Objekte für die Abbildung von Werten ein. Es lässt dadurch Werte nicht als Werte erkennen und vermindert Klarheit und Lesbarkeit des Quellcodes.

Sicherheit:

In Sprachen mit manueller Speicherverwaltung wie C++ ist die Benutzung einer nach dem Body-Handle-Idiom programmierten Klasse geradlinig und unkompliziert. Klienten müssen keine Programmierkonventionen zur Vermeidung von Alias-Effekten einhalten. Die komplexe Speicherverwaltung im Handle kann im Rahmen von Codeänderungen leicht zerstört werden, das macht das Verfahren fehleranfällig.

In Sprachen mit automatischer Speicherverwaltung (Java, C#, Eiffel, etc.) kann das Body-Handle-Idiom nur dazu genutzt werden, ein anderes Verfahren zum Eingrenzen von Seiteneffekten (Copy-on-write oder defensives Kopieren, s. o.) zu ergänzen und zu optimieren. Weil diese Ansätze ausschließlich auf Programmierkonventionen beruhen, ist ihre korrekte Anwendung schwer sicherzustellen und fragil gegenüber Codewartungen.

Effizienz:

Das Body-Handle-Idiom verringert den Speicherbedarf auf zwei Arten. Erstens reduziert es die erforderlichen Kopiervorgänge des Body anhand des Referenzzählers. Zweitens kopiert es in den meisten Fällen nur das leichtgewichtige Handle und nutzt den Body, soweit möglich, über mehrere Handles gemeinsam. Die daraus resultierende Speicherersparnis wirkt sich besonders bei schwergewichtigen Abstraktionen aus.

In Bezug auf das Laufzeitverhalten kann das Body-Handle-Idiom Probleme aufwerfen. Z. B. kann es in nebenläufigen Anwendungen aufgrund der erforderlichen Synchronisation zu erheblichen Performanceeinbußen kommen (siehe dazu Sutter [Sut02, Kap. 16]).

4.3.4 Mutable Companion

Ein weiteres Verfahren, um die o. g. technischen Nachteile des Value Object-Musters zu kompensieren, besteht darin, es um einen „Mutable Companion“ zu ergänzen. Das Verfahren wird z. B. beschrieben von Eckel [Eck02, Kap. A, p. 1057-1060]. Auch Bloch weist auf die Möglichkeit hin, die Performance unveränderlicher Klassen durch Einsatz eines Mutable Companion zu verbessern [Blo08, p. 77], ebenso Henney [Hen03b, p. 65].

Mit dem Mutable Companion wird zusätzlich zu einem Value Object eine zweite Klasse programmiert, die fachlich dem Value Object entspricht, aber über ändernde Methoden verfügt. Verarbeitungen mit vielen Zwischenergebnissen benutzen statt des Value Objects den Mutable Companion.

Ein Java-Codebeispiel (angelehnt an Eckel [Eck02]) illustriert die Programmierung und Benutzung eine Mutable Companion. Die Klasse `Immutable` mit den zugehörigen Methoden `add` und `multiply` ist als Value Object implementiert. Die Klasse `Mutable` bildet ihren Mutable Companion. Sie verfügt über die gleichen Methoden wie `Immutable`, anders als in der Klasse `Immutable` ändern diese das Element, für das sie aufgerufen werden. Jede der beiden Klassen verfügt über eine Methode, welche ein Element in ein „gleichwertiges“ Gegenstück der anderen Klasse umwandelt, `makeMutable` bzw. `makeImmutable`.

```
class Immutable {
    private int data;
    public Immutable(int val) { data = val; }
    public int read() { return data; }

    public Immutable add(int x) {
        return new Immutable(data + x);
    }
    public Immutable multiply(int x) {
        return new Immutable(data * x);
    }
    public Mutable makeMutable() {
        return new Mutable(data);
    }
}

class Mutable {
    private int data;
    public Mutable(int initVal) { data = initVal; }
    public int read() { return data; }

    public void add(int x) {
        data += x;
    }
    public void multiply(int x) {
        data *= x;
    }
    public Immutable makeImmutable() {
        return new Immutable(data);
    }
}
```

Der folgende Klienten-Code illustriert anhand der – inhaltlich gleichen – Methoden `modify1` und `modify2` eine Benutzung der Klasse `Immutable`, einmal ohne und einmal mit `Mutable Companion`.

```
Immutable modify1(Immutable inVal) {
    Immutable im = inVal;
    im = inVal.add(12);
    im = im.multiply(3);
    im = im.add(10);
    im = im.multiply(2);
    return im;
}
```

`modify1` verwendet ausschließlich die Klasse `Immutable`. Jeder Schritt (jede Zeile) erfordert auf technischer Ebene Speicher für das nächste Zwischenergebnis, während der Speicher des vorhergehenden Ergebnisses freigegeben werden kann. Das belastet den Garbage Collector.

```
Immutable modify2(Immutable inVal) {
    Mutable m = inVal.makeMutable();
    m.add(12);
    m.multiply(3);
    m.add(10);
    m.multiply(2);
    return m.makeImmutable();
}
```

`modify2` setzt den `Mutable Companion` ein. Ab der zweiten Zeile beinhaltet jeder Verarbeitungsschritt eine Änderung des vorhandenen Objektes, so dass keine Garbage collection notwendig ist.

Der `Mutable Companion` fungiert als Hilfsklasse. In der Regel wird er nur in der Implementation einer Methode verwendet, nicht in ihrer Schnittstelle. Seine Benutzung ähnelt der des Erbauer-Musters (*Builder* [GHJV98, Kap. 3]), bei dem ebenfalls ein Ergebnis in mehreren Schritten erstellt wird und der Klient den Ablauf dieser Schritte steuert.

Ein Beispiel für einen `Mutable Companion` aus den Java-Bibliotheken ist die Klasse `StringBuilder` (sie ersetzt die zuvor für ähnliche Zwecke gebräuchliche Klasse `StringBuffer`). Sie kann als `Mutable Companion` der Klasse `String` eingesetzt werden, um String-Verarbeitungen effizienter zu gestalten.

Werteigenschaften:

Der `Mutable Companion` ist per Konstruktion änderbar und wird erzeugt, er setzt somit keine der charakteristischen Werteigenschaften um. Das Value Object, dessen Ergänzung der `Mutable Companion` darstellt, wurde bereits in Abschnitt 4.2.1 diskutiert.

Einfachheit:

Mit einem Mutable Companion gibt es zwei Klassen, die – abgesehen davon, dass die eine unveränderlich und die andere änderbar ist – denselben Typ abbilden. Der Ansatz geht also mit Codeverdopplung und den entsprechenden negativen Auswirkungen auf die Wartbarkeit einher, etwa der Gefahr von Inkonsistenzen und der Notwendigkeit doppelter Pflege. Klienten benutzen *beide* Klassen, das Value Object und seinen Mutable Companion, anders als z. B. beim Body-Handle-Idiom, bei dem für Klienten nur eine Klasse, das Handle, sichtbar ist.

Das Verfahren setzt voraus, dass ein Mutable Companion verfügbar ist. Der Programmierer des Value Object muss diese Ergänzung also vorausschauend einplanen. Prinzipiell ist es zwar möglich, dass ein Klient bei Bedarf selbst einen Mutable Companion schreibt. Der damit verbundene Aufwand bedeutet aber eine hohe Hemmschwelle, und er birgt die Gefahr, dass jeder Klient einen eigenen Mutable Companion schreibt und damit redundanten Code produziert.

Klarheit:

Es ist verwirrend, wenn ein Klient für dieselbe Abstraktion zwei verschiedene Klassen benutzt. Dass der Mutable Companion lediglich eine Hilfskonstruktion darstellt, ist dem Code nicht zu entnehmen. Um die Vorgänge im Code korrekt zu deuten, muss der Leser das Verfahren kennen und erkennen.

Wie bei den zuvor beschriebenen klientenseitigen Verfahren kommt mit dem Mutable Companion zwecks Optimierung eine änderbare Klasse zum Einsatz. Damit ist im Code nicht erkennbar, dass er dazu dient, Werte abzubilden.

Für Klienten von Klienten (im Beispiel: für Aufrufer von `modify2`) tritt der Mutable Companion nicht in Erscheinung. Für sie stellt sich das Verfahren dar wie ein Value Object.

Sicherheit:

Der Mutable Companion beruht, wie die anderen in diesem Abschnitt aufgeführten Verfahren, auf Programmierkonventionen, in diesem Fall auf der oben genannten Richtlinie, den Mutable Companion nur in der Implementation einer Operation einzusetzen und nicht in ihrer Schnittstelle. Das macht das Verfahren fehleranfällig.

Effizienz:

Der Mutable Companion hilft, speichertechnische Nachteile des Value Object-Musters zu vermeiden und den Garbage Collector zu entlasten. Die Erzeugung eines Mutable Companion zu Beginn der Verarbeitung und die Umwandlung zurück in ein Value Object bei Ende der Verarbeitung kosten aber auch Speicher und Laufzeit. Dieser zusätzliche Ressourcenverbrauch ist gegen die Ersparnis im Laufe der

Verarbeitung abzuwägen. Der Einsatz eines Mutable Companion lohnt sich in der Regel bei komplexen Verarbeitungen mit vielen Zwischenschritten.

4.4 Vergleich der Verfahren

Keines der untersuchten Verfahren führt dazu, dass alle Werteigenschaften erfüllt sind. Der überwiegende Teil der Verfahren fokussiert auf Unveränderlichkeit. Ein Grund dafür liegt in ihrer leichten Umsetzbarkeit und in ihren technischen Vorteilen: Unveränderliche Elemente können gemeinsam genutzt werden, sparen damit Speicherplatz, vermeiden Alias-Effekte und erfordern bei konkurrierendem Zugriff keine Synchronisation. Aus diesen Vorteilen wird sogar die Richtlinie abgeleitet, Klassen nach Möglichkeit unveränderlich zu machen oder wenigstens ihre Änderbarkeit auf ein „notwendiges Minimum“ einzuschränken [Blo08, p. 79]. Aus Sicht dieser Arbeit stellen die technischen Vorteile aber lediglich einen (willkommenen) Kollateralnutzen dar. Werttypen sollten als unveränderlich programmiert werden, weil sie ihrer Natur nach unveränderlich *sind* und der Quelltext so die abgebildete Domäne realitätsgetreu widerspiegelt. Die damit einhergehenden technischen Vorteile sollten nicht die ausschlaggebende Motivation bilden. Für die Klarheit des Codes wäre es nachteilig, änderbare Objekte – allein motiviert durch die technischen Vorteile – als unveränderlich zu programmieren.

Nur wenige Muster setzen Unerzeugbarkeit um – möglicherweise, weil diese Eigenschaft in der Gedankenwelt etlicher Entwickler nicht präsent ist. Auch machen objektorientierte Programmiersprachen die Umsetzung von Unerzeugbarkeit aufwändig und schwierig. Die Elemente einer Klasse müssen immer erzeugt werden, jede Klasse hat einen Konstruktor, und es ist allenfalls möglich, ihn durch geeignete Programmiertechniken vor Klienten zu verbergen.

Weil die meisten Muster nur eine (das Fliegengewicht: zwei) der Werteigenschaften umsetzen – und die klientenseitigen Verfahren keine – leidet ihre Klarheit: Bei keinem der betrachteten Verfahren ist ein Werttyp im Quelltext als solcher erkennbar.

Tabelle 4.2 fasst die in diesem Kapitel untersuchten Verfahren zur Abbildung von Werttypen zusammen und stellt sie einander gegenüber. Sie schlüsselt auf, welche Verfahren welche Werteigenschaften umsetzen, gibt eine vergleichende Übersicht über ihre softwaretechnische Qualität und über ihre technische Effizienz.

Die Tabelle lässt grob einen negativen Zusammenhang zwischen Einfachheit/ Aufwand und Sicherheit/ Fehleranfälligkeit erkennen: Aufwändig und komplex umzusetzende Verfahren dagegen (wie das Fliegengewicht oder das Body-Handle-Idiom in C++) sind zumindest sicher und ohne zusätzliche Konventionen für ihre Klienten benutzbar. Die einfach zu programmierenden Verfahren sind fehleranfälliger.

Verfahren	eingehaltene Wert-Eigenschaften			softwaretechnische Bewertung			Effizienz Laufzeit/ Speicher- verbrauch
	Unver- änderlich- keit	kein Erzeugen	referentielle Transparenz/ Seiteneffekt- freiheit	Einfach- heit	Klar- heit	Sicher- heit	
Value Object	ja	nein	nein	+	-	-	--
Fliegengewicht	ja	ja	nein	--	-	+	-
Typesafe enum	nein	ja	nein	-	-	+	+
Defensives Kopieren	Klient ^a	nein	nein	+	--	--	+
Copy-on-write	Klient ^b	nein	nein	+	--	--	+
Body-Handle	Klient ^c	nein	nein	--	--	-	++
Mutable Companion	Klient ^d	nein	nein	+	--	-	+

Tabelle 4.2: Softwaretechnische Verfahren zur Abbildung von Werten

^aBeim defensiven Kopieren sorgt der Klient nicht für Unveränderlichkeit, sondern er grenzt lediglich Änderungen auf einen Kontext ein.

^bWeil bei Copy-on-write vor jeder Änderung eine Kopie erstellt wird, ist jedes Element nach seiner Erzeugung de facto unveränderlich.

^cIn C++ kann das gewünschte Verhalten im Body und Handle programmiert werden, Klienten müssen keine Benutzungskonventionen einhalten.

^dDer Klient steuert, wann er das Value Object und wann den Mutable Companion einsetzt.

Auch zwischen softwaretechnischer Qualität und technischer Effizienz ist ein negativer Zusammenhang erkennbar. Die Klienten-seitigen Verfahren erkaufen die beabsichtigte Optimierung des Speicherbrauchs durch geringere Klarheit und Lesbarkeit sowie verminderte Sicherheit, Benutzbarkeit und Wartbarkeit.

4.5 Zusammenfassung

Kapitel 4 untersuchte, wie Werttypen mit Hilfe von softwaretechnischen Verfahren abgebildet werden können. Es analysierte verschiedene Entwurfsmuster, Idiome und Programmierkonventionen, die sich dafür eignen, Werttypen in einer objektorientierten Sprache zu programmieren und die verbreitet dafür eingesetzt werden.

Die Verfahren lassen sich in zwei Gruppen einteilen: Klassen, die so programmiert sind, dass sie aus Sicht von Klienten ein weitgehend wert-gemäßes Verhalten zeigen (dazu zählen die Muster *Value Object*, *Fliegengewicht* und das *Typesafe Enum Pattern*) und Verfahren, bei denen Klienten ein wert-ähnliches Verhalten durch Benutzungskonventionen herstellen (dazu gehören *defensives Kopieren*, das *Copy-on-write-Verfahren*, das *Body-Handle-Idiom* und der *Mutable Companion*).

Keines der betrachteten Verfahren ist darauf ausgerichtet, alle Eigenschaften von Werten abzubilden. Viele Verfahren fokussieren ausschließlich auf Unveränderlichkeit, indem sie auf ändernde Operationen verzichten oder – im Fall von Klienten-seitigen Konventionen – Änderungen auf einen Kontext begrenzen. Unerzeugbarkeit wird nur von wenigen Verfahren umgesetzt, z. B. verbirgt das Fliegengewicht die Erzeugung und lässt so die zugehörigen Elemente aus Klientensicht unerzeugbar erscheinen. Seiteneffektfreiheit und referentielle Transparenz – die Kerneigenschaften, die die Zustandslosigkeit von Werten charakterisieren – werden von keinem Verfahren thematisiert.

Aus softwaretechnischer Sicht ist die mangelnde Klarheit und Lesbarkeit der größte Nachteil und der gemeinsame Nenner aller Verfahren: Die – ohnehin nur partiell umgesetzten – Werteigenschaften beruhen auf Konventionen und Restriktionen, Werttypen sind im Quelltext nicht als solche erkennbar. Das macht die Verfahren unsicher, die Werteigenschaften können bei Codeüberarbeitung leicht zerstört werden. Besonders fragil sind Verfahren, bei denen die betreffende Klasse aus Effizienzgründen ändernde Operationen zulässt und die Änderungen durch Benutzungskonventionen auf einen Kontext eingrenzt. Weil hier die Verantwortung für die Werteigenschaften bei den Klienten liegt, sind sie besonders fehleranfällig.

Kapitel 5

Unterstützung für Werte in bestehenden Programmiersprachen

In Programmiersprachen finden sich einige Konstrukte, die als Werttypen einzuordnen sind oder die einzelne Werteigenschaften sprachlich abbilden. Diese Sprachkonstrukte sind Untersuchungsgegenstand dieses Kapitels. Es befasst sich mit der Frage, ob bestehende objektorientierte Programmiersprachen Werttypen unterstützen, in welchem Umfang und wie. Es untersucht eine Reihe von Sprachkonstrukten, die gemäß der Definition von Kapitel 3 als Werttypen einzustufen sind oder mit deren Hilfe Werttypen definiert werden können. Des weiteren betrachtet es Sprachkonstrukte, die einzelne Werteigenschaften sprachlich sicherstellen und diskutiert ihre Möglichkeiten und Grenzen.

Zur Abgrenzung betrachtet dieses Kapitel auch einige Sprachmittel, die die erforderlichen Werteigenschaften weder vollständig noch partiell unterstützen, die aber oberflächliche Ähnlichkeiten mit Werttypen aufweisen und leicht mit ihnen verwechselt werden können.

5.1 Untersuchte Sprachen

Entsprechend der Ausrichtung dieser Arbeit konzentriert sich die Untersuchung auf objektorientierte, klassenbasierte, statisch getypte Sprachen. Die ausgewählten Sprachen bilden einen repräsentativen Querschnitt aktuell verwendeter oder im Rahmen von Forschungsprojekten entworfener Sprachen. Welche Sprachen einbezogen werden, richtet sich in erster Linie danach, ob sie über Konstrukte verfügen, die Werttypen unterstützen oder Ähnlichkeit mit Werttypen aufweisen.

Die Auswahl umfasst industriell eingesetzte Sprachen wie Java, C#, C++ und D („Mainstream-Sprachen“), Sprachen aus dem akademischen Umfeld, wie Eiffel, Sather, X10, Fortress und Blue („Forschungssprachen“), sowie, als Vertreter der objekt-funktionalen Sprachen, Scala.

Einige der genannten Sprachen wurden bereits in den Abschnitten 2.2.2 und 2.5 vorgestellt. Über die weiteren gibt die Aufstellung auf Seite 85 einen Überblick.

Die Untersuchung berücksichtigt keine dynamisch getypten Sprachen (wie Smalltalk oder Groovy). Ein dynamisches Typsystem erlaubt es nicht, Werttypen und Objekttypen im Quelltext voneinander zu unterscheiden (s. dazu auch Abschnitt 2.6). Aus dem gleichen Grund wurden auch keine sogenannten Scriptsprachen (wie Perl, Ruby, Python oder JavaScript), die ebenfalls dynamisch getypt sind, einbezogen und keine prototyp-basierten Sprachen, die darüber hinaus auf Klassen verzichten und ausschließlich Objekte unterstützen (wie Self oder Lua).

Funktionale Sprachen sind ebenfalls nicht Bestandteil der Untersuchung. Funktionale Sprachen sind zwar inhärent wert-orientiert, ihnen fehlen aber Mechanismen zur Abbildung von Objekten und änderbarem Zustand. Die Arbeit zielt darauf ab, Werte in ein objektorientiertes Umfeld zu integrieren. Eine isolierte Betrachtung von Werten und wert-orientierter Programmierung ist darum nicht zielführend.

Im Vordergrund der folgenden Ausführungen stehen solche Sprachkonstrukte, die de facto Werttypen sind oder zumindest eine oder mehrere Werteigenschaften unterstützen. Ein vollständiger Überblick über die genannten Sprachen ist nicht beabsichtigt.

5.2 Untersuchungskriterien

Eine umfassende Sprachunterstützung für Werttypen erfüllt zwei Kriterien, die vollständige Umsetzung der Werteigenschaften und Flexibilität.

Vollständige Umsetzung der Werteigenschaften: Ein mit Hilfe der Sprachunterstützung definierter Werttyp weist *alle* charakteristischen Eigenschaften von Werttypen (s. Definition in Abschnitt 3.3) auf, und die Sprache stellt ihre Einhaltung sicher.

Flexibilität: Die Sprachunterstützung ermöglicht dem Anwendungsprogrammierer, neue, den fachlichen Anforderungen entsprechende Werttypen zu definieren, und sie ist für alle Arten von Werttypen einsetzbar, nicht nur für eine eingeschränkte Gruppe. Beispielsweise soll die Sprachunterstützung flexibel genug sein, um so verschiedene Typen wie Jahreszeit, Datum oder Zeichenkette zu definieren.

Im Folgenden wird untersucht, inwieweit die genannten Kriterien für die jeweils betrachteten Sprachkonstrukte erfüllt sind.

Weitere in dieser Arbeit untersuchte Sprachen

Sather [SO96] entstand 1990 als Forschungssprache und wurde von einem Team unter der Führung von Steve Omohundro an der University of California, Berkeley, entwickelt. Sather basiert auf Eiffel, mittlerweile gibt es verschiedene Varianten, u. a. eine Weiterentwicklung der Universität Karlsruhe (Sather-K). Sather unterstützt Unveränderlichkeit auf Typebene („immutable classes“) und wurde aus diesem Grunde in die Untersuchung einbezogen.

Blue wurde 1997 von Michael Kölling für Lehrzwecke entworfen [KR97; Kö199]. Aus dem Blickwinkel dieser Arbeit ist die Sprache Blue relevant, weil sie ein eigenständiges Konzept für Typen mit einer festen Elementemenge und damit für Unerzeugbarkeit enthält.

X10 [Sar+12] ist eine 2004 vom IBM Thomas J. Watson Research Center entworfene Forschungssprache, die auf „High Productivity Computing“ und parallele Programmierung ausgerichtet wurde. Sie ist in dieser Arbeit u. a. wegen ihres Typkonstruktors „struct“ relevant, mit dem Typen mit unveränderlichen Elementen definiert werden können.

Fortress [All+08] wurde von einem Entwicklungsteam bei SUN Microsystems entworfen, maßgeblich beeinflusst von Guy Steele (der auch an der Entwicklung von Java beteiligt war). Die Sprache war als Nachfolger von Fortran geplant und schwerpunktmäßig für den technisch-wissenschaftliche Einsatz gedacht. Die Weiterentwicklung von Fortress wurde 2012 eingestellt. Fortress trennt (sogenannte) „value objects“ von „reference objects“ und wurde deshalb in die Untersuchung mit einbezogen.

Die Programmiersprache **D** [Ale10], 2006 entwickelt von W. Bright und A. Alexandrescu, ist ähnlich C++ auf systemnahe Programmierung und hohe Performance ausgerichtet und verfügt ebenfalls über ein speichernahes Sprachmodell. Die Sprache enthält sowohl objektorientierte als auch funktionale Elemente, u. a. ermöglicht sie die Definition von pure functions.

5.3 Sprachseitig vorgegebene Werttypen

5.3.1 Basisdatentypen

Die meisten Programmiersprachen enthalten eine Menge von „einfachen“ Datentypen, in der Regel Typen für Zahlen, Zeichen und bool'sche Werte. Beispielsweise definiert Java die acht Datentypen `boolean`, `byte`, `short`, `int`, `long`, `char`, `float` und `double`. In C# gibt es darüber hinaus Typen für vorzeichenlose Zahlen wie `uint` (=unsigned int). Auch C++ und D enthalten solche Typen. Eiffel kennt `INTEGER`, `REAL`, `CHARACTER` and `BOOLEAN`, sowie (in der Sprachspezifikation von 2006) einige „sized variants“ davon.

Java und C# bezeichnen derartige Typen als *primitive Typen* (primitive types), denn sie können – aus Sicht des Anwendungsprogrammierers – nicht weiter zerlegt werden. C++ oder Eiffel führen sie unter dem Begriff *Basisdatentypen* (basic types oder fundamental types), weil sie die Grundbausteine bilden, aus denen weitere, komplexere Typen zusammengesetzt werden. Wieder andere Sprachen, z. B. Sather, nennen solche Typen *vordefiniert* (built-in), da sie durch die Sprachspezifikation vorgegeben und nicht in einer Code-Bibliothek definiert sind. Zur sprachlichen Vereinfachung werden derartige Typen ab hier durchgängig als „Basisdatentypen“ bezeichnet.

Bereits der uneinheitliche Sprachgebrauch weist darauf hin, dass es sich bei den Basisdatentypen nicht um ein homogenes Konzept handelt. In einigen Sprachen stehen solche Typen außerhalb der Typhierarchie (z. B. in Java), in anderen sind sie in diese eingebunden (z. B. in Scala). Manche Sprachen legen die zu den Basisdatentypen zugehörigen Operationen in der Sprachspezifikation fest (z. B. Java), andere definieren sie ganz oder teilweise in Systembibliotheken (z. B. C# im Modul `System` oder X10 im package `x10.lang`. X10 nimmt aufgrund dessen sogar in Anspruch, keine „primitive classes“ zu haben, ähnliches gilt für Scala und Eiffel.)

Ungeachtet dieser Unterschiede kennzeichnet alle Basisdatentypen ein gemeinsames Merkmal, das sie gegenüber anderen, programmierer-definierbaren Typen abgrenzt und eine Sonderstellung begründet, nämlich ihre Verankerung in der jeweiligen Programmiersprache:

- Basisdatentypen sind in der Sprachspezifikation festgelegt. Ihre Namen sind reservierte Schlüsselworte der Sprache.
- Basisdatentypen verfügen oft über eine spezielle Syntax, die sich von der für programmierer-definierte Typen unterscheidet:
 - Ihre Exemplare werden über *Literale* angesprochen (Tokens wie `17`, `-3.14e2`, `0xFFFF`, oder `'a'`). Umfang und Syntax der zulässigen Literale legt die Sprachspezifikation fest.

- Die Operationen von Basisdatentypen werden syntaktisch meist über (Prefix- oder Infix-) Operatoren wie $+$, $-$, $*$, $<$ oder $>>>$ aufgerufen, auch diese gehören zum Sprachumfang.
- Ob und welche *Konvertierungen* zwischen diesen Typen zulässig sind, ob sie implizit oder explizit erfolgen, legt ebenfalls die Sprachspezifikation fest (z. B. widening oder narrowing conversions in Java).
- Oft gelten für Basisdatentypen *semantische Besonderheiten*, die für programmierer-definierte Typen nicht definiert werden können, z. B. die short-circuit-Ausführung für die bool'schen Operatoren $\&\&$ und $\|\|$, wie sie in Java vorgesehen sind.

Basisdatentypen sind Werttypen

Basisdatentypen weisen alle Merkmale von Werttypen auf, bedingt durch das Verhalten ihrer zugehörigen Operatoren und Literale:

- Die Elemente der Basisdatentypen sind *unveränderlich*. Die vordefinierten Operationen verändern nicht ihre Operanden, sondern sie liefern einen – von den Operanden abhängigen – Ergebniswert.
- Die Elemente der Basisdatentypen werden *nicht erzeugt*. Ein Literal, z. B. 27, 3.14 oder 'a', bezeichnet immer denselben, bereits vorhandenen Wert.
- Die Operationen der Basisdatentypen sind *frei von Seiteneffekten* und *referentiell transparent*. Z. B. verändert die Addition zweier Zahlen weder ihre Summanden noch andere Elemente, und sie liefert bei gleichen Eingaben immer das gleiche Ergebnis.

Die besonderen Regeln, denen Basisdatentypen unterliegen, und die Tatsache, dass fast alle objektorientierten (und auch prozeduralen) Sprachen über derartige Typen verfügen, können als Hinweis darauf interpretiert werden, dass auch imperative Sprachen nicht ohne Werttypen und wert-orientierte Verarbeitung auskommen.

Basisdatentypen sind nicht frei definierbar

Die Menge der verfügbaren Basisdatentypen ist durch die Sprachspezifikation festgelegt. Der Anwendungsprogrammierer kann keine weiteren derartigen Typen definieren.

Besonderheiten

Einige Merkmale von Basisdatentypen – Literale, Operatoren – finden sich vereinzelt auch bei programmierer-definierbaren Typen, was die Trennlinie zwischen beiden verwischt.

Z. B. verfügt der Typ `String` in Java über *Literale*, und Strings können mit dem Operator „+“ konkateniert werden. `String` ist aber als Klasse definiert (in den

Sprachbibliotheken, im Package `java.lang`). Das lässt `String` eine Zwitterstellung einnehmen.

C++ erlaubt, die durch die Sprache vorgegebenen Operatoren für beliebige Typen zu definieren („überladen“). Scala lässt ebenfalls eine Operator-Schreibweise für beliebige Methoden zu und interpretiert sie lediglich als eine syntaktische Sonderform eines Methodenaufrufs.

In C++ gelten auch Zuweisung (`=`), increment (`++`), kombinierte Zuweisungen (wie `+=`) oder `<<` („put to“) als Operatoren. Deshalb können Operatoren hier ihre Operanden ändern oder sonstige Seiteneffekte bewirken. Grund dafür ist das Sprachmodell von C++, es unterscheidet nicht zwischen der Änderung einer Variablen und der Änderung ihres zugehörigen Elementes.

5.3.2 Aufzählungstypen

Viele objektorientierte Sprachen unterstützen Aufzählungstypen durch einen dedizierten Typkonstruktor, z. B. C#, C++ und D, sie werden dort mit dem Schlüsselwort `enum` eingeleitet. Auch etliche prozedurale Sprachen, z. B. Pascal, ANSI-C und Ada, enthalten eine eigene Sprachunterstützung für Aufzählungen.

In der Definition eines Aufzählungstyp werden Bezeichner für die zugehörigen Elemente explizit aufgeführt. Beispielsweise definiert folgender Code in C# einen Aufzählungstyp namens `Season` mit vier Elementen:

```
enum Season { Spring, Summer, Autumn, Winter }
```

Aufzählungstypen werden vorwiegend zur Programmierung von Mehrfachverzweigungen benutzt, deren möglichen Alternativen bereits zur Übersetzungszeit bekannt sind (in Java oder C# entsprechen sie den `switch`-Anweisungen). Anders als Ganzzahlkonstanten, die ebenfalls manchmal für diesen Zweck benutzt werden, ist die Verwendung von Aufzählungstypen typsicher, weniger fehleranfällig und sie erhöht die Lesbarkeit und Selbst-Dokumentation des Codes (s. auch Abschnitt 4.2.3). Aufzählungstypen können durch die IDE unterstützt werden, was ihre Handhabung weiter erleichtert, z. B. durch das Vorschlagen oder Auto-Completion der einzelnen Elemente, oder die automatische Generierung der gesamten Verzweigung (s. a. CSharp Language Specification [ECM06a, Kap 8.11]).

Aufzählungstypen sind Werttypen

Aufzählungstypen sind in den o. g. Sprachen Werttypen:

- Die Elemente von Aufzählungstypen sind per Konstruktion *unerzeugbar*. Sie sind durch die Definition des Typs festgelegt, es können keine weiteren Elemente zur Laufzeit erzeugt werden.

- Die für Aufzählungstypen verfügbaren Operationen sind durch die Sprache vorgegeben. Im Wesentlichen handelt es sich, je nach Sprache, um Gleichheit und Ungleichheit (Operatoren `==` und `!=`), Kleiner/ Größer-Relationen (`<`, `<=`, `>`, `>=`), Addition/ Subtraktion (`+`, `-`) und Konvertierungen von und zu Ganzzahltypen, teils explizit, teils implizit. Alle verfügbaren Operationen sind frei von Seiteneffekten und referentiell transparent (sie liefern für gleiche Operanden immer das gleiche Ergebnis).

Für Aufzählungstypen in den o. g. Sprachen können keine Datenfeldern definiert werden. Damit sind auch *keine Änderung* ihrer Elemente möglich. Zusammen mit dem Verhalten der verfügbaren Operationen ergibt sich daraus die Einhaltung aller Werteigenschaften.

Java unterstützt ebenfalls Aufzählungstypen, dort sind sie aber nicht notwendigerweise Werttypen. Aufzählungstypen wurden ab Version 1.5 in Form des Typkonstruktors `enum` in den Sprachumfang von Java aufgenommen. Sie ermöglichen die Definition beliebiger Datenfelder und Methoden [Gos+14, Kap. 8.9]. Damit können Aufzählungen über einen Zustand verfügen und über Operationen, die ihn ändern. Die Sprache gewährleistet lediglich, dass zur Laufzeit keine weiteren Elemente erzeugt werden. Sie stellt weder die Unveränderlichkeit der Elemente noch die Seiteneffektfreiheit und referentielle Transparenz der Operationen einer `enum` sicher.

Einschränkungen von Aufzählungstypen

Aufzählungstypen eignen sich nur für Werttypen mit einer überschaubaren Menge von Elementen und einfacher Verarbeitungslogik, wie „Season“. Sie sind nicht hinreichend flexibel für die Unterstützung beliebiger Werttypen. Komplexer aufgebaute Werttypen wie „Datum“ oder „Zeichenkette“ können damit nicht definiert werden.

Besonderheiten

Aufzählungstypen in C# und C++ sind *Abwandlungen von Ganzzahltypen*. Das macht sich dadurch bemerkbar, dass zu Ganzzahlen gehörige Operationen – z. B. Addition oder Kleiner/Größer-Relation – auch für Aufzählungen verfügbar sind, obwohl sie dort fachlich nicht immer einen Sinn ergeben.

In C# sind die Elemente eines Aufzählungstyps *nicht* auf die explizit definierten Konstanten eingeschränkt. Jedes Element des zugrunde liegenden Ganzzahltyps (standardmäßig: `int`) hat ein Gegenstück im Aufzählungstyp. Eine Anweisung wie

```
Season season = Season.Summer + Season.Winter
```

ist zulässig. Sie liefert das Element von `Season`, das in seinem zugrunde liegenden Ganzzahltyp durch den Wert 4 codiert wird (`Spring` entspricht 0, `Summer` 1, `Autumn` 2, `Winter` 3, s. dazu [Mic10, Kap 1.10]). Trotz dieser ungewöhnlichen

Semantik sind die Aufzählungstypen von C# Werttypen. Ihre Benutzung ist auch typsicher; es ist z. B. nicht ohne explizite Konvertierung möglich, einen Wert des zugrunde liegenden Ganzzahltyps anstelle eines Elementes von `Season` zu verwenden. Für einen Anwendungsprogrammierer, der erwartet, dass der Typ `Season` nur die vier explizit definierten Elemente hat, kann das beschriebene Verhalten aber zu unerwarteten Effekten führen.

5.3.3 Ergebnis

In objektorientierten Programmiersprachen gibt es nur wenige Sprachkonstrukte, die Werttypen darstellen oder ihre Definition unterstützen, d. h. Sprachkonstrukte, die *alle* Eigenschaften von Werttypen aufweisen: Basisdatentypen und (mit Ausnahme von Java) Aufzählungstypen. Beide stellen keine flexible Unterstützung für beliebige Werttypen dar. Basisdatentypen können nicht vom Programmierer definiert werden, und eine Definition mittels Aufzählung ist nur für eine eingeschränkte Gruppe von Werttypen praktikabel.

Tabelle 5.1 fasst die Ergebnisse von Abschnitt 5.3 zusammen.

5.4 Partielle Unterstützung von Werteigenschaften

Mehrere Arbeiten setzen sich damit auseinander, wie Werttypen, die über einfache Fälle wie Basisdatentypen oder Aufzählungen hinausgehen, mit den Mitteln von Programmiersprachen abgebildet werden können.

Winkler untersucht diese Fragestellung in bezug auf C++ [Win10b]. Er kommt zu dem Ergebnis, dass zwar durch verschiedene Sprachmechanismen (wie `const correctness`, geeignete Parameterübergabemechanismen und die C++ inhärente Kopiersemantik) die Umsetzung eines wert-gemäßen Verhaltens möglich ist. Es erfordert aber einen hohen Schreibaufwand, und es belastet den Code mit „semantisch irrelevanten Effizienzdetails“ [Win10b, p. 61]. Im Ergebnis ist „... eine saubere Modellierung von Werten ... in C++ somit nicht realisierbar“ [Win10b, p. 65].

Witt analysiert mit gleicher Zielsetzung die objekt-funktionale Sprache Scala [Wit14]. Aufgrund ihrer funktionalen Wurzeln lässt sie Unterstützung für Werte und Zustandslosigkeit erwarten. Auch hier konnten zwar einzelne Sprachmittel identifiziert werden, die das Verhalten einer Klasse dem eines Werttyps annähern (z. B. `Case Classes`), oder solche, die partielle, teils auch nur oberflächliche Ähnlichkeiten mit Werttypen aufweisen (z. B. eine Operatorschreibweise, oder sogenannte `Value Classes`, die speichertechnische Optimierungen erlauben oder die Typsicherheit erhöhen können). Die kennzeichnenden Merkmale von Werttypen (Seiteneffektfreiheit und referentielle Transparenz) unterstützt die Sprache aber nicht, so dass im Ergeb-

Sprachkonstrukt	vorhanden in	Unver- änder- lichkeit	kein Erzeugen	Seiteneffekt- freiheit / referentielle Transparenz	flexible Definition beliebiger Typen
Basisdatentypen	Java, C#, C++, Eiffel, X10, ...	ja	ja	ja	nein (festgelegt durch die Sprache, keine Definition weiterer Basisdatentypen möglich)
Aufzählungstypen	C#, C++	ja	ja	ja	nein (nur für Typen mit wenigen Elementen praktikabel)
Aufzählungstypen	Java	nein	ja	nein	nein (nur für Typen mit wenigen Elementen praktikabel)

Tabelle 5.1: Sprachseitig unterstützte Werttypen in objektorientierten Sprachen

nis „in Scala bei der Umsetzung von Werttypen die charakteristischen Eigenschaften nicht garantiert ... werden“ [Wit14, p. 50].

Um einen breit gefächerten Überblick über die verfügbaren Möglichkeiten zur Abbildung von „Wert-Artigkeit“ zu geben, werden im Folgenden aus etlichen verschiedenen Sprachen die unter diesen Gesichtspunkten aussichtsreichsten Konstrukte ausgewählt, d. h. solche, die eine oder mehrere der geforderten Werteigenschaften sicherstellen. Soweit mehrere Sprachen gleiche oder ähnliche Konstrukte aufweisen, werden sie zusammengefasst und gemeinsam untersucht.

5.4.1 Unveränderlichkeit

Unveränderlichkeit von Variablen

Viele Sprachen unterstützen Unveränderlichkeit auf der Ebene von Variablen. Java kennzeichnet unveränderliche Variablen mit dem Modifier `final`, C++ mit dem Schlüsselwort `const`¹, D mit dem Qualifier `immutable`². Scala und X10 definieren unveränderliche Variablen mit `val`, änderbare dagegen mit `var`. Fortress drückt den Unterschied durch die Symbole aus, mit denen eine Variable initialisiert wird, unveränderliche Variablen benutzen dafür „=“, änderbare „:=“. Der Programmierer kann so bei der Definition einer Variablen festlegen, dass sie nach ihrer Initialisierung nicht mehr änderbar ist und diesen Umstand im Quelltext auch dokumentieren. Das macht den Code sicherer und verbessert seine Lesbarkeit.

Die weite Verbreitung solcher Sprachmechanismen lässt erkennen, dass ein Bedarf nach sprachlicher Unterstützung von Unveränderlichkeit vorhanden ist. Unveränderliche Variablen unterstützen aber keine Werttypen, sie sind sogar orthogonal zu ihnen: Variablen von einem Objekttyp können als unveränderlich definiert werden, umgekehrt können die Variablen von Werttypen änderbar sein (s. dazu auch Abschnitt 3.3.8).³

Unveränderliche Typen

Unveränderlichkeit auf Typ-Ebene wird seltener unterstützt. Sie findet sich vorwiegend in Sprachen aus dem akademischen Umfeld. Z. B. führt Sather mit „immutable classes“ einen Sprachmechanismus ein, der die Unveränderlichkeit aller Elemente

¹`const` markiert in C++ lediglich eine lokale Unveränderlichkeit. Seit C++11 gibt es das Schlüsselwort `constexpr`, damit kann eine Variable als statische Konstante, also als global unveränderlich definiert werden. S. auch Abschnitt 5.4.4.

²D verwendet `immutable` auch als Typkonstruktor; er bewirkt, dass sich die Unveränderlichkeit transitiv auf die Bestandteile fortsetzt.

³Unveränderliche Variablen können immerhin die Programmierung von Werttypen erleichtern. Z. B. kann in Java der Programmierer alle Datenfelder einer Klasse als `final` deklarieren und damit zum Ausdruck bringen, dass Änderungen an den Elementen dieser Klasse nicht möglich sein sollen (zu den Vor- und Nachteilen dieses Verfahrens s. Abschnitt 4.2.1).

garantiert: Wird in Sather eine Klasse als „immutable class“ definiert (statt als „class“), so sind damit alle ihre Datenfelder (im Sather-Sprachgebrauch: Attribute) unveränderlich.

Fortress und X10 enthalten ähnliche Typkonstruktoren. Fortress erlaubt die Definition von sogenannten „value objects“, die – im Gegensatz zu „reference objects“ – unveränderlich sind. In X10 beschreibt der Typkonstruktor `struct` unveränderliche Objekte, `class` dagegen veränderliche.⁴

Der zusätzliche Typkonstruktor von Sather (und analog auch der von Fortress und der von X10) kann als ein Schritt in Richtung Werttypen interpretiert werden. Er stellt mit Unveränderlichkeit eine essentielle Eigenschaft von Werttypen sicher.

Immutable classes sind aber keine Werttypen, da ihnen die anderen drei Charakteristika fehlen. Die Elemente dieser Typen werden erzeugt. Die Typen ihrer Datenfelder können änderbare Typen („reference classes“) sein, und ihre Operationen können grundsätzlich auf andere, änderbare Objekte zugreifen. Beides führt dazu, dass für diese Typen Seiteneffektfreiheit und referentielle Transparenz nicht sichergestellt ist. Aufgrund dessen bilden die unveränderlichen Typen auch kein geschlossenes Teilsystem, wie es bei Werttypen der Fall ist (s. Kap. 3).⁵

Ähnlich wie unveränderliche Variablen können auch unveränderliche Typen die Programmierung von Werttypen erleichtern; die nicht von der Sprache unterstützten Werteigenschaften erfordern dann entsprechende Programmierdisziplin. Einige Werttypen, z. B. komplexe Zahlen, sind in Sather-Bibliotheken als immutable classes implementiert (s. auch Abschnitt 5.5.1). Auch in X10 sind etliche Typen, die in anderen Sprachen Basisdatentypen und damit Teil der Sprache sind, in Systembibliotheken mit Hilfe von `struct` programmiert, z. B. `Int`, `Float` oder `UInt` (s. Abschnitt 5.3.1).

Weil immutable classes (ebenso structs und value objects) speichertechnischen Restriktionen unterliegen (s. Abschnitt 5.5.1), sind sie nicht flexibel genug, um beliebige Typen zu programmieren.

5.4.2 Keine Erzeugung

Anders als Unveränderlichkeit ist Unerzeugbarkeit auf Sprachebene wenig präsent. Sie ist zwar inhärentes Merkmal von Basisdatentypen (s. Abschnitt 5.3.1), in den meisten Sprachen auch von Aufzählungstypen (s. Abschnitt 5.3.2). Darüber hinaus

⁴Eine frühere Arbeit über X10 von Charles et al. [Cha+05] bezeichnet unveränderliche Typen noch als „value classes“, erst in einer späteren Fassung von X10 wurden sie in `struct` umbenannt.

⁵Riehle konstatiert – ausgehend von einem anderen Wert-Konzept, welches sich ausschließlich auf die Unveränderlichkeit der Elemente stützt – dass X10 und Fortress zu einer neuen Sorte objektorientierter Sprachen gehören, welche Werttypen sprachseitig unterstützen [Rie06, p. 6]. Legt man das Werttyp-Konzept von Kap. 3 zugrunde, so trifft diese Aussage nicht zu.

wird sie aber nur in geringem Umfang als relevante Eigenschaft von Typen angesehen und sprachseitig unterstützt.

Eine der wenigen Sprachen, die Nicht-Erzeugung thematisiert und im Typsystem verankert, ist *Blue*, 1997 von Kölling für Lehrzwecke entworfen [KR97; Köl99]. Blue unterscheidet *manifest classes* und *dynamic classes*. Alle Elemente einer manifest class sind mit der Definition der Klasse und damit zur Entwicklungszeit bekannt. Sie brauchen nicht, und können nicht, zur Laufzeit erzeugt werden [KR97, Kap. 3.1]. Eine dynamic class entspricht dem Typkonstruktor `class` in Sprachen wie Java, Eiffel oder C#. Ihre Elemente müssen explizit erzeugt werden (in Blue mit dem Schlüsselwort `create`),

Manifest classes stellen nur die Unerzeugbarkeit ihrer Elemente sicher. Andere Wert-Eigenschaften (Unveränderlichkeit, Seiteneffektfreiheit und referentielle Transparenz der Operationen) sind gemäß Spezifikation nicht ausdrücklich verlangt.

Manifest classes sind *de facto* Werttypen, da sie in Blue nur vordefinierten Typen und Aufzählungstypen umfassen. Die vordefinierten Typen – `Integer`, `Real`, `Boolean` und `String` – erfüllen alle Werteigenschaften. Für Aufzählungstypen sind nur Operationen vergleichbar denen von C# und C++ verfügbar (s. Abschnitt 5.3.2), so dass auch ihre Seiteneffektfreiheit und referentielle Transparenz sichergestellt ist.

Blue prägte mit „manifest classes“ einen neuen Begriff und machte darauf aufmerksam, dass eine feste Wertemenge eine relevante Eigenschaft von Typen darstellt. Die in Blue unterstützten manifest classes (Basisdatentypen und Aufzählungstypen) gehen nicht über die in Abschnitt 5.3 behandelten Sprachkonstrukte hinaus, die auch in etlichen anderen objektorientierten Sprachen vorhanden sind. Komplexere Typen wie „Datum“ oder „Zeichenkette“ können in Blue nicht als manifest class programmiert werden.

5.4.3 Gleichheit

Das Konzept von Werttypen ist eng mit dem Konzept von Gleichheit verknüpft (s. Abschnitt 3.3.10): Das Vorhandensein von Gleichheit ist Voraussetzung für die referentielle Transparenz und damit für wert-konformes Verhalten.

Einige Programmiersprachen unterstützen für spezielle Typen eine „wert-artige“ Gleichheit und rücken sie damit ein Stück näher an Werttypen.

Case Classes in Scala

Ein Beispiel dafür sind die „case classes“ von Scala. Ist eine Klasse in Scala als case class definiert, dann generiert die Sprachumgebung eine Implementation der Methode `equals`, die alle Felder („Klassenparameter“) paarweise miteinander ver-

gleich `[OSV11, Kap. 15.1]`. Passend dazu werden auch die Methoden `hashCode`, `toString` und `copy` generiert.

Weil in Scala der Operator „`==`“ intern die Methode `equals` aufruft, ergibt sich aus Klientensicht für case classes ein wert-artiges Verhalten: Beispielsweise erscheinen die Elemente einer case class `Datum` gleich, wenn sie den gleichen Inhalt („den gleichen Wert“) haben.

Die Ähnlichkeit von case classes mit Werttypen wird dadurch weiter verstärkt, dass ihre Elemente standardmäßig unveränderlich sind – die Datenfelder („Klassenparameter“) einer case class sind implizit als `val` definiert. Des weiteren kann bei einer case class die Erzeugung eines Objektes ohne das Schlüsselwort `new` aufgerufen werden (ermöglicht wird das durch die Generierung einer geeigneten `apply`-Methode im Companion-Objekt der case class). Dadurch erscheinen ihre Elemente aus Klientensicht unerzeugbar. Case classes wurden aber nicht zur Abbildung von Werttypen in die Sprache Java aufgenommen, sondern zur Unterstützung von Pattern matching `[OSV11, Kap. 15]`.

Case classes ermöglichen, mit geringem Aufwand eine Klasse zu programmieren, die sich wie ein Werttyp verhält. Das beschriebene Verhalten stellt lediglich einen von der Sprache vorgegebenen Standard dar, von dem der Programmierer einer Case Class auch abweichen kann. Die Klassenparameter können mit `var` statt mit `val` definiert werden, wodurch die Elemente einer Case Class änderbar werden. Das Verhalten des Operators `==` wird durch die Methode `equals` bestimmt, der Programmierer kann somit eine Gleichheit festlegen, die im Widerspruch zu einem wert-gemäßen Verhalten steht. Auch ist für die Elemente von Case Classes ein Vergleich mit der (in Scala für alle `AnyRef`-Typen verfügbaren) Methode `eq` prinzipiell möglich. `eq` unterscheidet verschiedene Objekte auch dann, wenn sie den gleichen Wert repräsentieren und zerstört dadurch das wert-konforme Verhalten der case class (s. dazu auch die Analyse von Witt `[Wit14, Kap. 5.1]`). Aus diesen Gründen sind Case Classes im Allgemeinen keine Werttypen.

Gleichheit bei unveränderlichen Typen

Die in Abschnitt 5.4.1 beschriebenen unveränderlichen Typen in X10 („`struct`“) und in Fortress („`value object`“) verfügen über einen sprach-generierten Vergleichsoperator, der paarweise ihre Datenfelder vergleicht und damit dem Verhalten von `equals` bei case classes in Scala entspricht. Das gilt ebenfalls in Sather für immutable classes, soweit diese für den Vergleich die in der Klasse `SYS` vordefinierte Klassenmethode `is_eq` verwenden. Durch Überschreiben der Exemplarmethode `is_eq` kann der Programmierer ein anderes Verhalten für den Vergleichsoperator „`==`“ implementieren und so vom vorgegebenen Standard abweichen.

Aufgrund der sprach-generierten Gleichheit verfügen die oben genannten unveränderlichen Typen über ein weiteres charakteristisches Merkmal von Werttypen. Dennoch sind diese Typen keine Werttypen, die Gründe dafür wurden in Abschnitt 5.4.1 erläutert.

Gleichheit für struct und expanded types

Auch für Typen wie `expanded types` in Eiffel und `struct` in C# und D generiert die Sprache eine auf den Datenfeldern beruhende und damit wert-gemäße Gleichheit. Diese – prinzipiell veränderlichen – Typen werden mit Wertsemantik umgesetzt und in Abschnitt 5.5.1 behandelt.

5.4.4 Seiteneffektfreiheit und referentielle Transparenz

In den meisten imperativen und damit auch in objektorientierten Sprachen kann grundsätzlich jede Operation Seiteneffekte bewirken oder referentiell opak sein. Nur wenige objektorientierte Sprachen unterstützen die Kerneigenschaften, die die Zustandslosigkeit von Werten begründen: Seiteneffektfreiheit und referentielle Transparenz.

Pure functions

Die Programmiersprache D enthält das Konzept der „pure functions“ [Ale10, Kap. 5.11.1] und damit einen Ansatz, die Zustandslosigkeit von Operationen zu unterstützen. Eine mit dem Modifier `pure` definierte „Funktion“ darf außer über ihre Parameter nicht auf änderbaren Zustand zugreifen. Der Compiler stellt sicher, dass eine pure function keine „impuren Funktionen“ aufruft und dass sie weder lesend noch schreibend auf änderbare Variablen außerhalb der Funktion zugreift.

Eine als `pure` definierte Funktion ist nicht notwendigerweise eine Funktion im Sinne von Abschnitt 3.3. In D kann auch eine mit dem Modifier `pure` definierte Operation auf änderbaren Zustand zugreifen, nämlich über ihre Parameter. Diese dürfen prinzipiell änderbar sein (z. B. Pointer oder Arrays). In solchen Fällen wird die Operation als „weakly pure“ bezeichnet. Nur wenn zusätzlich alle Eingabeparameter `immutable` sind (s. auch Abschnitt 5.4.1), ist die referentielle Transparenz und Seiteneffektfreiheit der Operation gewährleistet. Dann ist sie eine Funktion im Sinne der funktionalen Programmierung und wird als „strongly pure“ bezeichnet [Ale14].

Die Einführung von pure functions in die – weitgehend imperative – Sprache D ist in erster Linie durch die zu erwartenden Effizienzvorteile motiviert: Die garantierte Abwesenheit von Änderungen in Teilbereichen des Systems erlaubt zahlreiche Compiler-Optimierungen.

Strongly pure functions in D kommen Funktionen (im Sinne von Abschnitt 3.2) nahe. Sie sind aber nicht direkt mit Wertoperationen vergleichbar. D verlangt nicht, dass die Eingabe und Ausgabe von pure functions auf Werte beschränkt wird. Das Sprachmodell von D ist speichernah, ähnlich dem von C oder C++. Werte als abstraktes Konzept im Sinne von Kap. 3 kommen darin nicht vor.

Strongly pure functions in D stellen Zustandslosigkeit auf der Ebene einzelner Operationen sicher, nicht auf Typebene. Das Sprachmodell von D würde es erlauben, für eine Klasse sowohl pure (zustandslose) als auch impure (zustandsverarbeitende) Operationen zu definieren. Mit der Trennung von zustandslosen Werttypen und zustandsbehafteten Objekttypen, die dieser Arbeit zugrunde liegt, ist das nicht vereinbar.

Eine Unterstützung für pure functions wurde auch für C++ vorgeschlagen [Bro13]. In C++11 wurde das Schlüsselwort `constexpr` aufgenommen [Str14, p. 9]. Hintergrund ist auch hier in erster Linie technische Optimierung: als `constexpr` markierte Variablen sowie Aufrufe entsprechender Operationen können bereits zur Übersetzungszeit ausgewertet werden. `constexpr` kann aber auch dazu genutzt werden, eine Operation als seiteneffektfrei und referentiell transparent zu definieren, wobei ähnliche Bedingungen bezüglich der Eingabeparameter zu beachten sind wie für pure functions in D. Auch Fortran 95 verfügt über ein eigenes Schlüsselwort für pure functions [For97, Kap. 12.6].

5.4.5 Ergebnis

In objektorientierten Sprachen, vorwiegend in Forschungssprachen, gibt es Konstrukte, die einzelne Merkmale eines wert-gemäßen Verhaltens aufweisen: Typen, deren Elemente nicht verändert werden können oder nicht erzeugt werden oder Typen mit einer zu Werten passenden Gleichheit. Teilweise sind die Wert-Merkmale nicht auf Typebene angesiedelt, sondern – wie etwa im Fall von pure functions – auf der Ebene einzelner Operationen.

Die charakteristischen Eigenschaften von Werten sind über verschiedene Sprachkonstrukte verteilt. Sie fügen sich nicht zu einer vollständigen Unterstützung für Werttypen zusammen; in keiner der untersuchten Sprachen gibt es ein Konstrukt, das alle Werteigenschaften in sich vereint. Tabelle 5.2 fasst die Ergebnisse von Abschnitt 5.4 zusammen.

Sprachkonstrukt	vorhanden in	Unveränderlichkeit	kein Erzeugen	Seiteneffekt-freiheit / referentielle Transparenz	flexible Definition beliebiger Typen
Unveränderliche Typen	Sather, X10, Fortress	ja	nein ^a	nein	nein (Speicherbeschränkungen)
Unerzeugbare Typen	Blue	nein	ja	nein	nein (nur Basisdatentypen und Aufzählungen) ^b
Case Classes	Scala	nein ^c	nein ^d	nein	ja
unveränderliche Variablen	Java, Scala, X10, Fortress, ...	ja			Merkmale nicht auf Typebene, Variablen sind orthogonal zu Typen
pure functions	D, C++			ja ^e	Merkmale nicht auf Typebene, definierbar für einzelne Operationen

Tabelle 5.2: Sprachkonstrukte mit partieller Unterstützung von Werteigenschaften

^aaufgrund der Funktionsweise der generierten Gleichheit *wirken* die Elemente dieser Typen wie unerzeugbar
^baufgrund dieser Einschränkungen sind Manifest Classes in Blue *de facto* Werttypen
^cmit Standardeneinstellungen sind die Elemente unveränderlich, der Programmierer kann abweichend eine Case Class auch als änderbar definieren
^dmit sprach-generierter Gleichheit wirken die Elemente von Case Classes unerzeugbar, der Programmierer kann eine abweichende Gleichheit implementieren
^eSeiteneffektfreiheit und referentielle Transparenz sind nur für strongly pure functions sichergestellt

5.5 Abgrenzung: Wert-ähnliche Sprachkonstrukte

Viele Sprachmechanismen weisen Ähnlichkeit mit Werttypen auf, sei es aufgrund ihrer Wirkungsweise oder nur aufgrund von – irreführenden – Namensähnlichkeiten.

Ein Beispiel sind Value Classes in Scala. Sie bezeichnen ursprünglich die fest vorgegebenen Klassen in der Scala-Hierarchie, die den primitiven Typen des zugrunde liegenden Java-Systems entsprechen (zuzüglich eines Typs `Unit`) [OSV11, Kap. 11.1]. Mittlerweile – ab Scala Version 2.10 – können Programmierer auch eigene Value Classes definieren [OOPS12]. Diese sind aber keine Werttypen, sondern ein Mittel zur Performancesteigerung. Sie bewirken, dass in der zugrunde liegenden Implementation eine Instanziierung entfällt (s. dazu Witt [Wit14, Kap. 5.3]).

Dieser Abschnitt befasst sich mit zwei weit verbreiteten, wert-ähnlichen Sprachkonstrukten: *Wertsemantik* wird von etlichen objektorientierten Sprachen unterstützt; sie nimmt in diesem Kapitel aufgrund ihrer vielen Ähnlichkeiten mit und der subtilen Unterschiede zu Werttypen einen breiten Raum ein. *Wertparameter* stellen in den meisten Sprachen das Standardverfahren zur Parameterübergabe dar, sie weisen aber keinen Zusammenhang mit Werttypen auf.

5.5.1 Wertsemantik für programmierer-definierte Typen

Wertsemantik als Technik zur Speicherverwaltung

Wertsemantik und ihr Gegenstück Referenzsemantik (diese Begriffe werden u. a. verwendet von Meyer [Mey97, Kap. 8.8]) beschreiben zwei verschiedene *Implementierungstechniken für Variablen* in imperativen Sprachen. Eine Variable wird dabei in Anlehnung an Sebesta [Seb10, Kap. 5.3] aufgefasst als „... eine Abstraktion einer Speicherzelle oder einer Sammlung von Speicherzellen, charakterisiert durch ...“

- einen Namen,
- einen adressierbaren Speicherbereich,⁶
- einen Wert,
- einen Typ (in statisch getypten Sprachen),
- eine Lebensdauer,
- einen Gültigkeitsbereich.

Für Wert- und Referenzsemantik sind die beiden Merkmale *Speicherbereich* und *Wert* relevant. Es ist zu beachten, dass der Begriff „Wert“, so wie er von Sebesta gebraucht wird, eine andere Bedeutung hat als im Kontext dieser Arbeit: Sebesta

⁶Der Variablenbegriff von Sebesta weicht von dem Variablenbegriff dieser Arbeit ab (s. Abschnitt 3.2). Während Sebesta Variablen als abstrakten, mit zusätzlichen Attributen versehenen Speicher begreift, verzichtet der Variablenbegriff dieser Arbeit bewusst auf einen Bezug zum Speicher. Wertsemantik setzt einen speicherbezogener Variablenbegriff voraus.

versteht unter dem „Wert einer Variablen“ den Inhalt ihres zugehörigen Speicherbereiches, nicht (wie in Kap. 3) ein Element eines Werttyps. Um diese verschiedenen Konzepte sprachlich zu unterscheiden, soll der „Wert einer Variablen“ (nach Sebesta) statt dessen nachfolgend als ihre *Belegung* bezeichnet werden. Damit lassen sich Wertsemantik und Referenzsemantik folgendermaßen charakterisieren:

Bei *Wertsemantik* besteht die Belegung der Variablen aus einer Repräsentation des zugehörigen Elementes (d. h. des Wertes oder des Objektes).

Bei *Referenzsemantik* besteht die Belegung der Variablen aus der Adresse eines anderen (in der Regel auf dem Heap gelegenen) Speicherbereichs, und darin befindet sich eine Repräsentation des zugehörigen Elementes.

Wert-„Semantik“ ist damit ein Verfahren zur Speicherverwaltung. Entgegen ihrem Namen trifft sie keine Aussage über die Bedeutung einer Variablen. Eine prägnantere Bezeichnung, die in Zusammenhang mit systemnaher Programmierung gebräuchlich ist und den technischen Charakter des Begriffs zum Ausdruck bringt, lautet *direkte Adressierung* (und für Referenzsemantik *indirekte Adressierung*).

Die obige Beschreibung spricht von der „Repräsentation eines Elementes“; und es ist essentiell, ein Element und seine Speicherrepräsentation voneinander zu unterscheiden. Elemente, d. h. Werte oder Objekte, sind konzeptionelle Abstraktionen, sie existieren nicht im Speicher. Der Speicher enthält Bitmuster, die als Repräsentationen von Werten oder Objekten *interpretiert* werden können – oder, im Fall von Referenzsemantik, als Adressen anderer Speicherzellen.

Konsequenterweise sind auch die Vorgänge im Speicher und die Vorgänge auf konzeptioneller Ebene voneinander zu trennen. Bei einer Zuweisung $b := a$ wird die Belegung von a in den Speicherbereich von b kopiert. Im Fall von Wertsemantik entsteht dadurch eine weitere Speicherrepräsentation des Elementes, weshalb sie auch als Kopiersemantik bezeichnet wird (s. z. B. Meyer [Mey97, p. 263]). Es handelt sich um einen rein technischen Kopiervorgang. Auf konzeptioneller Ebene wird der zur Variablen a gehörende Wert *nicht* kopiert – Werte sind unerzeugbar und können damit auch nicht kopiert werden. Konzeptionell verweist die Variable b nach der Zuweisung auf *denselben* Wert wie die Variable a .

Sprachen mit Unterstützung für Wertsemantik

Variablen von Basisdatentypen (s. Abschnitt 5.3.1) werden in den o. g. Sprachen mit Wertsemantik verwaltet, Variablen von sogenannten „Referenztypen“ (so bezeichnet z. B. Java Klassen und Arrays) dagegen mit Referenzsemantik. Einige Sprachen ermöglichen *Wertsemantik auch für programmierer-definierbare Typen*:

- C# und D unterstützen Wertsemantik durch den Typkonstruktor `struct`.⁷ Anders als structs in X10 (s. Abschnitt 5.4.1) sind structs in C# und D nicht unveränderlich.
- Eiffel ermöglicht Wertsemantik durch *expanded types*.
- In C++ kann eine Variable ein Objekt einer `class` oder `struct` direkt enthalten („immediate variables“). Alternativ kann die Variable einen Zeiger („pointer“) auf ein Objekt enthalten. In C++ werden Zeiger, anders als in C# oder Eiffel, vom Programmierer explizit als solche gekennzeichnet und manuell verwaltet.
- Auch die Variablen der in Abschnitt 5.4.1 diskutierten unveränderlichen Typen – Sather *immutable classes*, Fortress *value objects*, X10 *struct* – werden mit Wertsemantik verwaltet.

Beispiel: C# `struct`

In C#, C++, Eiffel und D und sind die durch Wertsemantik verwalteten Typen änderbar und bereits aus diesem Grund keine Werttypen. Änderungen bei Wertsemantik-Typen wirken sich anders aus als Änderungen bei Referenzsemantik-Typen, sie bleiben auf einen Kontext beschränkt.

Das illustriert das folgende C#-Beispiel. Es definiert einen Typ namens `POINT` mit dem Typkonstruktor `struct` und legt damit Wertsemantik für ihn fest.

```
public struct POINT {  
    public int x;  
    public int y;  
}
```

Die Methode `Main` definiert eine Variable vom Typ `POINT` und belegt dessen Koordinaten.⁸

```
static public void Main() {  
    POINT p1;  
    p1.x = 4;  
    p1.y = 3;  
    ModifyPoint(p1);  
    Console.WriteLine ("In Main: ({0},{1})", p1.x, p1.y);  
}
```

⁷C# bezeichnet die mit Wertsemantik unterlegten Typen – dazu zählen neben `struct` auch Aufzählungstypen – als „value types“. Wie dieser Abschnitt im weiteren ausführen wird, gibt es trotz des ähnlich Namens zwischen structs in C# und dem in Kap. 3 präsentierten Begriff von Werttypen wesentliche Unterschiede.

⁸Am Rande zeigt das Beispiel eine weitere Besonderheit von `struct`: Die Variable `p1` ist implizit bereits initialisiert. Anders als bei Referenzsemantik-Typen ist es nicht notwendig, ein Exemplar von `POINT` explizit mit `new` zu erzeugen.

Die Methode `Main` ruft eine Methode namens `ModifyPoint` auf, welche die Koordinaten des übergebenen Punktes ändert (verdoppelt).

```
static public void ModifyPoint (POINT pt) {  
    pt.x *= 2;  
    pt.y *= 2;  
    Console.WriteLine ("In ModifyPoint: ({0},{1})", pt.x, pt.y);  
}
```

Die in der Methode `ModifyPoint` vorgenommen Änderungen bleiben auf diesen Kontext beschränkt. Aus einem anderen Kontext, z. B. in der Methode `Main`, sind sie nicht sichtbar, dort bleibt der Punkt unverändert. Die Methode `Main` erzeugt deshalb die Ausgabe:

```
In ModifyPoint: (8,6)  
In Main: (4,3)
```

Orthogonalität von Wertsemantik und Werttypen

Basisdatentypen sind Werttypen, ihre Implementation basiert auf Wertsemantik, aber die Wertsemantik ist *nicht ursächlich* für ihre Werteigenschaften. Vielmehr sind umgekehrt die Werteigenschaften von Basisdatentypen eine Voraussetzung dafür, dass sie mit Wertsemantik unterlegt werden und korrekt funktionieren können. Beispielsweise ist es unproblematisch, technisch mehrere Kopien ihrer Repräsentationen zu führen, weil ihre Elemente unveränderlich sind.

Durch Wertsemantik unterlegte Typen sind keine Werttypen, im Fall von *expanded types* in Eiffel und *structs* in C# und D schon allein aufgrund ihrer Änderbarkeit (s. o.). Auch die unveränderlichen Typen in Sather, X10, Fortress unterstützen nicht alle Werteigenschaften, insbesondere fehlt eine sprachliche Sicherstellung von Seiteneffektfreiheit und referentieller Transparenz.

Die beiden Begriffe „Werttyp“ und „Wertsemantik“ bewegen sich auf verschiedenen Abstraktionsebenen und sind orthogonal zueinander. „Werttyp“ beschreibt einen Typ anhand des Verhaltens seiner Operationen. „Wertsemantik“ beschreibt eine Technik zur Speicherverwaltung. Eine Unterstützung durch Wertsemantik ist für Werttypen weder notwendig noch hinreichend. Z. B. ist es grundsätzlich möglich, auch die Variablen von Werttypen mit Referenzsemantik zu unterlegen.

Speicheroptimierung und Beschränkungen

Wertsemantik für programmierer-definierbare Typen wurde primär zur Speicherplatzoptimierung und zur Performancesteigerung in die jeweilige Sprache aufgenommen. Ob für einen Typ Wert- oder Referenzsemantik zweckmäßig ist, hängt in erster Linie von seinem Speicherverbrauch ab, weniger davon, ob er Werte oder Objekte

beschreibt. Z. B. empfiehlt C#, `struct` für leichtgewichtige Abstraktionen einzusetzen und dadurch den Speicherverbrauch zu verringern:

„The use of structs rather than classes for small data structures can make a large difference in the number of memory allocations an application performs ... allowing ... in-line storage of lightweight structures.“[Mic10, Kap. 1.7]

Ähnliche Begründungen und Anwendungsleitlinien gibt es auch für X10 [Sar+12, Kap. 9] und Sather [SO96, Kap. 1.2.6].

Die Art der Speicherverwaltung ist ein weiterer Grund, warum sich Wertsemantik nur eingeschränkt für die Abbildung von Werttypen eignet. Um die angestrebte Effizienz zu erreichen, unterliegen Wertsemantik-Typen technischen Einschränkungen. Z. B. dürfen sie nicht zirkulär definiert werden und schließen damit Rekursion aus. Sie sind nur einsetzbar, wenn alle Elemente des Typs Speicherrepräsentationen mit einer festen, zur Übersetzungszeit bekannten Größe haben.

Mit Wertsemantik können Typen wie „Datum“ programmiert werden, nicht aber Typen wie „Zeichenkette“, deren Speicherbedarf unterschiedlich groß und zur Entwicklungszeit nicht vorhersehbar ist. Aus diesem Grund sind z. B. in Sather die Klassen `STR` (Zeichenketten) und `INTI` („infinite precision integers“) – vom Konzept her als Werttypen einzustufen – nicht als immutable class, sondern als reference class implementiert.

Programmierung von Werttypen mit Wertsemantik

Zwar sind mit Wertsemantik unterlegte Typen nicht per se Werttypen. Es ist aber in vielen Fällen möglich, sie für die Programmierung von Werttypen einzusetzen; aufgrund der Funktionsweise ihrer Gleichheit (s. Abschnitt 5.4.3) eignen sie sich dafür besser als Referenztypen. Die Sicherstellung aller Werteigenschaften liegt dann in der Verantwortung des Anwendungsprogrammierers, sie erfordert Programmierdisziplin und die Einhaltung von Konventionen. Der Programmierer eines solchen Werttyps muss

- auf ändernde Operationen verzichten,
- alle Operationen des Typs referentiell transparent und seiteneffektfrei programmieren, und
- eine geeignete Gleichheit für den Typ implementieren oder (soweit vorhanden) darauf achten, die sprachgenerierte Gleichheit nicht zu überschreiben.

Nachteile von Wertsemantik

Wertsemantik ist in den untersuchten Sprachen kein reines Implementierungsdetail. Sie fungiert nicht als Blackbox, *sie beeinflusst auch das für Klienten sichtbare Verhalten*. Die technische und die konzeptionelle Ebene sind nicht streng getrennt.

Mit Wertsemantik können die Vorgänge im Speicher auf konzeptioneller Ebene verschieden gedeutet werden. Die damit definierten Typen sind weder klar als Werttypen noch eindeutig als Objekttypen einzuordnen (s. dazu das obige Code-Beispiel):

Werden die Elemente des Typs als Objekte angesehen, dann müssen Zuweisungen und Parameterübergaben als Kopiervorgang interpretiert werden anstatt (wie sonst bei Objekttypen) als ein weiterer Name für dasselbe Objekt. Werden die Elemente dagegen als Werte aufgefasst, müssen „ändernde“ Operationen uminterpretiert werden. Sie beschreiben dann – entgegen dem im Quelltext sichtbaren Aufruf – nicht eine Änderung des jeweiligen Elementes, sondern sie entsprechen einer – im Code nicht explizit aufgeführten – Zuweisung eines anderen Wertes.

In beiden Fällen ist es notwendig, einigen Anweisungen im Code (Zuweisung, Parameterübergabe, Methodenaufruf, ...) bei Wertsemantik eine andere Bedeutung zuzuordnen als bei Referenzsemantik. Unterstützung für Wertsemantik führt dadurch zu Mehrdeutigkeiten, macht das Sprachmodell komplexer und reduziert die Lesbarkeit der Quelltexte. Die Lehrsprache Blue, die ein klares und einheitliches Sprachmodell in den Vordergrund stellt, verzichtet aus den geschilderten Gründen bewusst auf die Unterstützung von Wertsemantik [Köl99, Kap. 6.2.2, 6.15.6].

5.5.2 Wert-Parameter (Call-by-value)

Wertparameter als Übergabeverfahren

Wertparameter oder „Übergabe per Wert“ bezeichnet ein Verfahren für den Datenaustausch zwischen einer aufgerufenen Operation und ihrem Aufrufer (was durch die englische Bezeichnung „call-by-value“ noch klarer zum Ausdruck gebracht wird).

Eine Übergabe per Wert hat die gleiche Wirkung wie eine Zuweisung. Der aktuelle Parameter, d. h. der vom Aufrufer übergebene Ausdruck, wird ausgewertet und das Ergebnis an den formalen Parameter gebunden. Innerhalb der aufgerufenen Operation wird der formale Parameter wie eine lokale Variable gehandhabt (s. Sebesta [Seb10, Kap. 9.5.2.1]). Zuweisungen an den formalen Parameter innerhalb der Operation wirken sich nicht auf den Aufrufer aus. Als aktueller Parameter ist jeder Ausdruck zulässig, der auf der rechten Seite einer Zuweisung stehen kann, z. B. muss sein Typ zum Typ des formalen Parameters kompatibel sein.

Unterstützung für Wertparameter

Alle hier untersuchten Programmiersprachen unterstützen Wertparameter. Sie sind das Standardverfahren für die Parameterübergabe; soweit in einer Operation ein formaler Parameter keine weiteren Angaben enthält, erfolgt die Übergabe „per Wert“. In etlichen Sprachen (z. B. Java, Eiffel, Fortress, X10) bilden Wertparameter den einzigen verfügbaren Übergabemodus.

Einige Programmiersprachen, z. B. C#, unterstützen zusätzlich *Referenzparameter* (auch bezeichnet als „Übergabe per Referenz“ oder „call-by-reference“).⁹ Bei Referenzparametern teilen sich der aktuelle Parameter und der formale Parameter denselben Speicherbereich [Seb10, Kap. 9.5.2.4]. Er wird im Aufrufer unter dem Namen des aktuellen Parameters angesprochen, innerhalb der aufgerufenen Operation unter dem Namen des formalen Parameters. Bei dieser Form der Übergabe kommen als aktuelle Parameter nur Variablen in Betracht (oder allgemeiner, Ausdrücke, die auf einen Speicherplatz verweisen und denen somit etwas zugewiesen werden kann). Eine Zuweisung zum formalen Parameter innerhalb der aufgerufenen Operation verändert auch den aktuellen Parameter und beeinflusst so den Aufrufer.

Orthogonalität der Begriffe

Werttypen und Wertparameter sind voneinander unabhängige Konzepte auf unterschiedlichen Abstraktionsebenen. „Werttyp“ beschreibt einen Typ anhand des Verhaltens seiner Operationen; „Wertparameter“ ein speichertechnisches Verfahren bei der Parameterübergabe. Werte müssen nicht notwendigerweise „per Wert“ übergeben werden, und Objekte – oder allgemeiner, Elemente von „Referenztypen“ – nicht notwendigerweise „per Referenz“. Typen und Übergabeverfahren können im Prinzip orthogonal miteinander kombiniert werden.

Es besteht die Gefahr, *Referenzparameter* und *Referenzsemantik* (s. Abschnitt 5.5.1) miteinander zu verwechseln, und im Analogieschluss auch *Wertparameter* und *Wertsemantik*.¹⁰ Ursache für die Verwechslung ist möglicherweise die Wortähnlichkeit sowie der Umstand, dass in beiden Fällen ein Speicherbereich über mehrere Namen angesprochen werden kann. Dieser Effekt wird auch als *Aliasing* bezeichnet.

Die beiden Formen von Aliasing haben unterschiedliche Auswirkungen und sind nicht miteinander zu vermengen: Bei *Referenzparametern* teilen sich zwei Variablen – der formale und der aktuelle Parameter – denselben Speicherbereich. Das hat zur Folge, dass eine Zuweisung zu einer Variablen auch die andere Variable ändert. Dagegen hat bei *Referenzsemantik* (sofern der Parameter per Wert übergeben wird), jede Variable ihren eigenen Speicherbereich. Weil dort eine Adresse abgelegt ist, können beide Variablen auf denselben (dritten) Speicherbereich *verweisen*. Änderungen,

⁹Die Literatur führt weitere Verfahren der Parameterübergabe auf, z. B. *call-by-value-result* und *call-by-result* (zwei Varianten von call-by-reference), oder *call-by-name* und *call-by-need* (Übergabeverfahren mit verzögerter Ausführung) [Seb10, Kap. 9.5]. Sie sind für die in diesem Kapitel vorgenommene Abgrenzung nicht von Belang und werden im Folgenden nicht weiter betrachtet.

¹⁰Beispielsweise findet sich in manchen Lehrbüchern die (falsche) Aussage, dass „in Java ... primitive Datentypen immer als Wert (by value) und Referenzdatentypen als Referenz (by reference) übergeben werden“ (z. B. bei Saake und Sattler [SS03, p. 16] oder bei Pollak [Pol09, p. 31]). Tatsächlich kennt Java ausschließlich Wertparameter, auch bei Referenztypen. Siehe hierzu auch die Debatte im Wiki von Cunningham & Cunningham [Cun10].

die über die eine Variable dort bewirkt werden, sind dann auch bei Zugriff über die andere Variable sichtbar – was in der Regel auch bezweckt wird. Aber Änderungen der Variablen selbst, d.h. Zuweisungen an sie, beeinflussen die andere Variable nicht.

Die drei Konzepte – der Typ einer Variablen (Wert- oder Objekttyp), ihre Speicherverwaltung (Wert- oder Referenzsemantik) und bei Übergabeparametern das Übergabeverfahren (Wert- oder Referenzparameter) – sind orthogonal zueinander. Im Prinzip können sich daraus bis zu acht Kombinationen ergeben, die meisten Programmiersprachen unterstützen nur einen kleinen Teil von ihnen. In Java z. B. sind Objekttypen mit Referenzsemantik unterlegt und primitive Datentypen mit Wertsemantik, und beide werden ausschließlich per Wert übergeben.

Nachteile von Referenzparametern

Referenzparameter können zu einer Reihe von Problemen führen. Deswegen sieht es diese Arbeit nicht als Vorteil an, wenn eine Programmiersprache Referenzparameter unterstützt.

Referenzparameter sind auf eine zweiseitige Kommunikation ausgelegt. Oft wird aber nur eine einseitige Kommunikation benötigt, vom Aufrufer zur aufgerufenen Operation. Referenzparameter bergen dann die Gefahr von unbeabsichtigten Änderungen des aktuellen Parameters [Seb10, Kap. 9.5.2.4].

Die durch Referenzparameter hervorgerufenen Aliaseffekte erschweren Programmverifikationen und verringern die Lesbarkeit des Codes. Eine Zuweisung an den formalen Parameter in der aufgerufenen Operation hat auch Auswirkungen auf den Aufrufer, sie sind dort aber schwer erkennbar, da im Aufrufer keine entsprechende Zuweisung sichtbar ist. Eine ausführliche Erörterung dieser Problematik findet sich z. B. bei Krishnamurthi [Kri03, Kap. 14.3].

Das Nebeneinander von Wert- und Referenzparametern macht eine Sprache komplex: Bei der Deklaration einer Operation muss für jeden Parameter das gewünschte Übergabeverfahren festgelegt werden; die dafür erforderliche Syntax bläht den Quellcode auf. Beim Aufruf einer Operation können die verschiedenen Übergabemodi syntaktisch markiert werden (z. B. handhabt C# das so), doch das ist redundant zur Festlegung in der Deklaration und macht auch hier den Quelltext komplizierter. Werden beim Aufruf einer Operation die Übergabemodi dagegen nicht kenntlich gemacht, so können syntaktisch gleichförmige Aufrufe unterschiedliches Verhalten bewirken. Beide Varianten sind nachteilig für die Lesbarkeit der Quelltexte.

Einige Sprachen, z. B. Java und C#, lassen in der Implementation einer Operation Zuweisungen an den formalen Parameter auch dann zu, wenn es sich um einen Wertparameter handelt und die Zuweisung deshalb außerhalb der Operation keine Wirkung entfaltet. Das kann verwirrend sein, besonders, wenn die Sprache auch Referenzparameter unterstützt. Möglicherweise beabsichtigt der Programmierer mit

einer solchen Zuweisung, den aktuellen Parameter des Aufrufers zu ändern (was bei Wertparametern nicht funktioniert), oder der Leser interpretiert den Code falsch, weil er irrtümlich diesen Effekt damit verbindet. Solche Zuweisungen machen den Code damit fehleranfällig.

5.5.3 Ergebnis

Einige Sprachkonstrukte weisen Ähnlichkeiten zu Werttypen auf oder tragen ähnliche Bezeichnungen, sie können darum leicht mit Werttypen verwechselt werden. Wertsemantik und Wertparameter, zwei in etlichen objektorientierten Sprachen unterstützte Konstrukte, bewegen sich auf einer speichertechnischen Ebene: Wertsemantik legt fest, wie die Variablen eines Typs im Speicher verwaltet werden, Wertparameter die Vorgänge im Speicher bei der Parameterübergabe. Die Speicherverwaltung von Variablen und der Übergabemodus von Parametern sind keine logischen Konzepte. Sie sind orthogonal zu Werttypen und Objekttypen, die im Gegensatz zu ihnen vollständig von einer zugrunde liegenden Speicherverwaltung abstrahieren.

Sprachkonstrukte, die dem Anwendungsentwickler Einfluss auf die Speicherverwaltung ermöglichen, verringern das Abstraktionsniveau der Sprache. Aufgrund ihrer Nachteile werden sie in dieser Arbeit kritisch gesehen: Wertsemantik ermöglicht zwar die Programmierung von Typen, deren Verhalten dem von Werttypen nahe kommt, sie nehmen aber eine unklare Zwitterposition ein und führen zu einem komplexen Sprachmodell mit mehrdeutiger Semantik. Referenzparameter machen den Code schwerer nachvollziehbar und verringern seine Lesbarkeit.

5.6 Zusammenfassung

Kapitel 5 untersuchte eine repräsentative Auswahl bestehender objektorientierter und hybrider Sprachen im Hinblick auf ihre Unterstützung für das in Kap. 3 beschriebene Konzept von Werttypen. Keine der Sprachen kann Werttypen vollständig *und* flexibel abbilden, d. h. *alle* Werteigenschaften sicherstellen und gleichzeitig die Definition *beliebiger* Werttypen ermöglichen.

Nur wenige Sprachkonstrukte verfügen über alle charakteristischen Eigenschaften von Werttypen: Basisdatentypen und Aufzählungstypen. Erstere können nicht vom Programmierer definiert werden, letztere nur für eine eng begrenzte Gruppe von Werttypen.

Eine Reihe von Sprachmechanismen erlaubt eine flexiblere Definition verschiedener Typen, sie stellen aber nur einzelne Werteigenschaften sicher: unveränderliche Klassen (in Sather, Fortress oder X10), unerzeugbare Klassen (in Blue), oder

Klassen mit einer zu Werttypen passenden Gleichheit (u. a. in Scala). Einige Sprachkonstrukte unterstützen Werteigenschaften nicht auf der Ebene eines Typs, sondern für einzelne Variablen (sie können in etlichen Sprachen als unveränderlich definiert werden) oder für einzelne Operationen (pure functions in C++ oder D gewährleisten Seiteneffektfreiheit und referentielle Transparenz).

Ein weiterer Teil von Kapitel 5 widmete sich der Abgrenzung gegenüber Sprachkonstrukten, die Ähnlichkeiten mit Werttypen aufweisen und leicht mit ihnen verwechselt werden können, die aber keine der Werteigenschaften sicherstellen. Wertsemantik, in einigen Sprachen durch einen dedizierten Typkonstruktor unterstützt (z. B. `struct` in C#, X10 und D), erlaubt die Definition von Typen mit wertähnlichem Verhalten. Wertsemantik ist aber ein Speicherverfahren für Variablen, orthogonal zum Konzept von Werttypen. Wertparameter sind ebenfalls ein speichertechnischer Begriff, sie beschreiben ein Verfahren für die Parameterübergabe und sind unabhängig von Wert- oder Objekttypen. Wertsemantik und Wertparameter geben dem Anwendungsprogrammierer Mittel an die Hand, um auf Implementierungsdetails Einfluss zu nehmen. Sie führen dadurch zu einem komplexeren Sprachmodell und verringern die Nachvollziehbarkeit des Codes.

Bestehende objektorientierte Sprachen enthalten eine Reihe von Sprachbausteinen mit Werteigenschaften oder wert-ähnlichen Charakteristika. Diese fügen sich aber nicht zu einer vollständigen und flexiblen Unterstützung für Werttypen zusammen.

Kapitel 6

Ein Typkonstruktor für Werttypen

Werttypen werden in einem objektorientierten Umfeld meist mit Hilfe von softwaretechnischen Verfahren umgesetzt. Doch solche Verfahren sind (s. Kapitel 4) unsicher, unklar und darüber hinaus oft umständlich, aufwändig und wenig effizient. Bestehende objektorientierte Programmiersprachen (s. Kapitel 5) unterstützen nur wenige Spezialfälle von Werttypen oder nur einzelne Werteigenschaften.

Dieses Kapitel schlägt vor, das in dieser Arbeit zugrunde gelegte Konzept von Werttypen (s. Kapitel 3) auf programmiersprachlicher Ebene zu unterstützen. Es erarbeitet einen Sprachentwurf für eine Werttyp-Unterstützung und diskutiert, wo vorhanden, alternative Gestaltungen und deren Vor- und Nachteile. Auf diese Weise wird ein einzelnes, dediziertes Sprachkonstrukt entwickelt, das dem Anwendungsprogrammierer einen einfachen Umgang mit Werttypen ermöglicht und die Verantwortung für deren charakteristische Eigenschaften auf die Sprachumgebung überträgt.

Die Spezifikation einer vollständigen Programmiersprache ist nicht Gegenstand der Arbeit. Sie würde eine Vielzahl von Entwurfsentscheidungen erfordern, die nichts mit Werttypen zu tun haben und dadurch den hier gesetzten Rahmen sprengen. Die Arbeit folgt dem von Hoare empfohlenen Vorgehen, den Entwurf eines neuen Sprachelementes und den Entwurf einer Programmiersprache voneinander zu trennen und diese beiden Tätigkeiten nicht miteinander zu vermischen:

„The designer of a new feature should concentrate on one feature at a time. ... He should make sure that his feature mitigates some disadvantage or remedies some incompleteness of the language ...

The language designer should be familiar with many alternative features ... His task is consolidation not innovation.“[Hoa73]

6.1 Anforderungen

Aus den in Kapitel 4 festgestellten Mängeln von Programmierverfahren lassen sich die folgenden softwaretechnischen Anforderungen an eine Sprachunterstützung für Werttypen ableiten (s. dazu auch [RS11]):

Sicherheit: Die Sprachunterstützung soll alle charakteristischen Eigenschaften von Werttypen gewährleisten, möglichst bereits zur Übersetzungszeit.

Klarheit: Werttypen sollen im Quelltext als solche erkennbar sein.

Einfachheit: Werttypen sollen mit angemessenem Aufwand definiert und benutzt werden können. Für die Herstellung ihres wert-gemäßen Verhaltens soll bei der Programmierung kein Mehraufwand entstehen.

Weitere Anforderungen lassen sich aus den in Kapitel 5 vorgenommenen Untersuchungen bestehender Sprachen ableiten:

Umfassende Unterstützung verschiedener Werttypen: Die Sprachunterstützung soll umfassend und flexibel für ein breites Spektrum verschiedener Werttypen einsetzbar sein. Sie soll z. B. so unterschiedlich aufgebaute Werttypen wie Punkt, Zeichenkette oder Spielkartenfarbe abdecken.

Aus dem Ziel, Werte in ein objektorientiertes Umfeld zu integrieren, ergibt sich des weiteren die Forderung, dass ihre Sprachunterstützung sich an den Leitlinien objektorientierter Programmierung ausrichtet:

Trennung von Schnittstelle und Implementation: Ein Grundprinzip der Objektorientierung ist die Trennung von Schnittstelle und Implementation und damit verbunden das Verbergen von internen Datenstrukturen gegenüber Klienten („Kapselung“). Dieses Prinzip ist beim Sprachentwurf einer Werttyp-Unterstützung zu berücksichtigen.

6.2 Sprachliche Grundelemente und Syntax

Die in diesem Kapitel verwendeten Beispiele setzen das Vorhandensein einiger sprachlicher Grundelemente voraus: Variablen, Operationen und (Objekt-)Klassen. Dieser Abschnitt beschreibt die Syntax, in der sie in dieser Arbeit abgefasst werden. (Anhang A.1 enthält eine zugehörige Grammatik.)

Die gewählte Notation verfolgt das Ziel, dem zustandslosen Charakter von Werten gerecht zu werden und trotzdem möglichst wenig von der Notation, wie sie aus Java oder C# vertraut ist, abzuweichen. Diese beiden Ziele können in Konflikt zueinander geraten und dann Kompromisse erfordern.

Die **Deklaration** einer **Variablen** legt ihren Namen und ihren Typ fest. Der Name des Typs wird dem Variablen-Namen vorangestellt. Beispiel:

```
MyType v;
```

Die **Zuweisung** zu einer Variablen wird durch das Symbol „:=“ ausgedrückt, mit der Variablen auf der linken und dem zugewiesenen Ausdruck auf der rechten Seite. Beispiel:

```
v := exp;
```

In der **Definition** einer **Operation** wird ihr Rückgabetyt ihrem Namen vorangestellt, danach folgen in Klammern ggf. ihre formalen Parameter, danach in geschweiften Klammern ihre Implementation. Beispiel:

```
RType op(PType1 p1, PType2 p2, ...) { ... }
```

Operationen in Objektklassen (**Methoden**) enthalten einen impliziten Parameter für das Objekt, für das sie aufgerufen werden (entsprechend `this` in Java).

Operationen in Wertklassen (**Funktionen**) führen in der Parameterliste *alle* ihre Parameter auf, sie haben keinen impliziten Parameter.

Der **Aufruf** einer **Methode** folgt der aus Java bekannten „Punktnotation“, mit dem Objekt, an welchem die Operation aufgerufen wird, vor dem Punkt und der Liste der aktuellen Parameter hinter dem Operationsnamen. Beispiel:

```
obj.op(a1, a2, ...)
```

Der **Aufruf** einer **Funktion** besteht aus ihrem Namen, gefolgt von der Liste ihrer aktuellen Parameter. Beispiel:

```
op(a1, a2, ...) ...
```

Für Funktionsaufrufe kommt damit die in der Mathematik gebräuchliche Schreibweise zum Einsatz, z. B. `op(a)` statt `a.op()`. Das entspricht auch dem mathematischen Charakter von Werten. Die Punktnotation würde bei Operationen mit zwei oder mehr Parametern zu einer optischen Asymmetrie (z. B. `a.op(b)` statt `op(a,b)`) führen, die bei vielen Funktionen – z. B. Addition oder Größenvergleich – fachlich nicht gerechtfertigt ist und damit eher verwirrt.

So lässt bereits die Form des Aufrufs erkennen, ob eine Operation zum „funktionalen Kern“ oder zur „objektorientierten Schale“ eines Systems gehört.¹

¹Mit der Punktnotation nimmt der erste Parameter nicht nur syntaktisch eine Sonderstellung ein. In vielen objektorientierten Sprachen legt er fest, welche Implementation der Operation aufgerufen wird („single dispatch“); entscheidend ist dabei der Typ des Objektes zur Laufzeit. Dieses dynamische Binden ist nur erforderlich, wenn es für die Operation in einer Subklasse eine abweichende Implementation geben kann. Wie Kapitel 9 zeigen wird, stellt Implementierungsvererbung bei Werttypen keine Subtyp-Beziehung her. Dynamisches Binden der Operationen spielt deshalb bei Wertklassen eine geringere Rolle als bei Objektklassen. Das ist ein weiterer Grund, für Funktionen eine andere Aufruf-Syntax vorzusehen als für Objektoperationen.

Die **Definition** einer **Objektklasse** und ihrer Member folgt der Syntax von Java (geschweifte Klammern, Aufführen der Member im Klassenrumpf, etc.).

Beispiel:

```
class MyClass {  
    MType mv;  
    RType op(PType1 p1, PType2 p2, ...) {...}  
    ...  
}
```

Anweisungen und Variablen-Deklarationen werden (s.o.) mit einem Semikolon abgeschlossen.

Für die Namen von Variablen und Operationen wird Kleinschreibung, für die Namen von Typen und Klassen Großschreibung verwendet.

Die in diesem Kapitel aufgeführten Beispiele setzen voraus, dass grundlegende Typen für Ganzzahlen (`Int`), für Gleitkommazahlen (`Float`), für Zeichen (`Char`) und für Wahrheitswerte (`Bool`) vorhanden sind, mit zugehörigen Literalen (z. B. 1, 1.2, 'a', true) und mit Infix-Operatoren für Grundrechenarten (+, -, *, /), für Vergleiche (=, !=, <, >) und für logische Verknüpfungen (&&, ||).

6.3 Entwurf einer Sprachunterstützung für Werttypen

Im Folgenden wird, aufbauend auf den in Kapitel 3 beschriebenen Eigenschaften und den oben genannten Anforderungen, eine Sprachunterstützung für Werttypen entwickelt. Ein durchgängiges Beispiel, ein Werttyp `Point` für die Abbildung von Punkten in einer zweidimensionalen Ebene, illustriert die zugehörigen Details.

Die definierenden Eigenschaften von Werttypen und die Anforderungen legen eine Sprachunterstützung nicht eindeutig fest. Wo vorhanden, werden alternative Gestaltungen, ihre Vor- und Nachteile und ggf. zugrunde liegende Zielkonflikte diskutiert.

6.3.1 Dedizierter Typkonstruktor

Werttypen werden mit Hilfe von Wertklassen definiert; für Wertklassen steht ein dedizierter Typkonstruktor zur Verfügung. Die Definition einer Wertklasse wird mit einem Schlüsselwort eingeleitet, das sich von dem für die Definition einer Objektklasse unterscheidet, hier: mit dem Schlüsselwort `valueClass`.

Die Definition einer Wertklasse `Point` illustriert den Typkonstruktor:

```
valueClass Point {  
    ...  
}
```

Der Typkonstruktor für Objekttypen erhält konsequenterweise ein Schlüsselwort wie z. B. `objectClass`, nicht nur `class`. Das verdeutlicht, dass Werte und Objekte zwei verschiedene Abstraktionen sind und keine als Spezialfall der anderen angesehen werden kann.

Die Bezeichnung als *Wertklasse* und das Schlüsselwort `valueClass` weisen darauf hin, dass dieser Typkonstruktor sowohl einen Typ als auch eine zugehörige Implementation definiert, ebenso wie der Typkonstruktor `class` in Java.

6.3.2 Datenfelder

Innerhalb einer Wertklasse bilden Datenfelder die interne Struktur der Elemente ab. Mit einer Darstellung durch kartesische Koordinaten, `x` und `y`, sieht die Definition der Wertklasse `Point` folgendermaßen aus:

```
valueClass Point {  
    Float x;  
    Float y;  
    ...  
}
```

Die Datenfelder von Wertklassen unterliegen spezifischen Restriktionen:

*Sie sind immer unveränderlich, und ihre Typen müssen ebenfalls Werttypen sein.*²

Zuweisungen zu den Datenfeldern sind nicht möglich, sie werden bei der Übersetzung als Fehler markiert. Das stellt sicher, dass die Elemente der Wertklasse unveränderlich sind. Über die Datenfelder können nur Funktionen (ggf. anderer Werttypen) aufgerufen werden, was Zugriff auf änderbaren Zustand unterbindet.

Gemäß dem Prinzip der Kapselung ist die interne Repräsentation verborgen. Klienten können nicht direkt auf die Datenfelder einer Wertklasse zugreifen, sondern nur über die für die Wertklasse definierten Funktionen. Damit Klienten die Koordinaten lesen können, müssen in der Wertklasse `Point` entsprechende Funk-

²In der Forderung, dass Werttypen nur aus Werttypen bestehen, unterscheidet sich die vorliegende Arbeit von etlichen anderen Arbeiten. Die von Evans beschriebenen „value objects“ können „entities“ referenzieren [Eva04, p. 98]. Gemäß Kühne können „value objects“ „reference type objects“ als Subkomponenten haben [Küh99, p. 160]. Die key fields der von Vaziri et al. beschriebenen „relation types“, von denen „value types“ einen Spezialfall darstellen, dürfen beliebige Typen sein, auch „reference types“ [VTFD07, p. 69]. Es werden dort auch keine Vorkehrungen erwähnt, die durch die Einbindung von „reference types“ möglichen Seiteneffekte so einzugrenzen, dass sie für Klienten der „value objects“ nicht sichtbar werden. Abschnitt 6.3.4 wird potentielle Zugriffe auf Objekte innerhalb von Wertklassen als eine mögliche Entwurfsalternative nochmals aufgreifen.

tionen definiert werden. Innerhalb der Implementation einer Funktion kommt für den Zugriff auf ein Datenfeld der Wertklasse die Punktnotation zum Einsatz:

```
valueClass Point {  
    Float x;  
    Float y;  
    Float getX (Point p) {return p.x;}  
    Float getY (Point p) {return p.y;}  
    ...  
}
```

Entwurfs-Alternative: für Klienten lesbare Datenfelder

Um Schreibaufwand zu sparen, ist ein alternativer Sprachentwurf vorstellbar, nach dem die Datenfelder von Wertklassen immer auch für Klienten lesbar sind. Das würde für einfache Wertklassen wie `Point` die Programmierung vereinfachen und den Quelltext verkürzen.

Gegen diesen Entwurf sprechen mehrere Gründe:

- Eine solche Festlegung würde gegen das Prinzip der Kapselung verstoßen. Es wäre für Klienten erkennbar, welche Datenfelder eine Wertklasse hat. Die Implementation der Wertklasse kann dann nicht mehr geändert werden, ohne die Klienten zu beeinflussen.
- Nicht für alle Wertklassen ist ein lesender Zugriff auf die Datenfelder fachlich angemessen. Beispielsweise sollte in einer Wertklasse wie `Rational` (die rationale Zahlen darstellt) die interne Repräsentation durch Zähler und Nenner verborgen bleiben, wenn diese nicht eindeutig ist.
- Die Notwendigkeit, auch für die Datenfelder Zugriffsoperationen zu schreiben, führt zu einer einheitlichen Schnittstelle der Wertklasse. Sie besteht dann ausschließlich aus Operationen (statt aus Operationen *und* Datenfeldern).

Der Schreibaufwand für den Programmierer kann durch die Entwicklungsumgebung verringert werden, z. B. durch Generierung entsprechender Funktionen. Eine solche auf Wertklassen ausgerichtete IDE-Funktionalität ähnelt dem Generieren von get-Methoden, wie sie z. B. von Eclipse für die Exemplarvariablen von Java-Klassen unterstützt wird („Create Getters and Setters“ [Ecl14]).

6.3.3 Selektoren

Werte werden konzeptionell nicht erzeugt. Wertklassen sollen diese Eigenschaft im Code widerspiegeln, sie verfügen darum nicht über Konstruktoren. Es besteht aber für Klienten der Bedarf, auf ein Element einer Wertklasse zuzugreifen und dabei zu spezifizieren, um welchen (bereits vorhandenen) Wert es sich handelt.

Bei primitiven Datentypen übernehmen Literale diese Aufgabe. Eine vom Programmierer definierte Wertklasse hat dagegen keine Literale. Deren Rolle wird von Operationen übernommen, die ein Element der Wertklasse liefern und anhand ihrer Parameter festlegen, welches. Im Beispiel der Wertklasse `Point` kann es z. B. eine Operation namens `pointCartesian` geben, die ausgehend von zwei kartesischen Koordinaten den Punkt mit diesen Koordinaten liefert:

```
valueClass Point {  
    Float x;  
    Float y;  
    Float getX (Point p) {return p.x;}  
    Float getY (Point p) {return p.y;}  
    Point pointCartesian(Float x, Float y) { ... }  
    ...  
}
```

Eine Operation innerhalb einer Wertklasse, die ein Element der eigenen Klasse liefert, wird als *Wertselektor*, oder kurz als *Selektor* bezeichnet. Ein Selektor wählt („selektiert“) ein Element aus dem „Werteuniversum“ aller vorhandenen Werte, er erzeugt es aber nicht. Konsequenterweise liefert ein Selektor, wenn er mehrfach mit den gleichen Argumenten aufgerufen wird, immer das gleiche Element – wie es die referentielle Transparenz für die Operationen von Werttypen fordert. Hierin liegt ein wesentlicher Unterschied zu den Konstruktoren von Objektklassen. Diese liefern bei jedem Aufruf ein neues und damit anderes Element, auch bei Aufrufen mit denselben Parametern.

Das Konzept von Wertselektoren als einer eigenständigen Kategorie von Operationen wurde erstmals 2003 vorgestellt [Rit03]. Selektoren in Wertklassen können als das Gegenstück zu den Literalen primitiver Datentypen interpretiert werden – oder umgekehrt Literale als syntaktische Sonderform von Selektoren.

Eine Wertklasse kann mehrere Selektoren haben. Im Beispiel `Point` kann es z. B. einen weiteren Selektor geben, der ein Element von `Point` auf Basis seiner Polarkoordinaten liefert:

```
Point pointPolar(Float rho, Float theta) { ... }
```

Auch Selektoren ohne Parameter sind möglich. Ein solcher Selektor liefert immer denselben Wert und entspricht damit einer Konstante. Z. B. kann die Wertklasse `Point` einen parameterlosen Selektor haben, der den Nullpunkt liefert:

```
Point zeroPoint() { ... }
```

Jede Wertklasse muss über einen oder mehrere Selektoren verfügen, und mindestens ein Selektor darf nur Parameter haben, deren Typ nicht die Wertklasse selbst

ist. Anderenfalls wäre es für Klienten nicht möglich, auf die Elemente der Wertklasse zuzugreifen.

Selektoren von Wertklassen sind „normale“ Funktionen. Aus diesem Grund sieht der Sprachentwurf für die Definition eines Selektors keine eigenen Schlüsselworte und keine spezifische Syntax vor. Selektoren sind bereits daran erkennbar, dass ihr Rückgabetyt die eigene Klasse ist. Eine spezielle Kennzeichnung ist daher nicht erforderlich.

Eine spezifische Syntax für Selektoren kann sogar nachteilig sein. Würde z. B. festgelegt, dass der Name eines Selektors mit dem Namen der Wertklasse übereinstimmt (wie in Java die Namen von Konstruktoren bei Objektklassen), dann könnten nicht mehrere Selektoren mit den gleichen Parametertypen definiert werden, wie im obigen Beispiel `pointCartesian` und `pointPolar`.

Die eingangs für Funktionen festgelegte Aufrufsyntax bewirkt einen gleichförmigen Aufruf von Selektoren und Nicht-Selektoren. Der Einsatz der Punktnotation wäre dagegen nur für solche Funktionen möglich, die über mindestens einen Parameter des eigenen Typs verfügen, was auf viele Selektoren nicht zutrifft.

Der primäre Selektor

Während Selektoren aus Sicht von Klienten gewöhnliche Funktionen sind, die nach den gleichen Regeln aufgerufen werden wie die anderen Operationen einer Wertklasse, erfordert ihre *Implementierung* einen zusätzlichen, auf Wertklassen zugeschnittenen Sprachmechanismus. Weil der Rückgabetyt eines Selektors die eigene Wertklasse ist, muss innerhalb der Implementation eines Selektors ein Ausdruck dieses Typs angegeben werden. Ein solcher ist gemäß der bis hier beschriebenen Sprachunterstützung nicht vorhanden.

Für Wertklassen wird eine Operation postuliert,

- die mit einem eigenen Schlüsselwort (`select`) aufgerufen wird,
- die innerhalb der Implementation jeder Wertklasse zur Verfügung steht,
- deren Parameter in Anzahl, Typ und Reihenfolge den Datenfeldern der Wertklasse entsprechen, und
- die das Element der Wertklasse liefert, dessen Datenfelder den aktuellen Parametern entsprechen.

Z. B. liefert in der Wertklasse `Point` der Ausdruck

```
select(3.1, 2.5)
```

das Element von `Point`, dessen x-Koordinate 3.1 und y-Koordinate 2.5 beträgt.

Die Operation `select` wird als *primärer Selektor* bezeichnet. Sie ist ein Sprachprimitiv, d. h. sie wird von der Sprachumgebung generiert und steht automatisch zur Verfügung.

Mit dem primären Selektor sieht eine Implementation des obigen Beispiels `Point` folgendermaßen aus:

```
valueClass Point {  
    ...  
    Point pointCartesian(Float x, Float y) {  
        return select(x, y);  
    }  
    Point pointPolar(Float rho, Float theta) {  
        Float x := rho * cos(theta);  
        Float y := rho * sin(theta);  
        return pointCartesian(x, y);  
    }  
    Point zeroPoint() { return select(0.0, 0.0); }  
    ...  
}
```

Selektoren können andere programmierer-definierte Selektoren aufrufen statt des primären Selektors, wie im Beispiel `pointPolar`. Mindestens ein Selektor muss den primären Selektor aufrufen – anderenfalls wäre der Aufruf der Selektoren untereinander zirkulär und würde nicht terminieren.

Entwurfs-Alternative: Zuweisungen zu Datenfeldern in Selektoren

Eine Alternative zur Postulierung des primären Selektors besteht darin, innerhalb der Selektoren – und nur dort – Zuweisungen zu den Datenfeldern zuzulassen. Diese Entwurfsvariante ist in [Rit04] beschrieben.

Damit legen Zuweisungen in den Selektoren fest, welches Element der Selektor liefert. Auch diese Variante erfordert einen zusätzlichen Sprachmechanismus: eine Pseudovariablen, die den Rückgabewert bezeichnet und deren Datenfelder angesprochen werden können, im Folgenden mit dem Schlüsselwort `value` bezeichnet.

Basierend auf diesem Sprachentwurf würde der Selektor `pointCartesian` folgendermaßen implementiert:

```
Point pointCartesian(Float x, Float y) {  
    value.x := x;  
    value.y := y;  
    return value;  
}
```

Dieser Sprachentwurf weist gegenüber dem primären Selektor mehrere Nachteile auf:

- Er spiegelt das Konzept von Werttypen nicht klar wider. Der Code enthält Zuweisungen zu den Datenfeldern und damit Änderungen, obwohl Werte konzeptionell unveränderlich sind.
- Die Zuweisungen zu den Datenfeldern verteilen sich i. A. über mehrere Anweisungen. Die Festlegung des auszuwählenden Elementes ist damit über den

Code verstreut. Es ist nicht auf einen Blick erkennbar, welcher Wert selektiert wird. Der Code ist weniger prägnant als ein Aufruf des primären Selektors.

- Es besteht die Gefahr, dass der Programmierer für ein Datenfeld die Zuweisung vergisst oder dass sie nicht angesprochen wird. Der Code innerhalb eines Selektors kann beliebig komplex werden. Nicht in allen Fällen kann eine statische Analyse feststellen, ob für jedes Datenfeld eine Zuweisung dynamisch durchlaufen wird. Die Lösung ist damit fehleranfälliger als der primäre Selektor.
- Auch der alternative Sprachentwurf erfordert mit der Pseudovariablen `value` einen zusätzlichen Sprachmechanismus und zugehörige Handhabungsregeln (s. dazu auch [Rit04, p. 16]). Der alternative Entwurf ist damit nicht einfacher als der primäre Selektor.

Entwurfs-Alternative: Primärer Selektor aufrufbar für Klienten

Der primäre Selektor kann gemäß dem oben beschriebenen Entwurf nur innerhalb der Wertklasse aufgerufen werden, nicht von ihren Klienten.

Ein alternativer Sprachentwurf kann vorsehen, dass der primäre Selektor auch für Klienten aufrufbar ist. Eine solche Gestaltung kann Schreibarbeit einsparen und den Quelltext der Wertklasse verkürzen. Für einfach gebaute Werttypen wie `Point` wird oft ohnehin ein öffentlich zugänglicher Selektor benötigt, dessen Parameter den Datenfeldern des Werttyps entsprechen. Ist der primäre Selektor nur innerhalb der Wertklasse aufrufbar, dann muss ein gleichartiger Selektor manuell programmiert werden (seine Implementation besteht dann nur aus einem Aufruf des primären Selektors). Ist der primäre Selektor für Klienten aufrufbar, so entfällt dieser Aufwand.

Auch hier spricht das Prinzip der Kapselung dagegen, den primären Selektor für Klienten aufrufbar zu machen:

- Weil die Parameter des primären Selektors den Datenfeldern der Wertklasse entsprechen, würde deren Änderung sich auch auf den primären Selektor und damit auf die Schnittstelle der Wertklasse auswirken. Das gilt nicht nur für eine grundlegende Änderung der internen Repräsentation, z. B. bei der Wertklasse `Point` ein Wechsel von kartesischen zu polaren Koordinaten. Bereits eine andere Reihenfolge der Datenfelder, ein scheinbar nebensächliches Detail, würde Klienten beeinträchtigen.
- Es kann Wertklassen geben, für die der primäre Selektor nicht für Klienten aufrufbar sein *soll*. Ein Beispiel dafür ist eine Wertklasse `Rational`, deren interne Repräsentation aus Zähler und Nenner besteht. Falls der Programmierer hier nur teilerfremde Darstellungen zulassen will (s. Abschnitt 6.3.5), ist es erforderlich, in einem Selektor zunächst die Eingabeparameter zu kürzen, bevor der primäre Selektor aufgerufen wird.

- Ein für Klienten aufrufbarer primärer Selektor würde zu einer uneinheitlichen Schnittstelle führen. Der Name des primären Selektors wäre dann im Gegensatz zu allen anderen Operationen des Typs durch die Sprachspezifikation festgelegt („select“). Ein vom Programmierer vergebener Name kann auch die Bedeutung des primären Selektors klarer zum Ausdruck bringen. Im Beispiel `Point` ist der Name `pointCartesian` aussagekräftiger als `select`.

Zwar führt das manuelle Codieren eines Selektors, der lediglich den primären Selektor aufruft, zu einem erhöhten Schreibaufwand. Ein solcher Selektor kann aber prinzipiell durch die IDE generiert werden und so den Programmierer entlasten – ähnlich, wie z. B. Eclipse die Generierung von Konstruktoren für Java-Klassen unterstützt („Generate Constructor using Fields“ [Ecl14], s. dazu auch Abschnitt 6.3.2).

Keine Trennung von Exemplar- und Klassenoperationen

Selektoren sind keine „Klassenoperationen“. Alle Operationen innerhalb einer Wertklasse sind gleichrangig. Eine Unterscheidung von „Exemplarmethoden“ und „Klassenmethoden“, wie sie z. B. in Java zu finden ist, wird hier für Wertklassen nicht zugrunde gelegt.³

Eine Einordnung als Klassen- oder Exemplaroperation ist zu unterscheiden vom statischen oder dynamischen Binden einer Operation – das ein technisches Implementierungsdetail darstellt. Weil Funktionen anders aufgerufen werden als Methoden von Objektklassen – es gibt keine Punktnotation und damit kein Element, „an dem“ eine Operation aufgerufen wird – erfordern sie in der zugrunde liegenden Implementation andere Mechanismen zum Auffinden des zugehörigen Codes. Das gilt insbesondere für Selektoren, die nicht über Parameter vom Typ der eigenen Klasse verfügen. Dafür gibt es verschiedene Lösungsansätze, von denen zwei an dieser Stelle skizziert werden:

Die Wertklassen, deren Operationen aufrufbar sein sollen, werden im Klientencode durch entsprechende import-Anweisungen festgelegt. Eine solche import-Anweisung gibt an, welchem Namensraum (hier: welcher Wertklasse) der Code der benutzten Operationen zugeordnet ist. Sie entfaltet damit eine Wirkung vergleichbar dem `static import` in Java (s. dazu [AGH06]). Verschiedene, unabhängig voneinander entwickelte Wertklassen können über gleichnamige Funktionen verfügen. In solchen Fällen führen die eben beschriebenen import-Anweisungen zu Namenskonflikten.

Ein mögliches Verfahren zum Auflösen derartiger Namenskonflikte besteht darin, dem Aufruf einer Funktion den Namen der zugehörigen Wertklasse voranzustellen.

³Auch für Objektklassen sind Klassenmethoden nicht zwingend erforderlich. Einige Sprachen unterstützen keine Klassenmethoden, z. B. Eiffel oder Scala.

Die Benutzung einer Wertklasse wie `Point` oder `Rational` nimmt dadurch z. B. folgende Form an:

```
... Point.pointCartesian(x,y) ...  
... Rational.add(r1,r2) ...
```

Der vorangestellte Name der Wertklasse markiert hier nicht den Aufruf einer statischen Methode, sondern dient dazu, die zur aufgerufenen Funktion gehörende Klasse anzugeben. Er macht so auch den Code verständlicher. Z. B. können damit Ausdrücke wie:

```
... Point.zeroPoint ...  
... Kartenfarbe.kreuz ...
```

– beides Sonderformen von Selektoren – im Quelltext besser lesbar notiert werden.

Bei Nicht-Selektoren wie `add`, die über mindestens einen Parameter der eigenen Klasse verfügen, ist das Voranstellen des Klassennamens in der Regel nicht erforderlich. Hier liefert der Parametertyp ausreichend Hinweise auf die zugehörige Wertklasse. Ein Voranstellen des Klassennamens kann in solchen Fällen dem Programmierer als Option freigestellt werden. Auch für Selektoren, deren Zuordnung zu ihrer Wertklasse durch eine `import`-Anweisung gewährleistet ist, kann das Voranstellen des Klassennamens optional gemacht werden. Die beiden Verfahren – `import`-Anweisungen und Voranstellen des Wertklassennamens – können einzeln oder in Kombination miteinander eingesetzt werden.

Gültigkeitsprüfungen

Einige Rahmenwerkslösungen für Werttypen, z. B. das JWAM-Rahmenwerk [JWA03], sehen Gültigkeitsprüfungen für die Parameter der Operationen vor, welche einen Wert der eigenen Klasse liefern – übertragen auf die hier vorgeschlagene Sprachunterstützung also für Selektoren.

Das den Gültigkeitsprüfungen zugrunde liegende Konzept wurde von Züllighoven et al. [BLZ99] beschrieben; Fürter und Spreckelmeyer haben es in eine Werkzeugunterstützung für Werte integriert [FS02, p.54-58]: Zu jedem „Selektor“ – einer passend benannten Klassenmethode – gibt es eine Methode namens „`isValid`“. Sie hat die gleichen Parametern wie der „Selektor“ und prüft, ob die Parameter einen Wert des Typs liefern. `isValid`-Methoden können z. B. Kombinationen wie den 30. Februar 2016 für einen Typ `Datum` als ungültig erkennen. Bei einer Abbildung von Werttypen in Java müssen `isValid`-Methoden `static` sein, weil sie nicht „an“ einem Element des Typs aufgerufen werden können – ein solches soll ja erst ermittelt werden.

Derartige Gültigkeitsprüfungen sind i. A. auch in Wertklassen erforderlich. Eine eigene programmiersprachliche Unterstützung für sie ist jedoch in der hier vorge-

schlagenen Sprachunterstützung aus mehreren Gründen nicht vorgesehen:

- Eine Gültigkeitsprüfung wäre für jeden für Klienten aufrufbaren Selektor erforderlich. Weil die Selektoren vom Programmierer einer Wertklasse definiert werden, ist nicht vorhersehbar, wie viele Selektoren es gibt und welche Parameter sie haben. Insoweit ist eine Sprachunterstützung für solche Prüfungen schwer umsetzbar.
- Ein Parameter oder eine Kombination von Parametern, zu der ein Selektor keinen Wert ermitteln kann, stellt eine Verletzung einer Vorbedingung dar. Wenn eine Programmiersprache Vorbedingungen, z. B. im Rahmen eines Vertragsmodell (s. z. B. [Mey92]), nicht explizit unterstützt, kann die Gültigkeit der Parameter in der Implementation des Selektors geeignet behandelt werden, z. B. durch Prüfen und bedingtes Auslösen einer Ausnahme. Es ist nicht erforderlich, dafür eine sprachgestützte Operation einzusetzen.

6.3.4 Kein Zugriff auf Objekte

Werte dürfen nur auf Werte zugreifen, nicht auf Objekte. Es wird festgelegt, dass weder in der Schnittstelle der Wertklasse noch in ihrer Implementation Objekttypen benutzt oder Objektoperationen aufgerufen werden dürfen. Diese Einschränkung stellt sicher, dass Wertoperationen Funktionen, d. h. seiteneffektfrei und referentiell transparent sind. Weil das Sprachmodell Werttypen und Objekttypen voneinander trennt, kann die Sprachumgebung diese Einschränkung sicherstellen.

Die festgelegte Einschränkung schließt auch die Verwendung von Objektklassen innerhalb der Implementation von Funktionen aus. Sie ist damit restriktiver als die in Kapitel 3 hergeleitete Gesetzmäßigkeit, nach der Werttypen keine *Typabhängigkeiten* zu Objekttypen haben können, d. h. dass in der Signatur von Funktionen keine Objekttypen vorkommen dürfen. Sie ist auch umfassender als die in Abschnitt 6.3.2 beschriebene Einschränkung, gemäß der in einer Wertklasse die Typen der Datenfelder keine Objekttypen sein dürfen.

Entwurfs-Alternative: Benutzung von Objekten in der Implementation von Wertklassen

Das Unterbinden von Objekt-Zugriffen in Wertklassen ist hinreichend, aber nicht notwendig, um Seiteneffektfreiheit und referentielle Transparenz sicherzustellen. Die konzeptionellen Werteigenschaften können auch gewährleistet werden, wenn die Verwendung von Objekten zugelassen, aber auf geeignete Weise eingeschränkt wird.

Beispielsweise kann innerhalb einer Funktion, die eine Zeichenkette ermittelt, ein Objekt vom Typ `StringBuilder` benutzt werden, um das Ergebnis durch schrittweise Änderungen aufzubauen. Wird das `StringBuilder`-Objekt zu Beginn

der Verarbeitung erzeugt und am Ende in das Ergebnis umgewandelt, so sind aus Sicht des Aufrufers keine Seiteneffekte feststellbar.

Ein Werttyp für Zeichenketten kann einfach und effizient implementiert werden, wenn seine interne Repräsentation durch ein änderbares Zeichen-Array – also durch ein Objekt – dargestellt wird. Die Werteigenschaften können dadurch sichergestellt werden, dass das Zeichen-Array durch die Operationen der Wertklasse nicht von Klienten übernommen, nicht an Klienten weitergegeben und nach seiner Erzeugung nicht verändert wird – Bedingungen, die durch entsprechende Sprachregeln schwer sicherzustellen sind.

Besonders das Ausschließen von Veränderungen ist im allgemeinen nicht sprachseitig überprüfbar. Objektorientierte Sprachen lassen im Prinzip in allen Methoden Zustandsänderungen zu und kennen keine Sprachmechanismen, um nur lesende Operationen („Queries“) von Operationen mit Seiteneffekten („Mutator“) zu unterscheiden (s. dazu Abschnitt 3.3.7). Die Einführung einer solchen Trennung wäre eine erhebliche Modifikation im Sprachentwurf für *Objektklassen* und wird hier nicht weiter vertieft.

6.3.5 Gleichheit für Wertklassen

Gleichheit basiert bei Wertklassen auf den Datenfeldern. Stimmen alle Datenfelder überein, so muss der gleiche Wert vorliegen. Anderenfalls wäre der primäre Selektor nicht referentiell transparent, und dann könnte auch nicht die referentielle Transparenz der darauf aufbauenden Selektoren gewährleistet werden.

Der Sprachentwurf sieht vor, dass für Wertklassen die Sprachumgebung die Gleichheit zur Verfügung stellt. Sie vergleicht paarweise die korrespondierenden Datenfelder und verknüpft die Ergebnisse mit logischer Konjunktion. Im obigen Beispiel verhält sich der Vergleich⁴ zweier Ausdrücke vom Typ `Point`,

`p1 = p2`

so, als sei er folgendermaßen implementiert:

```
return p1.x == p2.x && p1.y == p2.y;
```

Jede Wertklasse verfügt automatisch über diese Gleichheit. Sie wird von der Sprachumgebung generiert, sie muss nicht (und kann nicht) vom Programmierer der Wertklasse implementiert werden.

Die beschriebene Gleichheit betrachtet übereinstimmende interne Repräsentationen als gleich und unterschiedliche interne Repräsentationen als ungleich. Sie eignet

⁴Als Symbol für die Gleichheit wird hier, angelehnt an die mathematische Schreibweise, abweichend z. B. von Java oder C#, das Symbol „`==`“ verwendet.

sich für viele einfach aufgebaute Wertklassen, wie Point (repräsentiert durch kartesische Koordinaten) oder Datum (repräsentiert durch Tag, Monat und Jahr).

Es gibt aber auch Wertklassen, bei denen ein konzeptioneller Wert über mehrere Repräsentationen verfügen kann. Ein Beispiel sind rationale Zahlen – die Menge aller Zahlen, die als Quotient zweier ganzer Zahlen darstellbar sind. Hier gilt z. B. $1/3 = 2/6 = 3/9 = 4/12 \dots$

Auch solche Werttypen können so programmiert werden, dass die eben beschriebene, sprach-generierte Gleichheit korrekt funktioniert. Es erfordert jedoch einen zusätzlichen programmiertechnischen Schritt: Der Programmierer der Wertklasse muss für eine eindeutige Repräsentation sorgen, er muss die interne Repräsentation auf eine *standardisierte Form* bringen. Im Beispiel der rationalen Zahlen kann das durch Kürzen auf eine teilerfremde Darstellung bewerkstelligt werden, z. B. mit Hilfe des Euklid'schen Algorithmus. Die Ermittlung der eindeutigen Darstellung ist in *jedem* Selektor durchzuführen, bevor damit der primäre Selektor aufgerufen wird.

Das folgende Code-Beispiel zeigt eine Implementation einer Wertklasse Rational nach diesem Verfahren. Der Selektor `getRational` kürzt erst die Eingabeparameter und ruft dann mit dieser eindeutigen Darstellung den primären Selektor auf:

```
valueClass Rational {
  Int zaehler;
  Int nenner;
  Rational getRational(Int z, Int n) {
    ...
    Int g := ggT(z,n); // größter gemeinsamer Teiler
    Int z1 := z/g;
    Int n1 := n/g;
    return select(z1,n1);
  }
  ...
}
```

Mit diesem Programmierschema können Aufrufe eines Selektors mit verschiedenen Parametern den gleichen Wert liefern. Z. B. ergeben `getRational(1, 3)` und `getRational(2, 6)` den gleichen Wert (nämlich den mit der internen Darstellung `zaehler=1, nenner=3`). Umgekehrt gibt es Zahlenpaare, die in dieser Wertklasse nicht als Repräsentation für rationale Zahlen benutzt werden (nämlich alle kürzbaren Zahlenpaare wie (2,6), (3,9), etc.).

Eine eindeutige Darstellung herzustellen ist nur möglich, weil der primäre Selektor vor Klienten verborgen ist (s. dazu Abschnitt 6.3.3). Wäre er auch für Klienten zugänglich, könnte sein direkter Aufruf zu ungekürzten, und damit zu mehrdeutigen, Darstellungen einer rationalen Zahl führen.

Baker [Bak93] und Bloch [Blo08] beschreiben ebenfalls die Programmierung einer eindeutigen Repräsentation; sie bezeichnen sie als „*kanonische Form*“. Ihr Vorteil

besteht darin, dass die standardmäßig von der Sprache generierte Gleichheit, deren Korrektheit gewährleistet ist, zum Einsatz kommen kann.

Entwurfs-Alternative: programmierer-definierte Gleichheit

Es ist nicht sicher, ob für jede Wertklasse eine kanonische Form existiert. Baker weist auch darauf hin, dass die Ermittlung einer kanonischen Form unter Effizienzgesichtspunkten Nachteile aufwerfen kann [Bak93, p. 18]. Bei komplex aufgebauten Werten kann es vorzuziehen sein, die Darstellung eines Wertes nicht zu standardisieren, sondern für einen konzeptionellen Wert mehrere Repräsentationen zuzulassen. Ein Beispiel dafür ist eine Wertklasse für die Abbildung von Mengen, deren interne Repräsentation durch red-black-trees (s. dazu Guibas und Sedgewick[GS78]) implementiert wird.

Alternativ zum oben vorgeschlagenen Entwurf, bei dem die Wertgleichheit immer sprachgeneriert ist, kann ein alternativer Sprachentwurf eine programmierer-definierte Gleichheit vorsehen. Dazu ist ein Sprachmechanismus erforderlich, mit dem der Programmierer einer Wertklasse eine vom Standard abweichende Implementation für die Gleichheit angeben kann (ähnlich einem Überladen des Gleichheitsoperators).

Für diesen Mechanismus wird hier das Schlüsselwort `equality` verwendet. Im Beispiel einer Wertklasse `Rational`, die auch ungekürzte Repräsentationen für ihre Elemente zulässt, kann die Gleichheit folgendermaßen programmiert werden:⁵

```
valueClass Rational {
  Int zaehler;
  Int nenner;
  ...
  // Definition der Gleichheit
  Bool equality (Rational r1, Rational r2) {
    return r1.zaehler * r2.nenner = r2.zaehler * r1.nenner;
  }
  ...
}
```

Dieser Mechanismus ermöglicht mehrere Repräsentationen für einen Wert; sie werden mit Hilfe der Gleichheit zusammengeführt. Ein solcher Sprachentwurf ist flexibler als die oben beschriebene, sprach-generierte Gleichheit. Gleichzeitig ist er weniger sicher, denn Gleichheit unterliegt einem komplexen Regelwerk:

Gleichheit muss eine *Äquivalenzrelation* bilden (d. h. reflexiv, symmetrisch und transitiv sein, siehe dazu auch Bloch [Blo08, Item 8]).

⁵Aus Gründen der Deutlichkeit wird die Gleichheit hier mit derselben Notation definiert wie jede andere Funktion. Die Angaben der Typen, `Rational` als Typ der beiden Parameter und `Bool` als Rückgabety, sind aber redundant, da sie für die Gleichheit bereits festgelegt sind. Im Prinzip wäre auch eine knappere Syntax vorstellbar, die auf diese Angaben verzichtet.

Gleichheit muss zu allen anderen Operationen des Werttyps passen. Wenn $a=b$ gilt, muss für jede Funktion f aufgrund der referentiellen Transparenz auch $f(a)=f(b)$ gelten. Die Einhaltung dieser Bedingung kann bei einer programmiererdefinierten Gleichheit aufwändig und komplex sein.⁶

Bei der zuvor beschriebenen, von der Sprachumgebung generierten Gleichheit sind diese beiden Bedingungen erfüllt. Bei einer manuell implementierten Gleichheit liegt die Einhaltung dieser Regeln dagegen in der Verantwortung des Anwendungsprogrammierers. Das macht die Gleichheit fehleranfällig. Auch macht der zusätzliche, für die Implementierung der Gleichheit notwendige Mechanismus die Sprachunterstützung komplexer und erhöht so den Lernaufwand.

6.3.6 Aufzählungs-Wertklassen

Die bis hier beschriebene Sprachunterstützung ermöglicht die Definition von Wertklassen, deren interne Darstellung aus ein oder mehreren Datenfeldern besteht. Diese Art der Definition wird im weiteren als „*strukturierte Wertklasse*“ bezeichnet.

Typen können auch mittels *Aufzählung* definiert werden (s. auch Abschnitt 5.3.2). Ein Beispiel dafür ist ein Typ für die Spielkartenfarben eines Skatspiels. Er kann z. B. in der Programmiersprache C folgendermaßen definiert werden [KR88]:

```
enum kartenfarbe {KREUZ, PIK, HERZ, KARO};
```

Aufzählungs-Werttypen erfordern *nicht* zwingend eine spezifische Sprachunterstützung. Sie können mit den eben beschriebenen Sprachmitteln abgebildet werden. Dazu wird eine strukturierte Wertklasse mit einem einzigen Datenfeld (z. B. vom Typ `Int`) und mehreren parameterlosen Selektoren, basierend auf unterschiedlichen Werten der Repräsentation, programmiert. Ein Werttyp für die Abbildung von Spielkartenfarben kann z. B. gemäß der in diesem Kapitel eingeführten Notation mit folgender Wertklasse definiert werden:

```
valueClass Kartenfarbe {  
    Int number;  
    Kartenfarbe kreuz() {return select(1);};  
    Kartenfarbe pik()   {return select(2);};  
    Kartenfarbe herz()  {return select(3);};  
    Kartenfarbe karo()  {return select(4);};  
}
```

Diese Programmierung ist, verglichen mit einer direkten Sprachunterstützung wie der in C, umständlich und lang; und sie stellt eine Überspezifikation dar. Dass Elemente des Typs `Kartenfarbe` durch den Typ `Int` repräsentiert werden und welche

⁶Kapitel 8 befasst sich eingehender mit den genannten Regeln und mit den Konsequenzen, die eine programmiersprachliche Trennung von Werttypen und Objekttypen auf die Konzepte Gleichheit und Identität hat.

Werte von `Int` welches Element von Kartenfarbe darstellen, hat keine Auswirkung auf das von Klienten beobachtbare Verhalten des Typs.

Nicht alle Aufzählungstypen sind Werttypen. Auch Objekttypen können durch Aufzählung ihrer Elemente definiert werden. Der Typkonstruktor `enum` in Java unterstützt das, indem er für Aufzählungen änderbare Datenfelder und ändernde Operationen zulässt (s. dazu auch Abschnitt 5.3.2).

Weil diese Arbeit die klare Trennung von Werttypen und Objekttypen zum Ziel hat, unterscheidet sie auch Aufzählungs-Werttypen und Aufzählungs-Objekttypen. Der nachfolgend beschriebene Sprachentwurf bezieht sich ausschließlich auf Aufzählungs-*Werttypen*.⁷

Sprachunterstützung für Aufzählungen

Aufzählungs-Wertklassen werden durch einen eigenen Sprachmechanismus unterstützt. Er wird mit dem Schlüsselwort `enum` eingeleitet und führt die zur Aufzählung gehörenden Elemente durch voneinander verschiedene Bezeichner auf. (Er ähnelt darin der oben aufgeführten Sprachunterstützung für Aufzählungen in C.)

Der obige Typ `Kartenfarbe` wird mit einem solchen Mechanismus definiert durch:

```
enum Kartenfarbe {kreuz, pik, herz, karo};
```

Die deklarierten Bezeichner – im Beispiel `kreuz`, `pik`, `herz` und `karo` – sind für Klienten zugänglich. Sie können als syntaktische Sonderform von Selektoren aufgefasst werden: Sie ermöglichen den Zugriff auf einen Wert des Typs und spezifizieren, welcher Wert auszuwählen ist. Sie sind referentiell transparent, denn sie ergeben für gleiche Parameter den gleichen Wert – wobei hier ein Grenzfall vorliegt: diese Selektoren haben keine Parameter und liefern darum *immer* den gleichen Wert.

Die Sprachumgebung generiert für Aufzählungen die Gleichheit, dabei sind die innerhalb einer `enum` definierten Bezeichner paarweise ungleich.

Der Vorteil dieser Sprachunterstützung liegt in ihrer Kürze, ihrer Aussagekraft und ihrer leichten Handhabung. Sie vermeidet die im obigen Beispiel beobachtete Überspezifikation und delegiert Einzelheiten der technischen Umsetzung – Repräsentation, interne Codierung für jedes Element – an die Sprachumgebung.

⁷Anhang A.2 vergleicht anhand von Java-Codebeispielen die Implementierung von Aufzählungs-Typen mit und ohne dedizierte Sprachunterstützung, jeweils für einen Aufzählungs-Werttyp und für einen Aufzählungs-Objekttyp. Die Gegenüberstellung zeigt, dass der Aufzählungs-Werttyp mit Hilfe der Sprachunterstützung knapper und prägnanter definiert werden kann als ohne sie. Der Aufzählungs-Objekttyp dagegen profitiert nur in vergleichsweise geringem Umfang von der Sprachunterstützung. Das legt es nah, im Sprachentwurf für Aufzählungs-*Objekttypen* auf eine dedizierte Sprachunterstützung zu verzichten.

Vor allem aber ermöglicht eine Sprachunterstützung für Aufzählungen die Definition von Typen, die nicht das Vorhandensein anderer Typen voraussetzen. Aufzählungs-Wertklassen können so den Startpunkt für die Programmierung weiterer Typen bilden. Aus ihnen können strukturierte Wertklassen zusammengesetzt werden, diese wiederum bilden Bestandteile von Objektklassen. Auf diese Weise lässt sich die Programmierung *aller* Typen in Bibliotheken verlagern. Vordefinierte Typen als Bestandteil der Programmiersprache sind im Prinzip nicht erforderlich, was den Umfang einer Sprachspezifikation deutlich reduzieren kann.

Für Aufzählungen können – wie für jede Wertklasse – weitere Funktionen definiert werden. Weil bei Aufzählungen keine Datenfelder vorhanden sind, muss sich eine solche Funktion auf die – sprachseitig vorgegebene – Gleichheit abstützen. Beispielsweise kann für Kartenfarbe eine Funktion definiert werden, welche die Punkte (beim Reizen im Skat) ermittelt:

```
Int punkte(Kartenfarbe k) {  
    if (k = kreuz) {return 12};  
    if (k = pik)   {return 11};  
    if (k = herz)  {return 10};  
    if (k = karo)  {return 9};  
}
```

Um solche Programmierungen prägnanter zu gestalten, kann für Aufzählungen eine *Mehrfachverzweigung* unterstützt werden (analog zur `switch`-Anweisung in Java oder der `case`-Anweisung in Pascal). Die Operation `punkte` kann damit folgendermaßen implementiert werden (hier werden für die Mehrfachverzweigung die Schlüsselworte `switch` für den Test und `case` für die Zweige zugrunde gelegt):

```
Int punkte (Kartenfarbe k) {  
    return  
        switch k  
            case kreuz: 12  
            case pik:   11  
            case herz:  10  
            case karo:   9  
}
```

Mehrfachverzweigungen sparen nicht nur Schreibaufwand ein und verkürzen den Quelltext. Sie erhöhen auch die Sicherheit des Codes, denn

- der zu prüfende Ausdruck (im Beispiel: `k`) muss nur einmal aufgeführt werden,
- die Reihenfolge der Zweige ist (anders als bei geschachtelten `if`-Anweisungen) vertauschbar, ohne das Ergebnis zu ändern,
- die Vollständigkeit der Zweige ist durch die Sprachumgebung überprüfbar,

- es kann sichergestellt werden, dass jeder Wert des Typs nur in einem Zweig verwendet wird,
- eine Mehrfachverzweigung kann eine Anweisung oder (wie im obigen Beispiel) ein Ausdruck sein.

Aufzählungs-Wertklassen und strukturierte Wertklassen definieren nicht zwei verschiedene Kategorien von Werttypen, sondern es sind zwei *verschiedene Verfahren*, einen Werttyp zu definieren und damit ein Implementierungsdetail. Für einen Werttyp mit wenigen Elementen, wie Kartenfarbe, ist in der Regel eine Definition mittels Aufzählung praktikabler. Für einen Werttyp mit vielen Elementen und umfangreichen Operationen, wie `Point` oder `Rational`, ist meist eine Definition durch eine strukturierte Wertklasse besser geeignet. Es kann Grenzfälle geben, für die beide Definitionsverfahren gleichermaßen in Frage kommen. Beispielsweise kann ein Werttyp `Farbe` durch folgende Aufzählung definiert werden:

```
enum Farbe {schwarz, weiss, rot, gruen,  
            blau, gelb, cyan, magenta};
```

Alternativ kann der rot/grün/blau-Anteil einer Farbe (gemäß additiver Farbmischung) in eine entsprechende Struktur des Typs eingehen und resultiert dann in folgender Definition:

```
valueClass Farbe {  
    Bool enthaeltRot;  
    Bool enthaeltGruen;  
    Bool enthaeltBlau;  
    Farbe schwarz() {return select(false, false, false);}  
    Farbe weiss() {return select(true, true, true);}  
    Farbe rot() {return select(true, false, false);}  
    Farbe gruen() {return select(false, true, false);}  
    Farbe blau() {return select(false, false, true);}  
    Farbe gelb() {return select(true, true, false);}  
    Farbe cyan() {return select(false, true, true);}  
    Farbe magenta() {return select(true, false, true);}  
}
```

Damit ein Wechsel zwischen den beiden Definitionen ohne Beeinträchtigung von Klienten möglich ist, muss die Sprachunterstützung bei parameterlosen Selektoren das Weglassen der – ohnehin leeren – Klammern erlauben. Ein Klient des Werttyps `Farbe` kann dann z. B. mit dem Bezeichner `rot` auf ein Element des Typs zugreifen, unabhängig davon, ob er durch eine strukturierte Wertklasse oder mittels Aufzählung definiert wurde.

Integration von Aufzählungen in den Werttypkonstruktor

Die Sprachunterstützung integriert Aufzählungen in den Typkonstruktor `valueClass` und verdeutlicht damit, dass auch sie Werttypen definieren. `enum` bildet damit *keinen* eigenständigen Typkonstruktor, sondern eine optionale Klausel innerhalb einer `valueClass`.

Die vollständige Definition für das (oben nur angedeutete) Beispiel Kartenfarbe lautet damit folgendermaßen:

```
valueClass Kartenfarbe {  
    enum {kreuz, pik, herz, karo}  
}
```

Auf diese Weise sind genau zwei Typkonstruktoren vorhanden, `valueClass` und `objectClass`, was die Zweiteilung des Typsystems klar zum Ausdruck bringt. Zusätzlich ermöglicht die Integration von Aufzählungen in den Typkonstruktor `valueClass` auch eine *gemischte Definition* für einen Werttyp. D. h. ein oder mehrere Elemente des Typs werden mittels Aufzählung definiert, und für die restlichen Elemente wird eine interne Struktur in Form von Datenfeldern angegeben. Eine solche Definition ist insbesondere für rekursiv definierte Typen nützlich, für die eine (einelementige) Aufzählung einen „Startwert“ festlegt. Beispielsweise kann ein Werttyp `String` durch den leeren String und Aufbau aller weiteren Elemente aus dem ersten Zeichen und dem nachfolgenden String definiert werden:

```
valueClass String {  
    enum {empty};  
    Char first;  
    String rest;  
    String fromCharacter (Char c) {  
        return select (c, empty);  
    }  
    String concat(String s1, String s2) {  
        if (s1 = empty) { return s2; }  
        else{ return select (s1.first, concat(s1.rest, s2)); }  
    }  
    ...  
}
```

6.3.7 Benutzung von Wertklassen

Klienten benutzen Wertklassen und Objektklassen auf die gleiche Weise: Sie deklarieren Variablen des entsprechenden Typs, und sie rufen zugehörige Operationen auf. In dieser Hinsicht ist der Sprachentwurf – abgesehen von den oben beschriebenen Unterschieden in der *Syntax* der Operationsaufrufe – orthogonal.

Das gilt gleichermaßen für Typprüfungen. Die Sprachumgebung überprüft die Typ-Korrektheit der Zuweisungen und Operationsaufrufe bei Werttypen und Objekttypen auf die gleiche Weise.

Die Besonderheiten von Werttypen ziehen einige zusätzliche Restriktionen nach sich, die sprachseitig überprüft werden können und dadurch die Sicherheit weiter erhöhen:

Keine isolierten Aufrufe von Funktionen

Jeder Aufruf einer Funktion muss in einen Kontext eingebunden sein (z. B. als rechte Seite einer Zuweisung oder als aktueller Parameter einer anderen Operation). Weil Funktionen frei von Seiteneffekten sind, würde ein isolierter Aufruf einer Funktion nichts bewirken, es muss sich dabei um einen Programmierfehler handeln.

Isolierte Aufrufe von Funktionen werden deshalb vom Compiler als Fehler zurückgewiesen.^{8 9} Z. B. wird folgender Code vom Compiler als fehlerhaft markiert:

```
Rational r1;  
Rational r2;  
...  
add(r1, r2); // Fehler, isolierter Funktionsaufruf
```

während folgender Code zulässig ist:

```
Rational r3;  
...  
r3 := add(r1, r2); // ok
```

Kein Vergleich typ-inkompatibler Ausdrücke

Weil Gleichheit für Wertklassen ein Sprachprimitiv ist, kann der Compiler sicherstellen, dass nur Ausdrücke des gleichen Typs – oder, im Fall eines polymorphen Typsystems, nur Ausdrücke kompatibler Typen – akzeptiert werden. Vergleiche von Ausdrücken unterschiedlicher oder inkompatibler Typen – z. B. zwischen `Point` und `Kartenfarbe` oder zwischen einer Wertklasse und einer Objektklasse – liefern per se immer `false`, es muss sich deshalb um Programmierfehler handeln. Der Compiler kann deshalb Vergleiche typ-inkompatibler Ausdrücke als fehlerhaft abweisen.

⁸Java enthält ähnliche Prüfungen für primitive Datentypen.

⁹Eine Programmiersprache, die bei Objektklassen die Trennung von seiteneffektfreien und seiteneffekt-behafteten Operationen („Queries und Mutator“) unterstützt, wie in Abschnitt 6.3.4 beschrieben, kann auch für die Aufrufe von Queries solche Prüfungen etablieren.

6.3.8 Überblick: Sprachunterstützung für Werttypen

Abschnitt 6.3 hat einen Typkonstruktor für Werttypen schrittweise entwickelt und durch Beispiele illustriert. Wie bereits die große Bandbreite der jeweils diskutierten Entwurfsalternativen deutlich gemacht hat, ist eine solche Sprachunterstützung nicht eindeutig. Die Gestaltung des Typkonstruktors kann nicht aus den charakteristischen Eigenschaften von Werttypen deduziert werden.

Nachfolgend sind die zum vorgeschlagenen Typkonstruktor gehörenden Sprach-elemente tabellarisch zusammengefasst:

Tabelle 6.1 enthält die Wertklassen-spezifischen Schlüsselworte,
Tabelle 6.2 beschreibt die für Wertklassen zur Verfügung gestellte Funktionalität,
Tabelle 6.3 listet die Sprachregeln auf, denen Wertklassen unterliegen.
Eine Grammatik des Sprachentwurfs (in EBNF) ist in Anhang A.1 angegeben.

Schlüsselworte	
Schlüsselwort für den Typkonstruktor	<code>valueClass</code>
Schlüsselwort für den Aufruf des primären Selektors	<code>select</code>
Schlüsselwort für Aufzählungen	<code>enum</code>
Schlüsselwort für programmierer-definierte Gleichheit	<code>equality</code>

Tabelle 6.1: Schlüsselworte für die Definition einer Wertklasse

Funktionalität	
<i>primärer Selektor</i>	basiert auf den Datenfeldern der Wertklasse
<i>Gleichheit</i>	verhält sich bei strukturierten Wertklassen wie ein Vergleich aller Datenfelder, erkennt bei Aufzählungen verschiedene Bezeichner als unterschiedlich, gleiche als gleich

Tabelle 6.2: Sprach-generierte Funktionalität für Wertklassen

Sprachregeln (vom Übersetzer prüfbar)
Datenfelder von strukturierten Wertklassen sind unveränderlich.
Die Typen der Datenfelder in Wertklassen sind Werttypen.
Die Typen der formalen Parameter und der Rückgabe von Wertoperationen sind Werttypen.
In der Implementation von Wertoperationen darf nicht auf Objektoperationen zugegriffen werden.
Jede Wertklasse hat mindestens einen Selektor.
Mindestens ein Selektor hat keinen Parameter der eigenen Wertklasse.
Bei strukturierten Wertklassen ruft mindestens ein Selektor den primären Selektor auf.
Wertoperationen müssen immer in einem Kontext aufgerufen werden, nicht isoliert.
Ein Vergleich zweier Wert-Ausdrücke ist nur bei gleichen oder kompatiblen Typen zulässig.

Tabelle 6.3: Sprachregeln für Wertklassen

6.4 Abgleich mit den Anforderungen

Der vorgeschlagene Typkonstruktor für Werttypen erfüllt die oben (s. Abschnitt 6.1) gestellten Anforderungen:

Sicherheit

Das beschriebene Regelwerk stellt sicher, dass ein mit `valueClass` definierter Typ die konzeptionellen Eigenschaften von Werttypen aufweist:

- Die Elemente einer strukturierten Wertklasse sind *unveränderlich*, weil ihre Datenfelder nicht verändert werden können, weil deren Typen ebenfalls Werttypen sind und deshalb für sie keine verändernden Operationen aufgerufen werden können. Die Elemente von Aufzählungs-Wertklassen sind unveränderlich, weil sie nicht über eine interne, veränderbare Struktur verfügen.
- Die Elemente einer strukturierten Wertklasse sind *unerzeugbar*, weil sie sich aufgrund ihrer Selektoren und in Zusammenhang mit der Funktionsweise der Gleichheit aus Klientensicht so verhalten, als existierten sie schon immer. Die Elementmenge von Aufzählungs-Wertklassen ist per se festgelegt.
- Die Operationen von Wertklassen sind *seiteneffektfrei* und *referentiell transparent* aufgrund der beiden zuvor genannten Regeln und weil innerhalb ihrer Implementation Aufrufe von Objektoperationen nicht zulässig sind.

Klarheit

Wertklassen sind aufgrund des eigenständigen Typkonstruktors im Quelltext von Objektklassen unterscheidbar. Die verschiedene Aufrufsyntax von Funktionen (mathematisch) und Objektoperationen (Punktnotation) hilft zusätzlich, sie auf einen Blick voneinander zu unterscheiden.

Einfachheit

Der Aufwand für die Definition und Benutzung einer Wertklasse ist dem für die Definition und Benutzung einer Objektklasse vergleichbar. Es fällt weder für den Programmierer der Wertklasse noch für ihre Klienten Aufwand für die Sicherstellung des wertartigen Verhaltens an; diese wird durch die Sprachregeln gewährleistet. Eine eigene Unterstützung für Aufzählungen und zugehörige Mehrfachverzweigungen tragen dazu bei, die Programmierung solcher Spezialfälle zu erleichtern und besser lesbar zu gestalten.

Umfassende Unterstützung verschiedener Werttypen

Der Typkonstruktor `valueClass` ermöglicht die Definition von Werttypen, die aus anderen Typen zusammengesetzt werden können, wie Punkt oder Datum. Er enthält eine gesonderte Unterstützung für Aufzählungs-Werttypen. Er lässt rekursive Definitionen zu; damit können auch Werttypen mit variabler Größe wie Zeichenketten oder Typen mit baumartiger Struktur abgebildet werden.

Trennung von Schnittstelle und Implementation

Weil die interne Repräsentation von Wertklassen immer verborgen ist – die Datenfelder sind gekapselt, der primäre Selektor darf nur innerhalb einer Wertklasse aufgerufen werden – kann die Implementation unabhängig von der Schnittstelle der Wertklasse variiert werden. Sogar ein Wechsel von einer strukturierten Wertklasse zu einer Definition mittels Aufzählung ist möglich, ohne Klienten anpassen zu müssen.

6.5 Zusammenfassung

Kapitel 6 schlug vor, Werttypen auf der Ebene der Programmiersprache zu unterstützen und beschrieb den Entwurf eines entsprechenden Sprachkonstruktes. Gestaltung und Funktionsweise einer Sprachunterstützung für Werttypen wurden schrittweise und anhand von Beispielen entwickelt. Soweit vorhanden, wurden alternative Entwürfe vorgestellt und ihre Vor- und Nachteile diskutiert.

Der Bedarf nach einer Sprachunterstützung wurde mit den Erkenntnissen der beiden vorangegangenen Kapitel begründet, nämlich dass (Kap. 4) eine Abbildung von Werttypen mit softwaretechnischen Verfahren unsicher, unklar und aufwändig ist, dass (Kap. 5) bestehende objektorientierte Programmiersprachen Werttypen aber

nur rudimentär unterstützen. Aus den beobachteten Mängeln wurden Anforderungen an eine Sprachunterstützung abgeleitet: sie soll sicher, klar und einfach sein, umfassend, flexibel und sich in ein objektorientiertes Umfeld eingliedern.

Werttypen werden mit Hilfe von Wertklassen programmiert. Für Wertklassen wurde ein dedizierter Typkonstruktor („*valueClass*“) vorgesehen, der gleichberechtigt neben einem Typkonstruktor für Objekttypen („*objectClass*“) steht. Dadurch bringt der Sprachentwurf die Zweiteilung des Typsystems klar zum Ausdruck.

Die Implementation einer Wertklasse beschreibt die interne Repräsentation ihrer Elemente (in Form von „Datenfeldern“) und die auf die Elemente anwendbaren Operationen („Funktionen“). Für Wertklassen wurde ein zur Übersetzungszeit überprüfbares Regelwerk aufgestellt, das die in Kap. 3 beschriebenen Werteigenschaften sicherstellt. Z. B. müssen die Datenfelder von Wertklassen unveränderlich sein, ihre Typen müssen ebenfalls Werttypen sein, in der Implementation von Wertklassen darf nicht auf Objekttypen zugegriffen werden, Gleichheit beruht bei Wertklassen auf einem Vergleich der Datenfelder. Für Wert-Aufzählungen wurde eine eigene Unterstützung vorgesehen, integriert in den Typkonstruktor für Werttypen.

Wertklassen verfügen, anders als Objektklassen, nicht über Konstruktoren. Kapitel 6 stellte das Konzept der Selektoren vor, begründete, warum die Implementierung von Selektoren ein eigenes Sprachprimitiv erfordert (den *primären Selektor*) und beschrieb dessen Gestaltung und Funktionsweise.

Die Benutzung durch Klienten erfolgt für Wertklassen und für Objektklassen prinzipiell auf die gleiche Weise. Die Besonderheiten von Werttypen implizieren einige zusätzliche Regeln. Diese können sprachseitig geprüft werden und so die Sicherheit weiter erhöhen.

Es wurde gezeigt, dass die vorgeschlagene Sprachunterstützung die charakteristischen Eigenschaften von Werttypen gewährleistet und die eingangs gestellten Anforderungen erfüllt. Der zu Kapitel 6 gehörende Anhang gibt eine Grammatik für die vorgeschlagene Sprachunterstützung an.

Kapitel 7

Eine exemplarische Implementation: Die Forschungssprache VJ

Dieses Kapitel stellt die Forschungssprache VJ vor.

VJ – abkürzend für Value-Java – erweitert die objektorientierte Sprache Java um programmiererseitig definierbare Werttypen.

VJ basiert auf dem in dieser Arbeit entwickelten Konzept von Werten und Werttypen (s. Kap. 3). Entwurf und Implementierung eines VJ-Compilers wurden im Rahmen einer Diplomarbeit von J. Rathlev [Rat06] initiiert, unter Co-Betreuung der Autorin. VJ wurde auf der Konferenz SE2007 vorgestellt [RRS07].

7.1 Ziele von VJ

VJ dient als Machbarkeitsstudie („proof of concept“): Die Sprache zeigt, dass das Konzept von Werttypen tragfähig ist und in einem objektorientierten Umfeld umgesetzt werden kann. Ein wesentliches Ziel von VJ besteht darin, die Handhabung von Werttypen für den Anwendungsprogrammierer klar und einfach zu machen. Aus der Einbettung von Werttypen in eine bestehende Programmiersprache ergibt sich des weiteren die Forderung nach Kompatibilität zur Ausgangssprache.

7.1.1 Nachweis der Machbarkeit

Mit der Entwicklung von VJ wird der Nachweis geführt, dass eine programmiersprachliche Unterstützung für Werttypen möglich ist und dass Werttypen eine objektorientierte Sprache sinnvoll ergänzen. Dazu wird der Ansatz gewählt, eine bestehende Programmiersprache, nämlich Java, um Werttypen zu erweitern. Zum Nachweis der praktischen Umsetzbarkeit gehört auch die Entwicklung eines VJ-Compilers und die Programmierung von repräsentativen, aussagekräftigen Beispielen für Werttypen in VJ. Die Code-Beispiele evaluieren die korrekte Funktionswei-

se des VJ-Compilers, sie veranschaulichen die Vorteile einer expliziten Werttyp-Unterstützung und sie vermitteln einen Eindruck vom Umgang mit Werttypen in einem objektorientierten Umfeld.¹

7.1.2 Einfache Programmierung

Ausgehend von der Komplexität, die bei einer Abbildungen von Werten durch softwaretechnische Verfahren wie Muster oder Programmierkonventionen zu beobachten ist, bildet eine einfache Programmierung mit kompaktem Code ein vorrangiges Ziel des VJ-Sprachentwurfs. Der Entwurf steht unter der Maßgabe, dass Werttypen in VJ ähnlich einfach und geradlinig zu handhaben sind wie Objekttypen in Java. Das schließt ein, dass der Programmierer eines Werttyps sich auf die fachlichen Spezifika konzentrieren kann und dass kein umfangreicher Code zur Herstellung und Sicherung der grundlegenden Werteigenschaften erforderlich ist.

7.1.3 Kompatibilität zur Java

Aus dem Ansatz, Java um Werttypen zu erweitern, ergibt sich die Anforderung nach Kompatibilität zu Java. Das betrifft sowohl die Quelltext-Kompatibilität – jedes gültige Java-Programm ist auch ein gültiges VJ-Programm – als auch die Bytecode-Kompatibilität – bestehender Java-Code kann ohne Neu-Übersetzung aus VJ-Code heraus benutzt werden. Letzteres ist besonders für Java-Standardbibliotheken relevant.

7.2 Sprachentwurf

VJ umfasst alle Sprachkonstrukte von Java. Dieser Abschnitt beschreibt die Sprachkonstrukte und Sprachregeln, die über Java hinausgehen und die VJ für die Unterstützung von Werttypen einführt.

Die werttyp-spezifischen VJ-Sprachkonstrukte werden im Folgenden schrittweise, anhand eines durchgängigen Beispiels entwickelt. Dafür dient hier ein Typ `Date` für die Abbildung von Daten (= Plural von *Datum*). Ein Element dieses Typs entspricht einem Tag, der durch die drei Angaben Jahr, Monat und Tag (=laufende Nummer innerhalb des Monats) eindeutig beschrieben werden kann. Über die drei genannten Merkmale hinaus hat ein Datum weitere Eigenschaften. Z. B. kann festgestellt werden, welche Position ein Datum im jeweiligen Jahr einnimmt („Tag des Jahres“) oder

¹Aufgrund des Charakters einer Machbarkeitsstudie ist der Einsatz von VJ in industriellen Projekten kein ausdrückliches Ziel der hier vorgestellten Arbeit; er wird aber nicht von vornherein ausgeschlossen. Kommerzielle Projekte setzen in der Regel für ihre Werkzeuge einen (vertraglich) abgesicherten Support voraus, den eine experimentelle Programmiersprache wie VJ nicht leisten kann.

um welchen Wochentag es sich handelt. Elemente des Typs `Date` können aufeinander bezogen werden, z. B. kann die Anzahl der Tage zwischen zwei Daten ermittelt werden, oder ob ein Datum vor einem anderen Datum liegt.

Der Übergang vom julianischen zum gregorianischen Kalender – der in verschiedenen Ländern zu verschiedenen Zeitpunkten stattfand – wird vernachlässigt; aus Gründen der Vereinfachung wird durchgängig von einem Datum im gregorianischen Kalender ausgegangen.

7.2.1 Sprachkonstrukte für Werttypen

VJ fokussiert auf *zusammengesetzte Werttypen*, d. h. solche, deren Elemente durch eine Menge von Eigenschaften charakterisiert werden können, wie `Datum` (Jahr, Monat und Tag), `Punkt` (x- und y-Koordinate), oder `Geldbetrag` (Betrag und Währung).

VJ unterstützt zusammengesetzte Werttypen durch *Wertklassen*. Dafür setzt VJ folgende Sprachmittel ein:

Eigener Typkonstruktor

Eine Wertklasse wird in VJ mit Hilfe eines eigenen Typkonstruktors definiert, eingeleitet mit dem Schlüsselwort `valueclass`. Objektklassen werden in VJ, bedingt durch die Kompatibilität zu Java, weiterhin mit dem Typkonstruktor `class` definiert. Wie eine Objektklasse beschreibt auch eine Wertklasse einen Typ und seine zugehörige Implementation.

Eine Wertklasse `Date` zur Beschreibung eines Datums-Typs wird z. B. folgendermaßen definiert:

```
valueclass Date {  
    ...  
}
```

Konstituierende Eigenschaften

Die Eigenschaften, die jedes Element des Werttyps festlegen, werden in VJ als *konstituierende Eigenschaften* bezeichnet. Sie werden innerhalb einer Wertklasse wie Exemplarvariablen einer Java-Klasse definiert. Mit den konstituierenden Eigenschaften `Jahr`, `Monat` und `Tag` (`year/ month/ day`) erhält man folgende Definition:

```
valueclass Date {  
    int year;  
    int month;  
    int day;  
    ...  
}
```

Eine Wertklasse kann zusätzlich zu den konstituierenden Eigenschaften Klassenvariablen definieren. Wie in einer Objektklasse beziehen sie sich auf die Wertklasse als Ganzes, nicht auf eines ihrer Elemente, und werden mit dem Modifier `static` markiert.

Alle innerhalb einer Wertklasse definierten Felder, die nicht `static` sind, werden als konstituierende Eigenschaften angesehen. Eigenschaften, die nicht konstituierend sind, sieht VJ nicht vor. Ein gesondertes Schlüsselwort zur Kennzeichnung konstituierender Eigenschaften ist darum entbehrlich.

Die konstituierenden Eigenschaften sind standardmäßig für Klienten lesbar. Abweichend davon können sie als `private` deklariert werden und sind dann nur innerhalb der Wertklasse zugänglich.

Methoden von Wertklassen

In einer Wertklasse können beliebige Exemplarmethoden definiert werden, mit dem Modifier `static` auch Klassenmethoden.

Im folgenden Beispiel definiert die Wertklasse `Date` eine Exemplarmethode `dayOfYear`. Aufgrund des komplexen Regelwerkes (Anzahl der Tage pro Monat, Berücksichtigung von Schaltjahren, etc.) ist ihre Implementation nur angedeutet:

```
valueclass Date {  
    ...  
    public int dayOfYear() {  
        int result = (this.month - 1) * 30 + this.day;  
        // Verarbeitung für Monate mit 31 Tagen,  
        // für Februar, für Schaltjahre  
        if (this.month == 2) { result = ... }  
        ...  
        return result;  
    }  
    ...  
}
```

Wertwähler

VJ generiert für jede Wertklasse eine Operation, die ein Element dieser Wertklasse anhand seiner konstituierenden Eigenschaften auswählt und an den Aufrufer liefert; sie wird als *Wertwähler* bezeichnet. Der Wertwähler ist nur implizit vorhanden, d. h. im Quellcode nicht sichtbar. Die Parameter des Wertwählers entsprechen in Typ und Reihenfolge den konstituierenden Eigenschaften, sein Rückgabetyt ist die Wertklasse selbst.

Der Aufruf des Wertwählers folgt einer eigenen Syntax. Er wird eingeleitet durch das Schlüsselwort `const`, gefolgt vom Namen der Wertklasse und der Angabe der

erforderlichen Parameter.²

Beispielsweise liefert die Anweisung

```
Date myDate = const Date(2006,2,13);
```

den 13. Februar 2006, also das Element der Wertklasse `Date` mit den konstituierenden Eigenschaften `year == 2006`, `month == 2`, `day == 13`.

Der Wertwähler ist immer für Klienten aufrufbar, unabhängig davon, welche Sichtbarkeit die konstituierenden Eigenschaften der Wertklasse haben.

Alternative Wertwähler, d. h. Operationen, die ein Element der Wertklasse basierend auf anderen als den konstituierenden Eigenschaften liefern, verfügen in VJ nicht über eine gesonderte sprachliche Unterstützung. Sie können als Klassenmethoden definiert werden; dieses Programmierschema entspricht dem Entwurfsmuster der Fabrikmethode (s. Gamma et al. [GHJV98]).

Beispielsweise kann eine Operation `dateYYYYDDD` programmiert werden, die ein Element der Wertklasse `Date` anhand des Jahres und des Tages innerhalb des Jahres liefert:

```
valueclass Date {  
    ...  
    public static dateYYYYDDD(int year, int dayOfYear) {  
        int month1 = ... // Monat ermitteln  
        int day1 = ... // Tag ermitteln  
        return const Date(year,month1,day1);  
    }  
    ...  
}
```

Sie wird aufgerufen durch

```
Date myDate = Date.dateYYYYDDD(2006,44); // 13.2.2006
```

Gleichheit bei Wertklassen

VJ unterscheidet bei Wertklassen nicht zwischen Gleichheit und Identität. Bei Wertklassen prüft der von VJ zur Verfügung gestellte Gleichheitsoperator „==“ die Gleichheit aller konstituierenden Eigenschaften, jeweils mit dem Operator „==“, und damit rekursiv, falls es sich bei diesen ebenfalls um Wertklassen handelt. (Er entspricht damit der in Abschnitt 6.3.5 beschriebenen kanonischen Gleichheit.) Die Funktionsweise des Operators „==“ stellt damit sicher, dass zwei Aufrufe des Wertwählers mit den gleichen Parametern das gleiche Ergebnis liefern, der Wertwähler

²`const` ist ein in Java reserviertes Schlüsselwort, aber bisher nicht in Gebrauch. Die Entwurfsentscheidung, den Wertwähler mittels `const` aufzurufen – statt z. B. mit `select` oder `value` – ist motiviert durch das Bestreben, Inkompatibilitäten zu bestehendem Java-Code auszuschließen.

also referentiell transparent ist.

Beispielsweise liefert in VJ der folgende Vergleich `true`:

```
if (const Date(2006,2,13) == const Date(2006,2,13)) {...}
```

Wertklassen verfügen in VJ über zwei Vergleiche; zusätzlich zum Operator „`==`“ gibt es für sie auch die Methode `equals`. Der Operator „`==`“ und die Methode `equals` verhalten sich bei Wertklassen fast gleich. Der Unterschied zwischen beiden besteht lediglich darin, dass der Operator „`==`“ immer ein Ergebnis `true` oder `false` liefert, während die Methode `equals` – da es sich um eine Exemplarmethode handelt – bei `null` als Receiver eine `NullPointerException` auslöst. Die Methode `equals` ist nicht verzichtbar, da sie in vielen Java-Bibliotheken für Vergleiche verwendet wird und aus Kompatibilitätsgründen zur Verfügung stehen muss.

Gültigkeitsprüfung

Es kann Wertklassen geben, für die nicht alle Kombinationen von konstituierenden Eigenschaften einen gültigen Wert beschreiben. Beispielsweise liefert die Kombination (2006/02/31) keinen gültigen Wert von `Date`. Um die ungültigen Kombinationen zu erkennen und auszufiltern, unterstützt VJ für Wertklassen die Methode `isValid`. Sie ist eine Klassenmethode, ihr Rückgabetyt ist `boolean` und ihre Parameter entsprechen, wie beim Wertwähler, in Typ und Reihenfolge den konstituierenden Eigenschaften der Wertklasse.

`isValid` ist vom Programmierer der Wertklasse so zu implementieren, dass sie für gültige Kombinationen von konstituierenden Eigenschaften `true` liefert und für ungültige `false`. VJ sorgt dafür, dass der Wertwähler als erstes die Methode `isValid` aufruft und im Fall des Ergebnisses `false` eine `NoSuchValueException` auslöst.

Beispielsweise unterscheidet die folgende `isValid`-Methode in der Wertklasse `Date` gültige und ungültige Kombinationen von `year`, `month` und `day`. Ihre Implementation ist aufgrund der umfangreichen Prüfungen – unterschiedlich lange Monate, Schaltjahre, etc. – hier nur angedeutet.

```
public static boolean isValid (int year, int month, int day) {  
    if (day < 1 || day > 31) {  
        return false;  
    }  
    ...  
    if ((month == 4 || month == 6 ..) && day == 31) {  
        return false;  
    }  
    ...  
    return true;  
}
```


Definiert der Programmierer einer Wertklasse keine `isValid`-Methode, entspricht das implizit einer `isValid`-Methode, die immer `true` liefert. Ist die Methode `isValid` als `public` definiert, können Klienten sie auch direkt aufrufen. Z. B. können sie damit vor dem Aufruf eines Wertwählers festzustellen, ob er ein gültiges Element der Wertklasse liefern würde.

Anhang B.1 fasst die bis hier aufgeführten Code-Fragmente zusammen und stellt den Quellcode der VJ-Wertklasse `Date` in zusammenhängender Form dar.

Aufzählungs-Werttypen

Die Wertklassen von VJ unterstützen nur zusammengesetzte Werttypen, keine Aufzählungs-Werttypen. Für Aufzählungen greift VJ auf den in Java bereits vorhandenen Typkonstruktor `enum` zurück; er wird unterstützt ab Java-Version 1.5, auf der VJ beruht.³

7.2.2 Sprachregeln

Um die charakteristischen Eigenschaften von Werten zu gewährleisten, unterliegen Wertklassen in VJ folgenden Einschränkungen:

Unveränderlichkeit der konstituierenden Eigenschaften

Konstituierende Eigenschaften können nicht verändert werden. Eine Zuweisung zu einer konstituierenden Eigenschaft löst einen Übersetzungsfehler aus.

Weil konstituierende Eigenschaften bereits gemäß Sprachentwurf unveränderlich sind, ist es nicht erforderlich, und nicht möglich, sie mit dem Modifier `final` zu versehen.

Nur Werttypen als konstituierende Eigenschaften

Der Typ einer konstituierenden Eigenschaft muss ebenfalls ein Werttyp sein. Als Werttypen zählen in VJ neben Wertklassen auch die primitiven Datentypen, Aufzählungen und die Klasse `java.lang.String` (s. u.).

Nur Werttypen als Parameter und Ergebnis von Methoden

Methoden von Wertklassen dürfen als Parameter nur Werttypen haben, und ihr Rückgabotyp muss ein Werttyp – oder ein Wertinterface (s. u.) – sein. Insbesondere darf der Rückgabotyp nicht `void` und kein Objekttyp sein.

³Weil der Typkonstruktor `enum` in Java – und damit auch in VJ – änderbare Datenfelder grundsätzlich zulässt, ist für Typen, die mittels `enum` definiert wurden, Seiteneffektfreiheit sprachseitig nicht garantiert. Der Sprachentwurf von VJ geht hier bewusst einen pragmatischen Kompromiss ein: Eine zusätzliche Sprachunterstützung, speziell für Aufzählungs-Werttypen, würde aufgrund des Vorhandenseins von `enum` dazu führen, dass die Sprache für Aufzählungen über zwei ähnliche Konstrukte verfügt und dadurch verwirren.

Keine geprüften Exceptions

Methoden von Wertklassen dürfen keine geprüften Exceptions („checked exceptions“) werfen.

Kein Zugriff auf Objekte

In der Implementation einer Wertklasse darf nicht auf Objekte zugegriffen werden: Objekterzeugung, Aufrufe von Klassenmethoden und Zugriffe auf Klassenvariablen von Objektklassen sind nicht zulässig.⁴

Ausnahmen sind Zugriffe auf `static final` Variablen, deren Typ ein Werttyp ist („Wertkonstanten“). Ebenfalls ausgenommen sind Aufrufe von Methoden der Klasse `java.lang.Math`. Diese Klasse stellt eine Reihe von mathematischen Funktionen zur Verfügung. Alle Methoden von `java.lang.Math` sind seiteneffektfrei und referentiell transparent; sie können deshalb innerhalb von Wertklassen aufgerufen werden, ohne die Werteigenschaften zu zerstören.

Keine Konstruktoren

Für eine Wertklasse können keine Konstruktoren definiert werden.

Keine Vererbung

Eine Wertklasse kann weder von einer anderen Klasse erben, noch kann sie als Basisklasse für eine andere Klasse verwendet werden.

Polymorphe Verwendung von Werten

VJ unterstützt die polymorphe Verwendung von Werten. In VJ können auch Wertklassen Interfaces implementieren. Da die Methoden von Wertklassen Einschränkungen unterliegen (s. o.), sind dafür nur solche Interfaces zulässig, in denen alle Methoden diesen Einschränkungen genügen. Daraus entsteht das – gedankliche – Konstrukt eines *Wertinterface*. Ein Wertinterface ist ein Interface, dessen Methoden die einschränkenden Bedingungen für Wertklassen erfüllen:

- ihre Parameter sind ausschließlich Werttypen,
- ihr Rückgabetyt ist ein Werttyp oder ein Wertinterface, und
- sie deklarieren keine `throws`-Klausel mit geprüften Exceptions.

Durch diese Regelung kann eine Wertklasse z. B. ein Interface wie `Comparable` implementieren und so auf `Comparable` basierende Sortier-Routinen verwenden.

Wertinterfaces sind in VJ nicht gesondert gekennzeichnet. Jedes Interface, das die genannten Bedingungen erfüllt, wird als Wertinterface angesehen und ist in der

⁴Aufrufe von Exemplarmethoden einer Objektklasse sind ebenfalls nicht zulässig, ebenso Zugriffe auf Exemplarvariablen eines Objektes. Sie müssen nicht explizit durch die Sprachregeln ausgeschlossen werden. Weil in Wertklassen die Parameter der Methoden keinen Objekttyp haben dürfen, und weil die Erzeugung von Objekten ausgeschlossen ist, sind sie nicht möglich.

`implements`-Klausel einer `valueclass` zulässig. Wertinterfaces können sowohl von Wertklassen als auch von Objektklassen implementiert werden. Sie bilden eine Teilmenge der „Objektinterfaces“, nicht einen Gegensatz zu ihnen.

Weil VJ es zulässt, dass Wertklassen (Wert-)Interfaces implementieren, ist die polymorphe Verwendung von Werten unter dem statischen Typ `Object` möglich. Deshalb sind alle Methoden, die in der Wurzelklasse `java.lang.Object` definiert sind, auch für Wertklassen verfügbar. VJ stellt sicher, dass sich diese Methoden in Wertklassen wert-konform verhalten.⁵

Integration der Klasse `String` als Werttyp in VJ

Aufgrund der weit verbreiteten Nutzung der Klasse `java.lang.String` und wegen ihrer engen Verzahnung mit den vorhandenen Java-Bibliotheken verzichtet VJ auf die Definition eines eigenen Typs für Zeichenketten. Die Klasse `String` darf in VJ, wie ein primitiver Datentyp, an allen Stellen verwendet werden, an denen ein Werttyp erwartet wird. `String` ist mit Java-Sprachmitteln definiert und damit eine Objektklasse („`class`“). Ihr Verhalten entspricht aber weitgehend dem einer Wertklasse⁶. Auch weist sie etliche Gemeinsamkeiten mit primitiven Typen auf, u. a. verfügt sie über Literale.

7.3 Der VJ-Compiler

Der VJ-Compiler wurde als Gemeinschaftsprojekt von J. Rathlev und der Autorin entworfen und realisiert. Als Bestandteil der betreuten Diplomarbeit [Rat06] wurde mit Entwurf und Implementierung des VJ-Compilers begonnen und eine erste lauffähige Version erstellt. Im Rahmen der vorliegenden Arbeit wurde der VJ-Compiler umstrukturiert und weiterentwickelt.

7.3.1 Ansatz und Werkzeuge

Der VJ-Compiler ist als Prä-Compiler konzipiert: Er übersetzt VJ-Code in Java-Code, ein Java-Compiler erzeugt daraus Bytecode.

Für die Entwicklung des VJ-Compilers wurde das Werkzeug ANTLR, ein Parser-Generator, eingesetzt (s. dazu [Par07] und [Rat06, Kap. 5]). Um eine VJ-Grammatik

⁵In VJ ist für alle Wertklassen auch die Methode `clone` verfügbar, da sie in `Object` definiert ist. Konzeptionell ist das Vorhandensein von `clone` in einer Wertklasse verwirrend, da Werte nicht erzeugt und damit nicht kopiert werden können. VJ schränkt die Benutzung von `clone` für Wertklassen soweit wie möglich ein: Ein Aufruf von `clone` außerhalb des eigenen Package führt zu einem Übersetzungsfehler. Innerhalb des eigenen Package löst er eine `CloneNotSupportedException` aus und damit einen Laufzeitfehler aus.

⁶Ausnahmen von einem strikt wert-artigen Verhalten der Klasse `String` bilden die Methoden `valueOf` und `copyValueOf`, die Arrays und damit Objekte als Parameter haben, sowie die Methoden `getBytes` und `split`, die bei jedem Aufruf ein neues Array erzeugen und ausgeben.

zu erhalten, wurde eine in ANTLR-Notation abgefasste Java-Grammatik um Produktionsregeln für VJ-spezifische Sprachkonstrukte erweitert (s. Rathlev [Rat06, p. 51 und p. 63–66]). ANTLR generiert aus dieser Grammatik einen Lexer und einen Parser. Sie verarbeiten einen in der jeweiligen Sprache – hier: in VJ – abgefassten Quelltext und erstellen daraus einen abstrakten Syntaxbaum (Abstract Syntax Tree = AST). Für die weitere Verarbeitung wurden mit ANTLR weitere Baumparser erstellt, die den VJ-AST in mehreren Schritten analysieren, umformen, anreichern und schließlich daraus Java-Code generieren.

7.3.2 Aufbau des VJ-Compilers

Die Übersetzung eines VJ-Quelltextes durchläuft die folgenden acht Schritte:

Schritt 1: JavaLexer - Lexikalische Analyse des Quelltextes

Zerlegt den Quelltext in Tokens.

Schritt 2: VJParser - Syntaktische Analyse

Ermittelt die syntaktische Struktur des Programms und stellt sie in Form eines abstrakten Syntaxbaums (AST) dar.

Schritt 3: ASTValidator - Aufbereitung des abstrakten Syntaxbaums

Ersetzt potentiell mehrdeutige Knotentypen, d. h. solche, die nur anhand der untergeordneten Knoten unterscheidbar sind, durch neue, eindeutige Knotentypen. Dieser Schritt dient dazu, die Verarbeitung der nachfolgenden Schritte zu vereinfachen.

Schritt 4: MMCreator - Aufbau eines Metamodells

Erstellt aus dem AST ein Metamodell der Klassen, Variablen und Methoden. Jeder deklarierte Bezeichner (Variable, Methode, ...) erhält einen Eintrag in einer Symboltabelle.

Schritt 5: TypeChecker - Typermittlung und Typ-Prüfung

Ordnet allen Einträgen in der Symboltabelle ihren zugehörigen Typ zu. Überprüft die auf Wertklassen bezogenen Regeln und erzeugt bei Bedarf entsprechende Fehlermeldungen.

Schritt 6: VJToJavaTransformer - Umwandlung in Java-Syntax

Wandelt die VJ-spezifischen Sprachkonstrukte in Java um. Das Ergebnis dieses Schrittes ist ein Java-AST.

Schritt 7: JavaEmitter - Generierung von Java-Quelltext

Gibt den Java-AST in Form von Java-Quelltext aus.

Schritt 8: Java-Compiler - Erzeugen von Bytecode

Anwenden eines Java-Compilers auf den in Schritt 7 erzeugten Java-Quelltext.

Die Schritte 1 bis 7 sind jeweils mit Hilfe einer ANTLR-Grammatik und, soweit erforderlich, zugehörigem Java-Code umgesetzt.

7.3.3 Prüfungen für Wertklassen

Der VJ-Compiler überprüft, ob der zu übersetzende VJ-Quellcode die Regeln für Wertklassen (s. Abschnitt 7.2.2) einhält. Die Entdeckung von Verstößen führt zum Abbruch des Übersetzungsvorgangs und zur Ausgabe entsprechender Fehlermeldungen. Die Überprüfung der Wertklassen-spezifischen Regeln sind über mehrere Schritte des VJ-Compilers verteilt:

- Ein Teil der auf Wertklassen bezogenen Regeln wird bereits in der VJ-Grammatik zum Ausdruck gebracht, z. B. die Einschränkung, dass eine Wertklasse keine Konstruktoren enthalten darf. Die Überprüfung dieser Regeln ist im – aus der VJ-Grammatik generierten – VJParser (Schritt 2 des VJ-Compilers) umgesetzt.
- Andere Regeln werden durch geeignete Übersetzungen in Java-Code und damit durch den VJToJavaTransformer (Schritt 6 des VJ-Compilers) sichergestellt (s. auch Abschnitt 7.3.4). Dadurch übernimmt der Java-Compiler (in Schritt 8) die eigentliche Überprüfung dieser Regeln. Z. B. wird die Unveränderlichkeit der konstituierenden Eigenschaften durch Hinzufügen des Modifiers `final` im generierten Java-Code erreicht.
- Typ-bezogene Regeln können nicht in der Grammatik ausgedrückt werden. Sie können auch nicht durch geeignete Umsetzung in Java-Code sichergestellt werden, denn die Typinformation wird bereits vor der Umwandlung in Java-Code benötigt. Die Entscheidung, ob die Verarbeitung fortgesetzt werden kann oder ob ein Fehlerfall vorliegt, hängt vielfach vom Typ eines Ausdrucks ab, beispielsweise bei der Überprüfung, ob der Parameter einer Wertoperation ein Werttyp ist (s. Abschnitt 7.2.2). Die Transformation des AST in Java-Code ist ebenfalls an vielen Stellen vom Typ der enthaltenen Ausdrücke abhängig. Z. B. entscheiden die Typen der an einem Vergleich mit dem Operator „`==`“ beteiligten Ausdrücke darüber, ob und wie dieser Vergleich in einen Aufruf der Methode `equals` übersetzt wird (s. Abschnitt 7.3.4). Typermittlung und typ-bezogene Regeln sind im `TypeChecker` (Schritt 5 des VJ-Compilers) manuell programmiert.

Die Typ-bezogene Verarbeitung nimmt einen erheblichen Anteil innerhalb der Verarbeitungslogik des VJ-Compilers ein. Ihre Aufgabe besteht in erster Linie darin, festzustellen, ob der Typ eines Ausdrucks ein Werttyp oder ein Objekttyp ist.

Aufgrund der Vielfalt und Komplexität der in Java, und damit auch in VJ, zulässigen Ausdrücke⁷ ist die dafür erforderliche Programmlogik umfangreich und komplex. In einigen Fällen stößt sie an Grenzen; es kann innerhalb eines AST Ausdrücke („Knoten“) geben, deren Typ im TypeChecker nicht ermittelt werden kann. Die Lücken in der Typermittlung sind dennoch unkritisch, weil sie nicht mit den Fällen kollidieren, in denen die Typinformation für eine Wertklassen-bezogene Verarbeitung benötigt wird:

- Innerhalb von Wertklassen – für konstituierende Eigenschaften, für Parameter und Rückgaben der Operationen – sind die zu überprüfenden Typen explizit deklariert. Hier muss nicht der Typ eines komplexen Ausdrucks ermittelt werden, so dass die problematischen Fälle nicht auftreten.
- In Klienten von Wertklassen wird die Unterscheidung von Werttypen und Objekttypen für die Umsetzung des Operators „==“ in die `equals`-Methode benötigt. Bei Operanden, die nicht eindeutig als Werttyp oder Objekttyp einzuordnen sind, wird hier eine Laufzeitprüfung in den generierten Code aufgenommen (s. Abschnitt 7.3.4), was ein korrektes Verhalten gewährleistet.

7.3.4 Umwandlung in Java-Code

Ein weiterer wesentlicher Bestandteil des VJ-Compilers ist die Übersetzung von Wertklassen-bezogenem Code in Java-Code, ausgeführt in Schritt 6 (VJToJavaTransformer). Transformationen sind a.) für Wertklassen und b.) für deren Klienten erforderlich.

a.) Der VJ-Compiler wandelt **Wertklassen** und ihre Bestandteile nach folgenden Regeln in Java-Code um:

Wertklassen

Das Schlüsselwort `valueclass` wird durch `class` ersetzt. Zusätzlich erhält die Klasse den Modifier `final`, was verhindert, dass eine andere Klasse von ihr erbt. Der VJ-Compiler ergänzt die so generierte Klasse um die Klausel „`extends vj.lang.Value`“. Die Klasse `vj.lang.Value` ist im VJ-Compiler definiert und enthält die Methoden von `Object` (`clone`, `finalize`, `toString`, ...), sie sind dort so implementiert, dass sie ein wert-artiges Verhalten sicherstellen.

Das Erben von `vj.lang.Value` bewirkt, dass die aus der `valueclass` generierte Klasse über alle Methoden von `Objekt` verfügt. `vj.lang.Value` als (technische) Oberklasse im generierten Code ermöglicht dem VJ-Compiler auch, Wert-

⁷Z. B. sind qualifizierte und nicht qualifizierte Zugriffe auf Exemplarvariablen und Methoden zu berücksichtigen, Erzeugung von Arrays und Zugriffe auf ihre Komponenten, generische Typen und `raw types`, überladene Methoden, etc.

klassen von Objektklassen zu unterscheiden (anhand der Prüfung „`instanceof vj.lang.Value`“).

Konstituierende Eigenschaften

Die konstituierenden Eigenschaften der Wertklasse werden zu Exemplarvariablen der korrespondierenden `class`. Der VJ-Compiler ergänzt für sie den Modifier `final` und stellt dadurch ihre Unveränderlichkeit sicher. Soweit eine konstituierende Eigenschaft nicht als `private` deklariert ist, wird sie im umgewandelten Code mit dem Modifier `public` versehen.

Gültigkeitsprüfung

Der VJ-Compiler generiert eine Klassenmethode `isValid` mit passenden, d. h. den konstituierenden Eigenschaften entsprechenden Parametern, falls eine solche nicht vorhanden ist. Sie hat den Rückgabotyp `boolean` und liefert immer `true`.

Wertwähler

Der VJ-Compiler generiert einen `public` Konstruktor, dessen Parameter in Name, Typ und Reihenfolge den konstituierenden Eigenschaften entsprechen. Als erste Anweisung ruft dieser Konstruktor die Methode `isValid` auf und löst ggf. eine `NoSuchValueException` aus. Anschließend weist er den Exemplarvariablen der Klasse – hervorgegangen aus den konstituierenden Eigenschaften – die jeweiligen Parameter zu.

`equals` und `hashCode`

Der VJ-Compiler generiert für die Wertklasse eine `equals`-Methode. Sie bildet die Wertgleichheit ab und ist `public`. Ihre Implementation prüft die übergebenen Parameter auf den passenden Typ, anschließend vergleicht sie die einander entsprechenden Exemplarvariablen, bei primitiven Datentypen oder bei einem `enum`-Typ mit dem Operator „`==`“, bei Wertklassen oder bei der Klasse `String` mit der Methode `equals`.

Die Methode `hashCode` wird ebenfalls auf Basis der Exemplarvariablen, passend zu `equals`, generiert. Sie implementiert das von Bloch [Blo08, p. 47] vorgeschlagene Verfahren zur Ermittlung eines Hashcode.

Anhang B.2 veranschaulicht die Transformationen anhand eines Beispiels. Er zeigt eine – für den Zweck der Illustration vereinfachte – VJ-Wertklasse `Date` und den Java-Code, den der VJ-Compiler gemäß den beschriebenen Transformationsregeln daraus generiert.

b.) Der VJ-Compiler wandelt **Klienten** von Wertklassen nach folgenden Regeln in Java-Code um:

Aufruf des Wertwählers

Der VJ-Compiler übersetzt einen Aufruf des Wertwählers in einen Aufruf des – aus den konstituierenden Eigenschaften generierten (s. o.) – Konstruktors.

Gleichheitsprüfung

Der VJ-Compiler wandelt einen Aufruf des Operators „==“ selektiv in einen Aufruf der Methode `equals` um, abhängig von den in Schritt 5 (TypeChecker) ermittelten Typen der Operanden:

- Sind die Operanden Wertklassen oder handelt es sich um die Klasse `String`, dann ersetzt der VJ-Compiler den Operator „==“; er sorgt dafür, dass statt dessen die Methode `equals` aufgerufen wird. Der Aufruf von `equals` erfolgt nicht direkt, sondern indirekt über die Klassenmethode `equal` der Klasse `vj.lang.Value`. Sie verhindert eine `NullPointerException`, falls einer der Operanden `null` ist.
- Sind die Operanden primitive Datentypen, Aufzählungen oder Objektklassen, so bleibt der Operator „==“ erhalten.
- Ist der statische Typ nicht ermittelbar (s. Abschnitt 7.3.3), oder verhindert der statische Typ eine Feststellung des Laufzeittyps (im Fall der Klasse `java.lang.Object` oder bei einem Wertinterface), so werden beide Vergleiche vorgesehen und zusätzlich eine dynamische Typprüfung (`instanceof` `vj.lang.Value`) generiert. Sie steuert zur Laufzeit, ob „==“ oder `equals` zum Einsatz kommt.

Der Operator „!=“ wird analog zum Operator „==“ umgewandelt.

Anhang B.3 illustriert anhand von Code-Beispielen die Übersetzung eines Wertwähler-Aufrufs und von mehreren Aufrufen des Operators „==“.

7.3.5 Umsetzung und Test

Der VJ-Compiler ist fertiggestellt und einsatzfähig. Sein Funktionsumfang umfasst alle in den Abschnitten 7.3.3 und 7.3.4 beschriebenen Prüfungen und Transformationen.

Der VJ-Compiler wurde durch verschiedene Anwendungsfälle getestet. Sie belegen, dass der VJ-Compiler gemäß den Vorgaben funktioniert, und sie illustrieren, auf welche Weise VJ die Programmierung von Werttypen kürzer, klarer und sicherer macht.

Anhang B.4 enthält vier ausgewählte Testfälle für den VJ-Compiler: Geldbeträge (`Money`), komplexe Zahlen (`Complex`), positive ganze Zahlen (`PositiveInteger`) und Intervalle über Gleitkommazahlen (`Range`). Jeder Testfall umfasst Code-Listings der jeweiligen VJ-Wertklasse, von zugehörigem Anwendungscode sowie jeweils den aus dem VJ-Code generierten Java-Code.

7.4 Bewertung und Folgerungen

Die Entwicklung der Forschungssprache VJ und eines zugehörigen Compilers haben den Nachweis erbracht, dass Werttypen durch eine Programmiersprache klar und sicher unterstützt werden können und dass es möglich ist, die für sie erforderlichen Konstrukte in eine objektorientierte Sprache zu integrieren.

7.4.1 Zielerreichung

Der folgende Abgleich mit den in Abschnitt 7.1 aufgestellten Zielen zeigt, dass diese erreicht wurden:

Nachweis der Machbarkeit

Mit VJ wurde eine Programmiersprache konzipiert und realisiert, die die einfache und geradlinige Programmierung von Werttypen in einem objektorientierten Umfeld ermöglicht. VJ baut auf der Sprache Java auf und unterstützt damit objektorientierte Programmierung; die VJ-spezifischen Erweiterungen unterstützen Werttypen entsprechend dem in Kapitel 3 beschriebenen Konzept. Die Entwicklung eines VJ-Compilers und die Programmierung von repräsentativen Fallbeispielen untermauern die praktische Umsetzbarkeit des Ansatzes. Sie erhärten die Tragfähigkeit des Werttyp-Konzeptes, sie belegen die Funktionsfähigkeit des VJ-Compilers und sie illustrieren, wie eine dedizierte Unterstützung für Werttypen die Programmierung erleichtert und sicherer macht.

Einfache Programmierung

Der gewählte Sprachentwurf vereinfacht die Programmierung von Werttypen erheblich. Der Programmierer einer Wertklasse muss keinen Aufwand für die Herstellung der charakteristischen Werteigenschaften investieren; deren Überwachung liegt in der Verantwortung der Programmiersprache. Ein großer Teil der Funktionalität von Wertklassen – wie der Wertwähler, die Wertgleichheit oder Zugriffe auf die konstituierenden Eigenschaften – wird von der Sprache standardmäßig zur Verfügung gestellt. Das ermöglicht eine effiziente Programmierung und führt zu knappen, prägnanten Quelltexten. Zusammengesetzte Werttypen wie `Datum`, `Punkt`,

Geldbetrag oder `KomplexeZahl` können so in VJ einfach und mit geringem Aufwand programmiert werden.

Kompatibilität zu Java

VJ ist Quellcode-kompatibel zu Java. Jedes gültige Java-Programm ist per Konstruktion auch ein gültiges VJ-Programm und kann mit dem VJ-Compiler übersetzt werden. VJ ist Bytecode-kompatibel zu Java. Bestehender Java-Code, insbesondere Java-Klassenbibliotheken, können ohne Neuübersetzung mit dem VJ-Compiler in Zusammenhang mit VJ-Code benutzt werden.⁸

7.4.2 Einfluss von VJ auf den Sprachentwurf dieser Arbeit

VJ setzt die in Kap. 6 aufgestellte Forderung nach einer programmiersprachlichen Unterstützung von Werttypen in eine konkrete Sprache um. Die in VJ getroffenen und durch die praktische Umsetzung evaluierten Entwurfsentscheidungen sind auch im Sprachentwurf dieser Arbeit zu finden; in einigen Punkten geht dieser über VJ hinaus.

Übereinstimmungen zwischen dem Sprachentwurf dieser Arbeit und VJ

Die in Abschnitt 6.3 beschriebene Sprachunterstützung für Werttypen stimmt in allen wesentlichen Merkmalen mit der Werttyp-Unterstützung von VJ überein:

- **Eigenständiger Typkonstruktor für Werttypen:**
Beide Entwürfe postulieren einen dedizierten Typkonstruktor für Werttypen, zusätzlich zum – als vorhandenen vorausgesetzten – Typkonstruktor für Objekttypen.
- **Selektoren/ Wertwähler:**
Beide Entwürfe stellen sicher, dass Wertklassen keine Konstruktoren haben. Kap. 6 führt das Konzept des primären Selektors ein, VJ das entsprechende Gegenstück, den Wertwähler.
- **Sicherstellung der Werteigenschaften:**
In beiden Entwürfen stellen Wertklassen-spezifische Restriktionen die Einhaltung der charakteristischen Eigenschaften von Werten sicher. Vergl. hierzu die Abschnitte 6.3 und 7.2.2.

⁸VJ ist nicht vollständig *verhaltenskompatibel* zu Java. Ein Java-Programm kann bei Übersetzung durch den VJ-Compiler ein anderes Ergebnis liefern als mit einer Übersetzung durch einen Java-Compiler. Die Unterschiede im Verhalten zwischen VJ und Java sind marginal, sie betreffen eine prinzipiell mögliche, praktisch aber wenig relevante Benutzung der Klasse `String`: Der Operator „`==`“ prüft in VJ bei String-Ausdrücken gemäß Vorgabe die Wertgleichheit, in Java dagegen die Identität der beiden String-Objekte. Verhaltenskompatibilität zu Java war keine Anforderung an VJ (s. Abschnitt 7.1). Sie würde aufgrund der für VJ getroffenen Entwurfsentscheidungen in Widerspruch zu der Forderung stehen, bestehende Java-Bibliotheken ohne Neu-Übersetzung in Kombination mit VJ-Code nutzen zu können.

- Unterstützung von Wertgleichheit:
Wertgleichheit spielt in beiden Entwürfen eine zentrale Rolle und wird für Wertklassen sprachseitig unterstützt. Die Sprache sorgt dafür, dass nur die Datenfelder respektive die konstituierenden Eigenschaften in die Implementation der Wertgleichheit einfließen, und dass ein – fachlich irrelevanter – Vergleich von technischen Speicheradressen nicht möglich ist.
- Keine Implementierungsvererbung:
Beide Entwürfe schließen Implementierungsvererbung bei Wertklassen aus.

Unterschiede zwischen dem Sprachentwurf dieser Arbeit und VJ

In einigen Punkten weicht der in Kap. 6 vorgestellte Sprachentwurf von VJ ab oder geht darüber hinaus. Die Unterschiede sind möglich und motiviert durch die verschiedenen Voraussetzungen der beiden Sprachentwürfe. VJ erweitert eine konkrete Sprache und ist damit in vielerlei Hinsicht den Vorgaben und impliziten Leitlinien der Ausgangssprache Java unterworfen. Der in dieser Arbeit präsentierte Sprachentwurf fokussiert, ausgehend von minimalen Annahmen über die objektorientierte Ausgangsbasis, auf ein einzelnes Sprachkonstrukt. Er zielt darauf ab, das in Kap. 3 definierte Konzept von Werttypen „in Reinkultur“ umzusetzen.

Die Abweichungen von und Erweiterungen gegenüber VJ betreffen folgende Details und haben folgende Gründe:

- Spektrum der unterstützten Werttypen:
Die in Kap. 6 beschriebene Sprachunterstützung deckt ein breiteres Spektrum von Werttypen als VJ ab. An einem Ende des Spektrums integriert sie Aufzählungen in den Werttypkonstruktor. VJ dagegen greift für Aufzählungen auf den in Java bereits vorhandenen Typkonstruktor `enum` zurück. Am anderen Ende des Spektrums ermöglicht der Sprachentwurf dieser Arbeit, auch Werttypen mit unterschiedlich großen Repräsentationen abzubilden, z. B. Zeichenketten, Mengen oder Listen. Er erreicht das, indem er rekursive Definitionen von Wertklassen ausdrücklich zulässt. Die Wertklassen von VJ sind dagegen auf die Abbildung von Werttypen ausgerichtet, deren konstituierende Eigenschaften einen anderen als den eigenen Typ haben (s. dazu Rathlev [Rat06, p. 59]); sie schließen eine rekursive Definition und unterschiedlich große Repräsentationen innerhalb einer Wertklasse aus.
- Strikte Trennung von Werten und Objekten:
Der Sprachentwurf der vorliegenden Arbeit ist auf eine strikte, im Quellcode erkennbare und von der Sprache durchgesetzte Trennung von Werten und Objekten ausgelegt. Für jeden Ausdruck kann statisch ermittelt werden, ob sein Typ ein Werttyp oder ein Objekttyp ist. Aus diesem Grund postuliert Kap. 6 keine gemeinsame Wurzelklasse für Wertklassen und Objektklassen und verzichtet auf das Konzept von Wertinterfaces. Auch die dedizierte Unterstützung

für Wert-Aufzählungen und ihre Integration in den Werttypkonstruktor trägt dazu bei, die Zweiteilung des Typsystems klar zum Ausdruck zu bringen.

- Nur ein Vergleich für Wertklassen:

Der in Kap. 6 beschriebene Sprachentwurf sieht für Wertklassen einen einzigen sprachunterstützten Vergleich vor, die Wertgleichheit. In VJ verfügen Wertklassen, bedingt durch die Einbettung in Java, über zwei Möglichkeiten zum Prüfen der Wertgleichheit: den Operator „==“ und die Methode `equals`.

- Implementierung der Wertgleichheit:

Der Sprachentwurf der vorliegenden Arbeit sieht für den Programmierer einer zusammengesetzten Wertklasse die Möglichkeit vor, das Verhalten der Wertgleichheit zu beeinflussen und eine eigene Implementation für sie anzugeben (s. Abschnitt 6.3.5).

- Kapselung:

Abschnitt 6.3.2 beschreibt einen Sprachentwurf, der die Datenfelder einer Wertklasse und den daraus hervorgehenden primären Selektor als Bestandteile der internen Implementation betrachtet und verbirgt. Sie sind, anders als in VJ, nicht für Klienten zugänglich. Dieser Ansatz überträgt das Prinzip der Kapselung – ein Wesensmerkmal objektorientierter Programmierung – auf Wertklassen; er erlaubt, die Repräsentation einer Wertklasse zu ändern, ohne Klienten anpassen zu müssen. Er ermöglicht, anders als der Ansatz der konstituierenden Eigenschaften, die Programmierung von Wertklassen mit mehrdeutiger Repräsentation, z. B. rationale Zahlen.⁹ Des weiteren können dadurch alle Selektoren gleichförmig aufgerufen werden; aus Klientensicht besteht keine Notwendigkeit, zwischen einem primären Selektor und alternativen Selektoren zu unterscheiden.

- Homogene Schnittstelle:

Basierend auf der Kapselung der internen Repräsentation vereinheitlicht der Sprachentwurf dieser Arbeit die Schnittstelle einer Wertklasse noch weiter. Klienten rufen *alle* Wertoperationen gleichförmig auf. Für sie gibt es keine syntaktische Unterscheidung zwischen Methoden und konstituierenden Eigenschaften, zwischen einem primären Selektor und alternativen Selektoren oder zwischen Selektoren und Nicht-Selektoren.

⁹Definiert man in VJ eine Wertklasse für rationale Zahlen mit Zähler und Nenner als konstituierenden Eigenschaften, dann führt die Funktionsweise von Wertwähler und Wertgleichheit zu inkorrektem Verhalten. Weil der Wertwähler in VJ für Klienten zugänglich ist, kann er für jede Kombination von Zähler und Nenner aufgerufen werden, z. B. für (2,3) und für (4,6). Die auf einem Vergleich aller konstituierenden Eigenschaften beruhende Wertgleichheit weist dann `const Rational(2,3)` und `const Rational(4,6)` als verschiedene Werte aus, obwohl es sich fachlich um die gleiche rationale Zahl handelt. Vergl. dazu auch die Abschnitte 6.3.2 bis 6.3.5.

7.5 Zusammenfassung

Kapitel 7 stellte die Forschungssprache VJ („Value-Java“) vor. VJ dient als Machbarkeitsstudie: Die Sprache weist nach, dass eine auf objektorientierte Programmierung ausgerichtete Sprache um Werttypen ergänzt werden kann. Sie veranschaulicht Handhabung und Vorteile einer Sprachunterstützung für Werttypen.

VJ erweitert die objektorientierte Sprache Java um einen Typkonstruktor für Werttypen und legt dafür den in dieser Arbeit beschriebenen Werttyp-Begriff zugrunde. Mit einem im Rahmen dieser Arbeit und einer daraus hervorgegangenen Diplomarbeit entwickelten VJ-Compiler wurde die Tragfähigkeit und praktische Umsetzbarkeit des Werttyp-Konzeptes nachgewiesen.

Ein vorrangiges Ziel von VJ ist einfacher, prägnanter Code. Der Sprachentwurf von VJ ist darauf ausgelegt, eine kompakte Programmierung von Werttypen zu unterstützen. Der gewählte Ansatz – die Erweiterung der Sprache Java – bedingt als ein weiteres Ziel Aufwärtskompatibilität: Innerhalb von VJ-Programmen kann Java-Code eingebunden werden.

Der Sprachentwurf von VJ sieht, aufbauend auf Java-Sprachkonstrukten, für Werttypen folgende sprachliche Erweiterungen vor: Es gibt einen eigenen Typkonstruktor für Werttypen („`valueclass`“). Eine Wertklasse enthält „konstituierende Eigenschaften“, sie legen zusammengenommen jeden Wert des Typs fest. Basierend auf ihnen generiert die Sprache einen „Wertwähler“ – er entspricht dem in Kap. 6 eingeführten primären Selektor und ist für Klienten aufrufbar – und zur Prüfung der Wertgleichheit die Methode `equals`, aufrufbar über den Operator „`==`“. Zur Sicherstellung der Werteigenschaften legt VJ spezifische Einschränkungen für Wertklassen fest.

Kapitel 7 stellte die Gliederung und die Funktionsweise des VJ-Compilers vor. Er ist als Prä-Compiler konzipiert und übersetzt VJ-Code in Java-Code. Der VJ-Compiler wurde durch verschiedenartige, repräsentative Fallbeispiele getestet.

VJ hat gezeigt, dass eine Sprachunterstützung für Werttypen in einer objektorientierten Sprache möglich ist und dass sie die Definition und Benutzung von Werttypen vereinfacht und sicherer macht. VJ bildet ein wesentliches Fundament für die in dieser Arbeit vorgeschlagene Sprachunterstützung; ihre Entwürfe stimmen in den zentralen Punkten überein. In einigen Details geht der in dieser Arbeit vorgestellte Sprachentwurf über den von VJ hinaus. Kapitel 7 hat gezeigt, welche Unterschiede in den Voraussetzungen zu diesen Modifikationen geführt haben.

Teil III

Auswirkungen der Wert-/ Objekttrennung

Kapitel 8

Gleichheit/ Identität bei Wert- und Objekttypen

Bereits die Kapitel 3, 6 und 7 haben gezeigt, dass Gleichheit für Werttypen eine zentrale Rolle spielt. In objektorientierten Programmiersprachen wirft der Umgang mit Gleichheit eine Reihe von Problemen auf, die bisher unter verschiedenen Gesichtspunkten diskutiert, aber nur unbefriedigend gelöst wurden.

Dieses Kapitel untersucht Gleichheit und Identität aus dem Blickwinkel der Trennung von Werten und Objekten. Er ermöglicht eine andere begriffliche Einordnung dieser Konzepte und darauf aufbauend die Verschmelzung von (Wert-)Gleichheit und (Objekt-)Identität zu einem einheitlichen Konzept. Das trägt dazu bei, etliche der Schwierigkeiten im Umgang mit Gleichheit zu vermeiden.

Abkürzend wird ab hier das auf der Trennung von Werten und Objekten basierende Sprachmodell, das in Kapitel 3 eingeführt und in den Kapiteln 6 und 7 durch geeignete Sprachunterstützung konkretisiert wurde, als *V-O-Modell* bezeichnet (V=Value, O=Object). Das erleichtert sprachlich die Gegenüberstellung zum herkömmlichen Sprachmodell der Objektorientierung, welches implizit davon ausgeht, dass alle Abstraktionen mit Hilfe von Objekten abgebildet werden können.

8.1 Bestandsaufnahme: Schwierigkeiten im Umgang mit Gleichheit

In objektorientierten Sprachen ist der Umgang mit Gleichheit von zahlreichen Schwierigkeiten geprägt. Z. B. muss Gleichheit einen *Vertrag* einhalten – einen Satz von Regeln, der die intuitiven Erwartungen an Gleichheit formalisiert. Bloch hat den Gleichheitsvertrag und Hinweise für seine Einhaltung in Java, d. h. bezogen auf die Methode `equals`, ausführlich beschrieben [Blo08, Item 8]. Auch in der Doku-

mentation von Scala [OSV11, p. 658] findet sich dieses Regelwerk für die Methode `equals`. In vielen weiteren Sprachen gibt es analoge Regeln.

Der Gleichheitsvertrag besagt, dass Gleichheit folgende Bedingungen erfüllen muss:

Gleichheitsvertrag:

1. Gleichheit muss für alle „non-null references“ eine Äquivalenzrelation bilden, d. h. sie muss reflexiv, symmetrisch und transitiv sein.¹
2. Gleichheit muss „konsistent“ sein, d. h. sie muss immer dasselbe Ergebnis liefern, solange keine Information verändert wurde, welche dem Vergleich zugrunde liegt.
3. Jeder Vergleich eines Elementes mit `null` muss `false` ergeben.

Im Folgenden werden einige Schwierigkeiten aufgeführt, die den Umgang mit Gleichheit erschweren.

8.1.1 Schwierigkeit 1: Gleichheit ist keine Äquivalenzrelation

Die Forderung, dass Gleichheit eine Äquivalenzrelation bilden muss (soweit nicht „null-Referenzen“ involviert sind), ist überraschend fehleranfällig. Ein Indikator für ihre Schwierigkeit ist, dass sich selbst in professionellem Code und in Lehrbüchern Implementationen von Gleichheit finden, die nicht symmetrisch oder nicht transitiv sind. Langer und Kreft [LK02a] sowie Vaziri et al. [VTFD07] führen einige Beispiele dafür auf.

Anhang C.2 zeigt anhand der Beispielklassen `Point` und `Colorpoint`, warum eine naive Implementierung der Gleichheit häufig fehlerhaft ist und dass die Programmierung als Äquivalenzrelation besonders in Zusammenhang mit Vererbungshierarchien Probleme aufwirft.

Es wurden verschiedene Verfahren zur vertragskonformen Implementierung der Gleichheit vorgeschlagen, z. B. von Bloch und Venners [BV02], Cohen [Coh02], Horstmann [Hor03], Langer und Kreft ([LK02a], [LK02b]), Odersky et al. [OSV09], Rupakheti und Hou ([RH08]), Stevenson und Phillips ([SP03]), Lahres und Raýman ([LR09, Kap. 4.4]). Sie beruhen auf Programmierkonventionen oder Programmiermustern, und häufig führen sie zu komplexem, schwer verständlichen Code. Oft beinhalten sie Typ-Prüfungen, die mehrfach durchlaufen werden müssen und sind dadurch ineffizient. Optimierungen zur Vermeidung der doppelten Prüfungen verkomplizieren den Code weiter.

¹Im Original-Wortlaut (wiedergegeben in Anhang C.1) besteht der Gleichheitsvertrag aus fünf Bedingungen, von denen die ersten drei die Regeln der Äquivalenzrelation enthalten. Zur besseren Vergleichbarkeit (s. Abschnitt 8.3.1) sind die drei ersten Bedingungen hier zu einer zusammengefasst.

8.1.2 Schwierigkeit 2: Gleichheit abhängig von änderbarem Zustand

Selbst eine Gleichheit, die den Vertrag einhält, kann Probleme aufwerfen, z. B., wenn sie von änderbarem Zustand abhängt. Das ist laut Gleichheitsvertrag zulässig. Er fordert gemäß Regel 2 lediglich, dass Gleichheit konsistent sein muss, d. h. sie muss nur so lange zum selben Ergebnis kommen, wie die verglichenen Elemente nicht verändert wurden.

Anhang C.3 führt ein Code-Beispiel von Odersky et al. auf. Es zeigt, auf welche Weise eine von änderbarem Zustand abhängige Gleichheit inkonsistentes Verhalten verursacht. Odersky et al. betrachten die Abhängigkeit von änderbarem Zustand als eine eigene Kategorie von Fallstricken in Zusammenhang mit der Implementation der Methode `equals`:

„The lesson to be drawn from this example is that when `equals` and `hashCode` depend on mutable state, it may cause problems for users. ... If you need a comparison that takes the current state of an object into account, you should usually name it something else, not `equals`.“[OSV09]

Vaziri et al. beschreiben dieses Phänomen ebenfalls [VTFD07]. Als Konsequenz fordern sie für Gleichheit einen revidierten Vertrag („revised equality contract“). Dieser legt fest, dass Gleichheit bei gleichen Eingaben *immer* das gleiche Ergebnis liefert und damit unabhängig von eventuellen Änderungen an den verglichenen Objekten sein muss.

Nelson et al. haben das Laufzeit-Verhalten von Objekten in bezug auf ihre Änderbarkeit und ihre Gleichheit mit Hilfe eines Profilers in 30 Open-Source Java-Projekten untersucht [NPN10]. Dabei stellten sie fest, dass Objekte, die in gleichheits-abhängigen Sammlungen verwaltet werden, *häufiger* eine von änderbarem Zustand abhängige Gleichheit aufweisen als in der „Gesamtpopulation“ (13% gegenüber 4%). Die Daten sind eingegrenzt auf anwendungs-spezifische Objekte, Objekte aus Java-Standardbibliotheken sind nicht mitgezählt. Allerdings wurden im Rahmen des Profilings keine entsprechenden Änderungen *nach* dem Einfügen der Objekte in die jeweilige Sammlung gefunden.

8.1.3 Schwierigkeit 3: Hashcode nicht passend zur Gleichheit

Elemente, die `equals` als „gleich“ einstuft, müssen auch den gleichen `hashCode` haben. D. h. wenn in Java

```
a.equals(b)
```

gilt, dann muss auch gelten

```
a.hashCode() == b.hashCode()
```

(s. dazu die API-Dokumentation [Ora14], Bloch [Blo08, Item 9] oder Odersky [OSV11, Kap. 30.2]).

Ein Programmierer, der die Methode `equals` überschreibt, muss deshalb auch `hashCode` passend dazu überschreiben – eine reine Konvention, die aus Unkenntnis oder durch Flüchtigkeitsfehler verletzt werden kann. Da in den Java-Bibliotheken viele Sammlungen für ihre interne Verwaltung auf den `hashCode` ihrer Elemente zurückgreifen, hat ein Verstoß gegen die `hashCode`-Regel inkonsistentes Programmverhalten zur Folge (s. z. B. Bloch [Blo08, Item 9]).

8.1.4 Schwierigkeit 4: Unterscheidbarkeit gleicher Elemente

Auch bei anderen Operationen als `hashCode` können verschiedene Ergebnisse für „gleiche“ Elemente überraschendes Programmverhalten hervorrufen. Anhang C.4 enthält ein Code-Beispiel, bei dem Fahrzeuge in eine Sammlung eingefügt werden und anschließend eine Liste der Fahrzeug-Besitzer erstellt wird. Das Beispiel illustriert, wie bei entsprechender Wahl der Gleichheit ein aus Klientensicht nicht deterministisches Verhalten resultiert.

8.1.5 Schwierigkeit 5: Benutzung eines „falschen“ Vergleichs

Viele objektorientierte, auch etliche hybride Sprachen unterstützen *zwei* Vergleichs-Operatoren oder -Methoden (s. Tabelle 8.1). Sie werden meist als Identität und Gleichheit bezeichnet.²

Sprache	Vergleiche	
Java	<code>==</code>	<code>equals</code>
Scala	<code>==</code>	<code>eq</code>
C#	<code>==</code>	<code>Equals</code>
Smalltalk	<code>==</code>	<code>=</code>
Eiffel	<code>=</code>	<code>is_equal</code>

Tabelle 8.1: Vergleichsoperatoren und -methoden in Programmiersprachen

Das Vorhandensein mehrerer Vergleiche birgt die Gefahr, den „falschen“ Vergleich zu verwenden und macht die Benutzung eines Typs dadurch fehleranfällig. Beispielsweise ist es in Java ein verbreiteter Fehler, `==` für Vergleiche von String-Ausdrücken zu verwenden, hier ist in der Regel `equals` der fachlich passende Vergleich.

²In vielen der hier aufgeführten Sprachen ist die Situation komplexer als in der Tabelle dargestellt. Z. B. verfügt Scala über die (überschreibbare) Methode `equals`, die ihrerseits von `==` aufgerufen wird. In C# gibt es zwei weitere Vergleiche, die beiden statischen Methoden `Equals` und `ReferenceEquals`. Eiffel kennt neben `==` und `is_equal` auch `~` und `===`, und das Verhalten der Vergleiche hängt u. a. davon ab, ob sich der Vergleich auf `reference types` oder `expanded types` bezieht, etc. Relevant ist hier, dass die Sprache zwei (oder mehr) Vergleiche unterstützt.

8.1.6 Schwierigkeit 6: Vergleich typ-inkompatibler Ausdrücke

Ein Vergleich von zwei Ausdrücken, deren Typen keine gemeinsamen Elemente haben, liefert immer `false` und ist deshalb nicht sinnvoll. Es muss sich dabei um einen Programmfehler handeln. Vaziri et al. führen für diesen „inadvertent test of incomparable types“ ein Praxisbeispiel an [VTFD07]. Die inkompatiblen Typen kommen dabei durch einen vergessenen Methodenaufruf zustande; der Fehler verursacht ein auf den ersten Blick unerklärliches Programmverhalten.

Die sichere Handhabung eines solchen Vergleichs wäre die Ausgabe eines Übersetzungsfehlers, wie sie z. B. Java bei einem Vergleich mit dem Operator „`==`“ erzeugt. Ist Gleichheit aber durch eine Methode wie `equals` implementiert (definiert in der Wurzelklasse `Object`, mit Parameter-Typ `Object`), dann kann der Compiler die inkompatiblen Typen nicht erkennen.

8.1.7 Schwierigkeit 7: Asymmetrie bei null-Referenzen

Horstmann hat darauf hingewiesen, dass sich eine Methode wie `equals` in Zusammenhang mit „Fehlwerten“ (in Java: `null`) inhärent asymmetrisch verhält [Hor03]. Ist die Variable `a` an ein Element gebunden, und die Variable `b` an die „null-Referenz“, dann liefert

```
a.equals(b)
```

gemäß Regel 3 des Gleichheitsvertrags das Ergebnis `false`, während

```
b.equals(a)
```

eine `NullPointerException` auslöst.

Aufgrund dieser Asymmetrie kann das Ergebnis von Vergleichen von der Reihenfolge der verglichenen Ausdrücke abhängen – die aber ggf. nur durch die jeweilige Implementation bedingt ist, z. B. wenn sukzessive die Bestandteile einer Menge verarbeitet werden. Die Folge ist indeterministisches Programmverhalten.

Klienten, die `NullPointerException`s vermeiden wollen, müssen den Aufruf von `equals` in Code einbetten, der berücksichtigt, dass eine oder beide Seiten `null` sein können. Beispielsweise prüft eine vom Eclipse-JDT generierte `equals`-Methode (s. [Ecl14], „Generate hashCode() and equals()“) eine Membervariable `a` der beteiligten Objekte nach folgendem Schema auf Gleichheit:

```
if (this.a == null) {  
    if (other.a != null)  
        return false;  
} else if (!this.a.equals(other.a))  
    return false;  
return true;
```

Die zusätzlichen Prüfungen blähen den Code auf. Sie machen die Benutzung der Gleichheit umständlich, fehleranfällig und schwerer lesbar.

8.1.8 Zusammenfassung der Schwierigkeiten

Die Nicht-Einhaltung der genannten Regeln führt weder zu Übersetzungs- noch zu Laufzeitfehlern, sondern dazu, dass das Programm sich anders verhält als beabsichtigt. Häufig treten die Auswirkungen in einem anderen Zusammenhang zutage. Das macht es schwierig, die Fehlerursache zu ermitteln.

Die beschriebenen Schwierigkeiten betreffen teils die Programmierer der Klasse, teils ihre Klienten. Tabelle 8.2 fasst sie zusammen.

	Schwierigkeit	Auswirkung
Verantwortlich: Programmierer der Klasse	Der Vertrag verlangt, dass Gleichheit eine Äquivalenzrelation bildet	Komplexer Code, in Zusammenhang mit Vererbung schwer einhaltbar
	Von änderbarem Zustand abhängige Gleichheit	Inkonsistentes Programmverhalten
	Gleiche Elemente mit unterschiedlichem Hashcode	Inkonsistenzen bei Verwaltung der Elemente in Sammlungen
	Unterscheidbarkeit gleicher Elemente	Indeterministisches Programmverhalten
Verantwortlich: Klienten	Vorhandensein mehrerer Vergleiche	Fehler bei Benutzung eines falschen Vergleichs
	Vergleich inkompatibler Typen	Kein Hinweis auf fehlerhaften Code durch den Compiler
	Handhabung von null	Asymmetrie, indeterministisches Verhalten, Vergleiche erfordern Prüfung auf null

Tabelle 8.2: Schwierigkeiten im Umgang mit Gleichheit

8.2 Ursachenanalyse: Fehlende Trennung von Werten und Objekten

Die Vielzahl der Schwierigkeiten in Zusammenhang mit Gleichheit sind ein Hinweis auf tiefer liegende Ursachen.

Ein wesentlicher Grund für die in Abschnitt 8.1 geschilderten Probleme ist die fehlende Trennung von Werten und Objekten. Sie führt zu einer begrifflichen Vermengung verschiedener Konzepte, sowohl in Quelltexten, in der Umgangssprache als auch in den Programmiersprachen selbst.

Daneben trägt auch der Gleichheitsvertrag in seiner oben beschriebenen Form zu den geschilderten Problemen bei. Er beschreibt die Eigenschaften von Gleichheit nicht eindeutig und lässt Raum für inkonsistentes Verhalten.

8.2.1 Simulation von Werttypen mit Objektklassen

Die fehlende Unterstützung für Werttypen zwingt den Programmierer, Werttypen mit Objektklassen umzusetzen.

Wenn die Programmiersprache kein Überladen von Operatoren unterstützt und keine Symbole als Methodennamen zulässt, erhält Gleichheit einen anderen Namen oder ein anderes Symbol als die Identität (z. B. in Java „equals“ statt „==“). Das führt zu dem Problem, dass für einen Typ zwei Vergleiche vorhanden sind und zu der Gefahr, sie zu verwechseln.

Weil Gleichheit häufig als „normale“ Exemplarmethode programmiert wird, löst sie im Fall von `null` als Receiver zwangsläufig eine `NullPointerException` (oder das jeweilige sprachliche Äquivalent dazu) aus. Das führt zu der beschriebenen Asymmetrie und zu der Notwendigkeit, dass Klienten sich selbst um die Handhabung von `null` kümmern und entsprechende Prüfungen einbauen müssen.

In der Regel ist der Parameter der Gleichheit ein in der Typhierarchie weit oben angesiedelter Typ (in Java: die Wurzelklasse `Object`). Das ist die Ursache dafür, dass auch Ausdrücke inkompatibler Typen miteinander verglichen werden können.

Die gebräuchliche Abbildung eines Werttyps mit dem Value Object Pattern (s. Abschnitt 4.2.1) führt zu einer semantischen Lücke. Mehrere Objekte können denselben Wert abbilden; es gibt keine 1:1-Entsprechung zwischen einem Wert des Typs und einem Objekt der Klasse. Ein Wert entspricht vielmehr einer *Äquivalenzklasse* der Objekte modulo Gleichheit. Die Anforderung, dass Gleichheit eine Äquivalenzrelation bilden muss, stellt sicher, dass diese Äquivalenzklassen wohldefiniert sind. Der Programmierer hantiert so mit mehreren Objekten, die aber lediglich *Repräsentationen* für einen Wert sind. Äquivalenzklassen sind rein gedankliche Gebilde und im Quelltext nicht erkennbar.

Das wird z. B. deutlich bei Fowler:

„One clear division between entities and values is that values override the equality method ... while entities usually don't. This is because you usually don't want to have more than one object representing the same conceptual entity within your processing context, however you don't care about multiple "5.0" objects.“[Fow]

Grogono und Chalin sowie Grogono und Santas dagegen stellen die notwendige Unterscheidung zwischen Werten und ihren Repräsentationen deutlich heraus:

„We cannot talk about, or compute with, values, but only with their representations. ...Values partition the set of representations into equivalence classes.“[GC94]

„We could ignore the difference between values and representations if there was a one-to-one correspondence between representation and values

but, in general, there is not. We therefore need ways of deciding when two representations denote the same value, and this is where equality enters the picture.“[GS93]

8.2.2 Synonyme und Homonyme

In Zusammenhang mit Objekten ist es sinnvoll, Gleichheit und Identität zu unterscheiden. Umgangssprachlich entspricht das der Unterscheidung zwischen „das gleiche“ und „dasselbe“. Z. B. bedeutet „Chris und Kim fahren das gleiche Auto“ etwas anderes als „Chris und Kim fahren dasselbe Auto“. Im ersten Fall gibt es zwei Fahrzeuge, die sich „hinreichend“ ähneln, im zweiten Fall betreiben Chris und Kim Car-Sharing und teilen sich ein einziges Fahrzeug.

Die Umgangssprache trennt nicht zwischen Werten und Objekten. In Zusammenhang mit Werten – z. B. in der Mathematik oder bei Geldbeträgen – hat „Gleichheit“ oder „das gleiche“ eine andere Bedeutung als im Kontext von Objekten. Eine Aussage wie „Chris und Kim verdienen das gleiche Gehalt“ besagt, dass es sich bei Chris’ Gehalt und Kims Gehalt um denselben Geldbetrag und damit um einen einzigen Wert handelt. Gleichheit, bezogen auf Werte, beinhaltet dieselbe Aussage wie Identität bei Objekten, es handelt sich um *Synonyme*.

Umgekehrt bedeutet Gleichheit umgangssprachlich für Werte etwas anderes als für Objekte. In den obigen Beispielen liegen bei den „gleichen Fahrzeugen“ zwei Objekte vor, die in relevanten Merkmalen (z. B. Hersteller und Modell) übereinstimmen. Bei den „gleichen Gehältern“ geht es dagegen um einen einzigen Geldbetrag. Gleichheit, bezogen auf Werte, beinhaltet eine andere Aussage als Gleichheit bei Objekten, es handelt sich um *Homonyme*.

Abbildung 8.1 illustriert diese sprachlichen Verflechtungen.

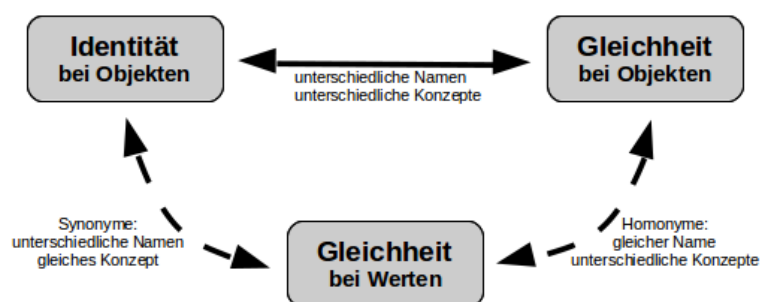


Abbildung 8.1: Gleichheit und Identität: Synonyme und Homonyme

Dass sich in der Umgangssprache in Bezug auf Wert-Gleichheit, Objekt-Gleichheit und Objekt-Identität Synonyme und Homonyme überlagern, macht es besonders

schwierig, die dahinter liegenden Konzepte gedanklich zu trennen.³

Wenn in Zusammenhang mit Objekten von Gleichheit die Rede ist, geht es darum, dass *zwei* Objekte in relevanten Merkmalen übereinstimmen.⁴ Diese Variante der Gleichheit ist kontextabhängig. Ein Objekttyp kann mehrere dieser Gleichheiten haben; je nach Sichtweise können zwei Objekte gleich oder ungleich sein. Welche Merkmale relevant sind, um zwei Objekte als „gleich“ einzustufen, hängt von – oft nur implizit vorliegenden – Anforderungen ab. Im obigen Beispiel der Fahrzeuge wurde z. B. unterstellt, dass Hersteller und Modell ausschlaggebend sind. Zwei gleiche Objekte sind immer – allein, *weil* es zwei sind – unterscheidbar, oft anhand von Eigenschaften, die für die jeweilige Gleichheit nicht relevant sind, mindestens aber anhand ihrer Identität. Die in den Abschnitten 8.1.2 und 8.1.4 beschriebenen Schwierigkeiten haben ihre Ursache darin, dass in der Methode `equals` eine solche Gleichheit zweier Objekte implementiert wurde. Weil „gleiche“ Objekte immer unterschieden werden können, und weil die Merkmale, anhand derer sie als gleich eingestuft werden, oft änderbar sind, kommt es zu den beschriebenen Effekten.

Wertgleichheit ist von einer Gleichheit zweier Objekte zu unterscheiden; sie stellt ein völlig anderes Konzept dar. Wertgleichheit ist notwendig, weil ein Wert, anders als ein Objekt, nicht über eine eigenständige Existenz verfügt und nur mit Hilfe von Repräsentationen dargestellt werden kann. In Zusammenhang mit Werten dient Gleichheit dazu, festzustellen, ob zwei Repräsentationen denselben konzeptionellen Wert darstellen. Diese Gleichheit ist, anders als Gleichheit bei Objekten, nicht abhängig von einem Kontext. Jeder Werttyp hat genau eine Gleichheit; sie legt implizit fest, was unter *einem* Wert zu verstehen ist.

Darin weist Wertgleichheit Gemeinsamkeiten mit Objektidentität auf. Objektidentität ist ebenfalls kontext-unabhängig und sie bestimmt, was unter *einem* Objekt zu verstehen ist. Anders als Wertgleichheit muss sie aber keinen Umweg über Repräsentationen gehen, da Objekte über eine eigenständige Existenz verfügen.

Im Folgenden stehen Wertgleichheit und Objektidentität im Fokus, ihre Gemeinsamkeiten und die Möglichkeiten, sie auf der Ebene der Programmiersprache zu unterstützen. Objektgleichheit wird von den weiteren Betrachtungen ausgeklammert. Sie spielt nur dort eine Rolle, wo der Gefahr der Verwechslung mit Wertgleichheit besteht und es essentiell ist, die Unterscheide klar herauszuarbeiten.

³Erschwerend kommt hinzu, dass natürliche Sprache per se ungenau ist. Sie kann allenfalls als Anhaltspunkt, nicht aber als zuverlässiges Unterscheidungskriterium dienen. Beispielsweise bezeichnet die englische Sprache eineiige Zwillinge als „identical twins“, doch handelt es sich bei ihnen um zwei verschiedene Personen (mit identischen Genen). Rechenberg belegt anhand etlicher Beispiele aus der Alltagssprache und aus der wissenschaftlichen Literatur, wie unpräzise die Begriffe „Identität/ identisch“ und „Gleichheit/ gleich“ benutzt werden [Rec06, Anhang A, p. 147–152].

⁴Diese Gleichheit findet sich zwar *meistens* bei Objekten, sporadisch aber auch in Zusammenhang mit Werten. Z. B. sagt man von zwei Personen, dass sie „am gleichen Tag“ Geburtstag haben, wenn Tag und Monat ihrer Geburtstage übereinstimmen. Im Kontext „Geburtstag“ werden implizit Tag und Monat eines Datums als „relevant“ angesehen und das Jahr vernachlässigt.

8.2.3 Vergleiche in Programmiersprachen

Die in vielen Programmiersprachen anzutreffende Unterstützung von zwei Vergleichen (s. Abschnitt 8.1.5) suggeriert, dass es zwei konzeptionell verschiedene Vergleiche gibt. Die Auffassung von Gleichheit und Identität als zwei einander entgegengesetzten Konzepten ist in der Informatik weit verbreitet. Sie ist u. a. zu finden bei Grogono und Sakkinen:

„There are two principal ways of comparing objects: we can check equality or identity.“[GS00]

bei Grogono und Santas:

„Identity is an a priori concept: an object is equal to itself. Equality is not an a priori concept: given two distinct objects, we need a rule that depends on the type of the object to decide whether they are equal.“[GS93]

und bei Khoshafian und Abnous:

„OO languages ... provide a simple identity test ... , which is different from the equality test ... The identity test checks whether the objects are the same. The equality test checks whether the contents of two objects are the same.“[KA90]

Meyer betrachtet die Fähigkeit, zwischen „identisch“ und „gleich“ zu unterscheiden, sogar als ein Wesensmerkmal objektorientierter Technologie:

„Being identical is not the same as being the same.... This ability to distinguish between the two notions is part of the modeling power of object technology.“[Mey97, p. 1052]

In anderen Arbeiten findet sich eine ähnliche Zweiteilung unter anderen Bezeichnungen. Felleisen et al. sprechen z. B. von intensionaler versus extensionaler Gleichheit [FFFK01], ebenfalls Grogono und Chalin [GC94], Moseley und Marks stellen intensionale und extensionale Identität einander gegenüber [MM06, p. 13].

Weil bestehende Programmiersprachen nicht zwischen Werten und Objekten unterscheiden, trennen sie auch nicht Wertgleichheit von einer Gleichheit zweier Objekte. Dadurch ist diese Unterscheidung auch in der Gedankenwelt vieler Programmierer nicht präsent.

Tabelle 8.3 schlüsselt exemplarisch das Verhalten der Vergleiche in Java auf:

Typ Operator/ Operation	==	equals
primitiver Datentyp	Wertgleichheit	<i>nicht verfügbar</i>
Klasse, bildet einen Werttyp ab, equals implementiert Wertgleichheit	<i>keine fachliche Bedeutung</i>	Wertgleichheit
Klasse, bildet einen Objekttyp ab, equals nicht überschrieben	Objektidentität	Objektidentität
Klasse, bildet einen Objekttyp ab, equals implementiert eine Objektgleichheit	Objektidentität	Objektgleichheit

Tabelle 8.3: Verhalten von == und equals in Java

Keiner der beiden in Java verfügbaren Vergleiche zeigt ein aus Sicht von Klienten einheitliches Verhalten. Die Methode `equals` kann Wertgleichheit, Objektidentität oder eine Objektgleichheit repräsentieren, abhängig davon, auf welchen Typ sie sich bezieht und wie sie vom Programmierer implementiert wurde. Der Operator „==“ bildet zwar bei primitiven Datentypen die Wertgleichheit ab und bei Klassen, die Objekttypen repräsentieren, die Objektidentität, und damit aus der Perspektive dieser Arbeit dasselbe Konzept. Bei Klassen, die Werttypen abbilden, hat er aber keine fachliche Bedeutung. Bei ihnen lässt er für Klienten irrelevante Implementierungsdetails durchschimmern und verursacht dadurch eine eigene Kategorie von Fehlern (s. Abschnitt 8.1.5).

In Sprachen, die Wertsemantik-Typen unterstützen (wie Eiffel mit `expanded types`, C# und X10 mit `struct`) ist die Situation nochmals komplizierter. In den meisten dieser Sprachen gibt es zwei (oder mehr) Vergleiche, von denen einige vom Programmierer überschrieben werden können. Die Standardimplementation eines Vergleichs hängt u. a. davon ab, ob es sich um einen Wertsemantik-Typ oder einen Referenztyp handelt. Auch hier besteht die Gefahr, Wertgleichheit und Gleichheit von Objekten zu vermengen. Weil Wertsemantik-Typen nicht notwendigerweise Werttypen sind und umgekehrt nicht alle Werttypen mit Wertsemantik-Typen abgebildet werden können (s. Kap. 5.5.1), bleibt auch hier für Klienten die intendierte Bedeutung eines Vergleichs unklar. Damit sind auch in diesen Sprachen die Konzepte Wertgleichheit und Objektidentität nicht deutlich erkennbar und nicht klar von einer Objektgleichheit unterscheidbar.

8.2.4 Konzept und Implementierung

Das in Java auftretende Problem, dass Gleichheit für Werttypen (die mit einer Objektklasse programmiert sein müssen) unter einem anderen Namen aufgerufen wird als Identität für Objekttypen – d. h. die Problematik der Synonyme – kann durch einen anderen Sprachentwurf gelöst werden. Beispielsweise ruft in Scala der Operator „==“ (definiert für Any und damit für alle Typen) die Methode `equals` auf. Durch geeignetes Überschreiben von `equals` kann der Programmierer einer Klasse dann das Verhalten des Operators „==“ beeinflussen und damit auch dafür sorgen, dass er die Wertgleichheit abbildet.

Die Unterscheidung zwischen Wertgleichheit und einer Objektgleichheit – d. h. die Problematik der Homonyme – ist auf der Ebene der Programmiersprache aber nicht lösbar, wenn Werte und Objekte sprachlich nicht getrennt sind. Eine Klasse, die die Methode `equals` basierend auf einem Vergleich von Datenfeldern implementiert, kann auf zwei Arten interpretiert werden: Es kann sich um die Abbildung eines Werttyps handeln, und `equals` implementiert die Wertgleichheit, oder es kann sich um die Abbildung eines Objekttyps handeln, und `equals` implementiert eine Gleichheit zweier Objekte.

So erschwert auch die Art und Weise, wie Wertgleichheit, Objektidentität und Objektgleichheiten implementiert sind, die gedankliche Einordnung dieser Konzepte:

- Wertgleichheit und Objektidentität weisen viele Gemeinsamkeiten auf. Sie können (s. Abschnitt 8.1) als synonyme Aussagen aufgefasst werden, einmal bezogen auf Werte, einmal bezogen auf Objekte. Sie sind aber programmiertechnisch oft verschieden umgesetzt. Wertgleichheit beruht auf einem Vergleich der Membervariablen und wird häufig durch den Programmierer der Klasse implementiert. Objektidentität vergleicht Referenzen oder technischen Adressen und ist meist ein Sprachprimitiv.
- Wertgleichheit und Objektgleichheit sind oft auf die gleiche Weise implementiert, nämlich als Vergleich korrespondierender Datenfelder. Sie verkörpern aber verschiedene Konzepte.

Weil die Implementation wenig über die Bedeutung eines Vergleichs aussagt, verzichtet diese Arbeit auch auf Begriffe wie „flache“ oder „tiefe“ Gleichheit, die in vielen Arbeiten eine wesentliche Rolle spielen, z. B. bei Khoshafian und Copeland [KC86], bei Kölling und Rosenberg [KR99] oder bei Grogono und Sakkinen [GS00]. Sie beschreiben lediglich Implementierungsvarianten, kein aus Klientensicht verschiedenes Verhalten.

8.2.5 Gleichheitsvertrag

Die Regeln des Gleichheitsvertrags sind nicht restriktiv genug, um inkonsistentes Verhalten zu unterbinden. Der Gleichheitsvertrag macht keine Aussage darüber, *was* Gleichheit ist, er legt Gleichheit nicht eindeutig fest. Nicht nur Wertgleichheit und Objektidentität erfüllen diesen Vertrag, sondern oft auch eine oder mehrere Objektgleichheiten. Das verursacht die in den Abschnitten 8.1.2 bis 8.1.7 aufgeführten Schwierigkeiten.

Der von Vaziri et al. vorgeschlagene abgewandelte Vertrag („revised equality contract“, s. Abschnitt 8.1.2) schließt Abhängigkeit von änderbarem Zustand aus. Diese verschärfte Fassung des Gleichheitsvertrags ist auch Bestandteil der für Scala beschriebenen Richtlinien zur Implementierung der Methode `equals` (s. Odersky [OSV11, p. 657].) Der abgewandelte Vertrag nach Vaziri behebt aber nur eines der in Zusammenhang mit Gleichheit auftretenden Probleme. Auch er schließt Objektgleichheiten ein und kann indeterministisches Verhalten hervorrufen.

8.3 Ein neues Sprachmodell für Gleichheit/ Identität

8.3.1 Modifizierter Gleichheitsvertrag

Im Folgenden wird ein modifizierter Gleichheitsvertrag vorgeschlagen. Seine Einhaltung behebt die in Abschnitt 8.1 aufgeführten Schwierigkeiten, und er bildet eine eindeutige und implementierungsunabhängige Beschreibung von Wertgleichheit und Objektidentität.

Modifizierter Gleichheitsvertrag:

1. Gleichheit muss eine *Äquivalenzrelation* bilden.
2. Gleichheit muss *unabhängig von änderbarem Zustand* sein.
3. Ein Vergleich von etwas mit `null` muss immer `false` liefern.
4. Gleichheit muss *ununterscheidbar* sein: Wenn `a` und `b` gleich sind, dann muss auch das Ergebnis jeder lesenden Operation `q`, angewendet auf `a` und `b`, gleich sein.

Wertgleichheit und Objektidentität erfüllen den modifizierten Gleichheitsvertrag. Dieser könnte darum mit gleicher Berechtigung als „Identitätsvertrag“ bezeichnet werden; in allen Bedingungen des Vertrages kann „Gleichheit“ durch „Identität“

ersetzt werden.⁵ Objektgleichheiten erfüllen den modifizierten Gleichheitsvertrag nicht, u. a. weil „zwei gleiche Objekte“ immer unterscheidbar sind (s. Abschnitt 8.1). Auf diese Weise schließt der modifizierte Gleichheitsvertrag Objektgleichheiten und das damit zusammenhängende, in Abschnitt 8.1 beschriebene, inkonsistente Verhalten aus.

Der modifizierte Gleichheitsvertrag geht in mehreren Punkten über den Vertrag von Bloch und über den von Vaziri hinaus:

- Bedingung 1 legt fest, dass Gleichheit immer eine Äquivalenzrelation bildet, nicht nur im Fall von „non-null references“.
- Bedingung 2 verlangt, dass der Vergleich von a und b immer das gleiche Ergebnis liefert, nicht nur, solange a und b unverändert sind. Er folgt in diesem Punkt dem „revised equality contract“ nach Vaziri et al. Anschaulich besagt Bedingung 2, dass Gleichheit keine zeitliche Komponente haben darf. Sind a und b jetzt gleich, dann auch zu jedem anderen Zeitpunkt.
- Bedingung 4 fordert: Wenn a und b gleich sind, dann darf kein Unterschied zwischen ihnen feststellbar sein („Ununterscheidbarkeit“). Umgekehrt formuliert, wenn eine lesende Operation q existiert, für die $q(a)$ und $q(b)$ ungleich sind, dann sind auch a und b ungleich: Gleichheit ist die „feingranularste“ aller Unterscheidungsmöglichkeiten (s. dazu Baker [Bak93]).

Der modifizierte Gleichheitsvertrag enthält ein universelles Bedingungswerk für Werttypen *und* Objekttypen. Würde er sich nur auf Werttypen beziehen, so wären Bedingung 2 und Bedingung 4 entbehrlich. Bedingung 2, die Forderung nach Unabhängigkeit von änderbarem Zustand, erübrigt sich, wenn die Elemente des Typs ohnehin unveränderlich sind. Bedingung 4 ist bei Werttypen immer erfüllt, sie ergibt sich aus der referentiellen Transparenz aller Wertoperationen.

Ununterscheidbarkeit, oben dargestellt anhand einer Operation q mit einem Parameter, gilt analog auch für Operationen mit zwei oder mehr Parametern. Ununterscheidbarkeit stellt einen Zusammenhang zwischen der Gleichheit und *allen lesenden* Operationen des Typs her. Die Bedingung bezieht sich nur auf lesende, d. h. seiten-effektfreie Operationen, weil sie für ändernde Operationen i. A. nicht erfüllbar ist. Wenn eine Methode m Änderungen bewirkt, können selbst zwei Aufrufe von $a.m()$ verschiedene Ergebnisse liefern.

Eine Änderung der Elemente kann sich auf das Ergebnis einer lesenden Operation q auswirken, sie beeinträchtigt dennoch nicht die Ununterscheidbarkeit. Ununterscheidbarkeit verlangt nicht, dass das Ergebnis einer lesenden Operation q zu

⁵Weil Objektidentität in der Regel ein Sprachprimitiv ist und nicht vom Programmierer einer Objektklasse implementiert werden muss (s. Abschnitt 8.3.4), entfällt der Bedarf nach Anweisungen, wie Identität zu implementieren ist.

allen Zeitpunkten gleich bleibt, sondern lediglich, dass $q(a)$ und $q(b)$ – zu einem festen Zeitpunkt – für gleiche a und b das gleiche Ergebnis liefern.

Ununterscheidbarkeit (Bedingung 4) ist etwas anderes als Unabhängigkeit der Gleichheit von änderbarem Zustand (Bedingung 2). Letztere verlangt, dass der Vergleich von a und b immer zum gleichen Ergebnis führt, unabhängig von den Änderungen, denen a und b unterworfen sind.

Die hashCode-Regel (s. Abschnitt 8.1.3) bildet einen Spezialfall der Ununterscheidbarkeit,⁶ sie ist aber nicht Bestandteil des Gleichheitsvertrags. Beispielsweise beschreiben Java und Scala den Gleichheitsvertrag in der API-Dokumentation der Methode `equals`. Die hashCode-Regel ist, getrennt davon, in der API-Dokumentation der Methode `hashCode` enthalten [Ora14].

Ein Beispiel für eine Implementierung, die Ununterscheidbarkeit *nicht* erfüllt, ist die folgende Wertklasse:

```
valueClass Rational {
  Int zaehler;
  Int nenner;
  Rational getRational(Int z, Int n) {
    return select (z,n);
  }
  // Definition der Gleichheit
  Bool equality (Rational r1, Rational r2) {
    return r1.zaehler*r2.nenner = r2.zaehler*r1.nenner;
  }
  Int getZaehler (Rational r) {
    return r.zaehler;
  }
  ...
}
```

Bei dieser Implementierung der rationalen Zahlen ist die interne Repräsentation nicht standardisiert (s. dazu Abschnitt 6.3.5), sie lässt auch ungekürzte Darstellungen zu. Die generierbare kanonische Gleichheit würde hier nicht korrekt funktionieren. Die Klasse benötigt deshalb eine vom Programmierer implementierte Gleichheit. Mit der Operation `getZaehler` können Klienten gleiche Werte unterscheiden, entgegen den Anforderungen des modifizierten Gleichheitsvertrags. Das zeigt das folgende Anwendungsbeispiel:

⁶Die hashCode-Regel hat möglicherweise deshalb einen hohen Bekanntheitsgrad erlangt, weil viele Klassen in den Java- und Scala-Bibliotheken, insbesondere Sammlungen, für die Speicherung und den Zugriff auf ihre Bestandteile die Methode `hashCode` einsetzen und sich dabei auf den beschriebenen Zusammenhang zwischen `equals` und `hashCode` verlassen.

```

Rational r1 := getRational(3,4);
Rational r2 := getRational(6,8);
if (r1 = r2) ... // true
if (getZaehler(r1) = getZaehler(r2)) ... // false

```

Ununterscheidbarkeit wird auch in anderen Arbeiten als ein Wesensmerkmal von Gleichheit angesehen.

Z. B. stellen Liskov und Guttag fest:

„it should be impossible to distinguish between equal objects“ [LG86]
 „if two objects are equals, it will never be possible to distinguish them
 in the future using methods of their type.“[LG01, p. 182]

Grogono und Santas bezeichnen eine Gleichheit, die Bedingung 4 erfüllt, als „congruence“:

„If there are operators, the equality predicate should be consistent
 with them. ...An equality predicate that is consistent with the operations
 of an algebra is called a congruence. ...
 if $E(x)$ is a term, then $a = b$ implies $E(a) = E(b)$.“[GS93]

Der modifizierte Gleichheitsvertrag legt – anders als der von Bloch und der von Vaziri – den Vergleich *eindeutig* fest. Jeder Typ kann nur einen Vergleich haben, der den modifizierten Gleichheitsvertrag einhält. Bei Objekttypen ist das die Objektidentität, bei Werttypen die Wertgleichheit.

Die Eindeutigkeit kann folgendermaßen nachgewiesen werden:

Eindeutigkeit des Vergleichs

Es wird gezeigt, dass zwei Operationen $eq1$ und $eq2$, die zwei Elemente eines Typs T vergleichen und den modifizierten Gleichheitsvertrag einhalten, genau die gleichen Elementpaare als gleich einstufen und deshalb gleich sind.
 Für alle a und b vom Typ T gilt $eq1(a,b)$ genau dann, wenn auch $eq2(a,b)$ gilt.

Beweis:

Wenn $eq1(a,b)$ **true** ergibt, dann gilt
 $eq2(a,b)$
 $= eq2(a,a)$ (aufgrund der Ununterscheidbarkeit von $eq1$)
 $=$ **true** (aufgrund der Reflexivität von $eq2$)

Wenn $eq2(a,b)$ **true** ergibt, dann gilt
 $eq1(a,b)$
 $= eq1(a,a)$ (aufgrund der Ununterscheidbarkeit von $eq2$)
 $=$ **true** (aufgrund der Reflexivität von $eq1$)

Der modifizierte Gleichheitsvertrag eignet sich deshalb nicht nur als Programmierrichtlinie, er ist auch eine implementierungsunabhängige Charakterisierung der Konzepte Wertgleichheit und Objektidentität.

8.3.2 Gleichheit und Identität als Operationen auf Ausdrücken

Abschnitt 8.2.2 hat gezeigt, dass Wertgleichheit und Objektidentität in vielerlei Hinsicht Gemeinsamkeiten aufweisen und deshalb als Synonyme betrachtet werden können:

Wertgleichheit enthält die Aussage, dass sich zwei Ausdrücke auf denselben Wert beziehen.
Objektidentität enthält die Aussage, dass sich zwei Ausdrücke auf dasselbe Objekt beziehen.

Wertgleichheit und Objektidentität nehmen aufgrund der in ihnen enthaltenen Aussage eine semantische Sonderstellung ein. Sie sind *Operationen auf Ausdrücken* – anders als die meisten anderen Operationen, die auf den von den Ausdrücken bezeichneten Elementen operieren.⁷

Deshalb sind verbreitet anzutreffende Formulierungen wie „zwei gleiche Werte“ oder „zwei Objekte sind identisch ...“ sprachlich irreführend: Sie scheinen die Aussage zu enthalten, dass *zwei* Elemente vorhanden sind. Bereits das Vorliegen von zwei Werten aber würde implizieren, dass sie nicht gleich sind, ebenso können zwei Objekte nicht identisch ein.

Auch die Umschreibung, Identität sei eine *Eigenschaft eines* Objektes, führt in die Irre. Sie ist z. B. zu finden bei Koshafian und Copeland:

„Identity is that property of an object which distinguishes each object from all others“[KC86])

oder bei Meyer:

„Object identity: A property that uniquely identifies an object independently of its current contents“[Mey97, p. 1198].

Diese Formulierungen lassen nicht erkennen, dass Identität *Ausdrücke* vergleicht, nicht Objekte. Des weiteren vernachlässigt sie, dass Identität nur anwendbar ist, wenn *zwei* Ausdrücke involviert sind.

⁷Auch die logische Negation der Gleichheit, die Ungleichheit, nimmt diese Sonderstellung ein. Die Zuweisung ist ebenfalls ein Sonderfall, keine „normale“ Operation. Indem sie der Variablen auf der linken Seite ein neues Element zuweist, operiert sie nicht auf dem Element, das diese Variable bezeichnet, sondern auf der Variablen selbst.

Die Signatur der Gleichheits- und der Identitätsprüfung – zwei Elemente des Typs als Eingabe, ein bool'scher Wert als Ergebnis – und die Verwendung eines Infix-Operators lassen ihre semantische Sonderstellung ebenfalls nicht erkennen. Eine solche Schreibweise findet sich auch bei „normalen“ Operationen, beispielsweise bei der kleiner-Relation für Gleitkommazahlen („ $a < b$ “) oder bei der Teilerfremdheit zweier ganzer Zahlen („ $a \perp b$ “). In den beiden letztgenannten Fällen geht es aber darum, die durch die Ausdrücke bezeichneten Werte zu vergleichen – in Bezug auf ihre Anordnung oder ihre Teilerfremdheit – nicht die Ausdrücke selbst.

Frege, dessen Arbeiten als grundlegend für Logik und formale Sprachen gelten, hat nachdrücklich auf die Sonderstellung von Gleichheit bzw. Identität hingewiesen. Demnach ist Gleichheit „eine Beziehung zwischen Namen oder Zeichen für Gegenstände“ [Fre92], nicht eine Beziehung zwischen den Gegenständen selbst. Ausführlicher begründet er das in der „Begriffsschrift“:

„Die Inhaltsgleichheit unterscheidet sich dadurch von der Bedingtheit und Verneinung, daß sie sich auf Namen, nicht auf Inhalte bezieht. Während sonst die Zeichen lediglich Vertreter ihres Inhaltes sind, sodass jede Verbindung, in welche sie treten, nur eine Beziehung ihrer Inhalte zum Ausdrucke bringt, kehren sie plötzlich ihr eignes Selbst hervor, sobald sie durch das Zeichen der Inhaltsgleichheit verbunden werden; denn es wird dadurch der Umstand bezeichnet, dass zwei Namen denselben Inhalt haben.“[Fre79]

Auch einige Arbeiten im Bereich der Informatik vertreten die Auffassung, dass Identität sich auf Bezeichner, oder allgemeiner, auf Ausdrücke bezieht, z. B. bei Züllighoven et al.:

„Identität bedeutet zunächst, dass auf programmiersprachlicher Ebene eindeutig festzustellen ist, ob zwei Bezeichner auf dasselbe Objekt verweisen.“[Zül98, p. 29]

Kent hebt ebenfalls hervor, dass Identität nicht auf den Objekten selbst operiert:

„A predicate of the form Identical(O1,O2) is hard to explain if O1 and O2 are meant to be the objects themselves. ... Distinguishability means we know this thing is not that thing, even if we don't know any distinguishing properties.“[Ken91]

Die Philosophie (Sprachphilosophie und analytische Philosophie) hat sich intensiv mit den Begriffen Identität und Gleichheit auseinandergesetzt; neben Frege haben sich u. a. Russel, Wittgenstein, Mauthner und Windelband mit ihnen befasst. Nicht alle Autoren sehen Identität und Gleichheit als ein- und denselben Sachverhalt an. In

den Eigenschaften der Identität lassen sie jedoch deutlich Wesenszüge von Objekten erkennen, und in denen der Gleichheit Wesenszüge von Werten. Wo die Philosophie Unterschiede zwischen Identität und Gleichheit konstatiert, betreffen sie nicht die Art und Weise des Vergleichens, sondern vielmehr die unterschiedliche Natur der verglichenen Gegenstände.

Beispielsweise definiert Windelband [Win10a] Identität als „seiende Gleichheit“, als „beharrende Gleichheit des wesentlichen Inhalts“. Identität wird „postuliert“ oder „normiert“, nicht formal logisch festgestellt. Sie kann i. A. nicht an einzelnen Merkmalen festgemacht werden. Im Laufe der Entwicklung eines Nusskerns zum Baum ändern sich z. B. seine Form, Größe und Materie. Diese Beschreibung lässt die Charakteristika von Objekten erkennen, welche grundsätzlich änderbar sind und die nacheinander verschiedene Zustände annehmen können.

Gleichheit dagegen ist nach Windelband eine „Urteilsform des reflektierenden Denkens“, sie bedeutet „kein reales Verhältnis der Gegenstände zueinander“. Das an die Gleichheit geknüpfte Abstrahieren von realen Gegenständen weist auf wertartige Kategorien (wie Anzahl, Größe, Farbe, Form) hin. „Das Maß dessen, was ... gleich sein muss, und dessen, was ... ungleich sein darf, ist keineswegs eindeutig bestimmt ... es hängt in jedem besonderen Falle von dem Standpunkt der Reflexion ab [Win10a].“ Diese Kontextabhängigkeit der Gleichheit lässt erkennen, dass hier Gleichheit im Sinne einer „Gleichheit zweier Objekte“ (s. Abschnitt 8.2.2) gemeint ist, d. h. dass hier ein oder mehrere als relevant eingestufte Merkmale übereinstimmen, dass aber zwei Gegenstände vorhanden und im Prinzip unterscheidbar sind.

8.3.3 Verschmelzung von Gleichheit und Identität

Die vorangegangenen Überlegungen ermöglichen, Wertgleichheit und Objektidentität zu einem einheitlichen Konzept zu verschmelzen. Dadurch hat im V-O-Modell jeder Typ nur einen Vergleich: Werttypen die Wertgleichheit, Objekttypen die Objektidentität. Gleichheit im Sinne einer „Übereinstimmung zweier Elemente in relevanten Merkmalen“ dagegen wird ausdrücklich ausgeklammert und ist nicht Bestandteil des V-O-Modells.

Ein einheitlicher Vergleich betont die in diesem Abschnitt herausgearbeiteten konzeptionellen Gemeinsamkeiten von Wertgleichheit und Objektidentität. Beide verkörpern bezogen auf ihren Typ die eindeutig festgelegte Operation, die den modifizierten Gleichheitsvertrag erfüllt (s. Abschnitt 8.3.1). Beide sind Operationen auf den beteiligten Ausdrücken, nicht auf den Elementen, auf die durch diese Ausdrücke Bezug genommen wird (s. Abschnitt 8.3.2).

Die Alternative wäre, für Wertgleichheit und für Objektidentität verschiedene Symbole vorzusehen. Ein Sprachentwurf mit zwei Vergleichssymbolen ist z. B. in Simula67 umgesetzt: Primitive Typen werden dort mit „=“ verglichen, Klassenty-

pen mit „==“. Auch die Symbole für die Zuweisungen unterscheiden sich, „:=“ wird in Zusammenhang mit primitiven Typen, „:-“ bei Klassentypen benutzt [DMN68]. Da der Anwendungsbereich jedes Symbols jedoch feststeht und keine Verwechslungsgefahr besteht, ist der Nutzen dieser Unterscheidung begrenzt. Sie ist deshalb als zu umständlich kritisiert worden, z. B. von Meyer [Mey97, p. 272] oder von Kühne [Küh99, p. 245].

Ein wesentlicher Vorteil eines einheitlichen Vergleichs liegt in einer einfachen Benutzerschnittstelle und einer Reduzierung potentieller Fehler. Wenn es nur einen Vergleich gibt, besteht keine Notwendigkeit, unter mehreren Vergleichen auszuwählen, damit entfällt die Gefahr, versehentlich einen unpassenden Vergleich zu verwenden (s. Abschnitt 8.1.5).

In einer Programmiersprache mit Unterstützung für parametrische Polymorphie („Generizität“) bietet ein einheitlicher Vergleich weitere Vorteile: Die Bestandteile von Sammlungen (Collections) – Typen wie Mengen (Set), Listen (List) oder Abbildungen (Map) – werden durch generische Typparameter ausgedrückt. Mit einem einheitlichen Vergleich können in der Implementation einer generischen Sammlung die darin enthaltenen Elemente miteinander verglichen werden, obwohl noch nicht feststeht, ob der Typparameter durch einen Werttyp oder einen Objekttyp instanziiert wird.

Eine Reihe von Arbeiten weist darauf hin, dass zwei Arten von Typen zu unterscheiden sind und jede einen auf sie zugeschnittenen Vergleich benötigt. Die meisten dieser Arbeiten unterscheiden dabei änderbare und unveränderliche Objekte – was der Trennung von Werten und Objekten nahe kommt, aber nicht mit ihr übereinstimmt (s. Kapitel 3).

Z. B. stellt Baker fest:

„Briefly, if an object is mutable, then we compare it using an "address-like" comparison, while if an object is immutable, then we recursively compare its components. ... EQ is correct for mutable cons cells and EQUAL is correct for immutable cons cells.“[Bak93]

Liskov und Guttag schliessen die unterschiedlichen Vergleiche aus der Forderung, dass gleiche Elemente nicht unterscheidbar sein dürfen:

„... it should be impossible to distinguish between equal objects. For mutable objects, this implies that equality is identity. Otherwise, it would be possible to distinguish apparently equal objects by making a change to one object and then checking whether the other object changed. Two immutable objects are equal if their abstract values are the same.“[LG86]

Grogono und Chalin betrachten extensionale Gleichheiten als angemessen für unveränderliche Objekte und intensionale Gleichheit für veränderliche:

„Immutable objects should be compared extensionally because we are concerned only about their values. ... Mutable objects should be compared intensionally because we are concerned with object identity.“[GC94]

Tabelle 8.4 (entnommen [GC94]) illustriert, dass sie intensionale Gleichheit für unveränderliche Objekte als „irrelevant“ und extensionale Gleichheit für änderbare Objekte als „irreführend“ ansehen.

Kind of Object	Kind of Comparison	
	Extensional	Intensional
Immutable	Appropriate	Irrelevant
Mutable	Misleading	Appropriate

Tabelle 8.4: Extensionale und intensionale Gleichheit nach Grogono und Chalin

Die genannten Arbeiten unterscheiden aber weiterhin zwei verschiedene Vergleiche. Es ist Sache des Programmierers, auf jeden Typ den passenden Vergleich anzuwenden und damit lediglich eine – fehleranfällige – Konvention. Das in dieser Arbeit vorgeschlagene Sprachmodell geht darüber hinaus, indem es Wertgleichheit und Objektidentität zu einem Konzept vereinigt und dafür sorgt, dass für jeden Typ nur der passende Vergleich verfügbar ist.

Die Auffassung von Gleichheit und Identität als Gegensatzpaar dominiert viele Arbeiten, und sie liegt implizit vielen Programmiersprachen zugrunde (s. Abschnitt 8.2.3). Es gibt aber auch einige Arbeiten, die Gleichheit und Identität als einheitliches Konzept ansehen.

Bereits Frege verwendet die Begriffe „Gleichheit“ und „Identität“ synonym:

„Ich brauche dies Wort (Gleichheit) im Sinne von Identität und verstehe "a = b" in dem Sinne von "a" ist dasselbe wie "b" oder "a und b fallen zusammen.“[Fre92]

Kent wendet das Konzept von Identität auch auf „nicht erzeugte“ Abstraktionen – sie entsprechen im V-O-Modell den Werten – an:

„The model of identity ... also applies equally to explicitly created objects and to eternally existing abstractions like numbers and extensional sets.“[Ken91]

Bacon schlägt mit der Sprache Kava eine effiziente Unterstützung für Werttypen wie komplexe Zahlen vor. Auf diese Typen findet in Kava nur Wertgleichheit Anwendung, und diese wird über den gleichen Operator aufgerufen wie Objektidentität:

„Values may only be distinguished by their contents, not by their identity. ... The == operator, when applied to values, returns true if and only if the contained values are equal.“[Bac03]

Vaziri et al. verwenden ebenfalls in ihrer Arbeit über „Declarative Object Identity“ die Begriffe Identität und Gleichheit synonym. Sie betrachten die Methode `equals` als Mittel, um eine von „==“ abweichende Implementation für *Identität* festzulegen („... customize the notion of object identity by overriding the `equals()` ... method“ [VTFD07]), und damit `equals` als gleichbedeutend mit Identität, nicht als Alternative oder Gegensatz.

Die Verschmelzung von Wertgleichheit und Objektidentität geht mit einem sprachlichen Problem einher. Die unterschiedliche Benennung – Gleichheit in Zusammenhang mit Werten, Identität in Zusammenhang mit Objekten – drückt die im V-O-Modell beabsichtigte Verschmelzung zu einem einheitlichen Konzept nicht deutlich aus. Ein einheitlicher Begriff wäre wünschenswert, ist aber aufgrund des gewachsenen Gebrauchs dieser Worte schwer umsetzbar. Das Wort „Gleichheit“ würde in Zusammenhang mit Objekten auf ein anderes Konzept (s. Abschnitt 8.2.2) hinweisen. Das Wort „Identität“ würde in Zusammenhang mit Werten die Gefahr der Verwechslung mit einem – fachlich irrelevanten – Vergleich von technischen Repräsentationen oder ihren Speicheradressen bergen.

Der Begriff *Gleichheit* wird im Folgenden ausschließlich im Kontext von Werttypen und damit im Sinne von Wertgleichheit verwendet. In Zusammenhang mit Objekttypen wird von *Identität* die Rede sein. Soweit der Kontext nicht feststeht, oder wenn nicht relevant ist, ob es sich um Werttypen oder Objekttypen handelt, wird auf den Schrägstrich-Begriff *Gleichheit/ Identität* zurückgegriffen.

8.3.4 Sprachunterstützung für Gleichheit/ Identität

Die Verschmelzung von Wertgleichheit und Objektidentität kann in einer auf dem V-O-Modell aufbauenden Sprache folgendermaßen unterstützt werden:

Dasselbe Symbol für Gleichheit und Identität

Wertgleichheit und Objektidentität werden durch dasselbe Symbol aufgerufen. Das bringt auf der Ebene der Programmiersprache zum Ausdruck, dass beide ein vergleichbares Konzept verkörpern. Im Folgenden wird für Gleichheit und Identität, wie in Kap. 6, der Infix-Operator „==“ verwendet.

Keine Unterstützung von Objektgleichheit

Eine Gleichheit von Objekten – im Sinne von „Übereinstimmung in relevanten Merkmalen“ – wird seitens der Programmiersprache nicht unterstützt. Sie kann, sofern sie

für einen Typ benötigt wird, wie jede andere fachlich motivierte Operation mit den vorhandenen Sprachmitteln unter einem passenden Namen programmiert werden. Die Sprache verhindert, dass eine Objektgleichheit unter dem für Gleichheit/ Identität reservierten Symbol implementiert wird. Damit schließt sie ein Verwechseln oder Vermengen dieser beiden unterschiedlichen Konzepte und das daraus resultierende inkonsistente Verhalten aus.

Gleichheit/ Identität als Sprachprimitiv

Gleichheit/ Identität ist ein eigenständiges Sprachkonstrukt. Die zentrale Bedeutung von Gleichheit und Identität und ihre in Abschnitt 8.3.2 beschriebene Sonderstellung rechtfertigen eine eigene Sprachunterstützung.

Gleichheit/ Identität kann insbesondere keine „normale“ Exemplarmethode sein. Das folgt aus Bedingung 1 und 3 des modifizierten Gleichheitsvertrags: Weil `(e = null)` `false` ergibt, muss aufgrund der Symmetrie auch `(null = e)` `false` ergeben. Eine Exemplarmethode wäre für ein erstes Argument von `null` nicht definiert und würde eine `NullPointerException` – oder, je nach Sprache, das entsprechende Äquivalent – auslösen.

Vergleich typ-inkompatibler Ausdrücke

Weil Gleichheit/ Identität von der Sprache unterstützt wird, können Vergleiche zwischen typ-inkompatiblen Ausdrücken (s. Abschnitt 8.1.6) bereits zur Übersetzungszeit als Fehler erkannt und zurückgewiesen werden. Das erhöht die Sicherheit der Sprache.

Vergleich mit null

Eine Sprachunterstützung für Gleichheit/ Identität kann `null`-Operanden gemäß den Vorgaben des modifizierten Gleichheitsvertrags behandeln, d. h. sicherstellen, dass ein Vergleich `false` ergibt, wenn genau einer der beiden Operanden `null` ist (s. Abschnitt 8.1.7). Das erspart aufwändige und fehlerträchtige Prüfungen auf `null` seitens der Klienten, und es gewährleistet die Symmetrie der Gleichheit/ Identität auch bei Vergleichen mit `null`.

Generierung durch die Sprachumgebung versus Programmierung

Um die Anforderungen an Gleichheit/ Identität sicherzustellen, sollte sie von der Programmiersprache zur Verfügung gestellt werden. Ihre Implementation sollte, soweit das möglich ist, automatisch durch die Sprachumgebung generiert werden.

Objektidentität kann von der Programmiersprache generiert werden. In den meisten objektorientierten Sprachen bildet Objektidentität ein Sprachprimitiv, auf dessen Verhalten der Programmierer keinen Einfluss nehmen kann. In der zugrunde liegenden technischen Implementation beruht Identität meist auf einem Vergleich

von Speicheradressen; Koshafian und Copeland beschreiben verschiedene Implementierungsvarianten [KC86]. Objektidentität ist gekoppelt an Objekterzeugung: bei jeder Erzeugung entsteht ein neues, von allen anderen Objekten verschiedenes Objekt (s. dazu Kent: „Object identity might be predicated on creation, with distinct objects arising from distinct creation events.“ [Ken91]).

Wertgleichheit kann, wie Abschnitt 6.3.5 gezeigt hat, in vielen Fällen ebenfalls durch die Programmiersprache generiert werden, und zwar:

- für Aufzählungs-Wertklassen,
- für strukturierte Wertklassen mit eindeutiger Repräsentation,
- für strukturierte Wertklassen mit potentiell mehrdeutiger Repräsentation, wenn der Programmierer sie auf eine standardisierte Darstellung bringt.

Nur strukturierte Wertklassen mit mehrdeutiger Repräsentation, bei denen keine kanonische Darstellung ermittelt werden kann oder soll, erfordern einen Sprachmechanismus für eine programmierer-definierte Implementation der Gleichheit (s. auch hierzu Abschnitt 6.3.5).

In der Literatur herrscht die Auffassung vor, dass Gleichheit ein semantisches Konzept darstellt, das nicht von der Sprachumgebung unterstützt und nicht von einem Compiler generiert werden kann (s. z. B. Grogono und Santas: „... a compiler cannot generate correct equality tests for user-defined types“ [GS93] oder Liskov und Guttag: „... extensional equality cannot be built into the language“ [LG86].)

Tabelle 8.5 fasst die im V-O-Modell möglichen Konstellationen zusammen. Sie verdeutlicht, dass Gleichheit/ Identität zwar nicht in allen, aber in vielen Fällen – in der Tabelle grün markiert – durch die Sprache generiert werden kann. Die sprachliche Trennung von Werttypen und Objekttypen ist dafür eine notwendige Voraussetzung: sie ermöglicht, die automatisch generierte Identität/ Gleichheit auf den jeweiligen Typ auszurichten.

Objektklasse	Wertklasse			
	Aufzählungs- wertklasse	strukturierte Wertklasse		
		mit eindeutiger Repräsen- tation	mit mehrdeutiger Repräsentation	
			kanonische Form	keine kanonische Form
sprach- generierte Identität	sprach- generierte Gleichheit	sprach- generierte (kanonische) Gleichheit	sprach- generierte (kanonische) Gleichheit	vom Programmierer definierte Gleichheit

Tabelle 8.5: Sprachgenerierte versus programmierer-definierte Gleichheit/ Identität

Im V-O-Modell ist Gleichheit/ Identität ein Sprachbaustein, nicht eine vom Programmierer frei definierbare Operation. Das trifft auch im Fall einer vom Programmierer implementierten Wertgleichheit zu. Auch sie ist größtenteils unter der Kontrolle der Sprache:

- Die Sprache legt das Symbol für die Gleichheit fest.
- Die Sprache bestimmt die Signatur der Gleichheit (zwei Elemente des jeweiligen Typs als Eingabe, Rückgabetyt Boolean).
- Die Sprache übernimmt (s. o.) die Prüfung auf Typkompatibilität der verglichenen Ausdrücke und das Verhalten der Gleichheit, wenn einer oder beide Operanden `null` sind.

Der Programmierer kann lediglich die Implementation festlegen. Innerhalb dieser kann nur auf die Datenfelder der Wertklasse zugegriffen werden. Ein Vergleich anhand einer technischen Adresse – wie er der Identität zugrunde liegt, und der gegen Ununterscheidbarkeit verstoßen würde – ist nicht möglich.⁸

Ein Sprachmechanismus zur Festlegung einer Implementation für die Wertgleichheit weist Parallelen zum Überschreiben von Methoden und zum Überladen von Operatoren auf, er ist aber weder das eine noch das andere. Der Programmierer erhält, wie beim Überladen oder Überschreiben, die Möglichkeit, einen bereits an anderer Stelle definierten Namen (das Symbol für die Gleichheit/ Identität) mit einer eigenen Implementation zu versehen. Anders als beim Überladen ist die Signatur festgelegt und kann nicht frei gewählt werden, es können auch nicht mehrere gleichnamige Operationen mit verschiedenen Signaturen für dieselbe Klasse definiert werden. Im Unterschied zum Überschreiben setzt der Mechanismus für die Implementierung der Gleichheit keine Subklassenbeziehung voraus (und auf technischer Ebene erfordert er kein dynamisches Binden).

Das V-O-Modell und die daraus resultierende Handhabung von Gleichheit/ Identität auf programmiersprachlicher Ebene führen dazu, die Verantwortlichkeiten zwischen dem Programmierer einer Wertklasse und den Programmierern von Klienten-code klar zu trennen. Nur die Wertklasse greift auf die hinter den Werten liegenden

⁸Die technische Umsetzung der Gleichheit in der zugrunde liegenden Implementation ist unabhängig davon, ob sie von der Sprachumgebung generiert wird oder ob der Programmierer der Wertklasse sie implementiert. Prinzipiell sind zwei Implementationen denkbar: Bei der ersten Variante lässt die Sprachumgebung mehrere speichertechnische Repräsentationen eines Wertes zu, und diese werden bei Bedarf durch die Gleichheitsformel verglichen. Bei der zweiten Variante sorgt die Sprachumgebung dafür, dass es zu einem konzeptionellen Wert maximal eine Speicherrepräsentation gibt, z. B. indem sie bei Aufruf eines Selektors (mit Hilfe der Gleichheitsformel) prüft, ob der gerade selektierte Wert bereits über eine Speicherrepräsentation verfügt, und falls ja, diese verwendet. Bei Aufruf der Gleichheit genügt es dann, zu prüfen, ob dieselbe Speicherrepräsentation vorliegt. Die erste Variante entspricht dem Value Object Pattern (s. Abschnitt 4.2.1), die zweite dem Fliegengewicht (s. Abschnitt 4.2.2) – die Umsetzung liegt aber unter der ausschließlichen Kontrolle der Sprachumgebung und ist für den Anwendungsprogrammierer nicht sichtbar. Variante 1 ist eher für leichtgewichtige, Variante 2 für schwergewichtige Wertrepräsentationen geeignet.

Repräsentationen zu. Der Programmierer der Wertklasse sorgt (im Fall einer nicht-standardisierten internen Darstellung) dafür, dass die Wertgleichheit mehrere Repräsentationen des gleichen Wertes korrekt zusammen führt und dass verschiedene Repräsentationen desselben Wertes sich in jeder Hinsicht gleich verhalten, d. h. anhand anderer Operationen der Wertklasse nicht unterscheidbar sind. Klienten sind dadurch vollständig von der Handhabung der Repräsentationen abgeschirmt. Klienten hantieren aus ihrer Sicht direkt mit den Werten – gedanklichen Konstrukten ohne eigene Existenz. Auf diese Weise ist der Umgang mit Werten und mit Objekten für Klienten in vielerlei Hinsicht ähnlich, obwohl es sich um fundamental verschiedene Abstraktionen handelt.

8.3.5 Vergleiche zu bestehenden Programmiersprachen

Einige Programmiersprachen weisen Parallelen zu dem hier beschriebenen Sprachmodell auf. Insbesondere Scala und Sather sind dem V-O-Modell in einigen Punkten ähnlich. Weil diese Sprachen Werte und Objekte nicht trennen, gibt es aber auch wesentliche Unterschiede. Im Folgenden wird ihr Umgang mit Gleichheit und Identität näher betrachtet.

Wie in Abschnitt 8.2.4 angemerkt, ruft in Scala der Operator „==“ die Methode `equals` auf. Darum können Objektidentität und Wertgleichheit – passende Implementierung von `equals` vorausgesetzt – unter demselben Symbol aufgerufen werden. In Scala können deshalb z. B. String-Ausdrücke semantisch korrekt mit „==“ verglichen werden. Auch kümmert sich in Scala die Methode „==“ (definiert in `Any` und nicht überschreibbar) um die Handhabung von `null` für den ersten Operanden. Weil Scala nicht zwischen Werttypen und Objekttypen unterscheidet, kann in der Methode `equals` – und damit unter dem Operator „==“ – aber auch eine Gleichheit zweier Objekte implementiert sein. Auch verfügt Scala mit der Methode `eq` („Referenzgleichheit“, definiert in `AnyRef`) über noch einen weiteren Vergleich, was die Gefahr von Verwechslungen birgt. Ein Vergleich typinkompatibler Ausdrücke mit „==“ oder `equals` wird vom Scala-Compiler nicht beanstandet. Das Paket `scalaz` definiert mit dem Operator „===“ einen typsicheren Vergleich, was die Anzahl der verfügbaren Vergleichsoperatoren nochmals vergrößert.

Sather unterstützt – anders als Java, C#, Scala oder Eiffel – nur einen Vergleich (`is_eq`, syntaktisch auch aufrufbar über den Operator „=“ [SO96]). Bereits diese Entwurfsentscheidung vermeidet einige der in Abschnitt 8.1 beschriebenen Fehlerquellen. Für „=“ alias „`is_eq`“ gibt es aber keine sprachseitige Standardimplementation. `is_eq` muss in jeder Klasse implementiert werden, was entsprechend fehlerträchtig ist. Prinzipiell ist es möglich, darin eine Gleichheit zweier Objekte zu programmieren. Auch unterbindet Sather keine Vergleiche typinkompatibler Ausdrücke. Sather unterscheidet zwei Arten von Klassen, „immutable classes“ und „reference

classes“. Diese Trennung entspricht aber nicht der von Werttypen und Objekttypen. Z. B. können einige Werttypen, wie Zeichenketten oder Ganzzahlen mit beliebiger Genauigkeit (Infinite precision integer), aufgrund technischer Restriktionen (s. Abschnitt 5.5.1) nicht als immutable class programmiert werden.

8.4 Beseitigte Probleme

Die Sprachmodelle, die Programmiersprachen wie Java, Scala, C#, Eiffel, Smalltalk, etc. implizit zugrunde liegen, kennen nur einen Typ („alles ist ein Objekt“), aber zwei (oder mehr) Vergleiche. Viele der in Abschnitt 8.1 aufgeführten Beispiele vermischen „wert-artiges“ Verhalten (den auf Inhalten basierenden Vergleich) und „objekt-artiges“ Verhalten (Erzeugbarkeit, Änderbarkeit) in einer Klasse. Das ist deshalb möglich, weil die Sprachmodelle der genannten Sprachen Werttypen und Objekttypen nicht voneinander trennen und deshalb nicht darauf hinwirken, Gleichheit/ Identität auf den jeweiligen Typ auszurichten. Die Schwierigkeiten bei der Implementierung und Benutzung von Gleichheit sind eine Folge einer solchen in sich widersprüchlichen Implementation.

Das V-O-Modell führt dazu, dass jeder Typ einen einzigen Vergleich hat. Der Vorteil dieses Modells liegt in seiner konzeptionellen Klarheit und in den Konstellationen, die es *nicht* zulässt. Der einheitliche Vergleich (Gleichheit für Werttypen/ Identität für Objekttypen) hat für alle Typen dieselbe Bedeutung: die Aussage, ob sich zwei Ausdrücke auf dasselbe Element beziehen. Das V-O-Modell schließt einen objekt-artigen Vergleich bei Werten und einen wert-artigen Vergleich bei Objekten aus. Es verhindert auf diese Weise, Wertgleichheit und Objektgleichheit miteinander zu vermengen und vermeidet so etliche der eingangs beschriebenen Schwierigkeiten.

Tabelle 8.6 stellt das V-O-Modell dem (impliziten) Modell von Sprachen wie Java, Scala, C#, etc. gegenüber.

Sprachmodell Java, Scala, C#, ...	V-O-Modell
„Alles ist ein Objekt“	Trenne Werte und Objekte
Gleichheit und Identität sind zwei verschiedene Konzepte	(Wert-)Gleichheit und (Objekt-)Identität sind zu einem Konzept verschmolzen
Gleichheit ist mehrdeutig (Wertgleichheit/ Objektgleichheit)	Trenne Wertgleichheit und Objektgleichheit als grundlegend verschiedene Konzepte
Gleichheit ist eine normale Methode (meist: Exemplarmethode)	Gleichheit ist (wie Identität) ein Sprachprimitiv

Tabelle 8.6: Gegenüberstellung der Sprachmodelle

Das V-O-Modell beseitigt einen großen Teil der Fehlerquellen, die den Umgang mit Gleichheit in objektorientierten Sprachen prägen. Tabelle 8.7 greift die in Abschnitt 8.1 zusammengestellten Schwierigkeiten auf und zeigt, auf welche Weise das V-O-Modell sie vermeidet oder reduziert.

	Schwierigkeit	V-O-Modell
Verantwortlich: Programmierer der Klasse	Der Vertrag verlangt, dass Gleichheit eine Äquivalenzrelation bildet	Sprach-generierte Gleichheit/ Identität bildet immer eine Äquivalenzrelation, bei programmierer-definierter Gleichheit keine Komplikation durch Vererbung.
	Von änderbarem Zustand abhängige Gleichheit	Sprach-generierte Gleichheit/ Identität hängt nicht von änderbarem Zustand ab. Programmierer-definierte Gleichheit gibt es nur bei Wertklassen, dort ist kein änderbarer Zustand vorhanden.
	Ununterscheidbarkeit/ Hashcode-Regel	Sprach-generierte Gleichheit ist immer ununterscheidbar (und erfüllt darum auch die Hashcode-Regel).
Verantwortlich: Klienten	Benutzung eines falschen Vergleichs	Im V-O-Modell nicht möglich, da nur ein Vergleich vorhanden.
	Vergleich inkompatibler Typen	Sprachumgebung erkennt Vergleiche inkompatibler Typen und markiert sie als Fehler.
	Handhabung von null	Sprachumgebung übernimmt korrekte Handhabung von null, gewährleistet Symmetrie.

Tabelle 8.7: Beseitigte Schwierigkeiten

Wie die Tabelle zeigt, sind die potentiellen Fehlerquellen für Klienten vollständig beseitigt. Fallstricke bei der Implementierung kann es, soweit der Vergleich von der Sprachumgebung generiert wird, ebenfalls nicht geben. Für etliche Klassen (Objekt-klassen, Aufzählungs-Wertklassen, strukturierte Wertklassen mit eindeutiger Reprä-sentation) ist somit eine korrekte Implementation der Gleichheit bzw. der Identität garantiert.

Für Wert-Klassen mit einer programmierer-definierten Gleichheit gibt der modi-fizierte Gleichheitsvertrag eine Leitlinie für die korrekte Implementierung. Die Be-dingungen der Äquivalenzrelation und der Ununterscheidbarkeit können i. A. nicht sprachseitig überprüft werden. Verglichen mit der in 8.1 beschriebenen Situation sind aber auch hier etliche Fehlermöglichkeiten ausgeschaltet:

- Gleichheit ist keine Exemplarmethode, sondern ein eigenständiges Sprach-konstrukt. Ihre korrekte Programmierung als Äquivalenzrelation wird nicht durch Wechselwirkungen mit einer Vererbungshierarchie verkompliziert.
- Bei Wertklassen muss sich der Vergleich auf die Datenfelder stützen, er kann nicht auf Referenzen oder technische Adressen zuzugreifen. Das verhindert, dass die Gleichheit verschiedene Speicherrepräsentationen ein- und desselben

Wertes als verschiedene Werte ausweist, was die Ununterscheidbarkeit verletzen würde.

- Weil die Datenfelder in Wertklassen unveränderlich sind, ist sichergestellt, dass die Gleichheit nicht von änderbarem Zustand abhängt.

8.5 Zusammenfassung

Kapitel 8 zeigte, welche Probleme in heutigen objektorientierten Sprachen den Umgang mit Gleichheit und Identität prägen und wie die Unterscheidung von Werten und Objekten dazu beiträgt, diese Probleme zu vermeiden. Das auf der Wert-/Objekt-Trennung beruhende Sprachmodell ermöglichte, mehrdeutige Verwendungen des Begriffs Gleichheit aufzuschlüsseln, die Konzepte Gleichheit und Identität anders einzuordnen und auf dieser Basis Wertgleichheit und Objektidentität zu einem Konzept zu verschmelzen.

Kapitel 8 startete mit einer Bestandsaufnahme. Sie trug eine Reihe von Schwierigkeiten zusammen, mit denen die Implementierung und die Benutzung von Gleichheit in objektorientierten Sprachen einhergehen. Die anschließende Ursachenanalyse ermittelte als wesentlichen Grund für die beschriebenen Probleme die fehlende Trennung von Werten und Objekten, außerdem den Vertrag, der die Anforderungen an Gleichheit festlegt.

Die fehlende Unterstützung von Werttypen führt dazu, dass sie durch Objektklassen abgebildet werden müssen und dabei häufig die Wertgleichheit in Form einer Exemplarmethode wie `equals` implementieren. Werttypen sind dann im Quellcode nicht als solche kenntlich; die (bei Objekttypen sinnvolle) Unterscheidung von Gleichheit und Identität führt bei Werttypen in die Irre. Programmiersprachen unterscheiden Gleichheit bei Werten und Gleichheit bei Objekten nicht klar voneinander. Auch in der Umgangssprache überlagern sich Synonyme und Homonyme, was die Gefahr birgt, verschiedene Konzepte miteinander zu vermengen und zusammengehörende Konzepte gedanklich zu trennen.

Die bisher in der Literatur vorgeschlagenen Verträge für Gleichheit sind nicht restriktiv genug, um inkonsistentes Verhalten von Gleichheit zu unterbinden. Sie legen Gleichheit auch nicht eindeutig fest, ihre Bedingungen werden meist auch von dem kontextabhängigen Konzept der Objektgleichheit erfüllt. Als Gegenentwurf schlug Kapitel 8 einen „modifizierten Gleichheitsvertrag“ vor, der den jeweiligen Vergleich – für Werte die Gleichheit, für Objekte die Identität – eindeutig charakterisiert und Inkonsistenzen verursachende Objektgleichheiten ausschließt.

Ausgehend von der Eindeutigkeit des modifizierten Gleichheitsvertrags und von der Beobachtung, dass sich Gleichheit bei Werttypen und Identität bei Objekttypen gleich verhalten, wurde das Sprachmodell konkretisiert: Es sieht nur einen einzigen

sprachgestützten Vergleich vor – historisch gewachsen bei Objekttypen als Identität, bei Werttypen als Gleichheit bezeichnet.

Wertgleichheit und Objektidentität erfordern aufgrund ihrer spezifischen Eigenschaften eine dedizierte Sprachunterstützung. Kapitel 8 griff die bereits in Kapitel 6 beschriebene sprachliche Gestaltung von Wertgleichheit auf und ordnete sie in das übergreifende Konzept eines einheitlichen Vergleichs ein.

Eine Gegenüberstellung der beiden Sprachmodelle – „Alles ist ein Objekt“ versus „Werte und Objekte sind fundamental verschieden“ – fasste die jeweils aus ihnen resultierenden Konzepte von Gleichheit und Identität zusammen. Sie zeigte, dass auf Basis der Wert-/ Objekttrennung die eingangs beschriebenen Schwierigkeiten größtenteils beseitigt und die restlichen erheblich reduziert werden können.

Kapitel 9

Subtyping und Subclassing bei Werttypen

Vererbung gilt als eine zentrale Säule, teilweise sogar als *das* prägende Merkmal der Objektorientierung. Beispielsweise betrachtet Wegner nur solche Sprachen als objektorientiert, die auch Vererbung unterstützen [Weg87, p. 169], ebenso Meyer [Mey97, p. 26].

Die Vererbung, die von den meisten objektorientierten Sprachen unterstützt wird, verbindet mit Subtyping und Subclassing verschiedene Konzepte und Mechanismen. In dieser Form ist sie auf Objekttypen ausgerichtet. Zu Werttypen dagegen passt sie nicht. Das V-O-Modell ermöglicht Anpassungen, die den Unterschieden zwischen Werttypen und Objekttypen sowie denen zwischen Subtyping und Subclassing Rechnung tragen. Dieses Kapitel zeigt, wie Subtyping – ein Bestandteil von Vererbung – auch bei Werttypen, dort aber mit einem anderen Sprachmechanismus, erreicht werden kann und wie dadurch die Sicherheit der Sprache erhöht wird.

9.1 Bestandsaufnahme: Werte und Vererbung

Von den Arbeiten, die die Verschiedenartigkeit von Werten und Objekten thematisieren (s. dazu Kap. 3: [Mac82], [EK95], [Wer98], [Bäu+98], [BLZ99], [KR99], [Hen00], [Fow03], [Eva04], [Hoo07], [Poe09], [LR09]), befasst sich ein großer Teil nicht oder nur am Rande mit Vererbung. Diejenigen Arbeiten, die zu Vererbung in Zusammenhang mit Werttypen Stellung beziehen, lassen sich in zwei Gruppen einteilen: einige befürworten Vererbung bei Werttypen, andere stufen sie als problematisch ein und plädieren dafür, sie zu unterbinden.

9.1.1 Befürwortung von Vererbung bei Werttypen

Van der Werf behauptet, dass Wertklassen von Wertklassen erben können, Objektklassen von Objektklassen und Objektklassen von Wertklassen. Nur dass eine Wertklasse von einer Objektklasse erbt, schließt er aus. Als Begründung führt er an, dass Objekte einen Zustand haben, Werte dagegen nicht:

„Just as a class of objects can inherit from other classes of objects, so can a class of values inherit from other classes of values. ... There is nothing wrong with an object class inheriting from a class of values. ... A value class cannot inherit from an object class.“[Wer98, p. 28]

Die von Bacon vorgeschlagene Sprache Kava unterstützt Klassen für Objekte, Klassen für Werte und (als Spezialfall davon) Aufzählungen. Vererbung wird für Aufzählungen ausgeschlossen, nicht aber für Wertklassen im allgemeinen – hier wird bei Vorhandensein von Vererbungsbeziehungen lediglich weniger stark optimiert:

„Values may be subclassed, but will generally not be optimized as highly if they are not final.“[Bac03, p. 190]

Die von Vaziri et al. vorgeschlagenen Relation types [VTF07] basieren darauf, zwei Arten von Datenfeldern zu unterscheiden, „key fields“ und „non-key fields“. Relation types, die nur über key fields verfügen, werden in diesem Sprachmodell als Werttypen bezeichnet. Für Relation types definieren Vaziri et al. Vererbung analog zu der Vererbung aus objektorientierten Sprachen wie Java:

„... subtypes may add (but not remove) additional key fields as well as other instance fields and methods. A subtype inherits methods and fields declared in a relation supertype.“[VTFD07, p. 60f]

Das impliziert, dass in diesem Programmiermodell Werttypen nicht von Objekttypen erben können, weil sie von diesen mindestens ein non-key field erhalten würden und dadurch keine Werttypen mehr wären. Alle anderen Vererbungsbeziehungen sind nach diesem Regelwerk zulässig.

Darüber hinaus postulieren Vaziri et al., dass die Elemente einer erbenden Klasse immer von den Elementen ihrer Basisklasse verschieden sind; dabei bezeichnet „ $=_R$ “ den durch die Definition eines Relation type induzierten, von der Sprache generierten Vergleich:

„The problems with defining an equivalence relation in the presence of subclassing are resolved by making relation types and their subtypes incomparable:
 $x =_R y$ yields false if x and y are not of exactly the same type.“[VTFD07, p. 59]

9.1.2 Ablehnung von Vererbung bei Werttypen

Kölling und Rosenberg beschreiben die Schwierigkeiten, die sogenannte „simple types“ im Sprachmodell in Bezug auf Erzeugung und Gleichheit bereiten. Sie verallgemeinern die Eigenschaften derartiger Typen zum Konzept der „Manifest classes“ (s. Abschnitt 5.4.2) und zeigen, dass der Einsatz von Vererbung bei Manifest classes Probleme und Widersprüche verursacht. Daraus ziehen sie die Schlussfolgerung, dass Vererbung und Manifest classes inkompatibel zueinander sind:

„The conclusion of this is: manifest classes and inheritance simply do not go together. It is important to emphasise that this is not a problem of our definition of manifest classes – it is an inherent problem of simple types in general.“[KR99, p. 219]

Henney publizierte mehrere Fachbeiträge über die spezifischen Eigenschaften von Werten und über ihre Abbildung in Java durch geeignete Idiome. Er schließt Vererbung in Zusammenhang mit Werten aus und begründete das mit den durch die Gleichheit aufgeworfenen Problemen:

„Object’s equals contract effectively forbids handling equality across different types ... Values should be non-subclassing and concrete.“[Hen02, p. 9]

„In common across most languages, values do not generally live in class hierarchies.“[Hen03a, p. 39]

9.2 Unterscheidung von Subtyping und Subclassing

Die eben genannten Arbeiten berücksichtigen nicht, dass in den meisten objekt-orientierten Programmiersprachen, u. a. in Java, C#, Scala, Eiffel oder Smalltalk, Vererbung zwei fundamental verschiedene Konstrukte koppelt, nämlich *Subtyping* und *Subclassing* (s. dazu u. a. Cook et al. [CHC90], Evered et al. [EKSM97] und Schmoltzky [Sch99]).

Beispielsweise hat in Java die Anweisung

```
class B extends A ...
```

zwei Auswirkungen:

- Die Klasse B übernimmt die Implementation von A, sie wird zu einer *Subklasse* von A.
- Elemente von B können an Stelle von Elementen von A verwendet werden, der Typ B wird zu einem *Subtyp* von A.

Aufgrund dieser engen Verzahnung von Subtyping und Subclassing ist das eine nicht ohne das andere programmierbar, und vielen Entwicklern ist nicht bewusst, dass es sich um verschiedene Beziehungen zwischen zwei Klassen oder zwei Typen handelt. Eine programmiersprachliche Trennung dieser beiden Konzepte ist prinzipiell möglich (s. Halbert/ O'Brien [HO87]) und aus softwaretechnischer Sicht vorteilhaft (s. Schmoltzky [Sch99]), wird aber nur in wenigen Sprachen konsequent unterstützt (z. B. in Sather [SOM94] oder in Theta [Lis+95]).

Weil die in Abschnitt 9.1 genannten Arbeiten Subtyping und Subclassing nicht unterscheiden, kann nicht beurteilt werden, ob sich ihr jeweiliges Befürworten oder Ablehnen von Vererbung bei Werttypen auf Subtyping, auf Subclassing oder auf beides bezieht.

9.2.1 Subtyping

Subtyping – auch bezeichnet als Subtyp-Polymorphie oder Inklusionspolymorphie (s. Cardelli und Wegner [CW85, p. 475]) – ist eine *semantische* Beziehung zwischen zwei Typen. Sie ist gekennzeichnet durch *Teilmengen-Semantik* – die Elemente des Subtyps S bilden eine Teilmenge der Elemente des Supertyps T – und durch das *Prinzip der Ersetzbarkeit* – die Elemente des Subtyps S können überall dort verwendet werden, wo ein Element des Supertyps T erwartet wird. Diese Auffassung von Subtyping ist u. a. zu finden bei LaLonde und Pugh [LP91], bei Simons [Sim02b], oder bei Pierce [Pie02]:

subset semantics: „... every value described by S is also described by T, that is, the elements of S are a subset of the elements of T.“

principle of safe substitution: „... any term of type S can safely be used in a context where a term of type T is expected.“ [Pie02, p. 182]

Im Folgenden wird unter Subtyping die beschriebene Teilmengen- und Ersetzbarkeitsbeziehung zwischen zwei Typen verstanden, unabhängig davon, ob es sich um Werttypen oder um Objekttypen handelt und unabhängig davon, wodurch sie zustande kommt.

Die „sichere Benutzung eines Ausdrucks vom Typ S“ beinhaltet u. a., dass die Ersetzung eines Ausdrucks vom Typ T durch einen Ausdruck vom Typ S *typsicher* sein muss. Jede für T verfügbare Operation muss auch für S verfügbar sein, jede Operation mit einem Parameter vom Typ T muss auch einen Parameter vom Typ S akzeptieren.

Subtyping ermöglicht Wiederverwendung auf Klientenseite: Klienten von T, d. h. der Code, der die Operationen von T aufruft, sind ohne Anpassung auch für alle Subtypen von T verwendbar. Das trägt dazu bei, Code-Redundanzen zu vermeiden.

9.2.2 Subclassing

Subclassing, auch bezeichnet als Implementierungsvererbung, ist ein Sprachmechanismus, mit dem eine Klasse aufbauend auf einer anderen Klasse implementiert werden kann.

Die erbende Klasse – Subklasse oder abgeleitete Klasse genannt – übernimmt alle Member der zugrunde liegenden Klasse – auch als Basisklasse oder Superklasse bezeichnet. Die Subklasse kann zusätzliche Membervariablen und Methoden definieren, und sie kann von der Basisklasse geerbte Methoden überschreiben, d. h. mit einer eigenen Implementation versehen. S. dazu Bruce und Pierce:

„A subclass may be defined from a class by either adding to or modifying the methods and instance variables of the original class.“[Bru02, p. 23]

„Most object-oriented languages achieve .. reuse of behaviors via ... a mechanism of subclassing that allows new classes to be derived from old ones by adding implementations for new methods and, when necessary, selectively overriding implementations of old methods.“[Pie02, p. 227]

Mittels Subclassing können Klassen inkrementell aufeinander aufbauend codiert werden. Sie bilden dann eine Hierarchie (bei Einfach-Vererbung eine Baumstruktur, bei Mehrfach-Vererbung einen gerichteten, azyklischen Graphen). Der Code für eine Methode kann in derjenigen Klasse angesiedelt werden, in der er auf sie und alle ihre Unterklassen Anwendung findet. Das trägt dazu bei, Code-Verdopplung zu vermeiden.

Subclassing ist ebenfalls eine Form von Code-Wiederverwendung, nämlich auf Implementierungsebene. Die durch Subclassing bewirkte Wiederverwendung unterscheidet sich von der auf Subtyping beruhenden Wiederverwendung: Beim Subclassing wird der Code der Basisklasse „wiederverwendet“, beim Subtyping dagegen der Klienten-Code der Basisklasse.

Subclassing ist je nach Programmiersprache unterschiedlich ausgestaltet. Es gibt verschiedene Zugriffsmodalitäten für Klienten und für Erben, sie werden durch verschiedene Zugriffsmodifier zum Ausdruck gebracht (z. B. in Java u. a. `public`, `protected`, `private` [Gos+11, Kap. 6.6]). Einige Sprachen erlauben Signaturanpassungen, andere nicht oder nur teilweise (z. B. Java nur für den Rückgabetypp [Gos+11, Kap. 8.4.5]). Eiffel erlaubt, Methoden in Subklassen mit anderen Namen zu versehen („feature renaming“ [Mey92, p. 81]). Für den hier im Mittelpunkt stehenden Zusammenhang zwischen Werttypen und Vererbung spielen diese Unterschiede keine Rolle.

9.2.3 Subclassing als ein Subtyping-Mechanismus

Eine Subtypbeziehung zwischen einem Typ *S* und einem Typ *T* beinhaltet die Aussage, *dass* die Elemente von *T* durch die von *S* ersetzbar sind, nicht aber, *wie* diese Ersetzbarkeit zustande kommt. In vielen „Mainstream-Programmiersprachen“ sind nominelle Typsysteme vorherrschend (s. Pierce [Pie02, p. 253]) und damit nominelles Subtyping. Das trifft insbesondere auf die hier betrachteten objektorientierten Sprachen wie Java, Eiffel, C++ oder C# zu. Damit gilt ein Typ *S* dann als Subtyp eines Typs *T*, wenn er als solcher *deklariert* wurde. Strukturelles Subtyping dagegen akzeptiert einen Typ *S* dann als Subtyp von *T*, wenn seine Struktur ihn als solchen kennzeichnet, d. h. wenn *S* alle Operationen von *T* mit der passenden Signatur besitzt. Nominelles Subtyping ist sicherer als strukturelles Subtyping und darum aus softwaretechnischer Sicht vorzuziehen: Es vermeidet ungewollte Subtypbeziehungen, die durch zufällige Namensgleichheiten zustande kommen („spurious subsumption“, s. Pierce [Pie02, p. 253]), ebenso das Nicht-Zustandekommen einer Subtypbeziehung aufgrund einer falschen Benennung oder einer vergessenen Methode. Im Folgenden wird darum von nominellem Subtyping ausgegangen.

Eine Subtyp-Beziehung zwischen zwei Typen *S* und *T* kann nicht *allein* mittels Deklaration hergestellt werden. Es ist ein Mechanismus erforderlich, der die Ersetzbarkeit gewährleistet, d. h. der sicherstellt, dass alle für *T* verfügbaren Operationen auch für *S* aufrufbar sind.

Subclassing ist ein solcher Mechanismus. Weil die Methoden und Datenfelder einer Klasse *A* auch in einer davon erbenden Klasse *B* vorhanden und für Klienten verfügbar sind – so als wären sie in *B* definiert – bildet der Typ *B* einen Subtyp von Typ *A*. Zahlreiche Beispiele aus Standardbibliotheken (z. B. `java.awt.Container` und `java.awt.Window`) oder Anwendungssystemen (z. B. `Person` und `Student`) illustrieren diesen Zusammenhang.

Eine Ausnahme bildet die `private` und die geschützte Vererbung, wie sie von C++ unterstützt wird [Str10, p. 742]. Private Vererbung bewirkt, dass die von der Basisklasse geerbten Methoden und Datenfelder nur innerhalb der Implementation der erbenden Klasse aufrufbar sind, im Fall von geschützter Vererbung auch in deren Subklassen; Klienten aber können nicht auf sie zugreifen. Private und geschützte Vererbung ziehen i. A. keine Ersetzbarkeit nach sich und führen damit *nicht* zur Herstellung einer Subtypbeziehung.

Im Folgenden wird unter Subclassing die uneingeschränkte Code-Übernahme einer Subklasse von einer Basisklasse verstanden, wie sie z. B. von Java, C# oder Eiffel unterstützt wird. Einschränkungen in Bezug auf die geerbten Member, wie bei privater oder geschützter Vererbung, sind im weiteren Verlauf der Arbeit von der Betrachtung ausgeklammert.

9.2.4 Subclassing bei Wertklassen

Bei den eben aufgeführten Beispielen (Container und Window, Person und Student) handelt es sich um Objekttypen, und hier zog das Subclassing eine Subtypbeziehung nach sich. Bei Wertklassen weist Subclassing einige Besonderheiten auf, die nachfolgend diskutiert werden.

Um die Wirkungsweise von Subclassing bei Wertklassen näher zu untersuchen, sind zwei Fälle zu unterscheiden: Im ersten Fall fügt die Subklasse der Basisklasse keine Datenfelder hinzu, sondern ergänzt oder überschreibt ausschließlich Operationen. Im zweiten Fall fügt die Subklasse der Basisklasse nicht nur Operationen hinzu, sondern auch ein oder mehrere Datenfelder.

Im ersten Fall repräsentieren die Basisklasse und die Subklasse dieselbe Wertemenge. Die Subklasse bildet dann eine (unechte) Teilmenge und einen Subtyp der Basisklasse. Um einer Klasse lediglich Operationen hinzuzufügen, erscheint die Bildung einer Subklasse und damit einer weiteren Klasse ein unangemessen hoher Aufwand. Auch können die zusätzlichen Operationen dann nur über einen Ausdruck der Subklasse aufgerufen werden. Hat ein Ausdruck, für den diese Operationen aufgerufen werden sollen, den Typ der Basisklasse, so wird weiterer Code für die notwendigen Konvertierungen erforderlich. Es gibt andere Sprachmittel, die ausschließlich dem Zweck dienen, einer Klasse weitere Operationen hinzuzufügen. Sie bringen diese Intention dadurch im Code klarer zum Ausdruck, und sie erfordern auch nicht die beschriebenen Konvertierungen. Dazu zählen z. B. *extension methods*, wie sie von C# unterstützt werden [Mic10, Kap. 10.6.9], oder *categories* in Objective-C [Koc11, Kap. 11]. Falls eine Sprache derartige Mechanismen nicht unterstützt, können die zusätzlichen Operationen in einer Hilfsklasse programmiert werden, wie sie z. B. `java.lang.Math` darstellt. Der erste Fall – eine Subklasse, die die Basisklasse um Operationen, nicht aber um Datenfelder erweitert – wird im Folgenden nicht weiter betrachtet.

Im zweiten Fall, d. h. wenn die Subklasse auch Datenfelder hinzufügt, bildet sie keinen Subtyp der Basisklasse, und ein Teil der geerbten Operationen kann Schwierigkeiten aufwerfen. Die Gründe dafür werden nachfolgend analysiert und anhand eines Code-Beispiels illustriert.

Das für diesen Zweck gewählte Beispiel ist in Java programmiert. Die aus dem Subclassing resultierenden Konstellationen sind unabhängig von der verwendeten Sprache und auch anhand von Java-Code nachvollziehbar.¹

¹Für den in Kap. 6 eingeführten Typkonstruktor `valueClass` verzichtet diese Arbeit – aus den im Folgenden erläuterten Gründen – auf die Einführung von Subclassing.

Als Fallbeispiel für Subclassing bei Wertklassen dienen zweidimensionale und dreidimensionale Punkte im euklidischen Raum, abgebildet durch die Klassen `Point2D` (Seite 195) und `Point3D` (Seite 196).

`Point3D` ist als Subklasse von `Point2D` programmiert und erbt somit deren Implementation. Beide Klassen verfügen u. a. über Operationen, die zwei Punkte der jeweiligen Klasse aufeinander beziehen, die z. B. den Abstand (`distance`) oder den Mittelpunkt (`middle`) zwischen zwei Punkten berechnen, außerdem über diverse Operationen für euklidische Transformationen, z. B. Strecken/ Stauchen (`stretch`) oder Punktspiegelung (`opposite`). Die Gleichheit (`equals`) ist in beiden Klassen als kanonische Gleichheit (d. h. als Vergleich aller Datenfelder, s. Abschnitt 6.3.5) implementiert.

Kein Subtyping

Im Folgenden wird nachgewiesen, dass die erbende Wertklasse keinen Subtyp ihrer Basisklasse bildet, und dass die Gleichheit zusammen mit dem Vorhandensein zusätzlicher Datenfelder ein wesentlicher Grund dafür ist.

Eine Implementierung von Gleichheit, die Vergleiche zwischen Basisklasse und Subklasse erlauben würde, scheitert an dem von Bloch beschriebenen Problem (s. dazu [Blo08, Item 9, p. 30] und Abschnitt 8.1.1/ Anhang C.2):

Es ist nicht möglich, Gleichheit so zu implementieren, dass

1. sie eine Äquivalenzrelation bildet *und*
2. Vergleiche zwischen Elementen der Basisklasse und der Subklasse möglich sind und `true` ergeben *und*
3. Elemente der Subklasse anhand der zusätzlichen Datenfelder als unterschiedlich erkannt werden.

Ein Verzicht auf Bedingung 1. würde gegen den Gleichheitsvertrag (s. Abschnitte 8.1.1 und 8.3.1) verstoßen und ist damit ausgeschlossen. Ein Verzicht auf Bedingung 3. hätte zur Folge, dass die zusätzlichen Datenfelder keinerlei Wirkung entfalten; damit würde de facto Fall 1 vorliegen, im Widerspruch zur Voraussetzung. Es muss also auf Bedingung 2. verzichtet werden; und damit haben die Wertemenge der Subklasse und die der Basisklasse keine gemeinsamen Elemente. Deshalb liegt keine Teilmengen- und infolgedessen auch keine Subtypbeziehung vor.

Im vorliegenden Fallbeispiel ist auch anschaulich klar, dass `Point3D` keinen Subtyp von `Point2D` bildet: Die Menge der dreidimensionalen Punkte ist keine Teilmenge der Menge der zweidimensionalen Punkte. Auch weitere Operationen des Beispiels zeigen, dass das Ersetzbarkeitsprinzip verletzt ist. Z. B. ist die Operation `distanceTo` nicht auf ein Element von `Point2D` und ein Element von `Point3D`

Code-Beispiel: Point2D

```
class Point2D {
    float x;
    float y;

    public static Point2D pointCartesian(float x, float y) {
        return new Point2D(x, y);
    }

    public static Point2D zeroPoint() {
        return pointCartesian(0.0, 0.0);
    }

    public boolean equals(Object other) {
        ...
        Point2D p = (Point2D)other;
        return (x == p.x && y == p.y);
    }

    public float getX() {
        return x;
    }

    public float getY() {
        return y;
    }

    public float distanceTo(Point2D p2) {
        return sqrt(pow(x-p2.x,2) + pow(y-p2.y,2));
    }

    public float distanceToZeroPoint() {
        return distanceTo(zeroPoint());
    }

    public Point2D opposite() {
        return pointCartesian(-x, -y);
    }

    public Point2D middle(Point2D p2) {
        return pointCartesian((x+p2.x)/2, (y+p2.y)/2);
    }

    public Point2D stretch(float factor) {
        return pointCartesian(x*factor, y*factor);
    }
    ...
}
```

Code-Beispiel: Point3D

```
class Point3D extends Point2D {
    // x and y are inherited from Point2D
    float z;

    public static Point3D pointCartesian(float x,
                                          float y,
                                          float z) {

        return new Point3D(x, y, z);
    }

    public static Point3D zeroPoint() {
        return pointCartesian(0.0, 0.0, 0.0);
    }

    public boolean equals(Object other) {
        ...
        Point3D p = (Point3D)other;
        return (x == p.x && y == p.y && z == p.z);
    }

    // getX and getY are inherited from Point2D
    public float getZ() {
        return z;
    }

    public float distanceTo(Point3D p2) {
        return
            sqrt(pow(x-p2.x,2) + pow(y-p2.y,2) + pow(z-p2.z,2));
    }

    public float distanceToZeroPoint() {
        return distanceTo(zeroPoint());
    }

    public Point3D opposite() {
        return pointCartesian(-x, -y, -z);
    }

    public Point3D middle(Point3D p2) {
        return
            pointCartesian((x+p2.x)/2, (y+p2.y)/2, (z+p2.z)/2);
    }

    public Point3D stretch(float factor) {
        return pointCartesian(x*factor, y*factor, z*factor);
    }
    ...
}
```


anwendbar; die Entfernung zwischen einem zweidimensionalen und einem dreidimensionalen Punkt ergibt fachlich keinen Sinn.²

Wiederverwendbarkeit des geerbten Codes

Subclassing dient in erster Linie der Wiederverwendung von Code. Bei Wertklassen ist die von einer Basisklasse *T* geerbte Implementation einer Operation nur dann in einer Subklasse *S* brauchbar, wenn das Verhalten dieser Operation in der Subklasse *S* ohne Zugriff auf die von *S* zusätzlich definierten Datenfelder auskommt.

Im Fallbeispiel aber müssen die meisten Operationen – `pointCartesian`, `zeroPoint`, `opposite`, `stretch`, `middle`, `distanceTo` – in der Klasse `Point3D` auch das zusätzliche Datenfeld, die *z*-Koordinate, einbeziehen. Nur `getX` und `getY` können die von `Point2D` geerbte Implementation unverändert übernehmen. Deren Implementationen enthalten nur eine einzige Zeile. Der aus ihrer Wiederverwendung resultierende Nutzen ist damit im Fallbeispiel eher marginal.

Im allgemeinen Fall – in dem über die Implementation keine detaillierte Aussage getroffen werden kann – liefern die Operationssignaturen Hinweise darauf, ob der Code der Basisklasse in der Subklasse verwendbar ist. Um das systematisch zu untersuchen, werden die Operationen einer Wertklasse *T* in vier Gruppen eingeteilt:

1. Operationen der *Gruppe 1* haben ein Ergebnis, aber keinen Parameter vom Typ *T*. Ihre Signaturen lauten $X \rightarrow T$, $X \times Y \rightarrow T$, etc.

Diese Operationen (es handelt sich konzeptionell um Selektoren) haben weder einen Parameter noch ein Ergebnis vom Typ *S* und sind deshalb in der Subklasse *S* nicht brauchbar. Ein Selektor der Klasse *S* muss in seiner Implementation alle Datenfelder der Klasse *S* ansprechen, und damit auch diejenigen, die *S* nicht von *T* erbt. (Im Fallbeispiel `Point2D`/`Point3D` illustrieren das die Operationen `pointCartesian` und `zeroPoint`.) Die Klasse *S* kann die Selektoren von *T* nicht übernehmen, sie muss alle ihre Selektoren selbst definieren und implementieren.

2. Operationen der *Gruppe 2* haben sowohl ein Ergebnis, als auch (mindestens) einen Parameter vom Typ *T*. Ihre Signaturen lauten $T \rightarrow T$, $T \times X \rightarrow T$, $T \times T \rightarrow T$, etc.

²Unter der Annahme, dass die Ebene der zweidimensionalen Punkte in den Raum der dreidimensionalen Punkte eingebettet wird, könnte die Entfernung zwischen einem zweidimensionalen und einem dreidimensionalen Punkt ermittelt werden. Die Menge der zweidimensionalen Punkte würde durch eine solche Einbettung zu einer Teilmenge der dreidimensionalen Punkte, und alle Operationen von `Point3D` wären dann auf die Elemente von `Point2D` anwendbar.

Doch auch mit einer solchen zusätzlichen Annahme illustriert das Fallbeispiel die Behauptung, dass Subclassing bei Wertklassen *nicht* zu Subtyping führt. Auf Basis der obigen Vererbungsbeziehung müsste `Point3D` einen Subtyp von `Point2D` bilden. Mit einer Einbettung aber verläuft die Subtypbeziehung in der entgegengesetzten Richtung.

In der Subklasse *S* erfordern diese Operationen in der Regel an *S* angepasste Signaturen, insbesondere den Ergebnistyp *S*. Weil die in der Klasse *T* definierte Implementation ein Element von *T* liefert, ist sie in der Subklasse *S* nicht brauchbar. (Im Fallbeispiel `Point2D/Point3D` gehören Operationen wie `opposite`, `stretch` oder `middle` zu dieser Gruppe.)

3. Operationen der *Gruppe 3* haben genau einen Parameter, aber kein Ergebnis vom Typ *T*. Ihre Signaturen lauten $T \rightarrow X$, $T \times X \rightarrow Y$, etc.

Operationen der Gruppe 3 können im Prinzip, d. h. auf Basis ihrer Signatur, in einer Subklasse wiederverwendet werden. Das ist aber nur eine notwendige, keine hinreichende Bedingung. Auch Operationen dieser Gruppe können in der Subklasse *S* einen Zugriff auf eines der zusätzlichen Datenfelder erfordern, wodurch der von *T* geerbte Code in *S* nicht mehr brauchbar ist. (Im Fallbeispiel `Point2D/Point3D` trifft das auf `distanceToZeropoint` zu.)

4. Operationen der *Gruppe 4* haben zwei (oder mehr) Parameter, aber kein Ergebnis vom Typ *T*. Ihre Signaturen lauten $T \times T \rightarrow X$, $T \times T \times T \rightarrow X$, $T \times T \times X \rightarrow Y$, etc.

Wenn die Parameter einer solchen Operation in der Subklasse auf den Typ *S* angepasst werden müssen, dann ist die aus *T* stammende Implementation bereits aufgrund ihrer Signatur nicht in *S* verwendbar. Die Operation entspricht dann einer binären Methode. (Im Fallbeispiel `Point2D/Point3D` ist `distanceTo` eine solche Operation.)

Operationen mit zwei oder mehr Parametern vom Typ *T* kann es auch in Gruppe 2 geben – wo die Wiederverwendbarkeit ihrer Implementation bereits aufgrund des Ergebnistyps in Frage gestellt ist. Abschnitt 9.4.2 beschreibt weitere mit binären Methoden einhergehende Schwierigkeiten.

Im Ergebnis sind die Operationen der Gruppen 1, 2 und 4 nicht in der Subklasse brauchbar, die der Gruppe 3 mit Einschränkung. Der Umfang der aus dem Subclassing resultierenden Codewiederverwendung ist damit, abhängig von den Operationssignaturen, in vielen Fällen gering.

Fehleranfälligkeit

Wie eben gezeigt wurde, sind mehrere Gruppen von Operationen bereits aufgrund ihrer Signaturen in der Subklasse nicht brauchbar. Weil beim Subclassing aber per Definition alle Operationen der Basisklasse geerbt werden, sind diese Operationen trotzdem in der Subklasse vorhanden. Damit bergen sie dort ein erhebliches Fehlerpotential.

Z. B. kann im Fallbeispiel die Methode `distanceTo` für ein Element vom Typ `Point3D` und mit einem Parameter vom Typ `Point2D` aufgerufen werden, ob-

wohl die Entfernung zwischen einem zweidimensionalen und einem dreidimensionalen Punkt fachlich keinen Sinn ergibt und darum einen Programmierfehler darstellt.

Gesamtbeurteilung

Subclassing ist auch bei Wertklassen prinzipiell möglich. Es führt dort aber (außer in einfachen Fällen, in denen die Subklasse die Basisklasse nur um Operationen erweitert) nicht zu Subtyping. Damit entfällt die Code-Wiederverwendung auf Klientenseite.

Die Wiederverwendung der Implementation ist nur für eine eingeschränkte Gruppe von Operationen möglich (bei Operationen, deren Implementation sich auch in der Subklasse nur auf die von der Basisklasse geerbten Datenfelder abstützt, was insbesondere bei einem Ergebnistyp der eigenen Klasse nicht möglich ist).

Nähere Untersuchungen und weitere praktische Beispiele müssen zeigen, ob bei Wertklassen die aus Subclassing resultierende Codewiederverwendung nützlich genug ist, um die Nachteile (Fehleranfälligkeit aufgrund der geerbten, aber nicht brauchbaren Operationen) aufzuwiegen. Der Fokus der Arbeit liegt im weiteren auf Subtyping bei Wertklassen.

In diesem Abschnitt wurde Subclassing zwischen Wertklassen betrachtet, in Abschnitt 9.2.3 Subclassing zwischen Objektklassen. Der Vollständigkeit halber soll erwähnt werden, dass es *gemischte* Subklassenbeziehungen – zwischen einer Wertklasse und einer Objektklasse oder umgekehrt – nicht geben kann:

Eine Objektklasse muss zwecks Initialisierung den Konstruktor der Basisklasse aufrufen. Das ist nicht möglich, wenn es sich bei dieser um eine Wertklasse handelt, weil diese nicht über Konstruktoren, sondern über Selektoren verfügt.

Würde umgekehrt eine Wertklasse von einer Objektklasse erben, würde sie deren (i. A. änderbare) Datenfelder und deren (i. A. referentiell opake und seiteneffekt-behaftete) Operationen übernehmen, was im Widerspruch zu den charakteristischen Eigenschaften von Werttypen stünde.

9.2.5 Subtyp-Beziehungen bei Werttypen

Subtyp-Beziehungen gibt es auch zwischen Werttypen. Z. B. können Ganzzahlen als eine Teilmenge der Gleitkommazahlen angesehen werden. Es ist sinnvoll, gemischte Ausdrücke wie „3+4.25“ oder „1.7*2“ zuzulassen, ohne dass der Programmierer explizite Konvertierungen von Ganzzahlen zu Gleitkommazahlen vornehmen muss. In ähnlicher Weise können Gleitkommazahlen in allen Kontexten verwendet werden, in denen eine komplexe Zahl zulässig ist.

Bei vordefinierten Typen erfolgt in solchen Fällen eine implizite Typumwandlung. Beispielsweise erlaubt Java, einen Ausdruck vom Typ `int` dort zu verwenden, wo ein

`float` erwartet wird. Zwischen beiden gibt es eine „widening primitive conversion“ [Gos+11, Kap. 5.1.2]. Auch andere Sprachen kennen solche impliziten Typumwandlungen. In C und C++ werden sie als *coercion* bezeichnet, C# verwendet wie Java dafür den Begriff *conversion*. Welche Umwandlungen zwischen welchen vordefinierten Typen möglich sind, ist in der Sprachspezifikation vorgegeben.

Einige Programmiersprachen unterstützen programmierer-definierte Typumwandlungen. Beispielsweise sind in Scala im Objekt `scala.PreDef` implizite Umwandlungen zwischen etlichen `AnyVal`-Typen definiert [Ode14, Kap. 6.26.1, 12.5.1], z. B. zwischen `Short` und `Int`, oder zwischen `Float` und `Double`. Sie haben die gleiche Wirkung wie widening conversions in Java.

Etliche Arbeiten weisen darauf hin, dass Subtyping nicht notwendigerweise Vererbung und dynamisches Binden voraussetzt, sondern dass Subtyping auch durch Typumwandlung („coercion“) erreicht werden kann, z. B. Pierce [Pie02, Kap. 15.6], Luo [Luo99], Kollár et al. [KPRS09] und Simons:

„A simple approach to interface satisfaction is subtyping. This is where an object of one type may safely be substituted where another type was expected ... This involves no more than coercing the supplied subtype object to a supertype and executing the supertype’s functions. The coerced object then behaves in exactly the same way as expected. ... Note that no dynamic binding is implied or required.“ [Sim02a, p. 57]

Die vier eben genannten Publikationen stammen aus dem funktionalen Umfeld. Das kann als ein weiterer Hinweis dafür angesehen werden, dass Typumwandlung bei funktionalen Abstraktionen (Werten) ein geeigneter Subtyping-Mechanismus ist.

In diesem Abschnitt standen Subtyp-Beziehungen zwischen Werttypen im Fokus, in Abschnitt 9.2.3 waren es Subtyp-Beziehungen zwischen Objekttypen. Der Vollständigkeit halber soll auch hier erwähnt werden, dass es keine gemischten Subtyp-Beziehungen, d. h. zwischen Werttypen und Objekttypen geben kann:

Ein Objekttyp als Subtyp eines Werttyps könnte, weil Objekte erzeugt werden, indirekt die Wertemenge seines Supertyps vergrößern. Auch könnte er im Supertyp vorhandene Operationen so redefinieren, dass deren Seiteneffektfreiheit oder referentielle Transparenz nicht mehr erfüllt wäre.

Umgekehrt haben die Operationen in Objekttypen in der Regel eine andere Signatur als in Werttypen (Objekttypen als Parameter oder `void` als Rückgabetyp), sie können deshalb i. A. keine Werttypen als Subtypen haben.

Tabelle 9.1 und Tabelle 9.2 fassen die Kombinationsmöglichkeiten und Wechselwirkungen von Werttypen/ Objekttypen und Wertklassen/ Objektklassen in Bezug auf Subclassing und Subtyping in Matrixform zusammen.

	Wertklasse	Objektklasse
Wertklasse	möglich, (kein Subtyping)	nein
Objektklasse	nein	ja (führt zu Subtyping)

Tabelle 9.1: Subklassenbeziehungen zwischen Wertklassen und Objektklassen

	Werttyp	Objekttyp
Werttyp	ja (nicht durch Subclassing)	nein
Objekttyp	nein	ja (mittels Subclassing)

Tabelle 9.2: Subtypbeziehungen zwischen Werttypen und Objekttypen

9.3 Subtyping-Mechanismus für Wertklassen: Inklusionsfunktion

Diese Arbeit schlägt vor, die Deklaration und Herstellung einer Subtyp-Beziehung zwischen zwei Wertklassen durch einen eigenen Sprachmechanismus zu unterstützen: eine Inklusionsfunktion.

9.3.1 Definition und Wirkungsweise der Inklusionsfunktion

Die Inklusionsfunktion legt fest, welcher Subtyp *S* in welchen Supertyp *T* „inkludiert“ wird; und sie gibt an, welches Element des Subtyps welchem Element des Supertyps entspricht.

Eine Inklusionsfunktion wird vom Anwendungsprogrammierer definiert. Die Sprache stellt dafür ein eigenes Schlüsselwort („include“) bereit. Hat z. B. eine Wertklasse `Complex` einen Selektor namens `getComplex`, der eine komplexe Zahl basierend auf ihrem Real- und Imaginärteil vom Typ `Float` liefert, dann kann eine Subtypbeziehung zwischen `Float` und `Complex` definiert werden durch:

```
Complex include(Float x) { return getComplex(x, 0.0); }
```

Die Inklusionsfunktion wird von der Sprachumgebung aufgerufen, implizit und bedarfsgesteuert, d. h. dann, wenn ein Ausdruck des Subtyps vorhanden ist, aber ein Ausdruck des Supertyps erwartet wird. Explizite Aufrufe der Inklusionsfunktion durch den Programmierer sind nicht vorgesehen. Beispielsweise wird der Code

```
Float f1 := 1.5;
Float f2 := 2.6;
Complex c0 := getComplex(3.7, 4.8);
...
Complex c1 := f1;
Complex c2 := add(c1, f2);
```

intern umgewandelt zu

```
...
// kein gültiger Code!
// veranschaulicht die Wirkung der Inklusionsfunktion
Complex c1 := include(f1);
Complex c2 := add(c1, include(f2));
```

Durch die Inklusionsfunktion können alle für den Supertyp (`Complex`) definierten Operationen auch für den Subtyp (`Float`) aufgerufen werden. Damit bewirkt eine Inklusionsfunktion zwischen zwei Typen *S* und *T* nominelles Subtyping:

- Sie macht die Elemente von *S* zu einer Teilmenge der Elemente von *T*.
- Sie sorgt dafür, dass jeder Ausdruck vom Typ *T* durch einen Ausdruck vom Typ *S* ersetzt werden kann.
- Sie kommt deklarativ zustande – durch Definition der Inklusionsfunktion – nicht durch die Struktur von Sub- und Supertyp.

Gemäß dem programmiersprachlichen Prinzip, dass eine Zuweisung `a:=b` eine anschließende Gleichheit von *a* und *b* implizieren sollte („equality after assignment“, s. dazu Meyer [Mey97, p. 264], Kölling und Rosenberg [KR99, p. 215]), gilt aufgrund der Funktionsweise der Inklusionsfunktion für alle Ausdrücke *s* vom Subtyp *S*:

```
s = include(s)
```

Es ist zu beachten, dass die Inklusionsfunktion nicht ein Element des Subtyps in ein Element des Supertyps umwandelt, sondern eine *Repräsentation* eines Wertes in eine andere Repräsentation *desselben Wertes*. Mit einer Inklusionsfunktion kann ein Wert mehrere Repräsentationen *mit unterschiedlichen Typen* haben. Ist z. B. zwischen einer Wertklasse `Float` und einer Wertklasse `Complex` eine geeignete Inklusionsfunktion definiert, dann sind die durch „1.5“ und „getComplex(1.5,0.0)“ bezeichneten Elemente als derselbe Wert aufzufassen.

Die hier gewählte Bezeichnung „Inklusionsfunktion“ dient dazu, den Zweck und die Wirkung dieses Sprachmechanismus deutlich zum Ausdruck zu bringen. Eine Bezeichnung wie „Konvertierungsfunktion“ oder „benutzer-definierte coercion“ dagegen würde die zugrunde liegende technische Implementierung in den Vordergrund rücken. Sie würde dadurch verwirren, denn auf konzeptioneller Ebene findet keine „Konvertierung“ statt: es gibt weder eine Veränderung des inkludierten Elementes noch eine Ersetzung dieses Elementes durch ein anderes.

Der Anhang führt weitere Code-Beispiele auf. Anhang D.1 zeigt, dass mit einer Inklusionsfunktion `Character` als Subtyp von `String` programmiert werden kann. Anhang D.3 stellt mit einer Inklusionsfunktion eine Subtypbeziehung zwischen zwei Aufzählungswertklassen, `Workday` und `Weekday`, her.

9.3.2 Regeln für die Inklusionsfunktion

Um eine Subtypbeziehung zu etablieren, muss eine Inklusionsfunktion zwei Regeln erfüllen, Injektivität und Verhaltenskonformität.

Injektivität

Damit die Inklusionsfunktion eine Teilmengenbeziehung zwischen dem Subtyp und dem Supertyp herstellt, muss sie verschiedene Elemente des Subtyps auf verschiedene Elemente des Supertyps abbilden. D. h. sie muss *injektiv* sein.

Verhaltenskonformität

Subtyp und Supertyp können über gleichnamige Operationen verfügen, oder über Operationen, die über dasselbe Operatorsymbol aufgerufen werden. Z. B. ist die Multiplikation sowohl für Ganzzahlen als auch für Gleitkommazahlen definiert, in der Regel mit demselben Operatorsymbol (Infixoperator „`*`“). Solche Operationen müssen im Subtyp und im Supertyp das *gleiche Verhalten* aufweisen. D. h. das Ergebnis der Multiplikation muss unabhängig davon sein, ob erst zwei Ganzzahlen multipliziert werden und das Ergebnis als Gleitkommazahl weiter verarbeitet wird, oder ob die Eingaben als Gleitkommazahlen interpretiert und diese multipliziert werden – *die Multiplikation und die Inklusionsfunktion müssen vertauschbar sein*. Für alle Ganzzahlen `i1` und `i2` muss gelten:

```
include(multiply(i1,i2)) = multiply(include(i1),include(i2))
```

Diese Vertauschbarkeit gilt analog für Operationen mit anderen Signaturen. Die Aufstellung auf Seite 204 beschreibt das Regelwerk für Operationen mit ein und mit zwei Parametern. (Das obige Beispiel – Multiplikation bei Ganzzahlen und Gleitkommazahlen – entspricht in dieser Aufstellung strukturell der Operation `g2`.)

Injektivität ist ein Spezialfall von Verhaltenskonformität. (Um das nachzuweisen, genügt es, in der Aufstellung auf Seite 204 für die Funktion `f2` die Gleichheit einzusetzen.)

Injektivität und Verhaltenskonformität können i. A. nicht durch die Sprachumgebung sichergestellt oder überprüft werden. Sie liegen in der Verantwortung des Anwendungsprogrammierers. Er muss dafür sorgen, dass diese Regeln erfüllt sind, d. h. er darf nur zwischen solchen Typen eine Inklusionsfunktion definieren, bei denen eine Teilmengenbeziehung vorliegt und das Ersetzungsprinzip gilt.

Regelwerk für Verhaltenskonformität

Verhaltenskonformität:

S und T sind Wertklassen, die durch eine Inklusionsfunktion

$\text{include}: S \rightarrow T$

verbunden sind.

R, R1, R2, Q, Q1, Q2 sind weitere Wertklassen, die als Parameter- oder Ergebnistypen in den Operationen dieser Wertklassen verwendet werden.

f1, g1, h1 sind Operationen mit einem Parameter,
die in S *und* in T definiert sind, mit den Signaturen:

$f1: T \rightarrow R, f1: S \rightarrow R$

$g1: T \rightarrow T, g1: S \rightarrow S$

$h1: Q \rightarrow T, h1: Q \rightarrow S$

Dann muss für alle s vom Typ S, q vom Typ Q gelten:

$T.f1(\text{include}(s)) = S.f1(s)$

$T.g1(\text{include}(s)) = \text{include}(S.g1(s))$

$T.h1(q) = \text{include}(S.h1(q))$

f2, g2, h2 sind Operationen mit zwei Parametern,
die in S *und* in T definiert sind, mit den Signaturen:

$f2: T \times T \rightarrow R, f2: S \times S \rightarrow R$

$g2: T \times T \rightarrow T, g2: S \times S \rightarrow S$

$h2: Q \times Q \rightarrow T, h2: Q \times Q \rightarrow S$

Dann muss für alle s1, s2 vom Typ S, q1, q2 vom Typ Q gelten:

$T.f2(\text{include}(s1), \text{include}(s2)) = S.f2(s1, s2)$

$T.g2(\text{include}(s1), \text{include}(s2)) = \text{include}(S.g2(s1, s2))$

$T.h2(q1, q2) = \text{include}(S.h2(q1, q2))$

Für Operationen mit drei und mehr Parametern ergeben sich analoge Regeln.

Die Forderung nach gleichem Verhalten einer Operation im Supertyp und im Subtyp entspricht dem Liskov'schen Substitutionsprinzip [LW94]. Verhaltenskonformität muss in dieser Arbeit nicht gesondert postuliert werden. Sie ist eine logische Konsequenz aus der Definition der Inklusionsfunktion und der für Gleichheit geforderten Ununterscheidbarkeit (s. Abschnitt 8.3.1). Aufgrund der Vielzahl der möglichen Operations-Signaturen – s. dazu die Aufstellung auf Seite 204 – sind die Nachweise umfangreich. Anhang D.2 enthält die Herleitung der Verhaltenskonformität für mehrere Operations-Signaturen.

Die Forderung nach Verhaltenskonformität ist auch im Umfeld von Programmiersprachen zu finden, die ähnliche Mechanismen wie die Inklusionsfunktion unterstützen. Beispielsweise gibt es in Fortress die Möglichkeit, *coercions* zu deklarieren. Hallett et al. fordern, dass eine coercion mit anderen Operatoren der beiden Typen vertauschbar ist:

„If coercion is intended to preserve the conceptual identity of a value, then we might expect conversion to commute with any operator that can be applied to the value both before and after conversion. For example, we might expect `int2float(4)/int2float(3)` and `int2float(4/3)` to evaluate to the same value.“[HLRS08, p. 167]

Ein Beispiel für eine *nicht* verhaltenskonforme Konvertierung ist die implizite Konvertierung von `int` zu `String` in Java. Beide Typen, `int` und `String`, verfügen über den Operator `+`. Für `int` bedeutet er die Addition, für `String` die Konkatenierung von Zeichenketten. Diese beiden Operationen sind nicht mit der Konvertierung vertauschbar, sie weisen unterschiedliches Verhalten auf.

`(x+y).toString()` und `x.toString()+y.toString()`
liefern i. A. verschiedene Ergebnisse.

Dadurch provoziert diese Konvertierung überraschendes Programmverhalten und macht die Programmierung fehleranfällig. Das illustriert das folgende Code-Beispiel:

```
i = 2;
j = 3;
System.out.println("Erster Summand: " + i);    // 2
System.out.println("Zweiter Summand: " + j);    // 3
System.out.println("Summe: " + i + j);          // 23 (!)
```

Ein weiteres Beispiel für einen Verstoß gegen die Verhaltenskonformität ist die Art und Weise, wie etliche Programmiersprachen mit der Division umgehen. Der zugehörige Operator `/` bedeutet z. B. in Java (und ebenfalls in C, C++, C#) für `int` die Ganzzahl-Division, für `float` dagegen die Division für Gleitkommazahlen.

Diese verhalten sich unterschiedlich: Ganzzahl-Division rundet (vereinfacht dargestellt) auf die nächst-kleinere ganze Zahl ab, Gleitkomma-Division dagegen nicht. Das folgende Java-Code-Beispiel illustriert ein daraus resultierendes Programmverhalten:

```
int i1 = 5;
int i2 = 2;
int i3 = i1/i2;    // Ganzzahl-Division, Ergebnis 2
float x1 = i1;
float x2 = i2;
float x3 = x1/x2; // Gleitkomma-Division, Ergebnis 2.5
```

Es ergibt sich eine Konstellation, in der `i1 == x1` und `i2 == x2` erfüllt ist, nicht aber `i1/i2 == x1/x2`.^{3 4}

Zwischen den beiden Beispielen gibt es einen wesentlichen Unterschied: Bei der Konvertierung von `int` zu `String` liegt keine Subtypbeziehung vor. Ganzzahlen bilden keine Teilmenge der Zeichenketten, sie haben ein unterschiedliches Verhalten, und es würde keinen Sinn ergeben, alle `String`-Operationen auch für `int` verfügbar zu machen. Das Verhalten des überladenen Operators „+“ ist lediglich ein weiterer Indikator für die Unverträglichkeit der beiden Typen.

Ganzzahlen dagegen können als Subtyp der Gleitkommazahlen aufgefasst werden. Sie bilden eine Teilmenge, und es erleichtert die Programmierung, wenn Gleitkomma-Operationen (wie Potenzieren, Sinus oder Wurzel) auch für Ganzzahlen aufrufbar sind. Im Beispiel wurden die Probleme dadurch verursacht, dass für zwei verschiedene Operationen – Ganzzahl-Division im Subtyp, Gleitkomma-Division im Supertyp – derselbe Name oder dasselbe Symbol gewählt wurde.⁵

9.3.3 Unterschiede zwischen Inklusionsfunktion und Coercion

Viele Programmiersprachen unterstützen coercions, d. h. implizite Konvertierungen zwischen Datentypen. Die hier vorgeschlagene Inklusionsfunktion weist Gemeinsamkeiten mit coercions auf: Sie beinhaltet – auf technischer Ebene – eine Konvertierung, und sie läuft ebenfalls implizit ab, d. h. sie ist im Quellcode nicht sichtbar.

³In Haskell dagegen führt der Operator „/“ auch bei Ganzzahlen die Gleitkomma-Division aus, 5/2 liefert hier das Ergebnis 2.5. Ganzzahl-Division hat in Haskell einen eigenen Namen („div“).

⁴Anhang D.3 führt ein weiteres Beispiel für eine gleichnamige, aber nicht verhaltenskonforme Operation in Supertyp und Subtyp aus: `next` bei Wochentagen und Werktagen.

⁵Wenn eine Inklusionsfunktion zwei unabhängig voneinander programmierte Wertklassen verbindet, können derartige Namenskollisionen nicht ausgeschlossen werden. Eine Lösung besteht darin, derartige Konflikte durch Umbenennung aufzulösen, d. h. mit einem Sprachmechanismus, ähnlich wie Eiffel ihn für geerbte Methoden unterstützt. Weil es in dieser Arbeit um die verschiedenen Subtyping-Mechanismen bei Wertklassen und bei Objektklassen geht, werden die Details eines solchen Mechanismus hier nicht weiter vertieft.

Zwischen der Inklusionsfunktion und den in die jeweiligen Sprachen eingebauten coercions bestehen aber wesentliche Unterschiede:

Die Inklusionsfunktion dient ausschließlich dem Zweck, eine Subtypbeziehung zwischen zwei Wertklassen herzustellen. Bei impliziten Typumwandlungen geht es dagegen primär darum, die Programmierung expliziter Typumwandlungen einzusparen und dadurch die Quelltexte kürzer zu gestalten. Das geht z. B. aus der Java Language Specification hervor:

„... the context may be able to accept a type that is related to the type of the expression; as a convenience, rather than requiring the programmer to indicate a type conversion explicitly, the Java programming language performs an implicit conversion from the type of the expression to a type acceptable for its surrounding context.“[Gos+11, Kap. 5]

Einige implizite Konvertierungen bewirken den gleichen Effekt wie eine Inklusionsfunktion, z. B. in Java, C# oder C++ die Umwandlungen von `int` in `long` oder von `float` in `double`. Abschnitt 9.2.5 hatte eingangs bereits auf den Ersetzungseffekt der widening conversions in Java hingewiesen.

Viele implizite Konvertierungen weisen aber keinerlei Zusammenhang mit Subtypbeziehungen auf. Konvertierungen von beliebigen Typen in `String`, wie sie z. B. Java unterstützt [Gos+11, Kap. 5.1.11], liegt i. A. keine Subtypbeziehung zugrunde. C++ konvertiert implizit zwischen fast allen Ganzzahl- und Gleitkommazahl-Typen, wobei es bei Bedarf die Nachkommastellen streicht (z. B. bei der Umwandlung von `float` zu `int`). Es unterstützt sogar Konvertierungen von `int` zu `bool`, oder von `double` zu `char`, die ebenfalls keinerlei Subtypbeziehung begründen.

9.3.4 Zuordnung der Inklusionsfunktion im Code

Eine Inklusionsfunktion kann unabhängig von den beiden dadurch verbundenen Klassen definiert werden. Sie muss nicht notwendigerweise in der Klasse des Subtyps oder der Klasse des Supertyps definiert sein.

In einigen der vorangegangenen Beispiele sind Code-Abhängigkeiten von der Klasse des Supertyps zur Klasse des Subtyps vorhanden. Z. B. besitzt die Wertklasse `Complex` einen Selektor mit zwei Parametern vom Typ `Float` (den Real- und den Imaginärteil). Das spricht dagegen, die Inklusionsfunktion in der Klasse des Subtyps zu platzieren: Dort würde sie aufgrund der bereits vorhandenen Abhängigkeiten in umgekehrter Richtung zu zyklischen Code-Abhängigkeiten führen.

Durch die Option, die Inklusionsfunktion unabhängig von Sub- und Supertyp zu definieren, kann auch eine Subtypbeziehung zwischen zwei Wertklassen hergestellt werden, die unabhängig voneinander entwickelt wurden, oder zwischen Werttypen,

deren Quellcode nicht änderbar ist, z. B. weil es sich um vordefinierte Typen der Sprache handelt oder weil sie Bestandteil einer nicht zugänglichen Bibliothek sind.

Befindet sich die Inklusionsfunktion weder in der Klasse des Subtyps noch in der Klasse des Supertyps, können die beiden Klassen am flexibelsten benutzt werden: einzeln, unabhängig von der jeweils anderen Klasse, oder bei Bedarf kombiniert als Sub- und Supertyp.

9.3.5 Typäquivalenz zwischen zwei Wertklassen

Inklusionsfunktionen müssen nicht kreisfrei verlaufen. Für zwei Wertklassen A und B kann es wechselseitige Inklusionsfunktionen geben. Sie stellen dann Typäquivalenz her, d. h. sie bewirken, dass die beiden Wertklassen gegeneinander austauschbar benutzt werden können.

Die beiden Inklusionsfunktionen sind in solchen Fällen nicht nur injektiv, sondern bijektiv, und die Inklusionsfunktion von A nach B muss die Umkehrfunktion der Inklusionsfunktion von B nach A sein. Mit diesem Verfahren können zwei Wertklassen, die denselben Typ verkörpern – z. B. aus verschiedenen Quellen oder von verschiedenen Programmierern – austauschbar gemacht werden. Ein Fallbeispiel für den Einsatz wechselseitiger Inklusionsfunktionen sind zwei Wertklassen für die Abbildung eines Datums, `Date` und `Fecha`:

(Der folgende Code verwendet die in Kap. 6 eingeführte Notation.)

```
valueClass Date {
    Integer day;
    Integer month;
    Integer year;
    ...
    Integer dayOfYear(Date d) { ... }
    ...
}

valueClass Fecha {
    Integer diasDesde19700101;
    ...
    Bool antes (Fecha f1, Fecha f2) { ... }
    ...
}
```

Durch wechselseitige Inklusionsfunktionen kann zwischen `Date` und `Fecha` Typäquivalenz hergestellt werden:

```
Fecha include (Date d) { ... }
Date include (Fecha f) { ... }
```

Der hier nur angedeutete Code der beiden Wertklassen `Date` und `Fecha` und der zugehörigen Inklusionsfunktionen ist in Anhang D.4 detaillierter und mit weiteren Operationen und Erläuterungen ausgeführt.

Wie das Beispiel zeigt, können die involvierten Wertklassen unterschiedlich implementiert sein: `Date` beruht auf einer Darstellung bestehend aus Jahr, Monat und Tag, `Fecha` repräsentiert ein Datum intern anhand der Anzahl der Tage seit dem 1.1.1970.

Durch die beiden Inklusionsfunktionen können alle für `Date` definierten Operationen auch für Ausdrücke vom Typ `Fecha` aufgerufen werden und umgekehrt. Auf diese Weise können Ausdrücke vom Typ `Date` und `Fecha` austauschbar verwendet werden. Es sind auch gemischte Ausdrücke möglich, bei Operationen mit zwei oder mehr Parametern können Ausdrücke beider Typen als aktuelle Parameter benutzt werden. Damit ist z. B. folgender Klienten-Code möglich:

```
Fecha f1 := ...
Date d1 := ...
...
... dayOfYear(f1) ...
... antes(d1, f1) ...
```

So kann Code, der `Date` benutzt, einfach und ohne zusätzlichen Aufwand mit anderem Code, der `Fecha` benutzt, interagieren. Ohne wechselseitige Inklusionsfunktionen wären solche Programme nicht kompatibel zueinander, sie würden in jedem Einzelfall explizite Konvertierungen erfordern.

9.3.6 Gegenüberstellung:

Inklusionsfunktion versus Subclassing

Das V-O-Modell ermöglicht, dieselbe semantische Beziehung – nämlich eine Subtypbeziehung – bei Wertklassen und bei Objektklassen durch unterschiedliche, auf die jeweiligen Typen zugeschnittene sprachliche Mechanismen zu unterstützen. Bei Wertklassen wird dafür die in Abschnitt 9.3.1 eingeführte Inklusionsfunktion eingesetzt, bei Objektklassen die – in objektorientierten Sprachen für diesen Zweck etablierte – Implementierungsvererbung („Subclassing“).

Abbildung 9.1 illustriert diesen Zusammenhang.

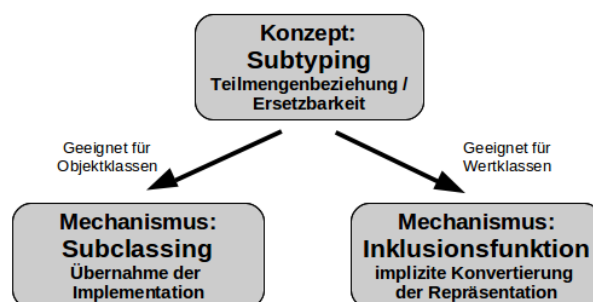


Abbildung 9.1: Sprachmechanismen für Subtyping

Ebenso wesentlich wie die Verfügbarkeit eines passenden Mechanismus für das Subtyping ist es, ungeeignete Mechanismen (s. Abschnitt 9.2.4) *nicht* zu unterstützen und dadurch Fehlerquellen auszuschalten. Die verschiedenartigen Subtyp-Mechanismen für Wertklassen und für Objektklassen sind darum eine weitere Motivation für die programmiersprachliche Trennung von Werttypen und Objekttypen.

Tabelle 9.3 stellt Implementierungsvererbung und Inklusionsfunktion einander gegenüber. Sie zeigt, dass diese beiden Sprachmechanismen unterschiedlich funktionieren, aber zum selben Ergebnis führen, nämlich zu einer Subtypbeziehung.

Auswirkung ...	... des Subclassing bei Objektklassen	... der Inklusionsfunktion bei Wertklassen
Operationen des Supertyps sind auch für Elemente des Subtyps aufrufbar, ...	... weil die Subklasse sie von der Basisklasse erbt.	... weil eine Repräsentation des Subtyps implizit in die Repräsentation des Supertyps konvertiert wird.
Im Subtyp können Operationen vorhanden sein, über die der Supertyp nicht verfügt, ...	... weil in einer Subklasse zusätzliche Operationen definiert werden können.	... weil Operationen, die nur in der Klasse des Subtyps definiert sind, für Elemente des Supertyps nicht aufrufbar sind.
Eine Operation kann im Supertyp und im Subtyp unterschiedlich implementiert sein, ...	... weil sie in der Subklasse überschrieben werden kann.	... weil im Subtyp und im Supertyp gleichnamige Operationen mit unterschiedlicher Implementation zulässig sind.
Für verschiedene Implementationen einer Operation kann die Sprachumgebung i. A. gleiches Verhalten nicht garantieren, ...	... das gilt bei Überschreiben von Methoden.	... das gilt, wenn in zwei verschiedenen Wertklassen zwei gleichnamige Operationen implementiert sind.

Tabelle 9.3: Wirkungsweise von Subclassing und Inklusionsfunktion

9.3.7 Vergleiche zu ähnlichen Sprachkonstrukten

Einige Programmiersprachen unterstützen programmierer-definierbare implizite Konvertierungen und damit Sprachmechanismen, die der Inklusionsfunktion ähneln. In Scala gibt es *implicit conversions* (s. Abschnitt 9.2.5), C# kennt *user-defined implicit conversions* [ECM06a, Kap. 17.9.3], Fortress unterstützt *user defined coercions* (s. Abschnitt 9.3.2), Eiffel hat *type conversions* in die Sprache aufgenommen [Mey01] [ECM06b, Kap. 8.15].

Der wesentliche Unterschied zwischen der Inklusionsfunktion und den genannten Konvertierungen in Scala, C#, Fortress und Eiffel besteht darin, dass die Inklusionsfunktion ausschließlich dem Zweck dient, zwischen zwei Wertklassen eine Subtyp-Beziehung herzustellen. Aus diesem Grund ist eine Inklusionsfunktion nur zwischen

Wertklassen zulässig. Nur das V-O-Modell mit seiner sprachliche Trennung von Werten und Objekten ermöglicht eine sprachliche Sicherstellung dieser Einschränkung. Conversions und coercions dagegen dienen nicht explizit dazu, eine Subtyp-Beziehung zu etablieren. Sie können auch nicht auf Werttypen eingegrenzt werden, weil Scala, C#, Fortress und Eiffel nicht zwischen Werttypen und Objekttypen unterscheiden.

In Scala werden implicit conversions oft dazu benutzt, einen Typ um zusätzliche Methoden „anzureichern“. Das Verfahren bietet sich insbesondere dann an, wenn der Quellcode einer Klasse nicht verfügbar ist oder nicht verändert werden darf. Z. B. können String-Operationen wie `reverse`, `capitalize`, etc., die in der Klasse `String` nicht vorhanden sind, in einer Klasse `RichString` programmiert werden. Mit einer impliziten Konvertierung von `String` zu `RichString` können die zusätzlichen Methoden auch für Ausdrücke vom Typ `String` aufgerufen werden (s. dazu [WP09, p. 186]). Sie erscheinen für den Aufrufer so, als wären sie in der Klasse `String` definiert. (Der Scala-Jargon bezeichnet dieses Verfahren als „Pimp my library“).

In C# fordern die Richtlinien für implicit user defined conversions keine Verhaltenskonformität, sondern lediglich, dass solche Konvertierungen keine Exception auslösen und dass sie keinen „Informationsverlust“ bewirken. Letzteres entspricht der Forderung nach Injektivität:

„In general, user-defined implicit conversions should be designed to never throw exceptions and never lose information. If a user-defined conversion can give rise to exceptions (for example, because the source argument is out of range) or loss of information (such as discarding high-order bits), then that conversion should be defined as an explicit conversion.“[ECM06a, Kap. 17.9.3]

In Fortress dienen user defined coercions einem ähnlichen Zweck wie die hier vorgeschlagene Inklusionsfunktion, nämlich eine Ersetzbarkeit zwischen einem Typ B durch einen Typ A herzustellen. Das lässt die folgende Aussage von Hallett et al. erkennen:

„allowing coercion from type A to type B should be a deliberate decision by the designer of B that in every circumstance that a function, method or operator expects an argument of type B, a value of type A may be used instead. In particular, every instance of A should represent the same conceptual entity as some instance of B.“[HLRS08, p. 167]

Hallett et al. stellen fest, dass bei Vorliegen einer coercion zwischen A und B die beiden Typen in einer *ähnlichen* Beziehung zueinander stehen wie Sub- und Supertyp. Sie betrachten aber Coercion und Subtyping⁶ als verschieden. Als Oberbegriff für beides etablieren sie den Begriff der „substitutability“:

„Thus, the relationship between B and A is similar to the one between B and its subtypes. We say that A is substitutable for B if either A is a subtype of B or A coerces to B.“[HLRS08, p. 167]

In Fortress ist es zulässig, dass zwischen mehreren Klassen sowohl Subtyp-Beziehungen (definiert mittels `extends`) als auch implizite Umwandlungen (definiert mittels `coerce`) vorhanden sind. So können die `extends`- und `coerce`-Beziehungen ein komplexes Geflecht bilden. Damit die Interaktionen dieser Beziehungen untereinander und mit Klientencode nicht zu Konflikten oder Mehrdeutigkeiten führen, definiert Fortress ein umfangreiches Regelwerk. Z. B. kann es zwischen zwei Klassen nicht gleichzeitig eine `extends`- und eine `coerce`-Beziehung geben, coercions werden, anders als andere Methoden, nicht vererbt, etc.

Bei überladenen Methoden (die in Fortress dynamisch, durch multiple dispatch aufgelöst werden) muss die Auswahl des passenden („most specific“) Codes sowohl `extends`-Beziehungen als auch coercions berücksichtigen.⁷ Anhang D.5 illustriert anhand eines Code-Beispiels, wie die Verflechtung von coercions und Subclassing die Komplexität des Codes erhöht und seine Nachvollziehbarkeit erschwert.

Auch Eiffel trifft Vorkehrungen, damit die durch Vererbung und die durch Conversion induzierte Ersetzbarkeit nicht zueinander in Konflikt geraten („Conformance-conversion exclusion principle“[Mey01, p. 29]). Da Eiffel kein Überladen von Methoden vorsieht, ist das Regelwerk in dieser Hinsicht einfacher als das von Fortress. Auf der anderen Seite verkomplizieren die `expanded Types` von Eiffel die Regeln. Diese gelten auch bei Vorliegen einer Vererbungsbeziehung als nicht konform zueinander und dürfen darum ineinander konvertiert werden.

Das in dieser Arbeit zugrunde gelegte V-O-Modell und die darauf aufbauende Inklusionsfunktion vermeiden die hier am Beispiel von Fortress und Eiffel beschriebene Komplexität, die aus der Interaktion von Vererbungs- und Konvertierungsbeziehungen resultiert. Weil es im V-O-Modell Implementierungsvererbung nur zwischen Objektklassen und Inklusionsfunktionen nur zwischen Wertklassen gibt, können die

⁶Fortress versteht unter Subtyping die durch Subclassing induzierte Ersetzbarkeit.

⁷Um sicherzustellen, dass eine spezifischste („most specific“) Methode immer eindeutig bestimmt werden kann, stellt Fortress weitere Regeln auf. Darin kommt einer dritten Art von Beziehung, der *excludes*-Beziehung, eine wesentliche Rolle zu. Sie besagt, dass die beiden beteiligten Klassen kein gemeinsames Element haben können. Dass es in Fortress drei Arten von Beziehungen (`extends`, `coerce`, `excludes`) zwischen zwei Klassen geben kann, erhöht die Komplexität der Sprache zusätzlich.

damit jeweils hergestellten Subtyp-Beziehungen nicht miteinander in Wechselwirkung treten. Anders als z. B. in Fortress oder Eiffel ist deshalb kein Regelwerk zum Ausschließen potentieller Konflikte erforderlich. Das zweigeteilte Typsystem macht so den Code besser nachvollziehbar und die Sprache leichter erlernbar.

9.4 Anwendung auf bekannte Probleme

Viele bekannte Probleme und Paradoxien in objektorientierten Sprachen haben ihren Ursprung in dem Versuch, für wert-artige Abstraktionen eine Subtypbeziehung mittels Subclassing herzustellen. Das geschieht zwangsläufig, wenn Werte nicht von Objekten unterschieden werden und darum für alle Abstraktionen derselbe – auf Objekte ausgerichtete – Mechanismus zur Verfügung steht.

Ein Beispiel dafür ist das „Kreis-Ellipsen-Dilemma“, ein weiteres sind die Probleme mit binären Methoden. Anhand dieser beiden bekannten Fallbeispiele wird veranschaulicht, wie die Verwendung von Wertklassen und der Einsatz einer Inklusionsfunktion Probleme vermeidet und die fachlichen Sachverhalte einfach und geradlinig abbildet.

9.4.1 Kreis-Ellipsen-Dilemma

Ein Beispiel für eine Situation, in der objektorientierte Modellierung an ihre Grenzen stößt, ist das vielfach erörterte *Kreis-Ellipsen-Dilemma*, u. a. diskutiert von Coplien [Cop92, p. 227], Martin [Mar96], Majorinc [Maj98] oder Henney [Hen01]. Dieses Dilemma befasst sich mit der Frage, ob und in welcher Richtung zwischen den beiden Klassen `Kreis` und `Ellipse` eine Vererbungsbeziehung programmiert werden sollte. Um die folgenden Erläuterungen nicht durch aufwändige Berechnungen der Ellipsen-Haupt- und Nebenachsen zu verkomplizieren, wird das Problem hier anhand der strukturell gleich aufgebauten, einfacher implementierbaren Beispiele `Quadrat` und `Rechteck` illustriert.

Das Problem

Quadrate können als spezielle Rechtecke angesehen werden: ein Quadrat „ist ein“ Rechteck, dessen Seiten alle gleich lang sind. Das spricht dafür, `Quadrat` von `Rechteck` erben zu lassen. Eine Methode `stretchX`, die eine Seite des Rechtecks um einen angegebenen Faktor streckt oder staucht, wirft dann Probleme auf. Aufgrund der Vererbung ist sie auch auf `Quadrate` anwendbar. Nach Ausführung dieser Methode ist aber ein Quadrat i. A. kein Quadrat mehr. Wird die Methode `stretchX` in der Klasse `Quadrat` überschrieben, so dass sie die Länge der anderen Seite ebenfalls anpasst, verletzt sie andere Nachbedingungen, z. B. dass die Fläche der Figur sich um den gleichen Faktor ändern muss.

Der umgekehrte Ansatz, Rechteck von Quadrat erben zu lassen – mit der Begründung, dass ein Rechteck über zwei Seitenlängen und damit über ein Attribut mehr verfügt als ein Quadrat, das nur eine Seitenlänge benötigt – würde bewirken, dass Rechtecke als spezielle Quadrate gelten und solche substituieren könnten. Eine dritte Lösung, die Quadrat und Rechteck als unabhängige Klassen ohne Vererbungsbeziehung programmiert, verhindert Code-Wiederverwendung. Es scheint keinen Ansatz zu geben, der alle Erwartungen an eine Lösung erfüllt.

Wenn die Sprache Werttypen von Objekttypen trennt, stellt sich das geschilderte Problem nicht. Der Programmierer muss sich dann bei der Programmierung von Quadrat und Rechteck für eine Abbildung als Werttyp oder als Objekttyp entscheiden. In beiden Fällen treten die o. g. Schwierigkeiten nicht auf.

Anhang D.6 enthält den Code für die beiden Programmierungsvarianten: Alternative 1 mit zwei Wertklassen und zugehöriger Inklusionsfunktion, Alternative 2 mit zwei Objektklassen, ohne Vererbungsbeziehung.

Alternative 1: Programmiere Quadrate und Rechtecke als Werte

Quadrate und Rechtecke werden als mathematische Figuren und somit als Werte aufgefasst und durch zwei Wertklassen – `SquareV` und `RectangleV` – implementiert. Eine Inklusionsfunktion bringt zum Ausdruck, dass `SquareV` ein Subtyp von `RectangleV` ist. Eine Funktion `stretchX`, definiert in der Wertklasse `RectangleV`, liefert dann das Rechteck, das sich aus der Änderung der Seite `x` um den angegebenen Faktor ergibt. `stretchX` ist aufgrund der Inklusionsfunktion auch auf Elemente von `SquareV` anwendbar, sie liefert immer ein Element von `RectangleV`.

Alternative 2: Programmiere Quadrate und Rechtecke als Objekte

Alternativ können Rechtecke und Quadrate als veränderliche Objekte aufgefasst und mit Hilfe von Objektklassen (`RectangleO` und `SquareO`) abgebildet werden. `SquareO` ist weder ein Subtyp noch eine Subklasse von `RectangleO`: Sie ist kein Subtyp, weil die Operationen von `SquareO` und `RectangleO` unterschiedliches Verhalten aufweisen; sie haben sogar unterschiedliche Schnittstellen. Sie ist keine Subklasse, weil die Implementation von `RectangleO` (mit zwei Seitenlängen `x` und `y`) nicht an die Klasse `SquareO` vererbt werden kann (deren Implementation nur eine einzige Seitenlänge benötigt).

9.4.2 Typisierung binärer Methoden

Für die Schwierigkeiten, die binäre Methoden in Zusammenhang mit Vererbung aufwerfen, wurden verschiedene Lösungen diskutiert. Im Folgenden wird gezeigt, dass die oben vorgeschlagene Inklusionsfunktion für das Problem der binären Methoden einen bisher noch nicht vorgeschlagenen Lösungsansatz liefert.

Das Problem

Unter einer binären Methode wird eine Methode *m* verstanden, die in der Klasse *A* definiert ist und (mindestens) einen Parameter des Typs *A* hat. D. h. der Receiver der Methode und ihr Parameter haben denselben Typ, die Methode entspricht folgendem Schema:

```
class A {  
    R m(A other) {...}  
}
```

Beispiele für binäre Methoden sind Addition und Multiplikation von Zahlen oder Teilmengen- und Schnittmengen-Bildung von Mengen.

Das Hauptproblem binärer Methoden besteht in ihrer Typisierung in Zusammenhang mit Vererbung: Eine Klasse *B*, die von *A* erbt und die Methode *m* überschreibt, muss ihren Parameter auf den Typ *B* anpassen, anderenfalls wäre sie keine binäre Methode mehr:

```
class B extends A {  
    R m(B other) {...}  
}
```

Aufgrund der Typänderung des Parameters⁸ bildet *B* keinen Subtyp von *A* mehr, wodurch eine – nach Bruce et al. wünschenswerte – Eigenschaft objektorientierter Sprachen verloren geht, nämlich die aus Subtyping resultierende Wiederverwendbarkeit:

„if subclasses correspond to subtypes, a subclass object can always be passed to a function or method expecting a superclass object, allowing reuse of code.“[Bru+95, p. 223]

Zusätzlich wirft auch der fehlende Zugriff auf die interne Repräsentation des Parameters Schwierigkeiten auf. Oft müssen in der Implementation einer binären Methode der Receiver und der Parameter symmetrisch behandelt werden. Das ist aber i. A. nicht möglich, weil die Methode *m* nur Zugriff auf die Repräsentation des Receivers, nicht aber auf die des Parameters hat. Die Signatur der binären Methode *m* stellt

⁸Die meisten objektorientierten Sprachen unterstützen eine solche Typänderung nicht.

nämlich nur sicher, dass der Parameter den Typ von A hat, nicht notwendigerweise aber dieselbe Klasse und damit dieselbe Implementation wie der Receiver.

Bekannte Lösungsansätze

Für die mit binären Methoden einhergehenden Probleme wurden unterschiedliche Lösungen vorgeschlagen, u. a. *Matching* (Bruce [Bru94]), *Precise typing* (Eifrig et al. [EST95]) oder verschiedene Varianten von *Multi-Methoden* (Mugridge et al. [MHH91], Chambers und Leavens [CL94], Castagna [Cas95]). Ein zusammenfassender Überblick über verschiedene Ansätze und ihre Vor- und Nachteile findet sich bei Bruce et al. [Bru+95]. Eine weitere, dort nicht enthaltene Lösung ist der von Schmolitzky et al. vorgeschlagene Ansatz, binäre Methoden mit Klassenmethoden statt Exemplarmethoden abzubilden und mit dem Typbezeichner `ThisType` dynamisches Binden für sie zu ermöglichen [SEKM00].

Alle genannten Ansätze zielen darauf ab, dass Subclassing trotz der o. g. strukturellen Schwierigkeiten zu Ersetzbarkeit führt. Um das zu erreichen, modifizieren sie entweder die Mechanismen des Subclassing oder sie reduzieren die Anforderungen an die Ersetzbarkeit, so dass im Ergebnis eine typsichere Ersetzung auch bei binären Methoden möglich wird. Keine der genannten Arbeiten versucht, eine Subtypbeziehung auf anderem Wege als durch Subclassing herzustellen.

Ein Beispiel soll die vorgeschlagenen Lösungsansätze illustrieren. Es zeigt anhand der Klassen `2DPointClass` und `3DPointClass` und der Methode `equal`, wie mit Hilfe der von Castagna vorgeschlagenen *gekapselten Multimethoden* [Cas95] bei binären Methoden eine typsichere Ersetzung erreicht werden kann.⁹ Bei dieser Lösung bestimmt der Parameter der Methode `equal` *dynamisch*, welcher Code aufgerufen wird (anders als z. B. in Java, wo gleichnamige Methoden mit verschiedenen Parametertypen überladene Varianten der Methode darstellen würden).

```
class 2DPointClass
  instance variables
    xValue: real
    yValue: real
  methods
    x: real is return (xValue)
    y: real is return (yValue)
    equal(p: 2DPoint): bool is
      return ( (xValue==p.x) && (yValue==p.y) )
end class
```

⁹Der Code geht zurück auf die Arbeit von Bruce et al. [Bru+95]. Er verwendet die dort eingeführte Notation und wurde hier aus mehreren Fragmenten der Original-Arbeit zusammengefügt.

```
class 3DPointClass subclass of 2DPointClass
  instance variables
    zValue: real
  methods
    z:real is return(zValue)
    equal(p: 2DPoint): bool is
      return(p.equal(self) )
    equal(p: 3DPoint):bool is
      return
        ( (xValue==p.x) && (yValue==p.y) && (zValue==p.z) )
end class
```

Bruce et al. stellen eine Liste von Kriterien auf, an denen sie Lösungen für das Problem der binären Methoden messen, s. Tabelle 9.4.

Beurteilungskriterien für Lösungen des Problems der binären Methoden, nach Bruce et al. [Bru+95, p. 222]

1. How expressive is the solution? In particular, to what extent does it allow reasonable subclassing and message sends?
2. Do subclasses always produce subtypes?
3. Do binary methods have privileged access to the argument's state?
4. Is program development modular? In particular, does adding a new class ever force modification to existing code, and can module interfaces be defined?
5. Are the receiver and argument of a binary method treated symmetrically?
6. Does the solution avoid unnecessary code duplication?

Tabelle 9.4: Beurteilungskriterien für Lösungen zu binären Methoden

Alle von Bruce et al. vorgestellten Ansätze weisen in Bezug auf mehrere dieser Kriterien Defizite auf. Es gibt nicht einen besten Ansatz, und keinen, der alle Kriterien erfüllt.

Binäre Methoden aus der Perspektive von Werttypen

Die Beispiele für binäre Methoden, die in den oben genannten Arbeiten herangezogen wurden – Gleichheit, Addition, Multiplikation, Grösser/Kleiner-Vergleich, Schnittmenge – sind Operationen, deren Receiver und Parameter als Werttypen eingestuft werden können: Zahlen, Punkte, Mengen. Insbesondere die Implementierung einer Gleichheit ist ein Hinweis darauf, dass – mangels direkter sprachlicher Unterstützung – eine Klasse dazu eingesetzt wurde, einen wert-artigen Typ abzubilden (s. dazu auch Kap. 8).

Abschnitt 9.2.4 hat bereits darauf hingewiesen, dass Subclassing bei Wertklassen nicht zu einer Subtyp-Beziehung führt. Die bei binären Methoden zu beobachtenden

Schwierigkeiten können deshalb auch als Symptome eines an anderer Stelle liegenden Problems aufgefasst werden:

- dem Einsatz eines für die beteiligten Typen ungeeigneten Mechanismus, um eine Subtyp-Beziehung herzustellen,
- dem (impliziten) Ziel, die Sprache so zu gestalten, dass Subclassing immer zu Subtyping führt,¹⁰
- und, als Ursache für die beiden vorgenannten Punkte, die fehlende Trennung von Werten und Objekten.

Im V-O-Modell, mit einer expliziten programmiersprachlichen Unterstützung für Werttypen und einer Inklusionsfunktion als zugehörigem Subtypmechanismus, werfen binäre Methoden keines der oben geschilderten Probleme auf. Das folgende Code-Schema illustriert, wie sich eine binäre Methode `m` bei zwei durch eine Inklusionsfunktion verbundenen Wertklassen `A` und `B` darstellt:

```
valueClass A {  
    R m(A a1, A a2) {...} // binäre Methode m  
    ...  
}  
  
valueClass B {  
    ...  
}  
  
A include(B b) { ... }
```

Mit dieser Programmierung ist *jede* Funktion mit Parametern vom Typ `A` auch für Parameter vom Typ `B` aufrufbar, und damit auch eine „binäre Funktion“ wie `m`. Jede Kombination von Parametern vom Typ `A` oder `B` ist bei einem Aufruf von `m` möglich:

```
// a1, a2 Ausdrücke vom Typ A,  
// b1, b2 Ausdrücke vom Typ B  
... m(a1, a2) ...  
... m(a1, b1) ...  
... m(b1, a1) ...  
... m(b1, b2) ...
```

Bei dieser Lösung gibt es auch nicht die oben geschilderten Probleme, auf die interne Repräsentation des Parameters zuzugreifen. Die Inklusionsfunktion sorgt

¹⁰Ein weiteres Problem des obigen Beispiels `2DPointClass/ 3DPointClass` besteht darin, dass dreidimensionale Punkte keinen Subtyp der zweidimensionalen Punkte bilden (s. Abschnitt 9.2.4). Im obigen Beispiel wird mit gekapselten Multimethoden unter erheblichem Aufwand eine typischere Ersetzbarkeit zwischen `2DPointClass` und `3DPointClass` hergestellt. Die Multimethode `equal` ist aber nicht transitiv und verstößt damit gegen den Gleichheitsvertrag. Solche Anomalien stellen das Ziel, dass Subclassing immer zu Subtyping führen sollte, in Frage.

dafür, dass auch der Wert des Subtyps bei Aufruf der binären Methode `m` in der von der Klasse `A` spezifizierten Repräsentation vorliegt.

Kontrastiert man diese Lösung mit den von Bruce et al. vorgestellten Ansätzen, z. B. den oben skizzierten gekapselten Multimethoden, so werden weitere Vorteile erkennbar:

- **Nur eine Implementation:** Bei Einsatz von Wertklassen und einer Inklusionsfunktion ist nur eine einzige Implementation der binären Methode (der Funktion `m` in der Wertklasse `A`) erforderlich.¹¹
Im Beispiel der gekapselten Multimethoden (s. o.) dagegen werden immer mehrere Implementationen der binären Methode benötigt, um die gewünschte Typsicherheit herzustellen. Im obigen Beispiel befindet sich eine Implementation von `equal` in der Basisklasse, zwei in der Subklasse. Würde man eine weitere Klasse `Point4D` als Subklasse von `Point3D` erstellen, so würden nochmals weitere Implementationen hinzukommen, s. dazu Bruce [Bru+95, p. 233].
- **Programmierung an zentraler Stelle:** Subtyping mittels Inklusionsfunktion setzt an einer zentralen Stelle an, nämlich an der Verbindung der beiden beteiligten Wert-Klassen. Dadurch können alle, insbesondere die binären, Operationen des Supertyps auch für Parameter des Subtyps aufgerufen werden. Die jeweiligen Operationen erfordern keinen zusätzlichen Programmieraufwand. Gekapselte Multimethoden und ähnliche Ansätze dagegen benötigen eine entsprechende Programmierung für jede einzelne binäre Methode.

Wertklassen mit einer zugehörigen Inklusionsfunktion erfüllen fünf der sechs Kriterien, die Bruce et al. zur Beurteilung des Problems der binären Methoden aufgestellt haben (s. Tabelle 9.4):

- Kriterium 1: Die Inklusionsfunktion lässt die Subtypbeziehung klar erkennen und sie erlaubt alle Varianten von Aufrufen.
- Kriterium 3: Durch die implizite Konvertierung hat die „binäre Methode“ auch auf die interne Repräsentation des Parameters Zugriff.¹²
- Kriterium 4: Die beiden Wertklassen können unabhängig voneinander definiert werden, so dass Änderungen der einen Klasse die andere nicht beeinträchtigen.
- Kriterium 5: Der Mechanismus der Inklusionsfunktion gewährleistet per se eine symmetrische Behandlung der beiden Parameter. Insbesondere ist für Aufrufe mit gemischten Parametern – einer vom Subtyp, einer vom Supertyp – kein

¹¹In der Wertklasse `B` kann es weitere Implementationen von `m` geben (soweit sie verhaltenskonform sind, s. Abschnitt 9.3.2). Sie sind aber nicht erforderlich, um `m` für Parameter vom Typ `B` aufrufbar zu machen.

¹²Bei Wertoperationen ist eine Unterscheidung zwischen „Receiver“ und sonstigen Parametern nicht vorgesehen (s. Kap. 6.2), darum stellt sich dieses Problem hier im Grunde genommen nicht.

gesonderter Code notwendig, der die Gefahr von Asymmetrien mit sich bringen könnte.

- Kriterium 6: Weil nur eine einzige Implementation erforderlich ist, kommt es nicht zu unnötiger Code-Duplikation.

Kriterium 2 („Do subclasses always produce subtypes?“) ist nicht erfüllt, genau das aber ist der Kern des Lösungsansatzes: Anstatt eine Subtypbeziehung durch Subclassing aufzubauen, wird dafür ein anderer – für Wertklassen besser geeigneter – Mechanismus eingesetzt.

Zusammenfassend erscheint das Problem der binären Methoden – soweit es in Zusammenhang mit Werttypen betrachtet wird – als ein Scheinproblem, hervorgerufen durch die Vernachlässigung der fundamentalen Unterschiede zwischen Werttypen und Objekttypen und durch die implizite Annahme, dass Subklassen immer zu Subtypen führen sollten. Mit Einsatz einer Inklusionsfunktion verursachen binäre Methoden bei Werttypen und ihren Subtypen keine Schwierigkeiten.

9.5 Zusammenfassung

Vererbung, ein zentrales Merkmal der Objektorientierung, koppelt in den meisten objektorientierten Sprachen zwei ineinander greifende, aber signifikant verschiedene Bestandteile aneinander, nämlich Subtyping und Subclassing. Subtyping ist eine semantische Beziehung zwischen zwei Typen: Die Elemente des Subtyps bilden eine Teilmenge der Elemente des Supertyps und können überall dort verwendet werden, wo ein Element des Supertyps zulässig ist. Subclassing ist ein Sprachmechanismus, der es ermöglicht, eine Klasse (die Subklasse) aufbauend auf einer anderen Klasse (der Basisklasse) zu implementieren.

Kapitel 9 zeigte, dass die Trennung von Werten und Objekten Anpassungen im Bereich der Vererbung möglich macht. Subclassing führt bei Objektklassen i. A. dazu, dass die zugehörigen Typen in einer Subtypbeziehung stehen. Bei Wertklassen dagegen führt Subclassing nicht zu Subtyping. Auch konnten Fallbeispiele angeführt werden, in denen der von der Basis-Wertklasse geerbte Code nur in geringem Umfang in der Sub-Wertklasse wiederverwendbar ist, dort aber ein hohes Fehlerpotential aufweist.

Subtyp-Beziehungen gibt es auch zwischen Werttypen. Z. B. können Ganzzahlen als Subtyp von rationalen Zahlen angesehen werden, oder Gleitkommazahlen als Subtyp von komplexen Zahlen. Kapitel 9 schlug einen dedizierten Sprachmechanismus (eine „Inklusionsfunktion“) für die Definition einer Subtypbeziehung zwischen zwei Wertklassen vor. Es beschrieb, welche Eigenschaften eine Inklusionsfunktion aufweisen muss (injektiv, verhaltenskonform) und zeigte anhand mehrerer Beispiele ihre Einsatzmöglichkeiten und Vorteile.

Einige Programmiersprachen (z. B. Scala, Fortress, C# und Eiffel) unterstützen programmierer-definierbare implizite Typumwandlungen. Diese Sprachmechanismen weisen Ähnlichkeiten mit einer Inklusionsfunktion auf. Wirkungsweise und Einsatzgebiete der impliziten Typumwandlungen unterscheiden sich aber von der Inklusionsfunktion, vor allem deshalb, weil die genannten Sprachen nicht zwischen Werttypen und Objekttypen unterscheiden.

Die Trennung von Werten und Objekten mündet in ein Sprachmodell, das Subtyping durch den jeweils passenden Sprachmechanismus unterstützt: Subclassing für Objektklassen, Inklusionsfunktionen für Wertklassen. Eine Gegenüberstellung von Subclassing und Inklusionsfunktion veranschaulichte, dass diese beiden Sprachmechanismen trotz ihrer Verschiedenheit etliche Gemeinsamkeiten aufweisen und dadurch zum gleichen Ergebnis, nämlich einer Subtypbeziehung, führen.

Viele bekannte Probleme der Objektorientierung sind auf den Versuch zurückzuführen, für wert-artige Abstraktionen eine Subtypbeziehung mittels Subclassing herzustellen. Das geschieht zwangsläufig, wenn Werte nicht von Objekten unterschieden werden und darum für beide derselbe – auf Objekte ausgerichtete – Mechanismus zur Verfügung steht. Beispiele dafür sind das „Kreis-Ellipsen-Dilemma“ oder die mit binären Methoden einhergehenden Probleme. Kapitel 9 veranschaulichte anhand dieser Beispiele, wie der Einsatz von Wertklassen und einer zugehörigen Inklusionsfunktion die fachlichen Sachverhalte einfach und geradlinig abbildet. Die Trennung von Werten und Objekten und ein auf die jeweiligen Typen ausgerichteter Subtyping-Mechanismus – und ebenso wichtig, das Unterbinden ungeeigneter Mechanismen – vermeiden auf diese Weise eine eigene Klasse von Fehlern.

Kapitel 10

Zusammenfassung und Ausblick

Diese Arbeit präsentierte ein Sprachmodell, das zwei fundamental verschiedene, zueinander komplementäre Konzepte miteinander verbindet: Werte und Objekte.

Die Eigenschaften, die eine Abstraktion als Wert oder als Objekt charakterisieren – Zustandslosigkeit bei Werten, änderbarer Zustand bei Objekten – sind Charakteristika der zugehörigen Typen. Konsequenterweise nimmt das Sprachmodell eine explizite Trennung von Werttypen und Objekttypen vor, sein Typsystem ist zweigeteilt. Die Anpassung anderer Sprachkonstrukte an die fundamentale Verschiedenheit von Werten und Objekten führt zu Änderungen im Sprachentwurf, die etliche potentielle Fehlerquellen objektorientierter Ansätze beseitigt. Damit verfügt das in dieser Arbeit beschriebene Sprachmodell über einen anderen „Baukasten“ von aufeinander abgestimmten Sprachkonstrukten als objektorientierte Sprachen. Es ermöglicht nicht nur, Werttypen klar und sicher zu programmieren, es verändert auch die Weltsicht der Softwareentwickler, die damit arbeiten.

10.1 Zusammenfassung

Teil 1 dieser Arbeit beantwortete die Frage, was Werte sind und worin sie sich von Objekten unterscheiden. Werte sind die Elemente zustandsloser Typen, sie verkörpern das funktionale Paradigma. Objekte dagegen sind die Elemente zustandsbehafteter Typen, sie sind damit dem imperativen Paradigma zuzuordnen.

Die Arbeit stellt den Begriff der „Zustandslosigkeit“ und damit der „Wertartigkeit“ auf eine belastbare Grundlage, indem sie eine Definition für Werttypen mit vier überprüfbaren Kriterien angibt: Unveränderlichkeit, Unerzeugbarkeit, Seiteneffektfreiheit und referentielle Transparenz. Diese Kriterien sind eng miteinander verflochten und stellen einander ergänzende Facetten von Zustandslosigkeit dar.

Werttypen und Objekttypen bilden mit dieser Definition klar unterscheidbare Gegensätze, es gibt keine Mischformen. Insbesondere sind Werte nicht dasselbe wie un-

veränderliche Objekte, und Wertoperationen („Funktionen“) sind nicht dasselbe wie seiteneffektfreie, aber dennoch zustandsabhängige Objektoperationen („Queries“).

Bereits aus der Definition lassen sich, unabhängig von einer konkreten Programmiersprache, Gesetzmäßigkeiten über das Zusammenwirken von Werten und Objekten ableiten. Z. B. stehen sie in einem asymmetrischen Abhängigkeitsverhältnis: Objekte können Werte benutzen, Werte aber keine Objekte. Im Ergebnis gliedert sich jedes nach diesem Ansatz entwickelte Softwaresystem in einen zustandslosen „funktionalen Kern“ und eine zustandsbehaftete „objektorientierte Schale“. Damit unterscheidet sich das Sprachmodell dieser Arbeit von dem rein funktionaler Sprachen, die vollständig zustandslos sind, und von dem imperativer Sprachen, die Zustand prinzipiell überall vorsehen.

Teil 2 der Arbeit wies nach, dass die Abbildung von Werttypen durch Programmiermuster, Programmierkonventionen und ähnliche Verfahren aus softwaretechnischer Sicht Schwachstellen aufweist. Solche Verfahren sind aufwändig zu programmieren, und teilweise bürden sie den Klienten die Verantwortung für das wert-gemäße Verhalten der Klasse auf. Sie lassen Werttypen nicht als solche erkennen. Das macht sie fehleranfällig, denn dadurch besteht die Gefahr, die Werteigenschaften bei einer Code-Überarbeitung zu zerstören.

Weiter wurde gezeigt, dass bestehende objektorientierte Programmiersprachen Werttypen nur rudimentär unterstützen. Sie ermöglichen keine flexible Definition beliebiger Werttypen, wie sie analog für Objekttypen möglich ist. Ähnliche oder ähnlich benannte Sprachkonstrukte einiger Sprachen (z. B. *value types* in C#) unterstützen nicht das in Teil 1 der Arbeit beschriebene Konzept zustandsloser Typen, sondern übertragen ein für primitive Datentypen gängiges Implementierungsverfahren („Wertsemantik“) auf programmierer-definierte Typen.

Weil weder softwaretechnische Verfahren noch bestehende Sprachen Werte zufriedenstellend abbilden, schlug diese Arbeit eine dedizierte Sprachunterstützung für Werttypen vor: einen eigenen Typkonstruktor („*valueClass*“). Die Arbeit beschrieb die Anforderungen an einen solchen Typkonstruktor. Er muss ermöglichen, Werttypen klar, sicher und mit angemessenem Aufwand zu programmieren und sich in ein prinzipiell objektorientiertes, d. h. zustandsbehaftetes Umfeld einfügen. Die Arbeit spezifizierte eine mögliche Ausgestaltung für einen solchen Typkonstruktor und wies nach, dass er die Anforderungen erfüllt. Die Verschiedenheit von Werttypen und Objekttypen spiegelt sich auch in ihrer Sprachunterstützung wider. Beispielsweise wurde zusammen mit dem Werttypkonstruktor das Konzept der Selektoren eingeführt – Operationen, welche die Nicht-Erzeugung von Werten sowohl gewährleisten als auch im Code erkennen lassen. In einer Sprache, die über einen Werttypkonstruktor verfügt, kann jeder beliebige Werttyp mit sprach-eigenen Mitteln definiert

werden; vordefinierte, „primitive“ Datentypen werden dadurch im Prinzip verzichtbar.

Mit der Entwicklung der Forschungssprache VJ lieferte die Arbeit ein konkretes Beispiel für eine objektorientierte Sprache mit dedizierter Unterstützung für Werttypen und wies die Machbarkeit des Ansatzes nach. Mehrere in VJ programmierte Beispiele illustrierten die Vorteile eines eigenen Typkonstruktors für Werttypen.

Teil 3 der Arbeit befasste sich mit den Auswirkungen der Wert-/Objekt-Trennung. Die fundamentale Verschiedenheit von Werten und Objekten führt zu Umgestaltungen anderer Sprachkonstrukte; im einzelnen wurde das anhand von Gleichheit und anhand von Vererbung ausgeführt.

Die Arbeit zeigte, dass viele in objektorientierten Sprachen auftretende Schwierigkeiten im Umgang mit Gleichheit auf die Abbildung von Werttypen durch Objektklassen zurückzuführen sind. Zusätzlich zeigte die Arbeit, dass der in vielen Sprachen geforderte „Vertrag für Gleichheit“ nicht restriktiv genug ist und vielen Problemen Vorschub leistet. Die Arbeit präsentierte eine modifizierte Fassung des Gleichheitsvertrags. Dieses restriktivere Regelwerk vermeidet die beschriebenen Probleme. Der modifizierte Gleichheitsvertrag stellt eine eindeutige Spezifikation dar: Jeder Typ kann nur einen Vergleich haben, der diesen Vertrag erfüllt, ein Werttyp die Wertgleichheit, ein Objekttyp die Objektidentität. Die Trennung von Werttypen und Objekttypen und die eindeutige Beschreibung durch den modifizierten Gleichheitsvertrag ermöglichen, Wertgleichheit und Objektidentität als ein einheitliches Konzept aufzufassen und miteinander zu verschmelzen. Eine sprachliche Unterstützung dieses einheitlichen Konzeptes kann einen großen Teil der beschriebenen Fehlerquellen beseitigen und andere reduzieren. Vor allem beeinflusst dieses Sprachmodell die Sichtweise des Programmierers. Statt zwei Bausteinen für Vergleiche (Gleichheit und Identität) gibt es nur einen einzigen.

Auch die für objektorientierte Sprachen als essentiell angesehene Vererbung erfährt unter dem Einfluss der Wert-/Objekt-Trennung Veränderungen. Implementierungsvererbung induziert bei Wertklassen – anders als bei Objektklassen – keine Subtypbeziehung. In den untersuchten Fallbeispielen war auch der von der Subklasse geerbte Code nur in geringem Umfang verwendbar, teils sogar fehlerträchtig. Subtypbeziehungen gibt es auch zwischen Werttypen. Sie können statt durch Implementierungsvererbung durch einen anderen Mechanismus – in der Arbeit als Inklusionsfunktion bezeichnet – hergestellt werden. Mehrere Beispiele (u. a. das in der Einleitung genannte Kreis-Ellipsen-Dilemma und die Typisierung binärer Methoden) belegten, dass bei Wertklassen der Einsatz einer Inklusionsfunktion eine einfache und robuste Programmierung ermöglicht, während die in objektorientierten Sprachen gängige Anwendung von Vererbung zu unlösbaren Problemen führen würde. Eine zusätzliche, für viele praktische Anwendungsfälle interessante Einsatzmöglich-

keit dieses Mechanismus stellen wechselseitige Inklusionsfunktionen dar: Damit können zwei unabhängig voneinander programmierte Wertklassen, die denselben Typ abbilden, gegeneinander austauschbar gemacht werden. Angestoßen durch die Unterscheidung von Werttypen und Objekttypen gelangt die Arbeit zu einem Sprachmodell, das Subtyping für beide Arten von Typen unterstützt, aber jeweils unterschiedliche Mechanismen dafür bereit stellt: für Werttypen die Inklusionsfunktion, für Objekttypen Implementierungsvererbung. Diese Gestaltung vermeidet etliche in objektorientierten Sprachen auftauchende Probleme.

10.2 Offene Fragen

Der Einfluss der Wert-/ Objekt-Trennung auf andere Sprachkonstrukte wurde exemplarisch anhand von Gleichheit und von Vererbung untersucht. Die Zweiteilung des Typsystems kann sich auf weitere, in dieser Arbeit nicht näher untersuchte Sprachbausteine, auswirken und dabei etliche Fragen aufwerfen. Das betrifft z. B. folgende Sprachkonstrukte:

Operatoren: Einige Sprachen, z. B. Java, kennen eine Operatorschreibweise (z. B. die Infixoperatoren $+$, $-$, $*$, etc.) nur für vordefinierte Typen und damit nur für Werttypen. In anderen Sprachen können Operatoren für beliebige Typen definiert werden (z. B. in C++ mit operator overloading [Str10]). Wieder andere Sprachen betrachten Operatoren lediglich als syntaktische Sonderformen für Methodenaufrufe (z. B. Scala [Ode14]).

Es sollte untersucht werden, wie eine auf der Trennung von Werttypen und Objekttypen aufbauende Sprache programmierer-definierte Operatoren unterstützen und dabei auf Werttypen eingrenzen kann. Sie würde damit für mathematische Abstraktionen wie komplexe Zahlen oder Matrizen vertraute Notationen unterstützen, gleichzeitig aber verhindern, dass ein zu umfassender – auch Objekttypen einbeziehender – Gebrauch von Operatoren zu schwer verständlichen Code führt.

Benutzerdefinierte Literale: Die meisten Sprachen kennen Literale nur für vordefinierte Datentypen. In einigen Sprachen gibt es Ansätze, auch benutzerdefinierte Literale zu unterstützen (z. B. in C++11 [Str13]). Das Sprachmodell dieser Arbeit interpretiert die Literale der primitiven Datentypen als syntaktische Sonderform von Selektoren.

Daraus ergibt sich der Ansatz, benutzerdefinierte Literale für Wertklassen zuzulassen und sie für den Aufruf von Selektoren einzusetzen. Auf diese Weise kann die Benutzung von Wertklassen hinsichtlich ihrer Schreibweise der von primitiven Datentypen weiter angenähert werden.

Nullability: Viele Sprachen kennen einen speziellen Bezeichner, der ausdrückt, dass eine Variable an kein Element des zugehörigen Typs gebunden ist. Java, C#, C++ und Scala verwenden dafür das Schlüsselwort `null`. Die Variablen von Objekttypen (genauer: von „Referenztypen“) sind dort immer „nullable“, die Variablen primitiver Datentypen (die de facto Werttypen sind) dagegen können nicht `null` sein. (Hoare bezeichnet rückblickend das Design von „null references“ als „Billion Dollar Mistake“ [Hoa09].)

Die unterschiedliche Umgang mit `null` bei Referenztypen und bei primitiven Typen wirft etliche Fragen auf: Ist es gerechtfertigt, die „nullability“ von Referenztypen auf Objekttypen und die „non-nullability“ primitiver Typen auf Werttypen zu verallgemeinern? Gibt es Gründe für den gegenteiligen Ansatz, Werttypen und Objekttypen in dieser Hinsicht gleich zu behandeln, d. h. in beiden Fällen die Deklaration von Variablen als „nullable“ oder „non-nullable“ zuzulassen? Ist die Bildung eines eigenen generischen Typs (zu finden in vielen funktionalen Sprachen, z. B. der `Maybe`-Typ in Haskell, s. [Mar10]) eine adäquate Lösung für den Umgang mit `null`?

Parametrische Polymorphie/ Generizität: Viele Programmiersprachen – sowohl funktionale als auch objektorientierte Sprachen – unterstützen parametrische Polymorphie, d. h. die Definition von Typen mit Typparametern.

Die Wert-/ Objekt-Trennung wirft die Frage auf, ob sich diese Trennung auch auf Typparameter erstrecken soll und welche Folgerungen sich daraus ergeben. Kann z. B. ein Typparameter eines generischen Typs immer sowohl durch Werttypen als auch durch Objekttypen instanziiert werden? Sollte ein Sprachmechanismus vorgesehen werden, der es ermöglicht, Typparameter explizit auf Werttypen oder auf Objekttypen einzuschränken?

Interfaces: Interfaces – reine Schnittstellentypen – sind vorwiegend in objektorientierten Sprachen zu finden. Sie beschreiben auf abstrakter Ebene die Schnittstelle von Objekttypen anhand der zugehörigen Operationen.

Ist dieses Konzept auf Werttypen übertragbar? Falls ja: soll sich die strikte Trennung von Werttypen und Objekttypen auch auf Interfaces erstrecken? Oder soll es zugelassen werden, dass ein Interface sowohl von Werttypen als auch von Objekttypen implementiert wird? Sind Werttypen aufgrund ihres funktionalen Charakters statt mit Interfaces besser vereinbar mit Typklassen, wie sie in funktionalen Sprachen (z. B. von Haskell) unterstützt werden?

Funktionale Sprachen enthalten etliche Bausteine, die es in imperativen Sprachen meist nicht gibt. Werttypen sind per Definition rein funktionale Abstraktionen, und ein Sprachmodell, das Werttypen und Objekttypen gleichermaßen unterstützt,

bringt ein Stück funktionale Programmierung in ein objektorientiertes Umfeld. Daraus ergibt sich die Frage, ob es vorteilhaft ist, in der Folge auch weitere „typisch funktionale“ Sprachkonstrukte mit aufzunehmen, und ob sie dann nur in Kombination mit Werttypen verfügbar sein sollten oder auch auf Objekttypen ausgeweitet werden können. Das betrifft z. B.:

Typinferenz: Viele funktionale Sprachen (z. B. ML, Miranda, Haskell) leiten den Typ eines Ausdrucks aus seinen Bestandteilen oder seinem Kontext ab, sie machen auf diese Weise explizite Typannotationen weitgehend verzichtbar. Typinferenz hat auch Eingang in hybride Sprachen gefunden (z. B. in OCaml, F# und Scala), mittlerweile wurde sie auch in einige objektorientierte Sprachen aufgenommen (z. B. in C# und in C++11).

Ist Typinferenz auf Wert-Ausdrücke und auf Objekt-Ausdrücke gleichermaßen anwendbar? Kann die Information, dass einige in einem Ausdruck enthaltene Abstraktionen Werte sind (oder dass einige involvierte Operationen Funktionen sind) die Typermittlung beeinflussen, z. B. sie erleichtern?

Verzögerte Ausführung (laziness, non-strictness): Verzögerte Ausführung als impliziter Standard findet sich ausschließlich in rein funktionalen Sprachen (z. B. in Miranda [Tur85] und in Haskell [Hud+92; Jon03]). In imperativen oder hybriden Sprachen dagegen gibt es lediglich einzelne Sprachbausteine, die verzögerte Ausführung beinhalten (z. B. call-by-name-Parameter in Scala [Ode14]). Ein Grund dafür ist möglicherweise, dass die Kombination von „laziness“ mit imperativen Sprachkonstrukten Sprachdesigner und Programmierer vor erhebliche Schwierigkeiten stellt; z. B. ist damit die Ausführungsreihenfolge praktisch nicht mehr vorhersehbar (s. dazu Launchbury [Lau93]). Kann verzögerte Ausführung in eine Sprache, die auf der Wert-/Objekt-Trennung aufbaut, übertragen werden? Sollte sie dort auf Werttypen eingeschränkt werden?

Funktionen/ Operationen höherer Ordnung: Sollen ausschließlich *Funktionen* höherer Ordnung unterstützt werden, oder auch „Objektoperationen höherer Ordnung“? Sollen die an eine Operation höherer Ordnung übergebenen Parameter oder ihr Ergebnis auf Funktionen beschränkt werden, oder können es auch Objektoperationen sein? Ist es nützlich, bei der Definition einer Operation höherer Ordnung die Parameter oder das Ergebnis gezielt auf Funktionen (oder auf Objektoperationen) einzugrenzen?

Etliche objektorientierte oder hybride Sprachen unterstützen bereits Operationen höherer Ordnung (z. B. Smalltalk durch Blöcke [GR89], Scala durch function types [Ode14], Eiffel in der Fassung von 2006 durch agents [ECM06b], Java ab Version 8 durch Lambda-Ausdrücke [Gos+14]). Weil diese Sprachen

Werttypen und Objekttypen nicht trennen, und deshalb auch (zustandslose) Funktionen nicht von (zustandsbehafteten) Objektoperationen, haben sich derartige Fragen dort bisher nicht gestellt.

Die in Teil 3 der Arbeit beschriebenen Änderungen im Sprachentwurf und die hier aufgeworfenen Fragen vermitteln einen Eindruck davon, dass die Unterscheidung von Werttypen und Objekttypen weitreichende Auswirkungen nach sich zieht. Die hier begonnene Liste der potentiell betroffenen Sprachbausteine ist sicher nicht vollständig.

10.3 Ausblick

Aktuell ist wieder ein wachsendes Interesse an funktionaler Programmierung zu verzeichnen, und es sind Bestrebungen zu beobachten, sie mit der – bisher in der Praxis dominierenden – objektorientierten Programmierung zu verbinden. Indikatoren dafür sind Erweiterungen bestehender objektorientierter Sprachen um funktionale Konstrukte (Lambdas, Typinferenz, s. o.) sowie der Erfolg von hybriden, d. h. objekt-funktionalen Sprachen wie OCaml, F# oder Scala. Diese Sprachen enthalten eine Vielzahl von „typisch objektorientierten“ und „typisch funktionalen“ Konstrukten. Allerdings sieht keine dieser Sprachen eine Trennung von (funktionalen) Werttypen und (imperativen) Objekttypen vor.

Die vorliegende Arbeit verfolgte einen anderen Ansatz: Sie konzentrierte sich auf die grundlegenden Charakteristika der beiden Paradigmen, Zustandslosigkeit versus Zustand, und verankerte sie am Typsystem. Das Ziel der Arbeit war, eine erkannte Lücke in objektorientierten Sprachen zu schließen, die Ausdrucksfähigkeit des Sprachmodells zu verbessern und Fehlerquellen zu reduzieren.

Als spekulativer Ausblick soll abschließend der Gedanke in den Raum gestellt werden, dass sich die hier ausgearbeitete Trennung von Werttypen und Objekttypen als Basis für eine objekt-funktionale Sprache eignet. Eine auf einem zweigeteilten Typsystem aufbauende Sprache kann beide Paradigmen „pur“ unterstützen: Sie garantiert im funktionalen Kern rein funktionale Programmierung, und sie bildet in der objektorientierten Schale „real world objects“ und ihre Zustandsänderungen geradlinig ab.

Anhänge

Anhang A

Code-Beispiele und Spezifikationen zu Kapitel 6

A.1 Grammatik der Sprachunterstützung für Wertklassen

Die verwendete Grammatik setzt folgende EBNF-Symbole und -Konventionen ein:

Eine *Produktionsregel* enthält ein Nichtterminal, gefolgt von einem Doppelpunkt; der zugehörigen EBNF-Ausdruck folgt auf der nächsten Zeile:

Nonterminal:

...

Ein *Terminalsymbol* steht in Anführungszeichen und wird fett gedruckt:

"..."

Das Pipe-Zeichen trennt *Alternativen*:

... | ...

Geschweifte Klammern markieren *Wiederholung*, 0 bis mehrmals:

{ ... }

Eckige Klammern kennzeichnen *Optionen*, einmal oder gar nicht:

[...]

In Bezug auf Objektklassen beschränkt sich die Grammatik auf grundlegende Bestandteile (Membervariablen, Methoden, Konstruktoren).

Aus Vereinfachungsgründen wird auf Sichtbarkeitsmodifier, abstrakte Methoden, Typparameter, Innere Klassen, Packages, Arrays, Typumwandlungen und Typprüfungen verzichtet.

Ebenfalls nicht enthalten sind Produktionsregeln für Integer-, Floatingpoint- und Character-Literale sowie für Ziffern, Groß- und Kleinbuchstaben.

```

Class :
    ObjectClass | ValueClass

ValueClass :
    "valueClass" Typename "{" [Enumeration] [DataFields] [Functions] "}"

Enumeration :
    "enum" "{" ListOfEnumIdentifiers "}"

ListOfEnumIdentifiers :
    EnumIdentifier { "," EnumIdentifier }

DataFields :
    { DataFieldDeclaration }

DataFieldDeclaration :
    Typename VariableName ";"

Functions :
    { FunctionDefinition }

FunctionDefinition :
    Typename OperationName [ "(" FormalParameters ")" ] Block

FormalParameters :
    DeclarationOfFormalParameter { "," DeclarationOfFormalParameter }

DeclarationOfFormalParameter :
    Typename VariableName

Block :
    "{" DeclarationOfLocalVariable | Statement "}"

DeclarationOfLocalVariable :
    Typename VariableName [ "!=" Expression ] ";"

MultiwayBranchExpression :
    "switch" SimpleExpression
    { "case" ListOfEnumIdentifiers ":" SimpleExpression }
    [ "otherwise" ":" SimpleExpression ]

MultiwayBranchStatement :
    "switch" SimpleExpression
    { "case" ListOfEnumIdentifiers ":" Block }
    [ "otherwise" ":" Block ]

FunctionCall :
    OperationName [ "(" ActualParameters ")" ]

ActualParameters :
    Expression { "," Expression }

```

```

ObjectClass :
    "objectClass" Typename "{" [MemberVariables] [Methods] "}"

MemberVariables :
    { MemberVariableDefinition }

MemberVariableDefinition :
    Typename VariableName [ ":=" Expression ] ";"

Methods :
    { MethodDefinition | ConstructorDefinition }

MethodDefinition :
    (Typename | "void" ) OperationName "(" [FormalParameters] ")" Block

ConstructorDefinition :
    Typename "(" [FormalParameters] ")" Block

MethodCall :
    [SimpleExpression "."] OperationName "(" [ActualParameters] ")"

ObjectCreation
    "new" Typename "(" [ActualParameters] ")"

Statement :
    VariableName ":=" Expression ";"
    | "return" Expression ";"
    | "if" Expression "then" Block [ "else" Block ]
    | "while" Expression "do" Block
    | MultiwayBranchStatement
    | SimpleExpression "." VariableName ":=" Expression ";"
    | "return" ";"
    | MethodCall ";"

Expression :
    SimpleExpression
    | CompoundExpression

SimpleExpression :
    Literal
    | VariableName
    | SimpleExpression "." VariableName
    | FunctionCall
    | "(" CompoundExpression ")"
    | "select" "(" ActualParameters ")"
    | "this"
    | MethodCall
    | ObjectCreation

CompoundExpression :
    MultiwayBranchExpression
    | SimpleExpression { InfixOperator SimpleExpression }

```


InfixOperator:
 "=" | "!=" | "+" | "-" | "*" | "/" | "<" | ">" | "&&" | "||"

TypeName:
 CapitalLetter TailIdentifier

VariableName:
 (LowerCaseLetter | Underscore) TailIdentifier

OperationName:
 LowerCaseLetter TailIdentifier

EnumIdentifier:
 LowerCaseLetter TailIdentifier

TailIdentifier:
 { CapitalLetter | LowerCaseLetter | Digit | Underscore }

Literal:
 IntegerLiteral
 | FloatingpointLiteral
 | CharacterLiteral
 | BooleanLiteral

BooleanLiteral:
 "true" | "false"

Underscore:
 "_"

A.2 Beispiele für Wert-Aufzählungen und Objekt-Aufzählungen

Um zu illustrieren, ob und in welchen Fällen eine Sprachunterstützung für Aufzählungen nützlich ist, werden im Folgenden zwei Code-Beispiele in Java aufgeführt. Jedes Beispiel ist einmal mit Sprachunterstützung (mit dem Typkonstruktor `enum`) und einmal ohne Sprachunterstützung (nach dem Typsafe `enum` pattern) programmiert.

Das erste Beispiel zeigt die Definition eines Aufzählungs-Werttyps (Kartenfarbe), das zweite Beispiel die eines Aufzählungs-Objektyps (Planet).

Beispiel 1a:

Definition des Aufzählungs-Werttyps Kartenfarbe mit Sprachunterstützung, mit dem Typkonstruktor `enum`:

```
enum Kartenfarbe {  
    KREUZ, PIK, HERZ, KARO  
}
```

Beispiel 1b:

Definition des Aufzählungs-Werttyps Kartenfarbe ohne Sprachunterstützung, mit dem typesafe-enum Pattern (s. auch Abschnitt 4.2.3):

```
class Kartenfarbe{  
    private Kartenfarbe() {}  
    public static final Kartenfarbe KREUZ = new Kartenfarbe();  
    public static final Kartenfarbe PIK = new Kartenfarbe();  
    public static final Kartenfarbe HERZ = new Kartenfarbe();  
    public static final Kartenfarbe KARO = new Kartenfarbe();  
}
```

Die Definition des Aufzählungs-Werttyps Kartenfarbe ist mit Sprachunterstützung kürzer, klarer und leichter verständlich als durch Einsatz von Programmier-techniken. Außerdem wirkt bei der zweiten Variante der Konstruktorauf-
ruf („`new Kartenfarbe()`“) verwirrend, weil Werte konzeptionell nicht erzeugt werden.

Beispiel 2a:

Definition des Aufzählungs-Objektyps Planet mit Sprachunterstützung, mit dem Typkonstruktor enum:

```
enum Planet {  
    MERCURY(0),  
    VENUS (0),  
    EARTH(8232486703),  
    MARS(0);  
    ...  
    private long inhabitants;  
    Planet(long inhabitants) {  
        this.inhabitants = inhabitants;  
    }  
    public long getInhabitants() { return inhabitants;}  
    public void setInhabitants(long inhabitants) {  
        this.inhabitants = inhabitants;  
    }  
}
```

Beispiel 2b:

Definition des Aufzählungs-Objektyps Planet ohne Sprachunterstützung, mit dem typesafe-enum Pattern (s. auch Abschnitt 4.2.3):

```
class Planet {  
    public static final Planet MERCURY = new Planet(0);  
    public static final Planet VENUS = new Planet(0);  
    public static final Planet EARTH = new Planet(8232486703);  
    public static final Planet MARS = new Planet(0);  
    ...  
    private long inhabitants;  
    Planet(long inhabitants) {  
        this.inhabitants = inhabitants;  
    }  
    public long getInhabitants () { return inhabitants; }  
    public void setInhabitants(long inhabitants) {  
        this.inhabitants = inhabitants;  
    }  
}
```

Beim Aufzählungs-Objektyp Planet ist der Code mit Sprachunterstützung nur unwesentlich kürzer als mit Einsatz des typesafe-enum-Patterns, der Programmieraufwand ist für beide Varianten annähernd gleich hoch.

Die Verwendung des Typkonstruktors enum erspart die Deklaration der einzelnen Elemente als Konstanten („public static final Planet“) und den jeweils zugehörigen Aufruf des Konstruktors („new Planet ...“). Die relative Ersparnis ist gering, weil Exemplarvariablen und zusätzliche Operationen einen erheblichen Teil des Codes ausmachen.

Das typesafe-enum-Pattern hat den Vorteil, dass eine Objektklasse zum Einsatz kommt und so deutlich macht, dass die Elemente des Typs Objekte sind.

Aufzählungen von Objekten können mit geringem Aufwand und ohne Einbuße an konzeptioneller Klarheit mit den vorhandenen Sprachmitteln programmiert werden. Das lässt es vertretbar erscheinen, bei Objekttypen auf eine dedizierte Sprachunterstützung für Aufzählungen zu verzichten.

Anhang B

Code-Beispiele und Spezifikationen zu Kapitel 7

B.1 Beispiel für eine mit VJ programmierte Wertklasse: Date

Das folgende Code-Beispiel fasst die in Abschnitt 7.2.1 präsentierten Codefragmente zusammen und listet die vollständige VJ-Wertklasse Date auf.

```
valueclass Date {  
    int year;  
    int month;  
    int day;  
  
    public int dayOfYear() {  
        int result = (this.month - 1) * 30 + this.day;  
        // hier Monate mit 31 Tagen berücksichtigen  
        // sowie Februar und Schaltjahre  
        if (this.month == 2) { result = ... }  
        ...  
        return result;  
    }  
  
    public static dateYYYYDDD(int year, int dayOfYear) {  
        int month1 = ... // Monat ermitteln  
        int day1 = ... // Tag ermitteln  
        return const Date(year, month1, day1);  
    }  
  
    public static boolean isValid (int year,  
                                   int month,  
                                   int day) {  
        if (day < 1 || day > 31) {  
            return false;  
        }  
        ...  
        if ((month == 4 || month == 6 ..) && day == 31) {  
            return false;  
        }  
        ...  
    }  
}
```

B.2 Beispiel: VJ-Wertklasse/ vom VJ-Compiler erzeugter Java-Code

Das folgende Code-Beispiel illustriert anhand der Wertklasse `Date` die in Abschnitt 7.3.4 beschriebene Übersetzung einer VJ-Wertklasse in eine Java-Klasse, d. h. das Ergebnis nach Schritt 7 des VJ-Compilers.

Weil die Code-Transformation im Vordergrund steht, sind diejenigen Methoden aus Abschnitt 7.2.1 weggelassen, für die der VJ-Compiler keine Veränderung vornehmen würde. Um die Generierung der Methode `isValid` zu illustrieren, ist hier in der Wertklasse `Date` keine solche Methode programmiert.

Aus der VJ-Wertklasse

```
valueclass Date {  
    int year;  
    int month;  
    int day;  
}
```

generiert der VJ-Compiler gemäß Spezifikation folgenden Java-Code:

```
final class Date extends vj.lang.Value {  
    public final int year;  
    public final int month;  
    public final int day;  
  
    public Date(int year, int month, int day) {  
        if (!isValid(year, month, day)) {  
            throw new NoSuchValueException();  
        }  
        this.year = year;  
        this.month = month;  
        this.day = day;  
    }  
  
    public static boolean isValid (int year,  
                                   int month,  
                                   int day) {  
        return true;  
    }  
  
    public boolean equals(Object o) {  
        if (o instanceof Date) {  
            return this.year == ((Date) o).year  
                &&    this.month == ((Date) o).month  
                &&    this.day == ((Date) o).day;  
        } else {  
            return false;  
        }  
    }  
  
    public int hashCode() {  
        int result = 17;  
        result = 37*result + year;  
        result = 37*result + month;  
        result = 37*result + day;  
        return result;  
    }  
}
```


Implementation der (Compiler-internen) Klasse `vj.lang.Value`:

```
package vj.lang;

/**
 * Base class for all value classes.
 */
public abstract class Value {

    @Override
    final protected Object clone()
        throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }

    @Override
    final protected void finalize() {
    }

    @Override
    public String toString() {
        return "";
    }

    public static final boolean equal(Value v1, Value v2) {
        if (v1 == null && v2 == null) {
            return true;
        } else if (v1 == null && v2 != null) {
            return false;
        } else if (v1 != null && v2 == null) {
            return false;
        } else {
            return v1.equals(v2);
        }
    }

    /**
     * String must get an equals-method on its own
     * because it is considered a value type
     * but the String class does not inherit from Value
     */
    public static final boolean equal(String v1, String v2) {
        if (v1 == null && v2 == null) {
            return true;
        } else if (v1 == null && v2 != null) {
            return false;
        } else if (v1 != null && v2 == null) {
            return false;
        } else {
            return v1.equals(v2);
        }
    }
}
```

B.3 Beispiel: Klientencode/ vom VJ-Compiler erzeugter Java-Code

Wertwähler-Aufruf im Klienten-Code

Der Aufruf des Wertwählers von Date:

```
myDate = const Date (2006, 5, 31);
```

wird vom VJ-Compiler umgewandelt in:

```
myDate = new Date (2006, 5, 31);
```

Gleichheitsprüfung im Klienten-Code

Der Vergleich zweier Ausdrücke, deren statischer Typ eine Wertklasse oder die Klasse `java.lang.String` ist, hier vom Typ `Date`:

```
Date d1; Date d2;  
...  
if (d1 == d2) { ... }
```

wird vom VJ-Compiler umgewandelt in:

```
Date d1; Date d2;  
...  
if (vj.lang.Value.equal(d1,d2)) { ... }
```

Der Vergleich zweier Ausdrücke, deren statischer Typ eine Aufzählung oder ein primitiver Datentyp ist, hier vom Typ `Himmelsrichtung`:

```
enum Himmelsrichtung {NORD, SUED, OST, WEST}
...
Himmelsrichtung h1; Himmelsrichtung h2;
...
if (h1 == h2) { ... }
```

wird vom VJ-Compiler nicht verändert, d. h. der umgewandelte Code stimmt mit dem VJ-Quelltext überein.

Der Vergleich zweier Ausdrücke vom statischen Typ `Object` oder eines Wertinterface:

```
Object o1; Object o2;
...
if (o1 == o2) { ... }
```

wird vom VJ-Compiler umgewandelt in:

```
Object o1; Object o2;
...
if ( ( (o1 instanceof vj.lang.Value)
      && (o2 instanceof vj.lang.Value))
    ? o1.equals(o2)
    : o1 == o2) { ... }
```

Mit dieser Übersetzung funktionieren auch die Fälle korrekt, bei denen zur Übersetzungszeit nicht festgestellt werden kann, ob Werttypen oder Objekttypen vorliegen.

Der Ergebnis-Code vermeidet auch `NullPointerExceptions`, weil der Operator `instanceof` für `null` das Ergebnis `false` liefert und damit den Aufruf von `equals` verhindert.

B.4 Testfälle für den VJ-Compiler

Die nachfolgenden Anwendungsbeispiele illustrieren das Verhalten des Typkonstruktors für Werttypen („valueclass“) und die Funktionsweise des VJ-Compilers.

Die Anwendungsbeispiele bilden folgende Werttypen ab:

`Money`

Ein Typ für Geldbeträge, bestehend aus einem Betrag und einer Währung.

`Complex`

Ein Typ für komplexe Zahlen und ihre Rechenoperationen.

`PositiveInteger`

Ein Typ für positive ganze Zahlen.

`Range`

Ein Typ für geschlossene Intervalle über Gleitkommazahlen.

Für jedes Anwendungsbeispiel sind aufgeführt:

- eine in VJ programmierte Wertklasse für den jeweiligen Werttyp,
- der Java-Code, den der VJ-Compiler aus der Wertklasse generiert,
- VJ-Klientencode, der die jeweilige VJ-Wertklasse benutzt,
- der Java-Code, den der VJ-Compiler aus dem Klientencode generiert.

Anwendungsbeispiel Money

VJ-Code Wertklasse:

```
package de.uni_hamburg.informatik.vjexamples.money;

public valueclass Money {
    int amount;
    public String currency;

    public Money add(Money m) {
        if (!sameCurrency(m))
            throw new RuntimeException("Currency mismatch");
        return const Money(this.amount + m.amount, currency);
    }

    public boolean sameCurrency(Money m) {
        return this.currency == m.currency;
    }
}
```

vom VJ-Compiler daraus generierter Java-Code:

```
package de.uni_hamburg.informatik.vjexamples.money;
public final class Money extends vj.lang.Value{
    final int amount;
    public final String currency;
    public Money add(Money m){
        if (!(sameCurrency(m))) throw new RuntimeException("Currency mismatch");
        return new Money((this.amount)+(m.amount), currency);
    }
    public boolean sameCurrency(Money m){
```

```
        return vj.lang.Value.equal(this.currency, m.currency);
    }
    public Money(int amount, String currency){
        (this.amount)=(amount);
        (this.currency)=(currency);
    }
    public final boolean equals(Object other){
        if ((other) instanceof Money) {
            Money otherValue = (Money) (other);
            if ((amount)!=(otherValue.amount)) return false;
            if (!(currency.equals(otherValue.currency))) return false;
            return true;
        }
        else {
            return false;
        }
    }
}
```

VJ-Klientencode:

```
package de.uni_hamburg.informatik.vjexamples.money;

public class MoneyUser {
    public static void main(String[] args) {
        Money m = const Money(10, "EUR");
        System.out.println(m.toString());
    }
}
```

vom VJ-Compiler daraus generierter Java-Code:

```
package de.uni_hamburg.informatik.vjexamples.money;
public class MoneyUser {
    public static void main(String[] args){
        Money m = new Money(10, "EUR");
        System.out.println(m.toString());

    }

}
```

Anwendungsbeispiel Complex

VJ-Code Wertklasse:

```
package de.uni_hamburg.informatik.vjexamples.math;
import java.lang.Math;

public valueclass Complex {
    public double real;
    public double imaginary;

    public Complex add(Complex valAdd) {
        return const Complex(this.real + valAdd.real,
            this.imaginary + valAdd.imaginary);
    }

    public Complex subtract(Complex valSub) {
        return const Complex(this.real - valSub.real,
            this.imaginary - valSub.imaginary);
    }

    public Complex multiply(Complex val) {
        return const Complex(
            this.real * val.real - this.imaginary * val.imaginary,
            this.imaginary * val.real + this.real * val.imaginary);
    }

    public Complex divide(Complex val) {
        return multiply(val.reciprocal());
    }

    public Complex reciprocal() {
        double denominator = (real * real) + (imaginary * imaginary);
        return const Complex(real / denominator, -imaginary / denominator);
    }
}
```



```

    }

    public double magnitude() {
        return Math.hypot(real, imaginary);
    }
}

```

vom VJ-Compiler daraus generierter Java-Code:

```

package de.uni_hamburg.informatik.vjexamples.math;
import java.lang.Math;
public final class Complex extends vj.lang.Value{
    public final double    real;
    public final double    imaginary;
    public Complex add(Complex valAdd){
        return new Complex((this.real)+(valAdd.real), (this.imaginary)+(valAdd.imaginary));
    }
    public Complex subtract(Complex valSub){
        return new Complex((this.real)-(valSub.real), (this.imaginary)-(valSub.imaginary));
    }
    public Complex multiply(Complex val){
        return new Complex(((this.real)*(val.real))-((this.imaginary)*(val.imaginary)),
                           ((this.imaginary)*(val.real))+((this.real)*(val.imaginary)));
    }
    public Complex divide(Complex val){
        return multiply(val.reciprocal());
    }
    public Complex reciprocal(){
        double denominator = ((real)*(real))+((imaginary)*(imaginary));

```

```
        return new Complex((real)/(denominator), -(imaginary)/(denominator));
    }
    public double magnitude(){
        return Math.hypot(real, imaginary);
    }
    public Complex(double real, double imaginary){
        (this.real)=(real);
        (this.imaginary)=(imaginary);
    }
    public final boolean equals(Object other){
        if ((other) instanceof Complex) {
            Complex otherValue = (Complex) (other);
            if ((real)!=(otherValue.real)) return false;
            if ((imaginary)!=(otherValue.imaginary)) return false;
            return true;
        }
        else {
            return false;
        }
    }
}
```

VJ-Klientencode:

```
package de.uni_hamburg.informatik.vjexamples.math;

public class ComplexUser {

    public static void main(String[] args) throws Exception {
        Complex c1 = const Complex(1.2, 2.3);
        Complex c2 = const Complex(1.2, 2.3);
        boolean same1 = (c1 == c2);
        String ausgabe = (same1? "gleich" : "nicht gleich");
        System.out.println("c1 und c2 sind " + ausgabe);
    }
}
```

vom VJ-Compiler daraus generierter Java-Code:

```
package de.uni_hamburg.informatik.vjexamples.math;
public class ComplexUser {
    public static void main(String[] args) throws Exception{
        Complex c1 = new Complex(1.2, 2.3);
        Complex c2 = new Complex(1.2, 2.3);
        boolean same1 = vj.lang.Value.equal(c1, c2);
        String ausgabe = (same1) ? ("gleich") : ("nicht gleich");
        System.out.println(("c1 und c2 sind ")+(ausgabe));

    }

}
```

Anwendungsbeispiel PositiveInteger

VJ-Code Wertklasse:

```
package de.uni_hamburg.informatik.vjexamples.math;

public valueclass PositiveInteger {
    public int value;

    public static boolean isValid(int value) {
        return value >= 0;
    }
}
```

vom VJ-Compiler daraus generierter Java-Code:

```
package de.uni_hamburg.informatik.vjexamples.math;

import vj.lang.NoSuchValueException;

public final class PositiveInteger extends vj.lang.Value{
    public final int value;
    public static boolean isValid(int value){
        return (value)>=(0);
    }
    public PositiveInteger(int value){
        if (!isValid(value)) {
            throw new NoSuchValueException();
        }
        (this.value)=(value);
    }
}
```

```
public final boolean equals(Object other){
    if ((other) instanceof PositiveInteger) {
        PositiveInteger otherValue = (PositiveInteger) (other);
        if ((value) != (otherValue.value)) return false;
        return true;

    }
    else {
        return false;
    }
}
}
```

VJ-Klientencode:

```
package de.uni_hamburg.informatik.vjexamples.math;

public class PositiveIntegerUser {

    public static void main(String[] args) throws Exception {
        String ausgabe;
        PositiveInteger p1 = const PositiveInteger(7);
        PositiveInteger p2 = const PositiveInteger(-1); // throws NoSuchElementException
        PositiveInteger p3 = const PositiveInteger(7);

        ausgabe = (p1==p2 ? "gleich" : "nicht gleich");
        System.out.println("p1 und p2 sind " + ausgabe);
        ausgabe = (p1==p3 ? "gleich" : "nicht gleich");
        System.out.println("p1 und p2 sind " + ausgabe);
    }
}
```

vom VJ-Compiler daraus generierter Java-Code:

```
package de.uni_hamburg.informatik.vjexamples.math;
public class PositiveIntegerUser {
    public static void main(String[] args) throws Exception{
        String ausgabe;
        PositiveInteger p1 = new PositiveInteger(7);
        PositiveInteger p2 = new PositiveInteger(-(1));
        PositiveInteger p3 = new PositiveInteger(7);
        (ausgabe)=((vj.lang.Value.equal(p1, p2)) ? ("gleich") : ("nicht gleich"));
        System.out.println("p1 und p2 sind "+(ausgabe));
        (ausgabe)=((vj.lang.Value.equal(p1, p3)) ? ("gleich") : ("nicht gleich"));
        System.out.println("p1 und p2 sind "+(ausgabe));

    }
}
```

Anwendungsbeispiel Range

VJ-Code Wertklasse:

```
package de.uni_hamburg.informatik.vjexamples.math;

public valueclass Range {
    public float start;
    public float end;

    public boolean isEmpty() {
        return start > end;
    }

    public float length() {
        if (this.isEmpty()) {
            return 0;
        } else {
            return (end - start);
        }
    }

    public boolean includes(float value) {
        return start <= value && value <= end;
    }

    public boolean includes(Range range) {
        if (range.isEmpty()) {
            return true;
        } else {
            return this.start <= range.start
                && range.end <= this.end;
        }
    }
}
```



```

    public boolean overlaps(Range range) {
        return this.includes(range.start)
            || this.includes(range.end)
            || range.includes(this.start)
            || range.includes(this.end);
    }

    public Range intersect(Range range) {
        float startR = (this.start > range.start ? this.start : range.start);
        float endR = (this.end < range.end ? this.end : range.end);
        return new Range(startR, endR);
    }

    public String toString() {
        return "Range (" + this.start + ", " + this.end + ") ";
    }
}

```

vom VJ-Compiler daraus generierter Java-Code:

```

package de.uni_hamburg.informatik.vjexamples.math;
public final class Range extends vj.lang.Value{
    public final float start;
    public final float end;
    public boolean isEmpty(){
        return (start)>(end);
    }
    public float length(){
        if ((this).isEmpty()) {
            return 0;
        }
    }
}

```

```

    }
    else {
        return (end)-(start);
    }

}

public boolean includes(float value){
    return ((start)<=(value))&&((value)<=(end));
}

public boolean includes(Range range){
    if (range.isEmpty()) {
        return true;
    }
    else {
        return ((this.start)<=(range.start))&&((range.end)<=(this.end));
    }
}

}

public boolean overlaps(Range range){
    return (((this).includes(range.start))||((this).includes(range.end)))
        ||(range.includes(this.start))||(range.includes(this.end));
}

}

public Range intersect(Range range){
    float startR = ((this.start)>(range.start)) ? (this.start) : (range.start);
    float endR = ((this.end)<(range.end)) ? (this.end) : (range.end);
    return new Range(startR, endR);
}

}

```

```
public String toString(){
    return (((("Range(")+(this.start))+(", "))+ (this.end))+(") "));
}
public Range(float start, float end){
    (this.start)=(start);
    (this.end)=(end);
}
public final boolean equals(Object other){
    if ((other) instanceof Range) {
        Range otherValue = (Range) (other);
        if ((start)!= (otherValue.start)) return false;
        if ((end)!= (otherValue.end)) return false;
        return true;
    }
    else {
        return false;
    }
}
}
```

VJ-Klientencode:

```
package de.uni_hamburg.informatik.vjexamples.math;

public class RangeUser {
    public static void main(String[] args) throws Exception{
        Range r1 = const Range (1.1f, 3.3f);
        Range r2 = const Range (2.2f, 4.4f);
        Range r3 = const Range (2.2f, 3.3f);
        Range r4 = r1.intersect(r1);
        Range r5 = r1.intersect(r2);

        if(r1.includes(1.1f)) {
            System.out.println(r1 + " enthält " + 1.1f);
        } else {
            System.out.println("error");
        }

        if(r1==r4) {
            System.out.println("Intersection von " + r1 + " mit sich selbst ergibt wieder " + r1);
        } else {
            System.out.println("error");
        }

        if(r5==r3) {
            System.out.println("Intersection von " + r1 + " und " + r2 + " ergibt " + r3);
        } else {
            System.out.println("error");
        }
    }
}
```

vom VJ-Compiler daraus generierter Java-Code:

```
package de.uni_hamburg.informatik.vjexamples.math;

public class RangeUser {
    public static void main(String[] args) throws Exception{
        Range r1 = const Range (1.1f, 3.3f);
        Range r2 = const Range (2.2f, 4.4f);
        Range r3 = const Range (2.2f, 3.3f);
        Range r4 = r1.intersect(r1);
        Range r5 = r1.intersect(r2);

        if(r1.includes(1.1f)) {
            System.out.println(r1 + " enthält " + 1.1f);
        } else {
            System.out.println("error");
        }

        if(r1==r4) {
            System.out.println("Intersection von " + r1 + " mit sich selbst ergibt wieder " + r1);
        } else {
            System.out.println("error");
        }

        if(r5==r3) {
            System.out.println("Intersection von " + r1 + " und " + r2 + " ergibt " + r3);
        } else {
            System.out.println("error");
        }
    }
}
```


Anhang C

Code-Beispiele und Spezifikationen zu Kapitel 8

C.1 Vertrag für Gleichheit in Java

Die Java-API-Dokumentation spezifiziert in der Klasse `java.lang.Object` für die Methode `equals` den folgenden Vertrag („equality contract“ [Ora14]):

The `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive*: for any non-null reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any non-null reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any non-null reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

C.2 Beispiel: Implementierung von equals in Vererbungshierarchien

Die Probleme, die die Implementierung von `equals` aufwirft, sollen anhand eines vereinfachten Java-Beispiels illustriert werden, einer Klasse `Point`, die Punkte im zweidimensionalen Raum darstellt:

```
class Point {  
    float x;  
    float y;  
    ...  
}
```

und einer Klasse `Colorpoint`, die von `Point` erbt und zusätzlich zu den Raumkoordinaten eine Farbe besitzt:

```
class Colorpoint extends Point {  
    Color c;  
    ...  
}
```

Eine naive Implementation der Gleichheit sieht folgendermaßen aus:

Für Point vergleicht sie die beiden Datenfelder x und y, für Colorpoint zusätzlich das Feld c.

Code-Variante 1

Gleichheit für die Klasse Point:

```
public boolean equals(Point other) {  
    return this.x == other.x  
        && this.y == other.y;  
}
```

Gleichheit für die Klasse Colorpoint:

```
public boolean equals(Colorpoint other) {  
    return this.x == other.x  
        && this.y == other.y  
        && this.c == other.c;  
}
```

Diese beiden Implementationen von `equals` sind kurz und übersichtlich, aber falsch, weil sie `equals` überladen statt zu überschreiben. Die Signatur von `equals` ist von der Wurzelklasse `Object` vorgegeben, ihr Parameter muss ebenfalls vom Typ `Object` sein.

Innerhalb der Implementation von `equals` ist deshalb der Typ des Übergabeparameters zu prüfen und ggf. in den passenden Subtyp umzuwandeln. Das macht den Code länger und unübersichtlicher.

Code-Variante 2

Gleichheit für die Klasse `Point`:

```
public boolean equals(Object o) {  
    if (!(o instanceof Point))  
        return false;  
    Point p = (Point)o;  
    return (p.x == this.x && p.y == this.y);  
}
```

Gleichheit für die Klasse `Colorpoint`:

```
public boolean equals(Object o) {  
    if (!(o instanceof Colorpoint))  
        return false;  
    Colorpoint cp = (Colorpoint)o;  
    return (super.equals(cp) && cp.color == this.color);  
}
```

Durch das Erben von `Point` wird `Colorpoint` zu einem Subtyp vom `Point`. Damit besteht auch die Möglichkeit, `Points` und `Colorpoints` miteinander zu vergleichen.

Der folgende Anwendungscode zeigt, dass die in Code-Variante 2 definierten Implementationen gegen die Symmetrie verstoßen:

```
Point p1 = new Point(1, 2);  
Colorpoint p2 = new Colorpoint(1, 2, Color.RED);  
... p1.equals(p2) ... // true  
... p2.equals(p1) ... // false - fehlende Symmetrie!
```

Um das zu beheben, wird die Gleichheit für `ColorPoint` erweitert. (Die Gleichheit für die Klasse `Point` bleibt wie in Variante 2.)

Code-Variante 3

Gleichheit für die Klasse `Colorpoint`:

```
public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;
    // if o is a normal Point, do a color-blind comparison:
    if (!(o instanceof Colorpoint))
        return o.equals(this);
    // o is a Colorpoint; do a full comparison:
    Colorpoint cp = (Colorpoint)o;
    return (super.equals(cp) && cp.color == this.color);
}
```

Wie der folgende Anwendungscode zeigt, verstößt die in Variante 3 definierte Implementation gegen die Transitivität:

```
Point p1 = new Point(1, 2);
Colorpoint p2 = new Colorpoint(1, 2, RED);
Colorpoint p3 = new Colorpoint(1, 2, BLUE);
... p1.equals(p2) ... // true
... p2.equals(p3) ... // true
... p1.equals(p3) ... // false - fehlende Transitivität!
```

Es ist aufgrund der Vererbungsbeziehung zwischen `Point` und `Colorpoint` problematisch, `equals` als Äquivalenzrelation zu programmieren.

Gemäß Bloch ist das ein prinzipielles Problem [Blo08, p. 38]. Demnach ist es nicht möglich, die Methode `equals` in einer Subklassen so zu implementieren, dass sie

1. eine Äquivalenzrelation bildet und
2. gemischte Vergleiche ermöglicht und
3. die zusätzlichen Eigenschaften der Subklasse beim Vergleichen einbezieht.

C.3 Beispiel: Gleichheit abhängig von änderbarem Zustand

Das folgende Java-Codebeispiel veranschaulicht, wie eine Gleichheit, die von änderbarem Zustand abhängt, inkonsistentes Programmverhalten hervorrufen kann.

Das Beispiel geht zurück auf einen Beitrag von Odersky et al. („How to Write an Equality Method ...“). Eine Java-Fassung findet sich in [OSV09], eine Scala-Fassung in [OSV11, p. 657].

Vorausgesetzt wird eine Klasse `Point` mit änderbaren Koordinaten, deren Methoden `equals` und `hashCode` gemäß den Anforderungen des Gleichheitsvertrags und der Hashcode-Regel umgesetzt sind.

```
public class Point {
    private int x;    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x) { this.x = x; }
    public void setY(int y) { this.y = y; }

    @Override public boolean equals(Object other) {
        boolean result = false;
        if (other instanceof Point) {
            Point that = (Point) other;
            result = (this.getX() == that.getX()
                      && this.getY() == that.getY());
        }
        return result;
    }
    @Override public int hashCode() { ... }
}
```

Die Gleichheit prüft die Übereinstimmung der beiden Koordinaten und hängt damit von änderbarem Zustand ab – was gemäß Gleichheitsvertrag zulässig ist.

Das folgende Anwendungsprogramm fügt einen Point `p` in ein HashSet `coll` ein. Der anschließende Test, ob `p` im HashSet `coll` enthalten ist, liefert (erwartungsgemäss) `true`.

Nachdem `p` verändert wurde (Erhöhen der x-Koordinate um 1) zeigt sich unerwartetes, in sich widersprüchliches Verhalten:

Das dem HashSet hinzugefügte Element ist nicht darin zu finden: `coll.contains(p)` ergibt `false` (meistens(!) – es ist prinzipiell möglich, dass die Prüfung `true` ergibt).

Iteriert man dagegen über das HashSet, taucht `p` darin auf: `containedP` ergibt `true`.

```
Point p = new Point(1, 2);
HashSet<Point> coll = new HashSet<Point>();
coll.add(p);
System.out.println(coll.contains(p));    // prints true

p.setX(p.getX() + 1);
System.out.println(coll.contains(p));    // prints false
                                         // (probably)

Iterator<Point> it = coll.iterator();
boolean containedP = false;
while (it.hasNext()) {
    Point nextP = it.next();
    if (nextP.equals(p)) {
        containedP = true;
        break;
    }
}
System.out.println(containedP);          // prints true
```

C.4 Beispiel: Unterscheidbarkeit gleicher Objekte

Dieses Java-Beispiel verdeutlicht, wie es zu inkonsistentem Verhalten führen kann, wenn gleiche Objekte anhand eines – nicht in die Gleichheit einfließenden – Merkmals unterschieden werden können.

Für das Beispiel wird vorausgesetzt, dass es eine Klasse „Fahrzeug“ gibt, und dass ihre Gleichheit auf Hersteller und Fabrikat beruht: Zwei Fahrzeuge werden als „gleich“ angesehen, wenn Hersteller und Fabrikat übereinstimmen.

Klasse Fahrzeug:

```
class Fahrzeug {
    String hersteller;
    String fabrikat ;
    String besitzer;

    public Fahrzeug (String hersteller,
                    String fabrikat,
                    String besitzer) {
        this.hersteller = hersteller;
        this.fabrikat = fabrikat;
        this.besitzer = besitzer;
    }

    public boolean equals(Object other) {
        if (!(other instanceof Fahrzeug))
            return false;
        Fahrzeug f = (Fahrzeug) other;
        return hersteller.equals(f.hersteller)
            && fabrikat.equals(f.fabrikat);
    }
    ...
}
```

Ein Anwendungsprogramm fügt die Fahrzeuge in eine Menge (`setF`) ein. Danach erstellt es eine Liste aller Fahrzeugbesitzer, durch Iterieren über diese Menge und Drucken des jeweiligen Besitzers.

Chris und Kim fahren das gleiche Fahrzeug: einen Audi A6.

Anwendungs-Code:

```
Set<Fahrzeug> setF = new HashSet<Fahrzeug>();

Fahrzeug fChris = new Fahrzeug("Audi", "A6", "Chris");
setF.add(fChris);

Fahrzeug fKim = new Fahrzeug("Audi", "A6", "Kim");
setF.add(fKim);

...

for (Fahrzeug fahrzeug : setF) {
    System.out.println(fahrzeug.besitzer);
}
```

Als Ergebnis wird nur einer der beiden Fahrzeugbesitzer, Chris *oder* Kim, ausgegeben. Wie das Resultat im Einzelfall aussieht, ist anhand des Quellcodes nicht vorhersagbar.

Die Menge `setF` enthält nur eines der beiden Fahrzeuge `fChris` und `fKim`, da die beiden gemäß `equals`-Methode gleich sind. Welches der beiden Fahrzeuge in der Menge enthalten ist, hängt von der Reihenfolge ab, in der sie eingefügt wurden und von dem Verfahren, mit dem die Menge ihre Elemente verwaltet (z. B. davon, ob bei gleichen Elementen das zuerst oder das zuletzt hinzugefügte in der Menge verbleibt). Das wiederum ist ein Implementationsdetail der Klasse `Set` und sollte für Klienten keine Rolle spielen.

Ursache des unvorhersehbaren Verhaltens ist das Vorhandensein eines Merkmals (hier: `Besitzer`), das auch bei „gleichen“ Objekten verschieden sein kann.

Anhang D

Code-Beispiele und Spezifikationen zu Kapitel 9

D.1 Beispiel Inklusionsfunktion: Character/ String

Das Beispiel zeigt, wie ein Werttyp für Zeichen als Subtyp eines Werttyps für Zeichenketten programmiert wird, und es illustriert Anwendungscode, der von dieser Subtypbeziehung Gebrauch macht.

Es wird vorausgesetzt, dass es einen Werttyp für Zeichen (`Character`) und einen Werttyp für Zeichenketten (`String`) gibt, und – der Einfachheit halber – dass die Elemente von `Character` mit Literalen wie `'a'` und die Elemente von `String` mit Literalen wie `"xyz"` angesprochen werden können.

`Character` bildet einen Subtyp von `String`, wenn ein Zeichen als eine Zeichenkette der Länge 1 aufgefasst wird. Z. B. wird das Zeichen `'a'` mit der Zeichenkette `"a"` identifiziert, etc.

Ist `stringForChar` ein Selektor der Wertklasse `String`, der zu einem `Character` den `String` liefert, der aus diesem Zeichen besteht, dann wird eine Subtyp-Beziehung zwischen `Character` und `String` durch die folgende Inklusionsfunktion hergestellt:

```
String include(Character c) { return stringForChar(c); }
```

Auf diese Weise stehen alle `String`-Operationen auch für Ausdrücke vom Typ `Character` zur Verfügung. Es sind auch gemischte Aufrufe mit `String`- und `Character`-Parametern möglich, ohne dass sie gesondert programmiert werden müssen, z. B. `String`-Verkettung (`concat`), lexikografischer Vergleich (`compare`) oder Anfang-/ Ende-Erkennung (`startsWith`, `endsWith`).

Der folgende Beispielcode zeigt, wie `String`-Operationen ohne Zusatzaufwand auch für `Character`-Ausdrücke aufgerufen werden können:

```
Character c1 := 'a'
Character c2 := 'n'
String s1 := "tomato"

// concat auf Character anwendbar
String s2 := concat(concat(c1,s1),c2)
...

// compare auf Character anwendbar
if (compare(c1,c2)) > 0 {...}
...

// endsWith auf Character anwendbar
if (endsWith(s2,c2)) {...}
...
```

D.2 Herleitung der Verhaltenskonformität

Verhaltenskonformität muss nicht postuliert werden. Sie kann folgendermassen aus den Regeln des modifizierten Gleichheitsvertrags hergeleitet werden:

Wie in Abschnitt 9.3.1 gezeigt wurde, gilt bei einer Inklusionsfunktion zwischen zwei Wertklassen S und T ($\text{include}: S \rightarrow T$) für alle Ausdrücke s vom Typ S

$$s = \text{include}(s)$$

Aufgrund der Ununterscheidbarkeit der Gleichheit (s. Kap. 8.3.1) gilt für jede lesende Operation f :

$$\text{wenn } a = b \text{ dann } f(a) = f(b)$$

Angewendet auf die verschiedenen Signaturen für Operationen in S und T ergeben sich daraus die in Abschnitt 9.3.2, Seite 204 aufgeführten Regeln:

Signatur $f1: T \rightarrow R, f1: S \rightarrow R$

aus $s = \text{include}(s)$

folgt $f1(s) = f1(\text{include}(s))$ (Anwenden der Ununterscheidbarkeit auf $f1$)

Signatur $g1: T \rightarrow T, g1: S \rightarrow S$

aus $s = \text{include}(s)$

folgt $g1(s) = g1(\text{include}(s))$ (Anwenden der Ununterscheidbarkeit auf $g1$)

und $g1(s) = \text{include}(g1(s))$ (Ersetzen von s durch $g1(s)$)

und damit $g1(\text{include}(s)) = \text{include}(g1(s))$

Signatur $h1: Q \rightarrow T, h1: Q \rightarrow S$

aus $s = \text{include}(s)$

folgt $h1(q) = \text{include}(h1(q))$ (Ersetzen von s durch $h1(q)$)

Die Verhaltenskonformität von Operationen mit zwei und mehr Parametern kann analog hergeleitet werden.

D.3 Beispiel Inklusionsfunktion: Workday/ Weekday

Das Beispiel zeigt, dass die Inklusionsfunktion sich auch dazu eignet, eine Subtyp-Beziehung zwischen Aufzählungs-Wertklassen herzustellen.

Mit Weekday wird eine Aufzählungs-Wertklasse für die sieben Wochentage und mit Workday eine Aufzählungs-Wertklasse für die sechs Werktage definiert:

```
valueClass Weekday {  
  enum {monday, tuesday, wednesday, thursday,  
        friday, saturday, sunday}  
}
```

```
valueClass Workday {  
  enum {monday, tuesday, wednesday, thursday,  
        friday, saturday}  
}
```

Die Werktage (Montag bis Samstag) bilden eine Teilmenge aller Wochentage (Montag bis Sonntag). Ein Werktag kann überall dort verwendet werden, wo ein Wochentag erwartet wird. Die folgende Inklusionsfunktion macht Workday zu einem Subtyp von Weekday:

```
Weekday include (Workday w) {  
  switch w  
    case monday:    Weekday.monday  
    case tuesday:   Weekday.tuesday  
    case wednesday: Weekday.wednesday  
    case thursday:  Weekday.thursday  
    case friday:    Weekday.friday  
    case saturday:  Weekday.saturday  
}
```

Eine Inklusionsfunktion ist auch dann möglich, wenn die beiden Aufzählungs-Wertklassen unterschiedliche Bezeichner für die Wochentage definieren, z. B. weil sie aus verschiedenen Quellen stammen. Beispiel:

```
valueClass Weekday {  
  enum {monday, tuesday, wednesday, thursday,  
        friday, saturday, sunday}  
}
```

```
valueClass JourOuvrable {  
  enum {lundi, mardi, mercredi,  
        jeudi, vendredi, samedi}  
}
```

Die Inklusionsfunktion sieht dann folgendermassen aus:

```
Weekday include (JourOuvrable w) {  
  switch w  
    case lundi:      Weekday.monday  
    case mardi:      Weekday.tuesday  
    case mercredi:   Weekday.wednesday  
    case jeudi:      Weekday.thursday  
    case vendredi:   Weekday.friday  
    case samedi:     Weekday.saturday  
}
```

Damit gilt z. B. `JourOuvrable.jeudi = Weekday.thursday`.

`JourOuvrable.jeudi` und `Weekday.thursday` sind dann nicht zwei verschiedene Werte, sondern zwei verschiedene Ausdrücke, die denselben Wert bezeichnen.

Auch im Beispiel `Weekday/Workday` kann sich das Problem ergeben, dass Operationen mit verschiedener Semantik den gleichen Namen tragen. Z. B. können die beiden Werttypen `Weekday` und `Workday` jeweils über eine Operation namens `next` verfügen, und für `Weekday` liefert `next` den nachfolgenden Wochentag, für `Workday` den nachfolgenden Werktag:

```
valueClass Weekday {  
  enum {monday, tuesday, wednesday, thursday,  
        friday, saturday, sunday}  
  
  Weekday next (Weekday w) {  
    switch w  
      case monday:    tuesday  
      case tuesday:   wednesday  
      case wednesday: thursday  
      case thursday:  friday  
      case friday:    saturday  
      case saturday:  sunday  
      case sunday:    monday  
    }  
  }  
}  
  
valueClass Workday {  
  enum {monday, tuesday, wednesday, thursday,  
        friday, saturday}  
  
  Workday next (Workday w) {  
    switch w  
      case monday:    tuesday  
      case tuesday:   wednesday  
      case wednesday: thursday  
      case thursday:  friday  
      case friday:    saturday  
      case saturday:  monday  
    }  
  }  
}
```

Die beiden Operationen namens `next` haben in `Weekday` und `Workday` *nicht das gleiche Verhalten*, sie sind nicht verhaltenskonform. Das beweist das folgende Gegenbeispiel:

Damit `next` verhaltenskonform ist, muss die Bedingungen

```
include(w.next()) = (include(w)).next()
```

für alle Elemente von `Workday` erfüllt sein.

Für `Workday.saturday` gilt:

```
include(Workday.saturday.next())
= include(Workday.monday) =
= Weekday.monday
```

aber

```
(include(Workday.saturday)).next()
= (Weekday.saturday).next() =
= Weekday.sunday
```

Solange `Weekday` und `Workday` getrennt voneinander benutzt werden, ist das unproblematisch. Die fehlende Verhaltenskonformität von `next` verhindert aber die Bildung einer Subtyp-Beziehung durch eine Inklusionsfunktion; oder sie macht sie zumindest komplizierter, weil der Namenskonflikt durch eine geeignete Umbenennung aufgelöst werden muss.

D.4 Beispiel Typäquivalenz: Date und Fecha

Das Beispiel zeigt, wie zwischen zwei Wertklassen, die den gleichen Typ abbilden – hier: den Typ `Datum`¹ – mit zwei wechselseitigen Inklusionsfunktionen Typäquivalenz hergestellt werden kann.

Das Beispiel geht davon aus, dass eine Wertklasse (`Date`) in einem englisch sprechenden Umfeld programmiert wurde, die andere (`Fecha`) in einem spanisch sprechenden. Das macht die beiden Wertklassen im nachfolgenden Code auch leichter unterscheidbar.

Vorausgesetzt wird das Vorhandensein eines Werttyps `Integer` mit Infix-Operatoren für die Addition (+), Multiplikation (*), Divisionsrest (%), kleiner-Vergleich (<) und größer-Vergleich (>). Des weiteren wird das Vorhandensein eines Werttyps `Bool` mit Infix-Operatoren für die logische Konjunktion (&&) und Disjunktion (||) vorausgesetzt.

¹Aus Vereinfachungsgründen wird vom aktuell in den meisten Ländern gebräuchlichen Gregorianischen Kalender ausgegangen, und der Übergang vom Julianischen zum Gregorianischen Kalender am 15.10.1582 nicht berücksichtigt.

Code der Wertklasse Date:

```
valueClass Date {
  Integer day;
  Integer month;
  Integer year;

  Date dateYMD(Integer year, Integer month, Integer day){
    // check validity of parameters
    if (day > 31) { ... }
    ...
    return select(day, month, year);
  }

  Date dateYDDD(Integer year, Integer dayOfYear){
    Integer m := ...
    Integer d := ...
    return select(d, m, year);
  }

  Date date1970(Integer noDaysSince19700101){
    Integer y := ...
    Integer m := ...
    Integer d := ...
    return select(d, m, yr);
  }

  Integer getDay(Date d) { return d.day; }

  Integer getMonth(Date d) { return d.month; }

  Integer getYear(Date d) { return d.year; }

  Integer getDayOfYear(Date d) { ... }

  Integer weekOfYear(Date d) { ... }

  Bool before(Date d1, Date d2) { ... }
}
```


Code der Wertklasse Fecha:

```
valueClass Fecha {
  Integer diasDesde19700101;

  Fecha fechaDesde19700001(Integer dias){
    return select(dias);
  }

  Fecha fechaAMD(Integer anno, Integer mes, Integer dia){
    ...
    dias := ...
    return select(dias);
  }

  Integer daDiasDesde19700101(Fecha f){
    return f.diasDesde19700101;
  }

  Integer daAnno(Fecha f) { ... }

  Integer daMes(Fecha f) { ... }

  Integer daDiaDeMes(Fecha f) { ... }

  Integer diferenciaDias(Fecha f1, Fecha f2){
    return f2.diasDesde19700101 - f1.diasDesde19700101;
  }

  Fecha sumaDias(Fecha f, Integer d){
    return select(f.diasDesde19700101 + d);
  }

  Bool antes(Fecha f1, Fecha f2){
    return f1.diasDesde19700101 < f2.diasDesde19700101;
  }
}
```

Inklusionsfunktion von Date nach Fecha:

```
Fecha include(Date d) {
  return fechaAMD(getYear(d), getMonth(d), getDay(d));
}
```

Inklusionsfunktion von Fecha nach Date:

```
Date include(Fecha f) {
  return date1970(daDiasDesde19700101(f));
}
```

Das Beispiel macht einige Besonderheiten bei Typäquivalenz von zwei Wertklassen deutlich:

Operationen in beiden Klassen

In den beiden Klassen `Date` und `Fecha` sind viele übereinstimmende Operationen unter verschiedenen Namen vorhanden: `getDay` entspricht `daDiaDeMes`, `getMonth` entspricht `daMes`, `getYear` entspricht `daAnno`, `before` entspricht `antes`. Diese Konstellation ist unkritisch. Nur der umgekehrte Fall – gleichnamige Operationen in beiden Klassen mit verschiedenem Verhalten – würde aufgrund der verletzten Verhaltenskonformität Probleme aufwerfen.

Bedingt durch die unterschiedliche interne Darstellung sind einige Operationen in `Date` deutlich einfacher und kürzer zu programmieren als ihr jeweiliges Gegenstück in der Klasse `Fecha`, z. B. `getYear` (`daAnno`). Umgekehrt gibt es Operationen, deren Implementation in `Date` aufwändiger ist als in `Fecha`, weil die interne Darstellung mit drei Datenfeldern sie verkompliziert, z. B. `before` (`antes`).

Operationen in nur einer Klasse

Sowohl in `Date` als auch in `Fecha` sind Operationen vorhanden, zu denen es in der anderen Wertklasse kein Gegenstück gibt. In `Date` sind das `dateYDDD`, `getDayOfYear` und `weekOfYear`, in `Fecha` sind es `daDiasDesde19700101`, `diferenciaDias` und `sumaDias`. Durch die beiden Inklusionsfunktionen sind diese Operationen auch für die jeweils andere Wertklasse aufrufbar.

Die für den Typ „Datum“ (verkörpert durch die beiden Wertklassen `Date` und `Fecha`) aufrufbaren Operationen bestehen aus der Vereinigungsmenge der in beiden Wertklassen definierten Operationen.

Ungültige Parameter für Selektoren

Bei Wertoperationen, insbesondere bei Selektoren, kann es ungültige Parameterkombinationen geben. Z. B. ist der 31. Februar 2016 kein gültiges Datum. Das ist eine Eigenschaft der jeweiligen Operation – nicht eine der Wertklasse. Z. B. müssen die (gleichbedeutenden) Selektoren `dateYMD` und `fechaAMD` die ungültigen Parameter berücksichtigen und passend behandeln (z. B. durch Auslösen einer Exception).

Umgekehrt führt ein Selektor, der ein Datum basierend auf der Anzahl der Tage seit einem festen Datum auswählt (`date1970` in der Wertklasse `Date`, `fechaDesde19700001` in der Wertklasse `Fecha`), immer zu einem gültigen Wert des jeweiligen Typs, unabhängig von der zugehörigen Wertklasse und ihrer internen Repräsentation.

D.5 Beispiel Fortress: extends- und coerce-Beziehungen

Das folgende Beispiel ist der Arbeit von Hallett et al. [HLRS08] entnommen. Es verdeutlicht, wie die Verflechtung von coercions und Subclassing die Komplexität des Codes erhöht und die Nachvollziehbarkeit erschwert.

Fortress-Code:

```
trait A end

object B(x:N) extends {A} end

object C(x:N)
  coerce (b: B) = C(0)
end

object D(x:N)
  coerce (a: A) = D(1)
  coerce (b: B) = D(2)
  coerce (c: C) = D(3)
end

object E(x:N)
  coerce (a: A) = E(4)
  coerce (d: D) = E(5)
end

object Tester(a: A)
  f (c: C):N = c.x
  f (d: D):N = d.x + 7
  f (e: E):N = e.x + 14
  run() = self.f(a)
end

Tester(B(6)).run()
```

Das Ergebnis des Programms lautet: 9

Begründung: die „spezifischste“ Definition, die auf `self.f(a)` angewendet werden kann, ist die mit Parametertyp D. Darum wird eine coercion von a zu D durchgeführt. Zur Laufzeit gilt `a = B(6)`, darum wird die coercion mit Parameter B aufgerufen, also `coerce (b: B) = D(2)`.

D.6 Beispiel Werte versus Objekte: Rechteck/ Quadrat

Der folgende Code zeigt Alternative 1:

Rechtecke und Quadrate werden als Werte aufgefasst und mit Hilfe der Wertklassen `RectangleV` und `SquareV` implementiert. Zwischen beiden gibt es eine Inklusionsfunktion.

```
valueClass RectangleV {
  Float x;
  Float y;
  RectangleV getRectangleXY (Float x, Float y) {
    return select(x,y); // primärer Selektor
  }
  Float getX(RectangleV r) {return r.x;}
  Float getY(RectangleV r) {return r.y;}
  RectangleV stretchX(RectangleV r, Float factor) {
    return select(r.x * factor, r.y);
  }
  ...
}

valueClass SquareV {
  Float a;
  SquareV getSquare (Float a) {
    return select(a); // primärer Selektor
  }
  Float getEdgeLength(SquareV s) {return s.a;}
  ...
}

RectangleV include (SquareV s) {
  return getRectangleXY(getEdgeLength(s), getEdgeLength(s));
}
```

Der folgende Code zeigt Alternative 2:

Rechtecke und Quadrate werden als Objekte aufgefasst und mit Hilfe der Objektklassen `RectangleO` und `SquareO` implementiert. Zwischen beiden gibt es keine Vererbungsbeziehung.

```
objectClass RectangleO {  
    Float x;  
    Float y;  
    Float getX() {return this.x;}  
    Float getY() {return this.y;}  
    void setX(Float x) {this.x := x;}  
    void setY(Float y) {this.y := y;}  
    void stretchX(Float factor) {  
        this.x := this.x * factor;  
    }  
    ...  
}  
  
objectClass SquareO {  
    Float a;  
    Float getEdgeLength() {return this.a;}  
    void setEdgeLength(Float a) {  
        this.a := a;  
    }  
    void stretch(Float factor) {  
        this.a := this.a * factor;  
    }  
    ...  
}
```

Die Klasse `RectangleO` bildet auch Quadrate ab; das sind diejenigen Elemente der Klasse, für die beide Seiten `x` und `y` gleich groß sind. Die Seiten eines solchen Quadrates sind unabhängig von der jeweils anderen Seite änderbar. Dadurch kann ein Element von `RectangleO` die Eigenschaft, ein Quadrat zu sein, verlieren (oder erhalten). „Quadratisch zu sein“ ist in dieser Klasse eine änderbare Eigenschaft, ebenso wie die Länge der beiden Seiten oder die Fläche.

Die Klasse `SquareO` dagegen gewährleistet die Eigenschaft, quadratisch zu sein, für alle ihre Elemente.

Aufgrund der Änderbarkeit der Form in der Klasse `RectangleO` ist das „Quadratisch-sein“ ein ungeeignetes Kriterium für die Bildung eines Subtyps. (Analog wäre es auch keine geeignete Modellierung, von einem Typ `Fahrzeug` einen Subtyp `BlauesFahrzeug` zu bilden, wenn die Farbe eines Fahrzeugs änderbar ist.)

Das „Quadratisch-sein“ gibt eine Auskunft über den aktuellen Zustand eines Rechtecks und kann durch eine geeignete sondierende Methode (`isSquare`) in der Klasse `RectangleO` zum Ausdruck gebracht werden:

```
objectClass RectangleO {  
    ...  
    Bool isSquare() {return x=y;}  
    ...  
}
```

Abhängig vom fachlichen Kontext können die Anforderungen an Quadrate bereits durch die Klasse `RectangleO` abgedeckt sein. In dem Fall ist eine gesonderte Klasse `SquareO` verzichtbar. Siehe dazu auch die Diskussion dieses Fallbeispiels von Oestereich [Oes09, p. 51].

Literaturverzeichnis

- [Abr78] Abrahams, P.: *The PL/I Programming Language*. Courant Mathematics und Computing Laboratory, U.S. Department of Energy, New York University, 1978.
- [AGH06] Arnold, K., Gosling, J. und Holmes, D.: *The Java Programming Language*. 4. Aufl. Upper Saddle River, NJ: Addison-Wesley, 2006.
- [Ale10] Alexandrescu, A.: *The D programming language*. Upper Saddle River, NJ: Addison-Wesley, 2010.
- [Ale14] Alexandrescu, A.: *Pure functions in D*. Persönliche Korrespondenz. 17. Juli 2014.
- [Ale79] Alexander, C.: *The Timeless Way of Building*. Center for Environmental Structure series. New York, NY: Oxford University Press, 1979.
- [All+04] Allen, E., Chase, D., Luchangco, V., Maessen, J.-W. und Steele Jr., G. L.: “Object-oriented Units of Measurement”. In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. (Vancouver, BC, Canada). New York, NY: ACM, 2004, S. 384–403.
- [All+08] Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele, G. und Tobin-Hochstadt, S.: *The Fortress Language Specification Version 1.0*. Techn. Ber. Sun Microsystems Inc., 31. März 2008.
- [AS96] Abelson, H. und Sussman, G.: *Structure and Interpretation of Computer Programs*. 2. Aufl. Electrical engineering and computer science series. Cambridge, MA: MIT Press, 1996.
- [Bac+57] Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A. und Nutt, R.: “The FORTRAN Automatic Coding System”. In: *Western Joint Computer Conference: Techniques for Reliability*. (Los Angeles, California). IRE-AIEE-ACM. New York, NY: ACM, 1957, S. 188–198.

- [Bac+63] Backus, J. W., Bauer, F. L., Green, J., Katz, C., McCarthy, J., Perlis, A. J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J. H., Wijngaarden, A. van und Woodger, M.: “Revised Report on the Algorithmic Language Algol 60”. In: *Commun. ACM* 6.1 (Jan. 1963). Hrsg. von Naur, P., S. 1–17.
- [Bac03] Bacon, D.: “Kava: A Java dialect with a uniform object model for lightweight classes”. In: *Concurrency - Practice and Experience* 15 (2003), S. 185–206.
- [Bac78] Backus, J. W.: “Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs”. In: *Commun. ACM* 21.8 (1978), S. 613–641.
- [Bak93] Baker, H. G.: “Equal rights for functional objects or, the more things change, the more they are the same”. In: *SIGPLAN OOPS Mess.* 4.4 (1993), S. 2–27.
- [Bal01] Balzert, H.: *Lehrbuch der Softwaretechnik. Software-Entwicklung*. 2. Aufl. Heidelberg: Spektrum Akademischer Verlag, 2001.
- [Bäu+98] Bäumer, D., Riehle, D., Siberski, W., Lilienthal, C., Megert, D., Syl-la, K.-H. und Züllighoven, H.: *Values in Object Systems*. Techn. Ber. Zurich, Switzerland: UBS AG, 1998.
- [BGW91] Bobrow, D., Gabriel, R. und White, J.: “CLOS: Integrating Object-Oriented and Functional Programming”. In: *Comm. ACM* 34.9 (1991), S. 28–38.
- [Bla+02] Black, A. P., Carlsson, M., Jones, M. P., Kieburtz, R. und Nordlander, J.: *Timber: A Programming Language for Real-Time Embedded Systems*. Techn. Ber. CSE-02-002. Beaverton, Oregon: Oregon Health & Science University, Apr. 2002.
- [Blo01] Bloch, J.: *Effective Java: programming language guide*. The Java series. Boston: Addison-Wesley, 2001.
- [Blo08] Bloch, J.: *Effective Java*. 2. Aufl. The Java series. Upper Saddle River, NJ: Addison-Wesley, 2008.
- [BLZ99] Bleek, W.-G., Lilienthal, C. und Züllighoven, H.: “Frameworkbasierte Anwendungsentwicklung (Teil 4): Fachwerte”. In: *ObjektSpektrum* 5 (Sep. 1999), S. 75–80.
- [BMS80] Burstall, R. M., MacQueen, D. B. und Sannella, D. T.: “HOPE: An Experimental Applicative Language”. In: *Proceedings of the 1980 ACM Conference on LISP and Functional Programming*. (Stanford University, California). New York, NY: ACM, 1980, S. 136–143.
- [Bro13] Brown, W. E.: *Proposing pure*. Standard C++ Foundation. Aug. 2013. URL: <http://isocpp.org/files/papers/n3744.pdf> (besucht am 11.07.2014).

- [Bru+95] Bruce, K., Cardelli, L., Castagna, G., Leavens, G. und Pierce, B.: “On Binary Methods”. In: *Theory and Practice of Object Systems* 1.3 (Dez. 1995), S. 221–242.
- [Bru02] Bruce, K.: *Foundations of object-oriented languages: types and semantics*. Cambridge, MA: MIT Press, 2002.
- [Bru94] Bruce, K.: “A paradigmatic object-oriented programming language: design, static typing and semantics”. In: *Journal of Functional Programming* 4.2 (1994), S. 127–206.
- [Bus+96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. und Stal, M.: *Pattern-Oriented Software Architecture: A System of Patterns*. Bd. 1. Chichester: Wiley, 1996.
- [BV02] Bloch, J. und Venners, B.: *Josh Bloch on Design, A Conversation with Effective Java Author*. artima developer. Jan. 2002. URL: <http://www.artima.com/intv/blochP.html> (besucht am 14.11.2010).
- [BW09] Biancuzzi, F. und Warden, S.: *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. Theory in Practice. O’Reilly, 2009.
- [Cas95] Castagna, G.: “Covariance and contravariance: conflict without a cause”. In: *ACM Transactions on Programming Languages and Systems* 17.3 (Mai 1995), S. 431–447.
- [CB96] Clack, C. und Braine, L.: “Introducing CLOVER: an Object-Oriented Functional Language”. In: *Proc. Eighth International Workshop on Implementation of Functional Programming Languages*. Springer-Verlag, 1996, S. 21–38.
- [Cha+05] Charles, P., Grothoff, C., Saraswat, V. A., Donawa, C., Kielstra, A., Ebcioğlu, K., Praun, C. von und Sarkar, V.: “X10: An Object-Oriented Approach to Non-Uniform Cluster Computing”. In: *Proceedings of the 2005 Conference on Object-Oriented Programming, Systems, Languages, and Applications*. (San Diego). New York, NY: ACM Press, 2005, S. 519–538.
- [CHC90] Cook, W. R., Hill, W. und Canning, P. S.: “Inheritance is not subtyping”. In: *POPL ’90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. (San Francisco, California). New York, NY: ACM, 1990, S. 125–135.
- [Chu36] Church, A.: “An Unsolvable Problem of Elementary Number Theory”. In: *American Journal of Mathematics* 58.2 (Apr. 1936), S. 345–363.
- [CL94] Chambers, C. und Leavens, G.: “Typechecking and Modules for Multimethods”. In: *ACM SIGPLAN Notices, OOPSLA ’94 Conference Proceedings*. (Portland, Oregon). New York, NY: ACM, Okt. 1994, S. 1–15.

- [Coh02] Cohen, T.: “How Do I Correctly Implement the equals() Method?” In: *Dr. Dobb’s Journal* (Mai 2002).
- [Cop92] Coplien, J. O.: *Advanced C++: Programming Styles and Idioms*. Reading, Massachusetts: Addison-Wesley, 1992.
- [CT09] Cesarini, F. und Thompson, S.: *Erlang programming: a concurrent approach to software development*. Beijing: O’Reilly, 2009.
- [Cun10] Cunningham & Cunningham Inc.: *Call By Reference*. 2010. URL: <http://c2.com/cgi/wiki?CallByReference> (besucht am 04.05.2013).
- [CW85] Cardelli, L. und Wegner, P.: “On Understanding Types, Data Abstraction, and Polymorphism”. In: *ACM Computing Surveys* 17.4 (1985), S. 471–522.
- [Deg11] Degner, R.: *TKeasy: Die unternehmensweite Java-Enterprise-Architektur der Techniker Krankenkasse*. Vortrag Universität Hamburg. 8. Nov. 2011. URL: http://swt-www.informatik.uni-hamburg.de/fileadmin/swt-redaktion/pdf_dateien/Techniker-Krankenkasse-TKeasy-UniHH-2011.pdf (besucht am 11.07.2014).
- [Dij72] Dijkstra, E. W.: “Notes on Structured Programming”. In: *Structured Programming*. Hrsg. von Dahl, O.-J., Dijkstra, E. W. und Hoare, C. A. R. London, UK: Academic Press, 1972, S. 1–82.
- [DMN68] Dahl, O.-J., Myrhaug, B. und Nygaard, K.: *SIMULA 67 Common Base Language*. Publ. No. S-2. Oslo, Norway: Norwegian computing center, 1968.
- [DN66] Dahl, O.-J. und Nygaard, K.: “SIMULA: An ALGOL-based Simulation Language”. In: *Communications of the ACM* 9.9 (Sep. 1966), S. 671–678.
- [Eck02] Eckel, B.: *Thinking in Java*. 3. Aufl. Upper Saddle River, NJ: Prentice Hall, 2002.
- [Ecl14] Eclipse Foundation: *Eclipse documentation*. 2014. URL: <http://help.eclipse.org/> (besucht am 31.08.2014).
- [ECM06a] ECMA International: *Standard ECMA-334 C# Language Specification, 4th edition*. Juni 2006.
- [ECM06b] ECMA International: *Standard ECMA-367 Eiffel: Analysis, Design and Programming Language, 2nd edition*. Juni 2006.
- [EK95] Eckert, G. und Kempe, M.: “Modeling with Objects and Values: Issues and Perspectives”. In: *ROAD (Report on Object Analysis & Design)* 1.5 (Jan. 1995), S. 20–27.

- [EKSM97] Evered, M., Keedy, J. L., Schmolitzky, A. und Menger, G.: “How Well do Inheritance Mechanisms Support Inheritance Concepts”. In: *Proc. JMLC '97, Linz, Austria*. Lecture Notes in Computer Science 1204. Springer-Verlag, 1997, S. 252–266.
- [EST95] Eifrig, J., Smith, S. und Trifonov, V.: “Sound polymorphic type inference for objects”. In: *Proceedings of the 1995 Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 1995, S. 169–184.
- [Eva04] Evans, E.: *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley, 2004.
- [FFFK01] Felleisen, M., Findler, R. B., Flatt, M. und Krishnamurthi, S.: *How to Design Programs: An Introduction to Programming and Computing*. Cambridge, MA: MIT Press, 2001.
- [FH14] Fogus, M. und Houser, C.: *The Joy of clojure*. 2. Aufl. Shelter Island, NY: Manning, 2014.
- [Flo79] Floyd, R. W.: “The Paradigms of Programming”. In: *Commun. ACM* 22.8 (Aug. 1979), S. 455–460.
- [For97] *Fortran 95 Standard, final draft (ISO/IEC 1539-1:1997)*. Okt. 1997. URL: http://gcc.gnu.org/wiki/GFortranStandards#Fortran_95 (besucht am 11.07.2014).
- [Fow] Fowler, M.: *EvansClassification*. ThoughtWorks. URL: <http://martinfowler.com/bliki/EvansClassification.html> (besucht am 17.08.2014).
- [Fow03] Fowler, M.: *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, 2003.
- [Fow09] Fowler, M.: *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, 2009.
- [Fre79] Frege, G.: *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle: Louis Nebert, 1879.
- [Fre92] Frege, G.: “Über Sinn und Bedeutung”. In: *Zeitschrift für Philosophie und philosophische Kritik* 100 (1892), S. 25–50.
- [Fri88] Friedrich, C.: *Schriftliche Arbeiten im technisch-naturwissenschaftlichen Studium*. Mannheim: Dudenverlag, 1988.
- [FS02] Fürter, M. und Spreckelmeyer, A.: “Konzepte und Ansätze zur Unterstützung der Implementierung fachlicher Werte in objektorientierten Programmiersprachen am Beispiel des Java-Rahmenwerks JWAM”. Diplomarbeit. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Apr. 2002.

- [GC94] Grogono, P. und Chalin, P.: “Copying, Sharing, and Aliasing.” In: *COODBSE*. 1994, S. 77–89.
- [GHJV98] Gamma, E., Helm, R., Johnson, R. und Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1998.
- [Goe+06] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D. und Lea, D.: *Java Concurrency in Practice*. Upper Saddle River, NJ: Addison-Wesley, 2006.
- [Goe03] Goetz, B.: *Java theory and practice: To mutate or not to mutate?* IBM developerWorks. 18. Feb. 2003. URL: <http://www.ibm.com/developerworks/java/library/j-jtp02183.html> (besucht am 25.02.2009).
- [Goe14] Goetz, B.: *JSR 335: Lambda Expressions for the Java™ Programming Language*. 4. März 2014. URL: <https://www.jcp.org/en/jsr/detail?id=335> (besucht am 11.07.2014).
- [Gos+11] Gosling, J., Joy, B., Steele, G., Bracha, G. und Buckley, A.: *The Java Language Specification, Java SE 7 Edition*. California: Oracle America, Inc., 2011.
- [Gos+14] Gosling, J., Joy, B., Steele, G., Bracha, G. und Buckley, A.: *The Java Language Specification, Java SE 8 Edition*. California: Addison-Wesley, 2014.
- [GR89] Goldberg, A. und Robson, D.: *Smalltalk 80: The Language*. Boston, MA: Addison-Wesley, 1989.
- [Gra97] Graham, P.: *ANSI Common Lisp*. München: Prentice Hall, 1997.
- [GS00] Grogono, P. und Sakkinen, M.: “Copying and Comparing: Problems and Solutions”. In: *ECOOP ’00: Proceedings of the 14th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2000, S. 226–250.
- [GS78] Guibas, L. J. und Sedgewick, R.: “A dichromatic framework for balanced trees”. In: *SFCS ’78: Proceedings of the 19th Annual Symposium on Foundations of Computer Science*. Washington, DC: IEEE Computer Society, Okt. 1978, S. 8–21.
- [GS93] Grogono, P. und Santas, P.: “Equality in Object Oriented Languages”. In: *EastEurOOPe’93*. Bratislava, Slovakia, 1993.
- [Gut77] Guttag, J.: “Abstract data types and the development of data structures”. In: *Commun. ACM* 20.6 (Juni 1977), S. 396–404.
- [Hen00] Henney, K.: “Patterns of Value”. In: *Java Report* 5.2 (Feb. 2000), S. 64–68.

- [Hen01] Henney, K.: “From Mechanism to Method: Total Eclipse”. In: *Dr. Dobb’s C/C++ Users Journal* (März 2001).
- [Hen02] Henney, K.: “Value-Based Programming in Java”. In: *JAOO* (Sep. 2002).
- [Hen03a] Henney, K.: “Objects of Value (Programmer’s Workshop column)”. In: *Application Development Advisor* 7.6 (Nov. 2003).
- [Hen03b] Henney, K.: “Unvollendete Symmetrie in Java”. In: *JavaSpektrum* (Mai 2003), S. 63–66.
- [HLRS08] Hallett, J. J., Luchangco, V., Ryu, S. und Steele, J. G. L.: “Integrating Coercion with Subtyping and Multiple Dispatch”. In: *Proceedings of the 2008 ACM symposium on Applied computing*. (Fortaleza, Ceara, Brazil). New York, NY: ACM, 2008, S. 166–170.
- [HO87] Halbert, D. und O’Brien, P.: “Using Types and Inheritance in Object-Oriented Languages”. In: *Proc. ECOOP ’87, Paris, France*. Lecture Notes in Computer Science 276. Springer-Verlag, 1987, S. 20–31.
- [Hoa09] Hoare, C. A. R.: *Null References: The Billion Dollar Mistake*. QCon London Keynote. 25. Aug. 2009. URL: <http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare> (besucht am 11.07.2014).
- [Hoa72] Hoare, C. A. R.: “Notes on Data Structuring”. In: *Structured Programming*. Hrsg. von Dahl, O.-J., Dijkstra, E. W. und Hoare, C. A. R. London, UK: Academic Press Ltd., 1972, S. 83–174.
- [Hoa73] Hoare, C. A. R.: *Hints on Programming Language Design*. Techn. Ber. STAN-CS-73-403. Stanford, CA, 1973.
- [Hoo07] Hoogendoorn, S.: “Die Implementierung von Value-Objekten”. In: *ObjektSpektrum* (Nov. 2007), S. 52–56.
- [Hor03] Horstmann, C.: *Some Objects are More Equal Than Others*. artima developer. Mai 2003. URL: <http://www.artima.com/weblogs/viewpost.jsp?thread=4744> (besucht am 03.12.2012).
- [Hor83] Horowitz, E.: *Fundamentals of Programming Languages*. New York, NY: W. H. Freeman & Co., 1983.
- [Hud+92] Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzmán, M. M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. und Peterson, J.: “Report on the Programming Language Haskell: A Non-strict, Purely Functional Language Version 1.2”. In: *ACM SIGPLAN Notices* 27.5 (Mai 1992), S. 1–164.
- [Hud89] Hudak, P.: “Conception, Evolution, and Application of Functional Programming Languages”. In: *ACM Computing Surveys* 21.3 (Sep. 1989), S. 359–411.

- [Hug84] Hughes, J.: “Why Functional Programming Matters”. In: *The Computer Journal* 32.2 (1984), S. 98–107.
- [IS99] Isbell, D. und Savage, D.: *Mars Climate Orbiter Failure Board Releases Report, Numerous NASA Actions Underway in Response*. 10. Nov. 1999. URL: http://nssdc.gsfc.nasa.gov/planetary/text/mco_pr_19991110.txt (besucht am 13.06.2014).
- [Ive62] Iverson, K. E.: *A Programming Language*. New York, NY: John Wiley & Sons, Inc., 1962.
- [Jon03] Jones, S. P., Hrsg.: *Haskell 98 Language and Libraries: The Revised Report*. Journal of functional programming. Cambridge University Press, 2003.
- [JW91] Jensen, K. und Wirth, N.: *Pascal User Manual and Report: ISO Pascal Standard*. Springer, 1991.
- [JWA03] *JWAM Value-Framework*. 2003. URL: <http://sourceforge.net/projects/jwamdomainvalue/> (besucht am 30.06.2013).
- [KA90] Khoshafian, S. und Abnous, R.: *Object orientation : concepts, languages, databases, user interfaces*. Wiley professional computing, 1990.
- [Kay93] Kay, A. C.: “The Early History of Smalltalk”. In: *The Second ACM SIGPLAN Conference on History of Programming Languages (HOPL-II)*. (Cambridge, MA). New York, NY: ACM, 1993, S. 69–95.
- [KC86] Khoshafian, S. N. und Copeland, G. P.: “Object identity”. In: *OOPLSA ’86: Conference proceedings on Object-Oriented Programming Systems, Languages, and Applications*. (Portland, Oregon). New York, NY: ACM, Sep. 1986, S. 406–416.
- [Ken91] Kent, W.: “A Rigorous Model of Object Reference, Identity, and Existence”. In: *Journal of Object-Oriented Programming* 4.3 (Juni 1991), S. 28–36.
- [KM96] Koenig, A. und Moo, B.: *Ruminations on C++ : a decade of programming insight and experience*. Redwood City, CA: Addison-Wesley Longman, 1996.
- [Koc11] Kochan, S. G.: *Programming in Objective-C*. 3. Aufl. Developer’s library. Upper Saddle River, NJ: Addison-Wesley, 2011.
- [Köl99] Kölling, M.: “The Design of an Object-oriented Environment and Language for Teaching”. Dissertation. Basser Department of Computer Science, University of Sydney, 1999.
- [KPRS09] Kollár, M., Peterka, O., Rysavý, O. und Skarvada, L.: *A Calculus of Coercive Subtyping*. Faculty of Informatics Masaryk University Brno FI MU Report Series FIMU-RS-2009-11. Nov. 2009.

- [KR12] Karmasin, M. und Ribing, R.: *Die Gestaltung wissenschaftlicher Arbeiten. Ein Leitfaden für Seminararbeiten, Bachelor-, Master- und Masterarbeiten sowie Dissertationen*. 7. Aufl. UTB. Wien: facultas.wuv, 2012.
- [KR88] Kernighan, B. W. und Ritchie, D. M.: *The C Programming Language*. 2. Aufl. Englewood Cliffs, NJ: Prentice Hall, 1988.
- [KR97] Kölling, M. und Rosenberg, J.: *Blue Language Specification, Version 1.0*. Techn. Ber. TR97-13. Monash University, Department of Computer Science und Software Engineering, 1997.
- [KR99] Kölling, M. und Rosenberg, J.: “On Creation, Equality and the Object Model”. In: *Proceedings of the 32nd International Conference on Technology of Object-Oriented Languages and Systems (TOOLS), Melbourne, Australia*. IEEE Computer Society, Jan. 1999, S. 210–221.
- [Kri03] Krishnamurthi, S.: *Programming Languages: Application and Interpretation*. Brown University, 2003.
- [Kuh70] Kuhn, T. S.: *The Structure of Scientific Revolutions*. University Of Chicago Press, 1970.
- [Küh99] Kühne, T.: “A Functional Pattern System for Object-Oriented Design”. Dissertation. Darmstadt University of Technology, 1999.
- [Lau93] Launchbury, J.: “Lazy Imperative Programming”. In: *Proceedings of the ACM SIGPLAN Workshop on State in Programming Languages, SIPL ’92*. (Copenhagen). 1993, S. 46–56.
- [Ler+13] Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D. und Vouillon, J.: *The OCaml system: Documentation and User’s Manual*. Sep. 2013. URL: <http://caml.inria.fr/pub/docs/manual-ocaml/> (besucht am 13.06.2014).
- [LG01] Liskov, B. und Guttag, J.: *Program Development in Java : Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2001.
- [LG86] Liskov, B. und Guttag, J.: *Abstraction and Specification in Program Development*. Cambridge, MA: MIT Press, 1986.
- [Lis+95] Liskov, B., Curtis, D., Day, M., Ghemawat, S., Gruber, R., Johnson, P. und Myers, A. C.: *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science. Cambridge, MA, Feb. 1995.
- [LK02a] Langer, A. und Kreft, K.: “Secrets of equals() - Part 1, Not all implementations of equals() are equal”. In: *JavaSolutions* (Apr. 2002).
- [LK02b] Langer, A. und Kreft, K.: “Secrets of equals() - Part 2, How to implement a correct slice comparison in Java”. In: *Java Solutions* (Juni 2002).

- [LL13] Ludewig, J. und Lichter, H.: *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. 3. Aufl. Heidelberg: dpunkt.Verlag, 2013.
- [Lou93] Loudon, K. C.: *Programming Languages: Principles and Practice*. Belmont, CA: Wadsworth Publ. Co., 1993.
- [LP91] LaLonde, W. R. und Pugh, J. R.: “Subclassing $\neq$ Subtyping $\neq$ Is-a”. In: *Journal of Object-Oriented Programming* 3.5 (Jan. 1991), S. 57–62.
- [LR09] Lahres, B. und Raýman, G.: *Objektorientierte Programmierung: das umfassende Handbuch*. 2. Aufl. Bonn: Galileo Press, 2009.
- [Luo99] Luo, Z.: “Coercive subtyping”. In: *Journal of Logic and Computation* 9.1 (1. Feb. 1999). Oxford University Press, S. 105–130.
- [LW94] Liskov, B. H. und Wing, J. M.: “A Behavioral Notion of Subtyping”. In: *ACM Transactions on Programming Languages and Systems* 16.6 (Nov. 1994), S. 1811–1841.
- [Mac82] MacLennan, B. J.: “Values and objects in programming languages”. In: *SIGPLAN Not.* 17.12 (1982), S. 70–79.
- [Maj98] Majorinc, K.: “Ellipse-Circle Dilemma and Inverse Inheritance”. In: *ITI 98, Proceedings of the 20th International Conference of Information Technology Interfaces*. Pula, 1998, S. 627–632.
- [Mar10] Marlow, S., Hrsg.: *Haskell 2010 Language Report*. 2010. URL: http://www.haskell.org/haskellwiki/Language_and_library_specification (besucht am 03.12.2012).
- [Mar96] Martin, R. C.: *The Liskov Substitution Principle*. Engineering Notebook, The C++ Report. März 1996.
- [McC78] McCarthy, J.: “History of LISP”. In: *SIGPLAN Notices* 13.8 (Aug. 1978), S. 217–223.
- [Mey01] Meyer, B.: “Conversions in an Object-Oriented Language with Inheritance”. In: *Journal of Object-Oriented Programming* 13.9 (Jan. 2001), S. 28–31.
- [Mey92] Meyer, B.: *Eiffel: the language*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1992.
- [Mey97] Meyer, B.: *Object-oriented software construction*. 2. Aufl. Upper Saddle River, NJ: Prentice-Hall, Inc., 1997.
- [MG] Müller, T. und Gauss, B.: *Richtlinien für die Anfertigung wissenschaftlicher Arbeiten am Fachgebiet Mensch-Maschine-Systeme*. Technische Universität Berlin. URL: http://www.mms.tu-berlin.de/fileadmin/fg268/Lehre/Richtlinien_Arbeiten_neu.pdf (besucht am 28.10.2014).

- [MHH91] Mugridge, W. B., Hosking, J. G. und Hamer, J.: “Multimethods in a statically-typed programming language”. In: *ECOOP '91 Conference Proceedings*. (Geneva, Switzerland). Lecture Notes in Computer Science 512. Springer-Verlag, 1991, S. 307–324.
- [Mic10] Microsoft Corporation: *C# Language Specification, Version 4.0*. Microsoft Corporation, 2010.
- [MM06] Moseley, B. und Marks, P.: “Out of the Tar Pit”. In: *Software Practice Advancement*. 2006.
- [MTH90] Milner, R., Tofte, M. und Harper, R.: *The definition of Standard ML*. Cambridge, MA: MIT Press, 1990.
- [Nor01] Nordlander, J.: *O'Haskell*. 2001. URL: <http://web.archive.org/web/20090517021445/http://www.cs.chalmers.se/~nordland/ohaskell/> (besucht am 07.08.2014).
- [NPN10] Nelson, S., Pearce, D. J. und Noble, J.: “Understanding the Impact of Collection Contracts on Design”. In: *TOOLS (48) Objects, Models, Components, Patterns*. Hrsg. von Vitek, J. Lecture Notes in Computer Science 6141. Springer, 2010, S. 61–78.
- [Ode14] Odersky, M.: *The Scala Language Specification, Version 2.9*. Techn. Ber. EPFL SWITZERLAND, 22. Mai 2014.
- [Oes09] Oestereich, B.: *Analyse und Design mit UML 2.3: Objektorientierte Softwareentwicklung*. 9. Aufl. München: Oldenbourg, 2009.
- [OOPS12] Odersky, M., Olson, J., Phillips, P. und Suereth, J.: *SIP 15: Value Classes*. Feb. 2012. URL: <http://docs.scala-lang.org/sips/completed/value-classes.html> (besucht am 11.07.2014).
- [Ora14] Oracle: *java.lang.Object API Documentation*. 2014. URL: <http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html> (besucht am 13.08.2014).
- [OSV09] Odersky, M., Spoon, L. und Venners, B.: *How to Write an Equality Method in Java*. artima developer. Juni 2009. URL: <http://www.artima.com/lejava/articles/equality.html> (besucht am 14.11.2010).
- [OSV11] Odersky, M., Spoon, L. und Venners, B.: *Programming in Scala : A comprehensive step-by-step guide*. 2. Aufl. Walnut Creek, Calif.: Artima Press, 2011.
- [Par07] Parr, T.: *The definitive ANTLR reference : building domain-specific languages*. Pragmatic Bookshelf, 2007.
- [PB96] Pomberger, G. und Blaschek, G.: *Software-Engineering: Prototyping und objektorientierte Software-Entwicklung*. 2. Aufl. München: Hanser, 1996.

- [Pie02] Pierce, B. C.: *Types and programming languages*. Cambridge, MA: MIT Press, 2002.
- [Poe09] Poetzsch-Heffter, A.: *Konzepte objektorientierter Programmierung : mit einer Einführung in Java*. Berlin: Springer, 2009.
- [Pol09] Pollak, D.: *Beginning Scala*. Berkeley, CA: Apress, 2009.
- [Qui60] Quine, W. V. O.: *Word and Object*. Cambridge, Mass: MIT Press, 1960.
- [Rat06] Rathlev, J.: “Ein Werttyp-Konstruktor für Java”. Diplomarbeit. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 2006.
- [Rec06] Rechenberg, P.: *Technisches Schreiben (nicht nur) für Informatiker*. 3. Aufl. Hanser, 2006.
- [RH04] Roy, P. van und Haridi, S.: *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
- [RH08] Rupakheti, C. R. und Hou, D.: “An empirical study of the design and implementation of object equality in Java”. In: *CASCON '08: Proceedings of the 2008 conference of the center for advanced studies on collaborative research*. (Ontario, Canada). New York, NY: ACM, 2008, S. 111–125.
- [Rie06] Riehle, D.: “Value object”. In: *PLoP '06: Proceedings of the 2006 conference on Pattern languages of programs*. (Portland, Oregon). New York, NY: ACM, 2006, S. 1–6.
- [Rit03] Ritterbach, B.: “Eigene Werttypen in Java”. In: *JavaSpektrum* (Juli 2003), S. 46–50.
- [Rit04] Ritterbach, B.: “Support for Value Types in an Object-Oriented Programming Language”. In: *5th Annual International Conference on Object-Oriented and Internet-Based Technologies*. (Erfurt). Hrsg. von Weske, M. und Liggesmeyer, P. Lecture Notes in Computer Science 3263. Berlin / Heidelberg: Springer, 2004, S. 9–23.
- [RRS07] Rathlev, J., Ritterbach, B. und Schmoltzky, A.: “Auf der Suche nach Werten in der Softwaretechnik: Werte und Objekte in objektorientierten Programmiersprachen”. In: *Software Engineering 2007*. (Hamburg). Hrsg. von Bleek, W.-G., Raasch, J. und Züllighoven, H. Bd. 105. LNI. GI, März 2007, S. 261–262.
- [RS03] Riehle, D. und Siberski, W.: *JValue - A Java Framework for Value Objects*. 2003. URL: <http://sourceforge.net/projects/jvalue/> (besucht am 30.06.2013).

- [RS11] Ritterbach, B. und Schmolitzky, A.: “Werttypen in objektorientierten Programmiersprachen: Anforderungen an eine Sprachunterstützung”. In: *Software Engineering 2011: Fachtagung des GI-Fachbereichs Softwaretechnik*. (Karlsruhe). Hrsg. von Reussner, R., Grund, M., Oberweis, A. und Tichy, W. Bd. 183. GI, Feb. 2011, S. 111–122.
- [RS14] Ritterbach, B. und Schmolitzky, A.: “Funktional versus Imperativ”. In: *JavaSpektrum* (März 2014), S. 8–11.
- [Sam81] Sammet, J. E.: “The Early History of COBOL”. In: *History of Programming Languages I*. Hrsg. von Wexelblat, R. L. New York, NY: ACM, 1981, S. 199–243.
- [Sar+12] Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. und Grove, D.: *X10 Language Specification, Version 2.3*. Techn. Ber. IBM Corporation, 15. Okt. 2012.
- [Sar93] Sargeant, J.: “Uniting Functional and Object-Oriented Programming”. In: *Proceedings of the First JSSST International Symposium on Object Technologies for Advanced Software*. London, UK: Springer-Verlag, 1993, S. 1–26.
- [Sch99] Schmolitzky, A.: “Ein Modell zur Trennung von Vererbung und Typabstraktion in objektorientierten Sprachen”. Dissertation. Universität Ulm, 1999.
- [Seb10] Sebesta, R. W.: *Concepts of Programming Languages*. 9. Aufl. Upper Saddle River, NJ: Pearson International, 2010.
- [SEKM00] Schmolitzky, A., Evered, M., Keedy, J. L. und Menger, G.: “Late-bound Pragmatical Class Methods”. In: *Journal of Object-Oriented Programming* 13.7 (Nov. 2000), S. 25–33.
- [SGC07] Syme, D., Granicz, A. und Cisternino, A.: *Expert F#*. Apresspod Series. Apress, 2007.
- [Sim02a] Simons, A. J. H.: “The Theory of Classification, Part 1: Perspectives on Type Compatibility”. In: *Journal of Object Technology* 1.1 (Mai 2002), S. 55–61.
- [Sim02b] Simons, A. J. H.: “The Theory of Classification, Part 4: Object Types and Subtyping”. In: *Journal of Object Technology* 1.5 (Nov. 2002), S. 27–35.
- [SO96] Stoutamire, D. und Omohundro, S.: *The Sather 1.1 Specification*. Techn. Ber. TR-96-012. Berkeley, CA: International Computer Science Institute, 18. Aug. 1996.
- [SOM94] Szyperski, C., Omohundro, S. und Murer, S.: “Engineering a Programming Language: The Type and Class System of Sather”. In: *Programming Languages and System Architectures*. Hrsg. von Gutknecht, J. Lecture Notes in Computer Science 782. Springer Verlag, 1994, S. 208–227.

- [SP03] Stevenson, D. E. und Phillips, A. T.: “Implementing object equivalence in Java using the template method design pattern”. In: *SIGCSE Bull.* 35.1 (2003), S. 278–282.
- [SS03] Saake, G. und Sattler, K.-U.: *Datenbanken & Java : JDBC, SQLJ, ODMG und JDO*. 2. Aufl. Heidelberg: dpunkt.verlag, 2003.
- [SS90] Søndergaard, H. und Sestoft, P.: “Referential transparency, definiteness and unfoldability”. In: *Acta Informatica* 27.6 (Mai 1990), S. 505–517.
- [Str00] Strachey, C.: *Fundamental Concepts in Programming Languages*. Higher-Order and Symbolic Computation. (original paper dated 1967). Apr. 2000.
- [Str10] Stroustrup, B.: *The C++ Programming Language*. Special Edition. Addison-Wesley, 2010.
- [Str13] Stroustrup, B.: *The C++ programming language: C++ 11*. 4. Aufl. Upper Saddle River, NJ: Addison-Wesley, 2013.
- [Str14] Stroustrup, B.: *A tour of C++*. Upper Saddle River, NJ: Addison-Wesley, 2014.
- [Str86] Stroustrup, B.: *The C++ programming language*. Boston, MA: Addison-Wesley Longman, 1986.
- [Sut02] Sutter, H.: *More exceptional C++: 40 new engineering puzzles, programming problems, and solutions*. Boston, MA: Addison-Wesley Longman, 2002.
- [Taf06] Taft, S. T., Hrsg.: *Ada 2005 reference manual: language and standards libraries*. Lecture notes in computer science. Berlin: Springer, 2006.
- [Tur36] Turing, A. M.: “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 2.42 (1936), S. 230–265.
- [Tur37] Turing, A. M.: “Computability and lambda-Definability”. In: *Journal of Symbolic Logic* 2.4 (1937), S. 153–163.
- [Tur85] Turner, D. A.: “Miranda: A Non-strict Functional Language with Polymorphic Types”. In: *Proc. of a Conference on Functional Programming Languages and Computer Architecture*. (Nancy, France). New York, NY: Springer-Verlag, 1985, S. 1–16.
- [Vos14] Voss, R.: *Wissenschaftliches Arbeiten leicht verständlich*. 3. Aufl. Schriftenreihe UTB. Konstanz: UVK Verl.-Ges., 2014.
- [VTFD07] Vaziri, M., Tip, F., Fink, S. und Dolby, J.: “Declarative Object Identity Using Relation Types”. In: *Proc. ECOOP 2007*. Hrsg. von Ernst, E. Lecture Notes in Computer Science 4609. Springer, 2007, S. 54–78.

- [Wad95] Wadler, P.: “Monads for Functional Programming”. In: *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques*. London, UK: Springer-Verlag, 1995, S. 24–52.
- [Wad97] Wadler, P.: “How to declare an imperative”. In: *ACM Computing Surveys* 29.3 (Sep. 1997), S. 240–263.
- [Wat90] Watt, D. A.: *Programming language concepts and paradigms*. Prentice Hall, 1990.
- [Weg87] Wegner, P.: “Dimensions of Object-Based Language Design”. In: *ACM SIGPLAN Notices, OOPSLA '87 Conference Proceedings*. (Orlando, Florida). New York, NY: ACM, Dez. 1987, S. 168–182.
- [Weg90] Wegner, P.: “Concepts and Paradigms of Object-oriented Programming”. In: *SIGPLAN OOPS Messenger* 1.1 (Aug. 1990), S. 7–87.
- [Wer98] Werf, P. van der: “Values and Objects Revisited”. In: *Journal Of Object Oriented Programming (JOOP)* 11.4 (Juli 1998), S. 25–34.
- [WH66] Wirth, N. und Hoare, C. A. R.: “A Contribution to the Development of ALGOL”. In: *Commun. ACM* 9.6 (Juni 1966), S. 413–432.
- [Wij+75] Wijngaarden, A. v., Mailloux, B. J., Peck, J. E. L., Koster, C. H. A., Sintzoff, M., Lindsey, C. H., Meertens, L. G. L. T. und Fisker, R. G.: “Revised Report on the Algorithmic Language ALGOL 68”. In: *Acta Informatica* 5 (1975), S. 1–236.
- [Win10a] Windelband, W.: “Über Gleichheit und Identität”. In: *Sitzungsberichte der Heidelberger Akademie der Wissenschaften, Philosophisch-historische Klasse, 14. Abhandlung* (1910), S. 1–24.
- [Win10b] Winkler, F.: “Benutzerdefinierte Werttypen in C++”. Diplomarbeit. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 8. Sep. 2010.
- [Wir71] Wirth, N.: “The Programming Language Pascal”. In: *Acta Informatica* 1 (1971), S. 35–63.
- [Wit14] Witt, E.: “Umsetzung von Werttypen in Scala”. Bachelorarbeit. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, Juni 2014.
- [WP09] Wampler, D. und Payne, A.: *Programming Scala - Scalability = Functional Programming + Objects*. Beijing: O'Reilly, 2009.
- [Zül05] Züllighoven, H.: *Object-oriented construction handbook - developing application-oriented software with the tools and materials approach*. dpunkt.verlag, 2005.
- [Zül98] Züllighoven, H.: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug & Material-Ansatz*. Heidelberg: dpunkt-Verlag, 1998.

Eigene Veröffentlichungen

Folgende Veröffentlichungen sind aus dieser Dissertation hervorgegangen:

Wissenschaftliche Publikationen

Ritterbach, B.: „Support for Value Types in an Object-Oriented Programming Language“. In: *5th Annual International Conference on Object-Oriented and Internet-Based Technologies*. (Erfurt). Hrsg. von Weske, M. und Liggesmeyer, P. Lecture Notes in Computer Science 3263. Berlin / Heidelberg: Springer, 2004, S. 9 – 23.

Rathlev, J., Ritterbach, B. und Schmolitzky, A.: „Auf der Suche nach Werten in der Softwaretechnik: Werte und Objekte in objektorientierten Programmiersprachen“. In: *Software Engineering 2007*. (Hamburg). Hrsg. von Bleek, W.-G., Raasch, J. und Züllighoven, H. Bd. 105. LNI. GI, März 2007, S. 261 – 262.

Ritterbach, B. und Schmolitzky, A.: „Werttypen in objektorientierten Programmiersprachen: Anforderungen an eine Sprachunterstützung“. In: *Software Engineering 2011: Fachtagung des GI-Fachbereichs Softwaretechnik*. (Karlsruhe). Hrsg. von Reussner, R., Grund, M., Oberweis, A. und Tichy, W. Bd. 183. GI, Feb. 2011, S. 111 – 122.

Sonstige Publikationen

Ritterbach, B.: „Eigene Werttypen in Java“. In: *JavaSpektrum* (Juli 2003), S. 46 – 50.

Ritterbach, B. und Schmolitzky, A.: *Funktional versus Imperativ*. In: *JavaSpektrum* (März 2014), S. 8 – 11.

Eidesstattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Dissertationsschrift selbst verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, den 5.12.2014

Beate Ritterbach