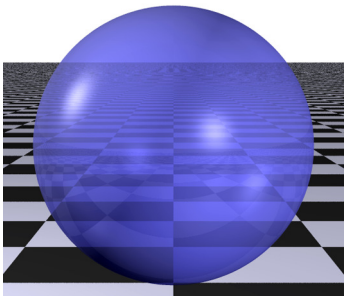
Objektorientierte Modellbildung zwischen Mathematik und Informatik

Von der Alltagssprache
zur
Informatischen und Mathematischen Modellbildung

L. Humbert, J. Kuhaupt, A. Filler, M. Weise & E. Niehaus



AT6FUI Capacity Buildung - Germany
Open Community Approach
Pilot Evolutionary Publications DE-2013-001-V1

9. Oktober 2013

Inhaltsverzeichnis

1 Eigenständige Modellierung – bestimmendes Moment jeder Bildung 10

1.1 Informatische Modellierung – Schlüssel und Ziel informatischer Allgemeinbildung . . . 11

1.1.1 Exkurs – Modelle – philosophisch betrachtet . . . 12

1.1.2 Informatik und ihre Didaktik: der Modellierungsprozess . . . 15

1.1.3 Modellierung und Informatische Literalität . . . 16

1.1.4 Arten der informatischen Modellierung . . . 20

1.2 Kerninformatik als Teil der informatischen Allgemeinbildung . . . 24

1.3 Woher kommen die Objekte? . . . 26

1.3.1 Fehler bei der Gestaltung von Informatiksystemen . . . 28

1.3.2 Konsequenzen für die Gestaltung von Lernprozessen . . . 29

1.4 Informatikunterricht – Modellierungsbeispiel . . . 30

1.5 Modellierungskompetenzen – Informatik und Mathematik . . . 34

1.6 Beschreibung einer Informatikunterrichtsreihe mit dem Schwerpunkt Unterrichtspraxis . . . 37

1.7 Zusammenfassende Thesen – Ausblick . . . 45

2 Objektorientierte Modellierung in der 3D-Computergrafik 58

2.1 Ein einführendes Beispiel . . . 59

2.2 Modellierung geometrischer Objekte . . . 61

2.2.1 Attribute und Methoden geometrischer Objekte . . . 61

2.2.2 Koordinatenbeschreibungen geometrischer Körper . . . 63

2.2.3 Geometrische Transformationen als Methoden . . . 66

2.2.4 Weitere Methoden zur Modellierung von Formen: Boolesche Operationen . . . 69
2.2.5 Ein komplexeres Beispiel: Klasse „schneekugel“ . . . 70
2.3 Fotorealistische Bilderzeugung . . . 72
2.3.1 Das Raytracing-Verfahren . . . 72
2.3.2 Lokale Beleuchtungsmodelle . . . 75
2.3.3 Transparenz und Brechung . . . 82
2.3.4 Zusammenfassung der berücksichtigten Wechselwirkungen; Grenzen des Raytracing-Verfahrens . . . 84
2.4 Einbeziehung in den Mathematik- und Informatikunterricht . . . 86

3 Objektorientierte Schreibweisen in der mathematischen Modellbildung 87
3.1 Einleitung zur mathematischen Formalisierung der Objektorientierung . . . 87
3.2 Klassen und Objekte . . . 91
3.2.1 Beziehung zwischen Objekt und Klasse . . . 92
3.2.2 Beziehung zwischen Klassen - Vererbung . . . 93
3.2.3 Beziehungen zwischen Objekten . . . 93
3.3 Klassen . . . 94
3.3.1 Graphische Darstellung und Bezeichnung . . . 95
3.3.2 Elternklassen . . . 98
3.3.3 Basisklassen . . . 98
3.3.4 Mehrfachvererbung . . . 99
3.3.5 Vererbungsabbildung . . . 100
3.3.6 Oberklassenmenge . . . 101
3.4 Zustände . . . 103
3.4.1 Zustandsabbildung . . . 103
3.4.2 Zustandsraum . . . 104
3.5 Methoden . . . 106
3.5.1 Protokoll einer Klasse . . . 109
3.5.2 Überschreiben von Methoden . . . 111

3.5.3 Klassenbibliothek 111
3.5.4 Zusammenfassung Klassen und Objekt 112
3.6 Lebenszyklus von Objekten 112
3.6.1 Objekte im Adressraum 113
3.6.2 Zeitabhängige Existenz von Objekten 115
3.7 Objektorientierte Programmierung mit JavaScript 122
3.7.1 Zustände und Methoden in JavaScript 122
3.7.2 Integration der Vererbung 123
3.7.3 Überschreiben von Methoden 124
3.7.4 Namenskonventionen für die Objektorientierung 125
3.7.5 Programmbeispiel mit Vererbung in JavaScript 128
3.8 Zusammenfassung: Mathematische Objektorientierung 134

4 Objektorientierte Modellierung und Spracherwerb 135
4.1 Spracherwerbstheorien und OOM 137
4.2 Phasen des Spracherwerbs und der Wortschatzentwicklung – Analogiebetrachtungen zur Objektorientierung 138
4.2.1 Phasierung des L2-Erwerbs 138
4.2.2 Voraussetzungen der Sprachentwicklung 143
4.2.3 Sprachentwicklung anhand der Wortschatzentwicklung . . . 147
4.2.4 Fuzzy-Theorie 148
4.2.5 Schriftspracherwerb 152
4.2.6 Spracherwerb und OOM-Kompetenzen 153
4.3 Schlussfolgerungen – Thesen 154

5 Fazit, Grenzen und Ausblick 156
Literatur 161

Einleitung

Die Objektorientiertung hat ihren Ursprung in der Informatik und nimmt in der Didaktik der Informatik einen besonderen Stellenwert für die informatische Modellbildung ein.

Ziel der folgenden Kapitel an der Schnittstelle zwischen Didaktik der Informatik und Didaktik der Mathematik ist es, die Objektorientierung aus den folgenden Blickwinkeln zu betrachten:

- ObjektOrientierte Modellierung (OOM) in der Didaktik der Informatik,
- Objektorientierung und dreidimensionalen Computergraphik in der Didaktik der Mathematik
- Mathematische Formalisierung der Objektorientierung und
- erste Schritte von der Umgangssprache zur einer objektorientierten Fachsprache an der Schnittstelle zur Linguistik.

In der zeitlichen Entwicklung stellen die ersten Schritte von der Umgangssprache zu einer objektorientierten Fachsprache die Frühphase in einem objektorientierten Lernprozess dar. Aus Darstellungserwägungen ist diese Frühphase im Allgemeinen nur sehr schwierig nachvollziehbar, wenn nicht gewisse objektorientierten Vorinformationen aus den anderen Kapiteln in den Grundzügen bekannt sind. Aus diesem Grund orientiert sich die Einordnung des letzten Kapitels 4 in die inhaltliche Gliederung nicht an einer zeitlichen Entwicklung eines objektorientierten Lernprozesses.

Inhaltliche Gliederung

Aus der oben genannt Motivation entsteht die Notwendigkeit, die Entwicklungen in der Informatikdidaktik und deren Beiträge zu einer eigenständigen Modellierung zu betrachten. Humbert und Kuhaupt haben daher in Kapitel 1 (S.10 ff.) die Leistungen der Informatikdidaktik für eine eigenständige Modellierung sowohl aus theoretischer Sicht als auch vor einem unterrichtspraktischen Hintergrund betrachtet. Aus dieser fachdidaktischen Sichtweise auf informatische Modellierungsthemen können allgemeinere Aspekte der Modellierung als bestimmendes Moment jeder Bildung abgeleitet werden, die sich in analytischen und konstruktiven Aspekten der Modellierung äußern.

Für die Mathematikdidaktik besteht daher das Ziel, diese Beiträge der Informatikdidaktik und die damit gewonnenen Erfahrungen in der Ausbildung von Schülerinnen und Schülern für die Mathematische Modellbildung nutzbar zu machen. Damit ist nicht gemeint, dass die Mathematik nun originäre informatische Lernziele übernimmt, sondern an der Schnittstelle zwischen Mathematik und Informatik die informatischen Repräsentation der Modellbildung auf geeignete mathematische Repräsentationsformen angewendet werden. In diesem Sinn ist nicht die Mathematik Hilfswissenschaft für andere Disziplinen, sondern Didaktik der Informatik wird hilfreicher Partner der Modellbildung in der Mathematikdidaktik.

Informatische Repräsentationen als Umsetzung der Modellierungsergebnis können ablauffähige Programme sein, die in der Mathematik in mathematische Repräsentationen als Produkt des Modellierungsprozesses ersetzt werden. Um diesen Transfer von der Informatik zur Mathematik leisten zu können wurde von Filler in Kapitel 2 (S.58 ff.) die 3D-Graphik betrachtet und Möglichkeiten der Anwendung der Objektorientierung auf mathematische Objekte untersucht. Objektstrukturen in Geometrie, deren Transformation durch Abbildungen und die Klassifizierung von geometrischen Objekten zeigen Realisationen von Objektbeziehungen, deren Effekte sich in den visuellen Produkten zeigen.

Die objektorientierte Modellierung in der 3D-Computergrafik wurde durch die Erstellung der 3D-Grafiken mithilfe einer OpenSource- Grafiksoftware ausgefÃ¼hrt, die mit Szenenbeschreibungen arbeitet. Die verwendet Software arbeitet mit einer Syntax, die aber nicht direkt objektorientierte Strukturen in Analogie zu anderen Programmiersprachen (wie z.B. Java) vollständig unterstützt. Dennoch liefert die objektorientierte Sichtweise eine enge Verbindung zwischen erzeugten sichtbaren 3D-Objekte und der syntaktischen Beschreibung von geometrischen Transformationen (z.B. Verschiebung, Spiegelung, Drehung, Streckung, ...) als *Methoden* und *Objekteigenschaften*, wie Farbe, Glanz, Transparenz und Brechungseigenschaften, die Zustände eines dargestellten Objektes definieren.

Abschließend wird in Kapitel 3 (S. 87 ff.) die symbolisch Repräsenetationsmöglichkeiten mit mathematischer Formalisierung untersucht, die die Produkte einer objektorientierten Analyse auch in einer mathematischen Notation sichtbar hält. ZIel dieses Abschnittes ist es, in Analogie zur Informatik die Strukturerhaltung im Übergang von den objektorientierten Modellierungsresultaten auch in objektorientierter Programmsyntax sichtbar zu halten. Dies ist insbesondere in einem Modellierungskreislauf wesentlich, wenn nach einer Implementa-

tion eines Modellierungsergebnisse weitere Optimierungsschritte des Modells notwendig werden. Durch die objektorientierte Strukturerhaltung in der Syntax können identifiziertbare Modellierungsfehler im implementierten Modellierungsprodukt besser in der syntaktischen Repräsentation des Produktes identifiziert werden. Die mathematische Formalisierung der Objektorientierung ist ein erster Schritt auf diesem Weg, um den informatischen Mehrwert der Objektorientierung auch in der mathematischen Modellbildung verfügbar zu machen. An dieser Stelle schließt sich der Kreis insbesondere mit dem Kapitel 1, da in der Informatikdidaktik entscheidende Vorarbeiten für die unterrichtlichen Zielsetzungen bereits erarbeitet wurden. Oft ist die Mathematik Hilfswissenschaft für informatische Werkzeuge. In diesem Fall wird die Informatikdidaktik zu einem Wegbereiter für didaktische Ansätze der objektorientierten Modellierungen innerhalb der Mathematik.

Formale Sprachen in der Mathematik und Informatik haben wie natürlich Sprachen ein Syntax und eine Semantik. Dieser Spracherwerb kann man als strukturäquivalent zum Erlernen einer Zweitsprache auffassen und die inhaltliche Fähigkeiten zur Objektorientierung in der Muttersprache in äquivalente Ausdrucksfähigkeiten in formaleren Sprachen zu überführen. Diesen linguistischen Blickwinkel auf die Objektorientierung nehmen Humbert und Weise in Kapitel 4 (S. 135 ff.) ein.

Evolutionäre Publikationen und OpenContent

Die einzelnen Kapitel wurden unter einer Creative Commons Lizenz veröffentlicht, die eine nicht- kommerzielle weitere Nutzung der Kapitel ermöglicht. Die Motivation für diesen Ansatz zielt auf einen Zugriff auf Informationen und Literatur, der nicht von dem Geldbeutel der Lernenden abhängen sollte. Damit kann man die Inhalte dieses Buches kostenfrei im Netz für nicht-kommerzielle Zwecke zugänglich machen. Käufer dieses Buches in einer gedruckten Version unterstützen durch ihren Kauf die Infrastruktur, die den freien Zugang zu den Inhalten im Web ermöglicht. Dieser OpenContent-Pilot wird ausgewertet und die Ergebnisse in Nachfolgeversionen dieses Dokumentes an dieser Stelle diskutiert.

Im Verlauf des Piloten in dieser Reihe wird sich ferner zeigen, wie evolutionäre Prinzipien einer Wikipedia-Artikel-Entwicklung auf Printausgaben übertragbar sind. Erste Voraussetzung dafür ist eine Print-On-Demand-Erzeugung der aktuellen Ausgabe. Mit einem Versionsmanagement können dann die Nutzer ent-

scheiden, welche Version diese in der Printausgabe erhalten möchten. Dies kann bis zu einer aktuellen Arbeitsversion von Einzelartikel reichen.

Die Partizipation an der Entwicklung des Buches kann über eigene neue Beiträge erfolgen oder als Ergänzung zu bestehenden Artikeln ggf. mit Einverständnis der Autorinnen und Autoren, die bisher an der Veröffentlichungen gearbeitet haben und in den Lizenzvereinbarung mit Creative Commons bisher keine Abwandlungen an des Artikels erlauben.

Korrekturhinweise unter Verbesserungsvorschläge können unter

`http://at6fui.weebly.com`

ab November 2013 (> Open Community Approach (OCA) > Evolutionary Publishing) angegeben werden. Diese werden dann in der nächsten Version aufgegriffen und diskutiert. Kritische Anmerkungen werden ebenfalls in den Band aufgenommen und Erweiterungen durch weitere Artikel können mit Verweis auf den Inhalt ebenfalls genannt werden. Zielsetzung ist in Analogie zu einem Wiki eine stetige Weiterentwicklung der Inhalte unter Einbeziehung der Nutzerinnen und Nutzer dieser Dokumente. Wie bei OpenSource-Entwicklungen kann so eine partizipatorische Weiterentwicklung unter Namensnennung für Dokumente ermöglicht werden. Der nächste Schritt wäre die Integration eines unabhängigen Begutachtungsprozesses, der einerseits die Anonymität der GutachterInnen sichert und die Gutachten aber auch direkt der Weiterentwicklung der Publikationen dient. Mit Genehmigung der Autoren werden die Gutachten auch als Anhang zur OCA-Publikation veröffentlicht werden. Unveröffentlichte Gutachten zählen nicht als Begutachtungsprozess. Positive Gutachten sollten mit Namen der GutachterInnen veröffentlicht, damit personelle Beziehungen zu den Autoren ggf. transparent werden. Insgesamt soll diese Publikationsart außerhalb von klassischen Journalen mit Begutachtungsverfahren als Option verstanden werden, Zugang zur Information ohne finanzielle Schranken mit einem transparenten Begutachtungsverfahren zu ermöglichen. Der OCA-Pilot bedient lediglich die erste Phase in dieser Entwicklung.

Insgesamt sind wissenschaftliche Beiträge zur Weiterentwicklung der ObjektOrientierte Modellierung (OOM) in der evolutionären Weiterentwicklung der Publikation willkommen.

Der Grad der Partizipation anderer AutorInnen orientiert sich auch an dem Interesse, den die Objektorientierung aus didaktischem Blickwinkel an der Schnittstelle zwischen mathematischer und informatischer Modellbildung besitzt.

Die Publikationsstrategie folgt einem Open Community Approach (OCA), der den freien Zugang zu Wissen und Publikationen ermöglicht. Neben der eigentlichen wissenschaftlichen Arbeit an Artikeln werden mit Publikationen als Wirtschaftgut auch ökonomische Ziele verfolgt. Diese können im Wesentlichen wie folgt klassifiziert werden.

- *Kommerzieller Standard: Die LeserInnen zahlen:* In gängigen Publikationsszenarien von Verlagen ist die Einreichung von wissenschaftlichen Artikel kostenfrei und die wissenschaftliche Gutachter sorgen ohne Honorar in einem Peer-Reviewing- Verfahren für die Qualtiätssicherung.
- *Kommerzielles OpenAccess: Die AutorInnen zahlen:* In einem OpenAccess-Szenario begegnen Verlage dem Wunsch des freien Zugangs zu Wissen und Erkenntnis mit einer Strategie, die dem Leser den freien Zugang zu den Publikationen gewährt. Da Verlage als Unternehmen im Kern gewinnorientiert sind, müssen dann die Autoren in der Regel nicht unerheblichen Beiträge für die Publikation in OpenAccess bezahlen.

Beide Szenarien sind aus Sicht eines OCA nicht ideal, da diese insbesondere für Entwicklungsländer durch die Kosten entweder im Standardfall der Zugang zu den besten und einflussreichsten Publikationen durch die finanzielle Hürde erschweren oder im Fall von OpenAccess die Autoren aus Entwicklungsländern ihre wissenschaftlichen Ergebnisse durch die Autorenkosten nicht eingereicht können. Immer die kostenlose Variante für LeserInnen und AutorInnen zu wählen ist ebenfalls keine Lösung, da die kostenfreie Einreichung von Artikel dann für die LeserInnen kostenpflichtig werden und damit gerade die wissenschaftliche Resultate *aus* Entwicklungsländern *für* Entwicklungsländern nicht ihre Adressaten erreichen können.

Diese Veröffentlichung zur Objektorientiertung ist kein Kernthema für Entwicklungsländer. Dennoch dies ein Pilot für diesen Typ von Publikationen, bei denen freie Verfügbarkeit im Web gewährleistet wird und lediglich die Verkaufszahlen des Buches die Kostendeckung der Publikationskosten decken müssen (administrative Kosten, Serverkosten, ...). Ob dieses Konzept langfristig bei freier Verfügbarkeit der Inhalte im Web zur Kostendeckung führt, kann zum jetzigen Zeitpunkt noch nicht beantwortet.

Abschließend möchte ich den anderen Autoren danken, dass sie dieser Piloten für ein echtes OpenAccess für LeserInnen und AutorInnen ermöglichen, da der

freie webbasierte Zugang zu der Veröffentlichung einer kommerziellen Verwertung der Inhalten entgegensteht.

Kapitel 1

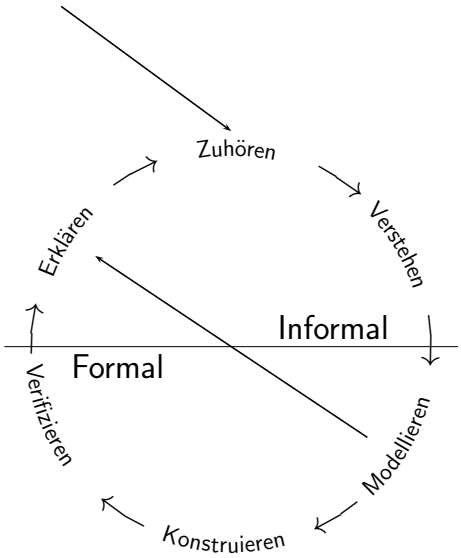
Eigenständige Modellierung – bestimmendes Moment jeder Bildung

Prof. Dr. Ludger Humbert
Fachgebiet Didaktik der Informatik
Bergische Universität Wuppertal
humbert@uni-wuppertal.de

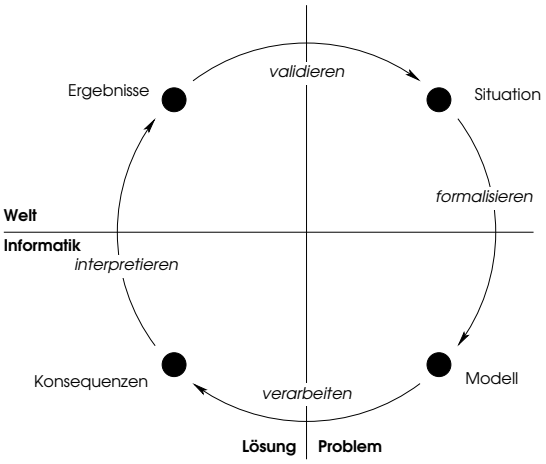
Johannes Kuhaupt
Evangelisches Gymnasium Lippstadt
kuhaupt@eg-lippstadt.de

Dieses Teildokument steht unter der »Creative Commons Lizenz« BY-NC-ND. Dies bedeutet: bei weiterer Verwendung des Textes sind die Namen der Autoren zu nennen, die Weiternutzung darf ausschließlich nicht kommerziell erfolgen, das Dokument darf nicht bearbeitet werden. Weitere Details zu den Creative Commons Lizenzen finden sich unter: `http://creativecommons.org/licenses/?lang=de`

Gut – wir sind Informatiker, Informatikfachdidaktiker und Informatiklehrer – so gesehen ist die thematische Klammer für diesen Beitrag recht deutlich dadurch zu charakterisieren, dass wir bei der Modellierung von der informatischen Modellierung ausgehen. Die Vielgestaltigkeit des Modellierungsbegriffs ist eine große Herausforderung für die Begriffsbildung, die durch den Fachbezug auf die Informatik eine Perspektive erhält, der wesentlich eine konstruktive Dimension in-



(Klaeren & Sperber, 2007, S. 6)



(Humbert, 2006, S. 14)

Abbildung 1.1: Der Prozess der Modellierung in der Informatik

newohnt. Aus diesem Bezug lässt sich die Zuschreibung der ingenieurwissenschaftlichen Arbeitsweise der Informatik ableiten. Dass die Wissenschaft Informatik zugleich auch eine Struktur- und eine Geisteswissenschaft ist, macht die Einordnung in den Wissenschaftskanon nicht gerade einfacher und dies mag einer der Gründe sein, die bisher dazu führten, dass dem Schulfach Informatik der notwendige Zugang zur allgemeinen Bildung verwehrt wurde.

1.1 Informatische Modellierung – Schlüssel und Ziel informatischer Allgemeinbildung

In dem vorliegenden Beitrag wird die informatische Modellierung – im Unterschied zu dem Ausweis in den Bildungsstandards Informatik (GI, 2008) – als Prozess verstanden, der die Implementierung und den Test der entwickelten Problemlösung umfasst (vgl. Abbildung 1.1, S. 11) – nur so entsteht der vollständige Modellierungskreis(lauf).

In den Bildungsstandards Informatik wird die Implementierung bei den Prozessen zusammen mit der Modellierung explizit ausgewiesen (Prozessbereich: Modellieren und Implementieren). Dieser explizite Ausweis hat seine Ursache darin, dass es bei der Ent-

wicklung der Bildungsstandards Informatik ein wichtiges Anliegen war, zu vermeiden, dass ein unvollständiger Modellierungsbegriff auf Seiten der Lehrkräfte dazu führen könnte, dass nur ein begrenzter Teil des Modellierungsprozesses unterrichtlich umgesetzt wird. Da diese Gefahr durchaus besteht, ist der explizite Ausweis nachvollziehbar.

1.1.1 Exkurs – Modelle – philosophisch betrachtet

In einem Beitrag zur computergenerierten Simulation macht KRÄMER deutlich, wie vielschichtig der Begriff der Modellierung ist (Krämer, 2011, S. 310f):

> Die Verwendung des Modellbegriffes scheint von kaum zu überblickender Vielgestaltigkeit: ein Spielzeugauto, das Modell beim Aktzeichnen, die Konstruktionszeichnung eines Hauses, das Diagramm über Organisationsstrukturen einer Firma, ein Atommodell, Gleichungen für Planetenbahnen, ein Begriffssystem, das die Axiome einer Theorie erfüllt: all dies kann jeweils als ›Modell‹ charakterisiert werden und diese Aufzählung ließe sich immer weiter fortsetzen. Gibt es trotz dieser Mannigfaltigkeit so etwas, wie eine ›Familienähnlichkeit‹ zwischen all diesen Versionen des Modellseins? Vier Gesichtspunkte bieten sich dafür an:
>
> (1.) Proportionalität: Schon die begriffsgeschichtliche Herkunft von ›Modell‹ aus dem lateinischen ›modulus‹ der Verkleinerungsform von ›modus‹: Maß weist in die Richtung von etwas ›Maßstäblichem‹. Es geht um eine Ähnlichkeit in der Form, in der Struktur, in der Proportion, demgegenüber die objekthaften, also substanziellen Gesichtspunkte zugunsten der relationalen zurücktreten. Eine solche ›Ähnlichkeit in der Form‹ nannten die Griechen: ›analogia‹.
>
> (2.) Handlungsbezug: Diese Ähnlichkeit darf allerdings nicht missverstanden werden: Sie ist nicht ›von Natur aus gegeben‹, sondern wir machen etwas zu einem Modell. Der Modellcharakter ist in einem Urteilsakt fundiert. Nur für handelnde Subjekte und im Zusammenhang ihrer praktischen oder theoretischen Interessen, kann etwas zu einem bestimmten Zeitpunkt ein Modell sein. Modelle rechtfertigen sich durch die Tätigkeiten, die wir mit und an ihnen vollziehen können.

(3.) Vereinfachung: ein durchgängiger Zug beim Modellieren ist die Reduzierung und Verkürzung. Es gibt zwischen dem, was Modell ist und dem, wofür bzw. wovon es Modell ist, stets ein Komplexitätsgefälle: das Modell hat kleiner, überschaubarer, einfacher, abstrakter zu sein. Allerdings zielt diese Vereinfachung und Veranschaulichung zumeist auf das ›Wesentliche‹. Daher erfüllen Modelle oftmals die Funktion einer Idealisierung.

(4.) Autonomie: Etwas kann Modell für etwas oder von etwas sein, sowohl als Vorbild, wie als Nachbild fungieren. Zumeist allerdings verbinden sich der normativvorschreibende und der abbildlichbeschreibende Aspekt. Unerachtet dieser Bezugnahmen steht außer Frage, dass das Modell eine eigenständige Realität bildet, innerhalb deren in freier Gestaltung etwas ausprobiert und erkundet werden kann. Im Fluchtpunkt dieser Aspekte zeichnet sich eine für Modellierungen weichenstellende Einsicht ab, die auch eine Grundlage ist, um über computergenerierte Simulationen nachzudenken: Innerhalb der Wissenschaften können Modelle als Werkzeuge des Erkenntnishandelns eingesetzt werden.

aus (Krämer, 2011, S. 310f)

Die Überlegungen von KRÄMER lassen sich mit Zielrichtung und Modellierungsabsicht im Prozess der Entwicklung und Umsetzung informatischer Modelle nicht vollständig zur Deckung bringen. Die bei KRÄMER zum Ausdruck gebrachte Perspektive der Autonomie von Modellen (vgl. ihren Punkt (4.)) ist eine durchaus spannende Facette, die allerdings für den Modellierungskontext der Informatik bereichsspezifisch erweitert werden muss: Die Wirksamkeit der informatischen Modellierung entfaltet sich nämlich durch die immanente Änderung des Modellierungsbereichs, da die Modellierung in der Implementierung diesen Bereich verändert.

Um einen Bezug zur allgemeinen Bildung herzustellen, wird – ausgehend von dem oben weit gefassten informatischen Modellierungsbegriff – die Eigenschaft der Rückwirkung in den modellierten Bereich bedeutsam: Bildung verändert die Sicht auf die Welt – die informatische Modellierung verändert die Welt; es spricht also vieles dafür, sich diesem wirkmächtigen Prinzip als notwendigem Bestandteil jeder allgemeinen Bildung zu nähern. Diese Feststellung ist zunächst nicht weiter verwunderlich, allerdings ist es überraschend, dass die dadurch zum Aus-

druck gebrachten Möglichkeiten bis heute in der Diskussion um die allgemeine Bildung nur wenig Berücksichtigung finden.

Es ist festzustellen, dass der informatischen Bildung und damit der informatischen Modellierung auch ohne Verfügbarkeit von Informatiksystemen in der Hand der Schülerinnen und Schüler der allgemeinbildende Charakter zugesprochen werden muss, ebenso wie der mathematischen Bildung der allgemeinbildende Charakter – unabhängig von »Rechenmaschinen« (oder anderen technischen Rechenhilfen) – zuzuschreiben ist. Informatische Allgemeinbildung orientiert sich damit nicht primär an der Zieldimension »Informatiksysteme bedienen«; vielmehr besteht der Nutzen der Informatiksysteme im Kontext der Modellierung in der Prüfung der entwickelten Lösungsvorschläge.

Selbstverständlich soll (und darf) das Lebensweltargument nicht ignoriert werden: In der Lebenswelt der Schülerinnen und Schüler sind Informatiksysteme omnipräsent. Diese Situation hat allerdings für die Diskussion um die Allgemeinbildung durchaus nicht nur positive Effekte, wird doch damit eine Zuschreibung vorgenommen, die in der falschen Aussage gipfelt: »Damit Schüler Informatiksysteme **bedienen** können, bedarf es des Informatikunterrichts«.[1] Jedes Informatiksystem ist Ergebnis informatischer Modellierung. Ein Grundverständnis für die Welt setzt somit voraus, dass jede mündige Bürgerin und jeder mündige Bürger den Prozess der informatischen Modellierung versteht. Dieses Verständnis wird nicht durch einfache Erklärungen erzielt, sondern es bedarf der Eigentätigkeit, die darin besteht, dass Probleme aus der Lebenswelt der Schülerinnen und Schüler mit Hilfe der informatischen Modellierung aufgeschlossen und mit Mitteln der Informatik bearbeitet und gelöst[2] werden.

Im Laufe der Zeit wurden in der Fachwissenschaft Informatik verschiedene Modellierungsansätze entwickelt, die – sachlich nicht korrekt – als Paradigmen bezeichnet werden[3]. Es ist zielführender, die Ansätze als »informatische Modellierungssichten« zu bezeichnen. Dabei gehen wir davon aus, dass die Church-Turing-These zutrifft und damit die Modellierungssichten bezüglich ihrer Mächtigkeit äquivalent sind.

[1] Bedienen steht hier im Gegensatz zu **verständnisvoll und absichtsvoll nutzen**.

[2] Dabei sind auch Grenzen der durch die Informatik (und damit von Informatiksystemen) lösbaren Probleme lerngruppenadäquat und fachlich korrekt zu diskutieren.

[3] Diese Einschätzung bezieht sich auf den KUHNschen Paradigmenbegriff.

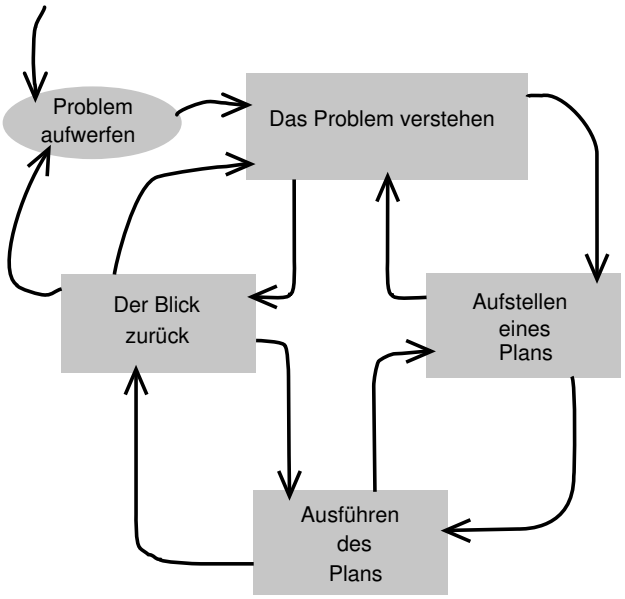


Abbildung 1.2: Der Problemlöseprozess in der Mathematik nach PÓLYA

1.1.2 Informatik und ihre Didaktik: der Modellierungsprozess

Eine verallgemeinernde Sicht auf Modellierungsprozesse wird in Form von Kreisprozessen präsentiert (vgl. Abbildung 1.1, S. 11). Sie werden in der Fachwissenschaft Informatik in Grundlagenwerken (siehe z. B. (Klaeren & Sperber, 2007)) als zentrales Element ausgewiesen. Wie THOMAS in seiner fachdidaktisch orientierten Dissertation (Thomas, 2002) quantitativ nachweisen konnte, steht informatische Modellierung im Zentrum der Fachwissenschaft.

Im Zusammenhang mit der Entwicklung der Bildungsstandards Informatik wird der Prozess der informatischen Modellierung als Kreisprozess ausgewiesen. Fachliche und fachdidaktische Darstellungen unterscheiden sich in der gewählten Sprache – nicht aber strukturell – voneinander. Bei der didaktischen Diskussion um die mathematische Modellierung finden wir ebenfalls die Darstellung als Kreisprozess (vgl. (Neumann, Heinze, Ufer & Neumann, 2011, S. 605) [4]).

In beiden Darstellungen der Abbildung 1.1 (S. 11) kann die 1989 von dem niederländischen Informatiker DIJKSTRA (1930–2002) geforderte Trennung zwischen dem formalen Kern der Arbeit der Informatik und dem sogenannten »Gefälligkeits-

[4] In dem von NEUMANN, HEINZE und UFER 2011 verfassten Beitrag (Neumann et al., 2011) wird die informatische Modellierung nicht erwähnt.

problem« durch eine Brandmauer deutlich gemacht werden. Diese Trennung zwischen »der Welt« und »der Informatik« kann durch die waagerechte Linie in den Grafiken gekennzeichnet werden. Damit konzentriert sich – nach Meinung von DIJKSTRA – die Aufgabe der Informatik auf die Erfüllung der formalen Anforderungen, die durchaus komplex genug sind, um wissenschaftlich bearbeitet zu werden. DIJKSTRA setzt damit in der in (Denning, 1989) dokumentierten Diskussion einen Kontrapunkt zu den anderen Diskutanten, die die Bearbeitung der Probleme des Anwendungsbereichs in der Informatik berücksichtigen wollen und diese als Bestandteil der Arbeit in der Informatik betrachten.

Der zwischen der Informatik und der Mathematik aufscheinende Unterschied tritt im Fall der Modellierung deutlich zutage: In der Fachwissenschaft Informatik werden Heuristiken zum informatischen Problemlösen thematisiert und diskutiert. Eine solche Darstellung findet in Grundlagenveranstaltungen für Mathematikstudierende keinen Platz. Dabei stellt der Problemlöseprozesses und seine Thematisierung eine notwendige Voraussetzung für das Verständnis bereit (Pólya, 1945). Die von PÓLYA vorgenommene Strukturierung des Problemlöseprozesses ist in Abbildung 1.2 (S. 15) dargestellt.

Im Unterschied zur Fachwissenschaft Informatik[5] finden wir in der Mathematik außerhalb der Didaktik kaum eine fachlich geführte Auseinandersetzung um den Prozess der mathematischen Modellierung, die Einfluss auf die fachliche Ausbildung des Nachwuchses hat.

1.1.3 Modellierung und Informatische Literalität

Frühe informatikfachdidaktische Überlegungen (vgl. exemplarisch (Bauersfeld, Otte & Steiner, 1977)) durchzieht die Erkenntnis, dass dem informatischen Modellierungsprozess[6] didaktisch wesentlich Aufmerksamkeit gewidmet werden muss. Da die Ergebnisse des Modellierungsprozesses als Quelltexte in Programmiersprachen formuliert werden, ist eine Verständigung auf möglichst orthogonale Konzepte unabdingbar. Seit Beginn der Entwicklung konkreter Programmiersprachen wird diese Verständigung sehr unterschiedlich gehandhabt und führte bisher noch nicht zu einer gemeinsamen Kommunikationsgrundlage. Im Folgenden werden einige der Baustellen dokumentiert, die für die Fachdidaktik bedeut-

[5] Vgl. exemplarisch die Tagungsreihe »Modellierung« der Gesellschaft für Informatik: `http://www.informatik.uni-trier.de/~ley/db/conf/modellierung/index.html`

[6] Zu Beginn bezog sich die Ausgestaltung primär auf den Prozess der Softwareentwicklung.

sam sind. Die verschiedenen Ebenen sind hier mit Begriffen bezeichnet worden, um eine Schichtung (hier sequenziell bottom-up dargestellt) zu ermöglichen. Die Reihenfolge ist so gewählt worden, weil sich gerade für die atomare Ebene offenbar zur Zeit kaum jemand zu interessieren scheint, diese konstitutiv für die Informatik ist und es angezeigt ist, hier – auch gegen Widerstände – mit der Arbeit zu beginnen.

Der Prozess der Herausbildung der Literalität ist für Strukturwissenschaften konstitutiv. Folgend gehört dieser Teil der Fachsozialisation zu jeder fachlichen Grundbildung. Dies muss sich auf die gesamte Bildungsbiographie beziehen. Aber auch aus fachlichen Sicht bedarf es hier einiger Anstrengungen, die zu erledigen sind, wenn informatische Bildung gelingen soll.

Bei Umsetzungen der mit der informatischen Modellierung entwickelten Ergebnisse kommt dem Aspekt der informatischen Literalität auf der Mikroebene eine bisher offenbar vernachlässigte Qualität zu, daher wird diesem Bereich im Folgenden etwas breiter dargestellt.

Informatische Literalität – die Mikroebene (I)

Fachlich bedeutsame Eigenschaften sind auf der Zeichenebene einheitlich so zu verwenden, dass das Prinzip der semiotischen Orthogonalität gewahrt wird. Der Prozess der Zuschreibung genau einer Bedeutung für ein Zeichen im Fachkontext ist wesentlicher Bestandteil der Entwicklung und der Konturierung einer Fachsprache auf der Zeichenebene. Die Zeichenebene ist eine der atomaren Ebenen der Literalität.

Die syntaktische Einbindung und erst recht die Semantik des $\boxed{=}$-Zeichens an verschiedenen Stellen der Informatik soll untersucht werden. Folgende Bedeutungen können mit diesem Zeichen im Zusammenhang mit der praktischen und der angewandten Informatik verbunden werden: (i) Tabellenkalkulation: Einleitung der Berechnungsvorschrift für den Inhalt einer Zelle, (ii) Programmiersprachen (ALGOL-Familie): Vergleichsoperator, (iii) Programmiersprachen: Wertzuweisung, ...

Zunächst finden wir – geschichtlich betrachtet – andere Zeichen, die für die Wertzuweisung Verwendung finden: Zwischen 1942 und 1945 entwickelte ZUSE die erste höheren Programmiersprache – den Plankalkül. Dort verwendet ZUSE ein eigenes Symbol für die Zuweisung $\boxed{\Rightarrow}$.

Bei dem Vorschlag für ALGOL 58 (IAL – International Algebraic Language) wur-

de von RUTISHAUSER die Einführung des Symbols $\boxed{\leftarrow}$ für die Wertzuweisung angeregt (Beer, 2006, S. 13). Der Vorschlag für die Notation der Wertzuweisung durch $\boxed{\leftarrow}$ wurde in Form des $\boxed{:=}$ realisiert; vorlaufend und folgend in anderen Programmiersprachen (von FORTRAN bis Java) als $\boxed{=}$.

Damit leistet die Informatik – durch eine fachdidaktisch höchst fragwürdige Darstellung – Fehlvorstellungen Vorschub: Es ist nicht eindeutig, was unter dem $\boxed{=}$ zu verstehen ist. Für Programmiersprachen, die nicht der ALGOL-Sprachfamilie zugerechnet werden, wurde das $\boxed{=}$ für die Wertzuweisung gewählt. Damit ist es nötig, für den Vergleichsoperator »ist gleich« ein anderes Zeichen auszuwählen. Üblicherweise wird das $\boxed{==}$ festgelegt.

Informatische Literalität – die Mikroebene (II)

Eine ebenfalls atomare Ebene ist die Entscheidung für eine Lese- und [Ab-]Arbeitsrichtung. Wir befinden uns kulturell im Abendland und die Leserichtung ist kulturell von links nach rechts festgelegt – auch dies hat unmittelbar Einfluss auf das Orthogonalitätsprinzip; wird diese Richtung aufgekündigt, müssen sehr gute Gründe als ursächlich angegeben werden. Hierzu ist festzustellen, dass ZUSE die Leserichtung von links nach rechts auch als Arbeitsrichtung ansieht, daher hat er eine Festlegung für die Richtung der Zuweisung in den Rechenplänen von links nach rechts durch $\boxed{\Rightarrow}$ vorgenommen. Diese »Richtungsentscheidung« wird mit ALGOL umgekehrt – bisher sind mir keine Belege bekannt, die erläutern, warum diese Entscheidung so getroffen wurde.

Informatische Literalität – die Mikroebene (III)

Für die Formulierung der Ablaufstrukturen in Algorithmen wird eine weitere fachdidaktisch bedeutsame Diskussion geführt. Durch den provokant gewählten Titel **Go To Statement Considered Harmful** (vgl. (Dijkstra, 1968)) nimmt WIRTH als zuständiger Editor der Communications of the ACM eine Pointierung vor, die DIJKSTRA nicht so scharf formuliert hätte. Allerdings wird damit die wichtige fachliche Diskussion um die »strukturierte Programmierung« eingeleitet.

Informatische Literalität – die Makroebene (I)

Mit der »schrittweisen Verfeinerung« (»stepwise refinement« –in (Wirth, 1971)) finden wir wenige Jahre später Überlegungen zur Strukturierung der nächsten

Ebene durch Modularisierung und Hierarchisierung von Problemlöseprozessen. Beide Elemente haben sich als überdauernde Prinzipien der Modellierung herausgestellt.

Informatische Literalität – die Makroebene (II)

Eine weitere Stufe in der Entwicklung fachlicher Elemente im Zusammenhang mit der Gestaltung des Modellierungsprozesses wird von PARNAS die Idee der Kapselung von Elementen vorgelegt (Parnas, 1972), die folgend nur über sauber definierte Schnittstellen mit der Umgebung kommunizieren können.

Konsequenz für die Lehre

Die so entwickelte fachliche Strukturierung von Teilen des informatischen Modellierungsprozesses hat Konsequenzen für die Gestaltung der Abläufe zur Bearbeitung von Problemen. Diese fachlichen Gesichtspunkte müssen auch in der Lehre ihren Niederschlag finden. Daher wird der Interdependenz zwischen Inhalt und Prozess durch projektorientierte Arbeitsformen (auch im Informatikunterricht) Rechnung getragen. Diese didaktischen Überlegungen basieren auf fachlichen[7] und fachdidaktischen Erkenntnissen.

Ziele von Modellen – abhängig von der Bezugswissenschaft

Modelle der Geisteswissenschaften und der Naturwissenschaften stellen Muster der Welterklärung und -interpretation bereit – bei der informatischen Modellierung jedoch haben wir es mit der Umkehrung dieser Verhältnisse zu tun. Jede/r Modellierende kann gestaltend an und in der Zukunft mittun. Unsere Zielmaßgabe ist nicht primär darin zu sehen, neue oder andere Erklärungen zu suchen und zu finden. Es ist vielmehr das Ziel der informatischen Allgemeinbildung, ein Verständis der Welt zu erreichen, das – bereits in der Schule – in der eigenständigen Veränderung wirksam wird. In der Mathematik ist dieses Phänomen – allerdings nicht auf der Ebene und als Bestandteil der allgemeinen Bildung – wohlbekannt: das durch »meine« Axiome aufgespannte Universum zeigt beweisbar Eigenschaften, die mir bedeutsam sind.

[7] Die projektorientierte Arbeitsweise ist Bestandteil der Ausbildung und der Berufspraxis von Informatikerinnen und Informatikern.

Die informatische Modellierung ermöglicht es allen Schülerinnen und Schülern, die Primärerfahrung der Gestaltung in einer Form zu machen, die in anderen Fachkontexten in dieser Weise nicht möglich ist.

1.1.4 Arten der informatischen Modellierung

Über die Zeit wurde die informatische Modellierung in verschiedener Weise ausgestaltet. Betrachten wir ausgewählte Arten der informatischen Modellierung unter ihrem Bezug zu den häufig »gewünschten« sogenannten Standardanwendungen. Das sind Werkzeuge, die für alle möglichen(?) – vormals papiergebundenen – Arbeiten in einem Büro konzipiert wurden. Die Gestaltung ist unter fachdidaktischen und bildungsorientierten Maßgaben grundsätzlich fragwürdig, da sie keinem Bildungsziel verpflichtet ist, sondern Büroprozesse abbildet und diese unterstützen soll. Es ist festzustellen, dass verschiedene Arten der Modellierung als Grundlage herangezogen werden müssen, um die diesen Werkzeugen und damit ihrer Nutzung zugrundeliegenden Ideen zu verstehen und zielführend einsetzen zu können (vgl. Tabelle 1.1). Allerdings sind einige Bemerkungen zu dieser Tabelle angezeigt:

- Fehleinschätzungen zu notwendigen Bedingungen für die objektorientierte Modellierung haben dazu geführt, dass die (verkürzte) These vertreten wird, dass »die Welt aus Objekten besteht« und es daher nicht notwendig sei, bei der adäquaten objektorientierten Modellierung kerninformatische Grundlagen zu berücksichtigen.
- Sowohl E-Mail wie auch Hypertexte werden der objektorientierten Modellierung zugeordnet, obwohl beide ohne eine zustandsorientierte Modellierung der benutzten Protokolle keine Funktionalität haben. Insofern zeigt sich, dass die Beschränkung auf eine Art der Modellierung informatischer

Art der Modellierung	**Bezug zu Informatiksystemen**
funktionale Modellierung	Tabellenkalkulation
wissensbasierte Modellierung	Datenbanken, Expertensysteme
objektorientierte Modellierung (OOM)	Dateisystem, Textverarbeitung, E-Mail, Hypertexte, Präsentationen
zustandsorientierte Modellierung	Automaten, Benutzungsschnittstellen

Tabelle 1.1: Arten informatischer Modellierung – Bezug zu Informatiksystemen

> Systeme nicht zielführend ist. Für die Zuordnung der Benutzungsschnittstelle von Informatiksystemen gilt entsprechend: Die Entwicklung von Benutzungsschnittstellen wird i. d. R. objektorientiert vorgenommen.

Bei dieser Strukturierung der Arten informatischer Modellierung darf die grundlegende Einbettung jeder informatischen Modellierung in einen informatischen Fachkontext nicht unterschlagen werden. Der Fachkontext stellt eine notwendige Randbedingung für die fachlich korrekte und daher didaktisch zu berücksichtigende Dimension dar.

Zur Einlösung des Anspruchs: Wer kann Informatik unterrichten?

Jede informatische Modellierung benötigt ein gerüttet Maß an Kerninformatik, die im Zusammenhang mit anderem Fachunterricht (z. B. Mathematik) weder zeitlich noch von der Primärqualifikation der Lehrkräfte geleistet werden kann. Diese Grundlage fehlt auch Lehrkräften in anderen Ländern, wie MILLER in seiner Dissertation ausführt (Miller, 2004). Die Begrifflichkeit in der englischsprachlichen Literatur ist bisher wenig entwickelt – daher findet der Terminus »computer programming« und nicht modelling/modeling Verwendung. Allerdings ist klar, dass die Programmierung nicht ohne eine Einbettung in den Modellierungsprozess gelingen – geschweige denn verstanden werden – kann (vgl. die entsprechenden Bermerkungen von DENNING in Abschnitt 1.2).

> Probably the greatest impediment to wider adoption of computer programming in school settings is the lack of adequate teacher education. Few teachers take it upon themselves to learn programming, and fewer still master it well enough to incorporate it into either their existing classes, or new classes specifically designed to teach programming. Thus, research is needed to provide guidance as to how to rectify this situation: Require programming in teacher education programs? Offer continuing education credits? Seek out computer professionals that are willing to teach? Develop student mentors to assist teachers with programming activities? I doubt that any solution will result in revolutionary changes, but if promising evolutionary approaches can be identified, then eventually there should be enough teachers comfortable with computer programming that students will not have difficulty using their programming knowledge in

classroom learning activities (Miller, 2004, S. 245f).[8]

Die Schulinformatik hat – nicht nur im Bundesland Nordrhein-Westfalen – mangels qualifizierter Lehrkräfte lange gebraucht, um von den bzgl. der Modellierung trivialen und im Sinne der informatischen Allgemeinbildung wenig zielführenden Beispielen[9] zur informatischen Modellierung zu gelangen. Desiderate dieser Fehlentwicklung finden sich bis heute in vielen Materialien und Lehrwerken.

Die Verbindung zwischen einer Problemsituation und einer Lösung in Form eines Informatiksystems besteht in der informatischen Modellierung. Um die Lösung umzusetzen, bedarf es geeigneter Werkzeuge in Form von graphischen Beschreibungssprachen (z. B. ER-Modellierung, UML, Struktogramm) und Programmiersprachen. Diese sind allerdings mehrheitlich nicht unter Berücksichtigung didaktischer Fragen und Anforderungen entworfen worden. WIRTH, der sich um die Entwicklung von Programmiersprachen unter didaktischen Gesichtspunkten verdient gemacht hat (er entwickelte u. a. die Programmiersprachen ALGOL-W, Euler, Pascal, Modula-2 und Oberon), stellt fest (vgl. (Wirth, 2002, S. 3 des PDF-Dokuments)):

> What can be done? It is very difficult indeed to counteract a global trend as an individual. I think of the trend to shift from long term planning to short term profits, from learning in order to understand to applying skills for immediate action. Considering our own subject, the field of computing and programming, an academic institution's ultimate goal must be much wider than the mastery of a language. It must be nothing less than the art of designing artifacts to solve intricate problems. Some call it the art of constructive thinking. It is in this context that the availability of an appropriate tool, a properly designed language, is of importance. It assumes the role of a theory on which we base our methods. How can anyone learn to design properly and effectively, if the formalism, the foundation, is an overwhelming, inscrutable mess? How can one learn such an art without master examples worth studying and following?

[8] Es muss darauf hingewiesen werden, dass MILLER mit »programming« auch die Aktivitäten bezeichnet, die bei uns mit der informatischen Modellierung assoziiert werden.

[9] Zu nennen sind hier exemplarisch »Hello World«, Temperatureinheitenumwandlungen, Lösen quadratischer Gleichungen oder auch die Bestimmung des »Body Mass Index«.

Es ergeben sich gleich mehrere Handlungsnotwendigkeiten: zunächst gilt es, herauszuarbeiten, wie Wege von einer Problemsituation zur Lösung – didaktischen Anforderungen genügend – unterrichtlich erschlossen werden können. Die Wege sind so zu gestalten, dass am Wege liegende fachbezogene Instrumente (Gegenstände und Methoden) zur Unterstützung und zur Realisierung in den Lernprozess eingewoben werden. Außerdem ist den Schülerinnen und Schülern der Blick über das Spezifische hinaus auf allgemeine Strategien zur Problembewältigung im Fachkontext zu ermöglichen.

Es bietet sich an, dabei die Erfahrungen der Profession reflektiert einfließen zu lassen. Ein Beispiel stellt die grundlegende Vorgehensweise zur Entwicklung von Informatiksystemen dar. Bereits in der ersten Veröffentlichung zu dem sogenannten Wasserfallmodell zum Softwareentwicklungsprozess (Royce, 1970) wird auf die iterative Struktur dieses Prozesses hingewiesen. Die Rezeption dieses konstitutiven Elements findet im Rahmen der partizipativen Softwareentwicklung statt (vgl. (Floyd, Mehl, Reisin, Schmidt & Wolf, 1987) – die Überlegungen stehen im Zusammenhang mit staatlichen Maßnahmen zur seinerzeit angestrebten »Humanisierung der Arbeitswelt«).

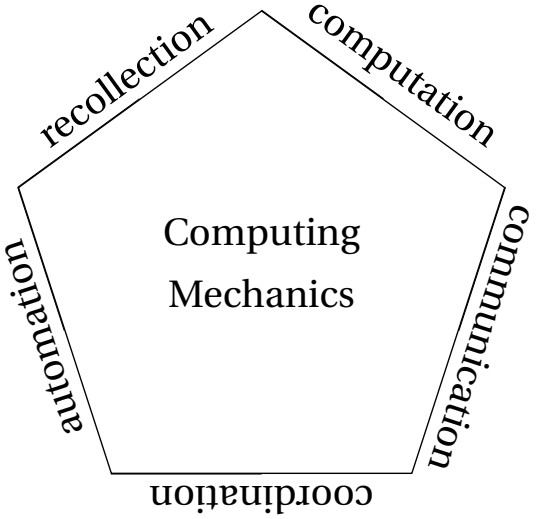


Abbildung 1.3: Die fünf Fenster von DENNING – nach (Denning, 2003, S. 18)

1.2 Kerninformatik als Teil der informatischen Allgemeinbildung

Neben Arbeiten zu den für die informatische Allgemeinbildung zu berücksichtigenden Fachmethoden und -gegenständen finden sich Überlegungen, um die **großartigen Prinzipien der Informatik** zu thematisieren. DENNING identifiziert die Punkte, die eine Fokussierung der Darstellung und Kommunikation der Wissenschaft Informatik[10] gestatten:

»It is time for us to make ourselves known by saying **our mechanics**, **our design principles**, and **our practices**. It is time to stop hiding the enormous depth and breadth of our field« (Denning, 2003, S. 19f, **Hervorhebungen** lh).

Aus der Physik wird der Begriff »mechanics« entlehnt, um die großartigen Prinzipien der Informatik zu fassen. DENNING identifiziert »The five windows of computing mechanics« als »Computation, Communication, Coordination, Automation, Recollection« (vgl. (Denning, 2003, S. 17f) und Abbildung 1.3, S. 23).

DENNING versteht seinen Vorschlag als Reaktion auf die mißlungenen Versuche, Informatik für Studierende besser zugänglich zu machen. Anstatt weiter Listen von Gegenständen vorzuschlagen, die curricular abgedeckt werden sollen, sind die hinter den Gegenständen stehenden Prinzipien zur Strukturierung aufzudecken und heranzuziehen. Darüber hinaus wird wesentlich hervorgehoben, dass Informatik nicht mit Programmierung gleichgesetzt werden darf:

»For years, many others have seen our field as programming. Through our 1989 Computing as a Discipline report (Denning et al., 1989) we hoped to encourage new curricula that would overcome this misleading image. But this was not to be« (Denning, 2003, S. 19f).

Bemerkenswert ist in diesem Zusammenhang der im Jahr 1992 veröffentlichte Aufruf von BRAUER und BRAUER an die Wissenschaftsgemeinde der Informatik, der vier Arten von Unterstützungssystemen ausweist, die durch die Informatik gestaltet werden (vgl. Abbildung 1.4, S. 25). Ausgehend von einer Analyse der Entwicklung der Fachwissenschaft über die Zeit dokumentieren BRAUER und BRAUER hier die Notwendigkeit der Weiterentwicklung (vgl. (Brauer & Brauer, 1992, S. 12) und weisen die fünf »Co«s – »communication, cooperation, collaboration,

[10] DENNING meint tatsächlich Informatik – dies hat er an anderer Stelle sehr deutlich dokumentiert: »The European synonym for computer science—informatics—more clearly suggests the field is about information processes, not computers« (Denning, 2005, S. 28).

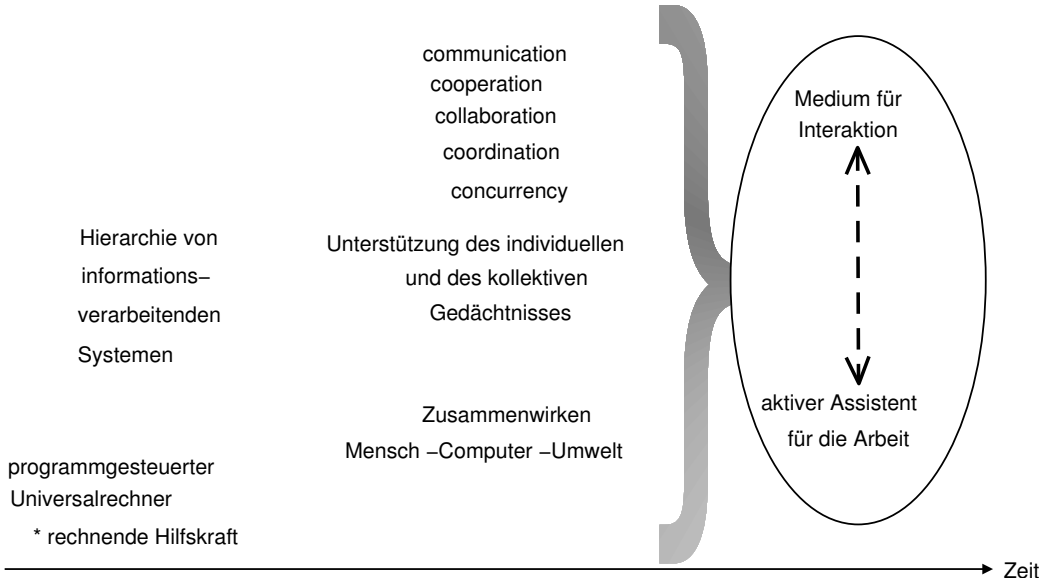


Abbildung 1.4: Wissenschaftliche Herausforderungen für die Informatik

coordination, concurrency« – Hervorhebungen von lh) als einen Bereich der Weiterentwicklung zur Strukturierung der Fachwissenschaft Informatik aus.

Zusammenfassend ist festzustellen, dass zunehmend in der Fachwissenschaft erkannt wird, dass eine Umorientierung notwendig ist, die sich grundlegenden Fachstrukturen verpflichtet. Allerdings ist bis heute nicht erkennbar, welche Folgen die hier dokumentierten Forderungen für die Weiterentwicklung der Fachwissenschaft und vor allem ihrer Lehre tatsächlich haben. Zwei Linien sind deutlich erkennbar: Einmal (vgl. BRAUER und BRAUER) wird »vom Ende« her argumentiert, d. h. eine Antwort auf die Frage gesucht: Wozu soll die Wissenschaft Informatik gesellschaftlich beitragen? In der Beantwortung dieser Frage wird eine Struktur entwickelt, bei der – über die Zeit betrachtet – die Sicht auf Elemente der Wissenschaft geordnet werden und die Perspektive, dass Informatiksysteme als ein Ergebnis der wissenschaftlichen Arbeit Menschen in ihrer Arbeit unterstützen und als Medium für die Interaktion zwischen Menschen dienlich sein sollen (vgl. Abbildung 1.4, S. 25). Die Perspektive von DENNING hingegen besteht darin, »aufzuräumen« und eine Strukturierung fachlicher Elemente unter gemeinsamen, einheitlichen Leitlinien vorzunehmen. Beide Ansätze führen zur Klärung der Frage, wie die Fachwissenschaft inhaltlich und perspektivisch gestaltet werden kann.

Für die Gestaltung der allgemeinen informatischen Bildung ist nunmehr die Fra-

ge zu klären, wie (= Prozesse) Gegenstände (= Inhalte) den Schülerinnen und Schülern im Bildungsprozess nahe gebracht werden sollen/können/müssen. Einige grundlegende Hinweise zur Beantwortung werden durch die Bildungsstandards Informatik (vgl. (GI, 2008)) gegeben. Obwohl dort nicht ausschließlich der Objektorientierung das Wort geredet wird, ist deutlich, dass dieser Modellierung eine prioritäre Rolle zuwächst. Damit ist allerdings nicht die Frage beantwortet, wie Objekte in der Informatik und erst recht nicht, wie Objekte in der informatischen Bildung gefunden werden oder gefunden werden sollen. Darüber hinaus gibt es ein Problem, das bereits in der Lehre der Informatik zu Schwierigkeiten geführt hat: Wie kann von der Objektsicht auf die Klassensicht abstrahiert werden. Da wir wissen, dass dieser Schritt selbst für Informatikstudierende eine Hürde im Lernprozess ist, die häufig genug nicht vernünftig gemeistert wird, sollten Strategien gefunden werden, die diese Schritt für Schülerinnen und Schüler handhabbar gestalten.

Wie in anderen Fällen, sollte nicht verabsäumt werden, die innerinformatische Diskussion zu berücksichtigen, bevor dieser Schritt in Arbeit genommen wird.

1.3 Woher kommen die Objekte?

Die durchgängige Objektorientierung ist unter wissenschaftstheoretischen Aspekten durchaus kritisch zu betrachten:

In der objektorientierten Modellierung – zur Zeit der wohl meistpraktizierte Weg zur fachlich orientierten informatischen Allgemeinbildung – setzen die Protagonisten darauf, dass durchgängig »eine Sprache« – nämlich »die Sprache der Objekte«[11] benutzt wird. **Durchgängig** bezeichnet hier den kompletten Weg, der durch den Modellierungskreis (vgl. Abbildung 1.1, S. 11) vorgezeichnet ist.

Hier allerdings liegt eine Quelle für viele – bisher in der Didaktik nicht diskutierte und antizipierte – Mißverständnisse und Fehlvorstellungen[12]. Nähern wir uns dem Problem aus der fachwissenschaftlichen Perspektive, so ist festzustellen, dass im Jahr 1994 von dem Arbeitskreis zur Terminologie der Softwaretechnik in der Gesellschaft für Informatik grundlegende Zuarbeiten vorgelegt wurden (vgl. (Hesse, Barkow, Braun, Kittlaus & Scheschonk, 1994)), die in der Fachdidaktik bis heute nicht sorgfältig rezipiert werden.

[11] Und nicht etwa »die Sprache der Klassen«.

[12] Auf eine dieser Fehlvorstellungen wurde oben hingewiesen: »Die Welt besteht aus Objekten«.

Philosophische Position/Disziplin	Paradigma	Erklärung(en)
Naiver Realismus	Schlaraffenland	Objekte »sind da«, »fliegen uns zu«
Externer Realismus	Garten Eden, »God's Eye View«	Objekte = Früchte in Nachbars Garten
Abbildtheorie	Spiegelkabinett	Beobachter schafft »Abbild« der »Realität«
Interner Realismus	»cookie cutter«, Töpferwerkstatt	Objekte werden vom Betrachter durch Handeln »geformt« (Putnam, 1999)
Konstruktivismus	Dichterwerkstatt	Sprachhandlungen sind maßgeblich zur Objektbildung
Mathematische Logik	Briefmarken-sammlung	Objekte sind Elemente eines formal definierten Individuenbereichs
Objektorientierte Programmierung	Sandkasten	Objekte sind formale, symbolische Entitäten (Goldberg & Robson, 1983)
Objektorientierte Analyse (OOA)	Sensationszirkus	Ontologische (»real-weltliche«) Objekte werden mit sprachlichen identifiziert

Tabelle 1.2: Objekte – philosophisch betrachtet (nach (Hesse & Braun, 2001, S. 781))

Wir gestatten uns hier einen Ausflug in die Begriffswelt »Objekt«.

> Zum Objektbegriff gibt es viele unterschiedliche Zugänge. In der Informatik besonders populär ist die Ansicht, dass die Objekte »von Natur aus gegeben« seien und von den System-Entwicklern nur »aufgelesen« und adäquat »abgebildet« werden müssten. Setzt man sich mit einem solchen Standpunkt kritisch auseinander, so wird sofort die außerordentliche Rolle offenbar, die Sprache bei der Identifikation und Behandlung von Objekten spielt.
>
> Schließlich setzt jeder Umgang mit Objekten eine Bezugnahme voraus – und das geschieht neben der sprachlichen Beschreibung durch Zeichen – Worte, Symbole, Bilder – oder was immer es uns ermöglicht, auf Objekte Bezug zu nehmen (Hesse & Braun, 2001, S. 776).

Aus diesen grundlegenden Bemerkungen werden Fragen abgeleitet, die verschiedene Arten des Zugangs zu der Objektsicht charakterisieren (aus (Hesse & Braun, 2001, S. 777)):

- Wo kommen die Objekte her – wer bestimmt, was ein Objekt ist und was nicht?
- Welche sprachlichen Mittel werden benutzt, um Objekte zu bezeichnen und zu charakterisieren?
- Wie verhalten sich Zeichen und die durch sie bezeichneten Objekte zueinander?

Um diese Fragen einer Klärung zuzuführen, können die bei der informatischen Modellierung auftretenden Fehler qualifiziert werden. Die Analyse der Fehler sollte Hinweise auf eine notwendige Schwerpunktverschiebung bei den Elementen der informatischen Modellierung liefern.

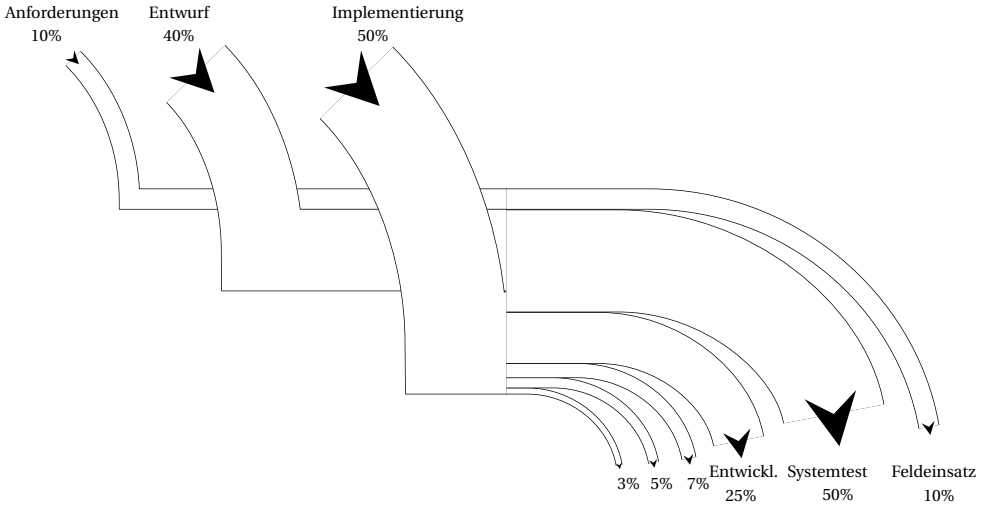


Abbildung 1.5: Eintrag und Behebung von Fehlern der Modellierung (nach (Möller & Paulish, 1993))

1.3.1 Fehler bei der Gestaltung von Informatiksystemen

Die – im Zusammenhang mit der Objektorientierung – aufscheinenden und zu klärenden Fragen haben durchaus die Qualität, den Modellierungsprozess nachhaltig zu beeinflussen. In vielen Werken zur objektorientierten Modellierung wird diesen Fragen nicht Rechnung getragen – typischerweise »fallen die Objekte vom Himmel« (vgl. Tabelle 1.2). Werden Heuristiken (bis hin zu Vorgehensmodellen) angegeben oder gefordert, so mögen diese im Sinne der späteren Implementierung für Einzelfälle durchaus zu angemessenen Ergebnissen führen, allerdings

fehlt in der Regel die kritische Betrachtung der Annahmen und vor allem fehlen Überlegungen zur tatsächlichen Verallgemeinerungsfähigkeit für die Heuristik. Autoren grundlegender Werke zur Fachwissenschaft Informatik beschäftigen sich nicht oder kaum mit solchen Fragen.

Schauen wir uns Daten zum Softwareentwicklungsprozess an, müssen wir feststellen, dass in allen Phasen der Entwicklung Fehler in die Modellierung eingebracht werden. Allerdings werden in jeder Phase auch Fehler beseitigt (vgl. Abbildung 1.5, S. 28). Dabei ist unter ökonomischen Gesichtspunkten zu beachten, dass die Fehlerbeseitigung in den verschiedenen Phasen Kosten verursacht, die um Größenordnungen von einander abweichen (vgl. Tabelle 1.3).

Wichtig wäre es also, gerade zu Beginn des Modellierungsprozesses sehr genau darauf zu achten, dass Entscheidungen über die Gestaltung der ersten Schritte sehr breit diskutiert werden. Diese Sicht ist bei Überlegungen, die von Objekten ausgehen, »die ohnehin schon da sind, und nur gefunden werden müssen« nicht zu erwarten. Dies ist der ersten Fragestellung aus (Hesse & Braun, 2001, S. 777), die dort als »ontologisch-erkenntnistheoretischer Zugang« zu Objekten charakterisiert wird, zuzuordnen.

Da mit der informatischen (i. d. R. objektorientierten) Modellierung tatsächlich die Grundlage für die Erstellung realer Systeme gelegt wird, handelt es sich durchaus um einen kritischen Punkt. Wir wissen, dass Softwareprojekte nach wie vor in großer Zahl völlig oder teilweise scheitern. Der Anteil erfolgreich abgeschlossener Projekte liegt – seit Jahren kaum verändert – bei ungefähr $\frac{1}{3}$. Wir wissen darüber hinaus, dass nach einer vorgenommenen Analyse der Aufwand zur Korrektur in Abhängigkeit von der vergangenen Zeit enorm steigt (vgl. Tabelle 1.3). Festzuhalten ist, dass 50% der Fehler beim Systemtest und 10% im Feldeinsatz festgestellt werden (vgl. Abbildung 1.5, S. 28).

1.3.2 Konsequenzen für die Gestaltung von Lernprozessen

Wir kommen nicht umhin, die seit Beginn der Informatik festgestellten (und bisher nicht grundlegend geklärten) Fragen der informatischen Literalität zum Ausgangspunkt für didaktisch gestaltete Anforderungen an Programmiersprachen als einem wesentlichen Element der informatischen Modellierung ernsthaft zu überdenken.

Phase der Fehlerauffindung	Fehlerbehebungs-kosten (in € pro Fehler)	Faktor
Entwurfsreview	150–250	1
Implementierung/Entwicklungstest	1.500–15.000	10 – 100
Systemtest	3.000–30.000	20 – 200
Feldeinsatz	5.000–50.000	33 – 333

Tabelle 1.3: Kosten der Fehlerbehebung in Abhängigkeit von der Phase der Entdeckung (nach (Möller & Paulish, 1993))

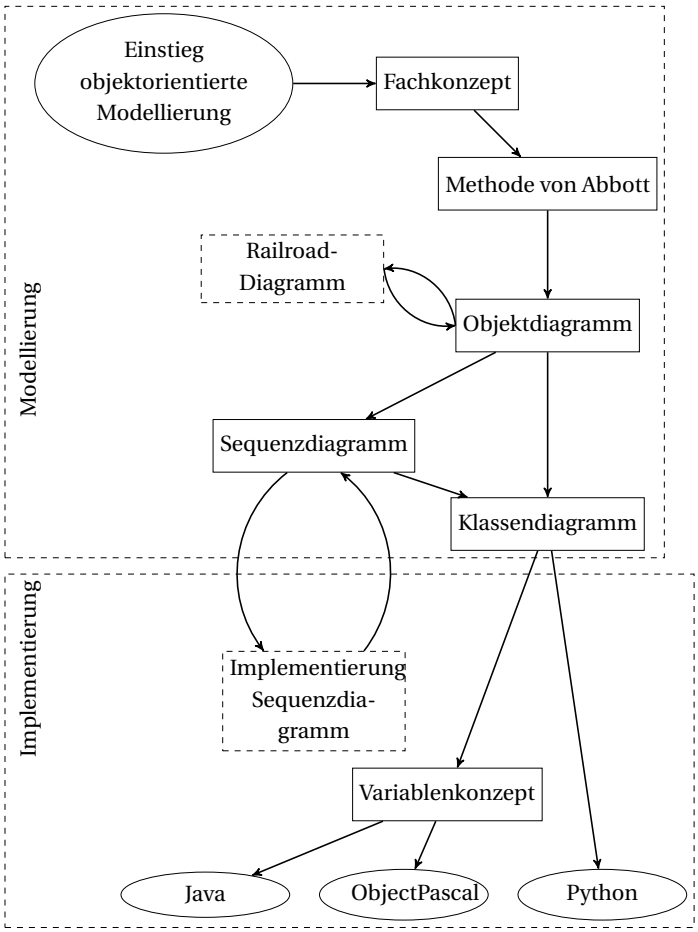
1.4 Informatikunterricht – Modellierungsbeispiel

Kann unter diesen Bedingungen Informatikunterricht so gestaltet werden, dass zumindest ein Teil der Anforderungen an die informatische Modellierung erfüllt wird? Es ist aus den oben formulierten Gründen für das Mißlingen der Modellierung angezeigt, eine klare Schwerpunktsetzung der objektorientierten Modellierung auf die Betrachtung der Objekte vorzunehmen.

Daher ist für den Informatikunterricht ein Vorgehensmodell zu wählen, das dieser Schwerpunktsetzung gerecht wird und damit den Modellierungsprozess der Schülerinnen und Schüler begleitet. Ein Vorschlag für eine Abfolge von Schritten ist in Abbildung 1.6, S. 31 dargestellt.

Im Folgenden werden die einzelnen Schritte dieser Abfolge exemplarisch an einer mit Schülerinnen und Schülern durchgeführten Modellierung illustriert. Für eine problemhaltige Situation (vgl. Abbildung 1.7, S. 47) wird das informatische Fachkonzept modelliert. Dazu wird das Problem mit der Methode von Abbott auf Kandidaten für Objekte untersucht. Diese werden auf Objektkarten dokumentiert und es wird diskutiert, wie die Objekte interagieren, um das Problem zu lösen und damit den dynamischen Aspekt der Problemlösung auf der Objektebene zu klären (vgl. Abbildung 1.8, S. 48).

Wie im Vorgehensmodell (vgl. Abbildung 1.6, S. 31) dokumentiert, sollte an dieser Stelle mit den Schülerinnen und Schülern die formale Konstruktion von gültigen Bezeichnern thematisiert werden. Im Zusammenhang mit der syntaktischen Beschreibung der Programmiersprache Pascal wurde von WIRTH das Element der »Railroad«-Diagramme eingesetzt, um mit einer grafisch eindeutigen Darstellung die Grammatik der Sprache zu beschreiben. Diese Darstellung eignet sich



(aus (Kuhaupt, 2012, S. 36) – (Boettcher et al., 2007, S. 221))

Abbildung 1.6: Vorgehensmodell für den Informatikunterricht

besonders gut, um durch »Nachfahren« eine gültige Konstruktion zu entwickeln – aber auch, um eine gegebene Konstruktion auf Korrektheit zu prüfen.

Die Regeln, nach denen korrekte Ableitungen von Wörtern einer formal definierten Sprache erzeugt werden können, werden in den Problembearbeitungsprozesses an der Stelle eingebunden, an der sie benötigt werden.

Bei der Interaktion verändern Objekte ihren Zustand und ggf. ihre Beziehungen zu anderen Objekten. Damit stellt ein Objektdiagramm in seiner statischen Form immer einen Schnappschuß dar, der eine konkrete Situation im Ablauf zeigt. Um die Dynamik geeignet abzubilden und darzustellen, zu dokumentieren und zu erproben, empfehlen sich Objektspiele, die protokolliert werden. Aus den Proto-

kollen lassen sich Sequenzen von Aktionen ableiten, die zu Sequenzdiagrammen zusammengefasst werden.

Für die vorgestellte Problemstellung wurde ein Kontext gewählt, den Schülerinnen und Schüler aus ihrer Lebenswelt kennen. Da sie aber typischerweise eine – aus Sicht der Informatik – fachlich korrekte Sicht auf diesen Gegenstand nicht kennen, begleiten zusätzliche Hinweise den Prozess der Bearbeitung. In dem Informatikkurs, dessen Ergebnisse hier dokumentiert werden, wurde im Unterricht nicht mit ortsfesten Informatiksystemen, sondern mit Mobiltelefonen (Smartphones) gearbeitet. Die im konkreten Unterricht verwendeten Systeme weisen eine einheitliche Verzeichnisstruktur auf (vgl. Abbildung 1.9, S. 49). Diese Situation ist für eine Verallgemeinerung nicht tragfähig. Die Aufgabenstellung kann so verändert werden, dass die Schülerinnen und Schüler für ihre eigenen Systeme eine ähnliche Problemsituation formulieren und bearbeiten.

Die geschilderte Problemsituation und ihre Lösung kann mit Hilfe einer »vorher« – »nachher« Darstellung der Verzeichnisstrukturen (inkl. der hier in Rede stehenden Dateiobjekte) vorgenommen werden. In der Abbildung 1.10 (S. 50) wird dieses Ergebnis der Überlegungen der Schülerinnen und Schüler vorgestellt.

Zwischenstand der Bearbeitung

Mit den Schülerinnen und Schülern wurde besprochen, wie eine Darstellung der Verzeichnisstruktur der Situation als »Baum« aussieht. Die in dem Informationsblatt zu Betriebssystemen und Verzeichnissen (vgl. Abbildung 1.9, S. 49) vorgenommene Visualisierung wurde von den Schülerinnen und Schülern als Vorlage für die Verzeichnisstruktur herangezogen. Am Ende der Unterrichtsstunde standen die in der Abbildung 1.10 (S. 50) dargestellten Verzeichnisbäume an der Tafel.

Der fachliche Aufschluß erfolgt durch die einzelnen Schritte in einem Sequenzdiagramm, wie es in Abbildung 1.11 (S. 51) dokumentiert ist.

Die Übersetzung von Sequenzdiagrammen in eine objektorientierte Prorammiersprache kann mit einem festen Satz von Regeln erfolgen, so dass bereits in diesem Durchgang ablauffähige Programme erstellt werden können. Allerdings sind für diese Programme von der Lehrkraft die entsprechenden Objekte vorzubereiten, so dass die Schülerinnen und Schüler diese über eine Schnittstelle nutzen können, um die erstellten Programme zu erstellen und zu testen, um damit zu überprüfen, ob ihre Modellierung zweckmäßig ist.

Das Sequenzdiagramm wurde in ablauffähigen Quellcode übersetzt. Dieser wird

hier nicht dokumentiert. Er kann unter `http://is.gd/avatus` eingesehen werden.

Erst nachdem die Schülerinnen und Schüler den Objektbegriff soweit internalisiert haben, dass sie in Lage sind, den Analyseprozeß von der Situationsbeschreibung bis zum Sequenzdiagramm an einem anderen Beispiel selbstständig durchführen können, ist der Übergang zu der nächsten Abstraktionsstufe (vgl. Abbildung 1.12, S. 52) angezeigt.

Mit diesem Vorgehen soll erreicht werden, dass die Schülerinnen und Schüler die zentralen Unterschiede zwischen Objekt und Klassen verstehen und bei der Modellierung angemessen berücksichtigen können. In den gängigen Einführungen und in den uns vorliegenden Lehrwerken für den Einsatz in der Sekundarstufe I und die Sekundarstufe II für die Hand des Schülers wird dieser Aspekt nicht berücksichtigt.

Das Vorgehen zur Klassifizierung folgt damit nicht der in einführenden Werken für Informatikstudierende verwendeten Bauplanmetapher, sondern geht bei der Begriffsbildung von den Objekten aus: aus der Modellierung des konkreten Problemkontextes werden Objekte abgeleitet. In der Arbeit mit diesen Objekten stellt sich heraus, dass (und dies ist bei der Konstruktion der problemhaltigen Situation bereits zu berücksichtigen) es immer mindestens zwei Objekte gibt, die über die gleichen Attribute und Methoden verfügen und sich damit als Kandidaten für den Schritt der Klassifizierung anbieten. In der Abbildung 1.12 (S. 52) ist das Ergebnis dieses Klassifizierungsprozesses dargestellt.

Der hier dargestellte Prozess wurde in den zurückliegenden Jahren in den Fachseminaren Informatik der Studienseminare – Zentren für schulpraktische Lehrerausbildung – Hamm und Arnsberg entwickelt und verfeinert. In diesem Zusammenhang sind referierte Beiträge für die Tagungen Informatik und Schule 2005 in Dresden, 2007 in Siegen und einige zweite Staatsarbeiten (Hausarbeiten im Rahmen der zweiten Phase der Lehrerbildung) entstanden, auf die hier verwiesen werden soll. Unter `http://is.gd/ihv4D3` findet sich eine Liste der Arbeiten mit einigen Verweisen auf die vollständigen Dokumente.

1.5 Modellierungskompetenzen – Informatik und Mathematik

Betrachten wir das im letzten Abschnitt vorgestellte Modellierungsbeispiel aus der Informatik, wird deutlich, dass der informatische Modellierungsprozess die Chance bietet, sowohl die Komplexität des Faches zu erfahren als auch einen didaktisch gestalteten Prozess zu initiieren, mit dem die Problemsituation durch die Schülerinnen und Schüler konstruktiv aufgelöst wird. Am Weg liegende Fachfragen werden in den Prozess der Bearbeitung so integriert, dass Kompetenzen im Kontext aufgebaut werden, die bezüglich der Lösung des konkreten Problems benötigt werden.

Als Unterstützung für den Modellierungsprozess finden sich vielfältige fachliche grafische Beschreibungsmittel, von denen einige in diesem Beitrag in Abbildungen (vgl. Abb. 1.8, S. 48, Abb. 1.11, S. 51 und Abb. 1.12, S. 52) dokumentiert sind. Hinzu kommen die erwähnten »Railroad«-Diagramme zur Syntaxbeschreibung, die ebenfalls hier problembezogen dargestellten Baumstrukturen (vgl. Abb. 1.9, S. 49 und Abb. 1.10, S. 50) aber auch Struktogramme zur programmiersprachenunabhängigen Darstellung von Abläufen, die zur Festlegung der Aktionen in den Methoden benötigt werden.

In den Bildungsstandards Informatik (vgl. (GI, 2008, S. 13)) werden im Prozessbereich **Modellieren und Implementieren** folgende Kompetenzen ausgewiesen.

Schülerinnen und Schüler aller Jahrgangsstufen

- erstellen informatische Modelle zu gegebenen Sachverhalten
- implementieren Modelle mit geeigneten Werkzeugen
- reflektieren Modelle und deren Implementierung

In dem oben dargestellten Beispiel werden selbstverständlich weitere inhaltliche und prozessbezogene Kompetenzbereiche berührt. Unsere Darstellung in diesem Beitrag richtet allerdings den Fokus auf den Prozessbereich **Modellieren**. Hier lässt sich sehr deutlich ablesen, dass die Erstellung des informatischen Modells, sowie die Implementierung und Reflexion dieses Modells gefordert sind. Damit findet sich der vollständige Modellierungskreis (vgl. Abbildung 1.1, S. 11) als Element der Bildungsstandards Informatik.

Die Untersetzung der Bildungsstandards für Jahrgangsstufengruppen detailliert die Anforderungen für den Prozessbereich Modellieren und Implementieren (hier exemplarisch 5–7, vgl. (GI, 2008, S. 19)).

Schülerinnen und Schüler der Jahrgangsstufen 5 bis 7

- betrachten Informatiksysteme und Anwendungen unter dem Aspekt der zugrunde liegenden Modellierung
- identifizieren Objekte in Informatiksystemen und erkennen Attribute und deren Werte

Damit werden die Grundlagen auch für unser oben dargestelltes Modellierungsbeispiel gelegt. Gerade die Fülle der Detailelemente, die es bei der informatischen Modellierung zu berücksichtigen gilt, macht deutlich, wie notwendig eine biografisch langfristig angelegte informatische Allgemeinbildung ist, die den Prozess der informatischen Modellierung umfasst.

Wenden wir uns der Mathematikdidaktik zu, müssen wir zunächst feststellen, dass – wie bereits oben (vgl. Abschnitt 1.1.2) dargestellt, die Diskussion um die mathematische Modellierung nicht aus dem Fach gespeist wird, sondern im Wesentlichen eine Domäne der jüngeren Fachdidaktik ist. Symptomatisch zeigen dies die von MAASS aufgeworfenen Fragen der Mathematikdidaktik:

1. Was sind Modellierungskompetenzen?
2. Inwieweit können Lernende der Klasse 8 Modellierungskompetenzen erwerben?

Als Ergebnis der Untersuchung dieser Fragen kommt MAASS zu der folgenden Definition von Modellierungskompetenzen:

> Modellierungskompetenzen umfassen die Teilkompetenzen zur Durchführung der einzelnen Schritte des Modellierungsprozesses, Metakognitive Modellierungskompetenzen, die Kompetenz, Sachprobleme zu strukturieren und zielorientiert bei der Lösung vorzugehen, die Kompetenz, mit Bezug auf die Modellierung zu argumentieren und diese Argumentation zu verschriftlichen sowie Kompetenzen,

> die Möglichkeiten, die die Mathematik zur Lösung von realen Problemen bietet, zu erkennen und sie positiv zu beurteilen«
> (Maaß, 2005, S. 367).

Die so charakterisierten mathematischen Modellierungskompetenzen von Schülerinnen und Schülern stellen zu großen Teilen Elemente dar, die in der Informatik den Prozessbereichen »Begründen und Bewerten«, »Strukturieren und Vernetzen«, »Kommunizieren und Kooperieren« sowie »Darstellen und Interpretieren« zugeordnet werden (vgl. Bildungsstandards Informatik (GI, 2008)). Was bleibt, ist die Abarbeitung eines Plans, nämlich der »Durchführung der einzelnen Schritte des Modellierungsprozesses«. Es fehlt deutlich das »Entwickeln und Aufstellen des Plans«[13], der abgearbeitet wird – dies ist in der Informatik ein zentrales Moment der Modellierung und der ausgewiesenen Modellierungskompetenzen.

Das Schulfach Mathematik ist im Unterschied zum Schulfach Informatik durchgängig in allen Schularten und in allen Schulstufen etabliert. Nun stellt sich die Frage, welche Vorarbeiten aus dem Schulfach Mathematik zur Modellierungskompetenz im Schulfach Informatik beitragen können.

Die bereits oben dokumentierte Sicht der Fachdidaktik Mathematik auf Modellierungsprozesse macht deutlich, dass nicht zu erwarten ist, dass Schülerinnen und Schüler den konstruktiven Prozess der Modellierung im Sinne des »Aufstellen eines Plans« zur Lösung außermathematischer Probleme mit Hilfe der Mathematik bewältigen können – dies scheint eine deutliche Überforderung des Anspruchs an den Mathematikunterricht zu sein.

Im Schulfach Mathematik findet eine Form der Begriffsbildung statt, die – ähnlich der Arbeit der Informatik – klaren logischen und formalen Ansprüchen genügen muss. Der Erwerb der Kompetenzen zur funktionalen Modellierung aus der Mathematik ist für die Informatik bedeutsam, da diese als Voraussetzung für die zielgerichtete Nutzung von Tabellenkalkulationssystemen benötigt wird. Ist der Funktionsbegriff durch die Mathematik vorbereitet, hat die Informatik eine qualifizierte Basis, um die informatische funktionale Modellierung mit den Schülerinnen und Schülern zu entwickeln.

Betrachten wir die von MAASS formulierten mathematischen Modellierungskompetenzen kritisch, wird deutlich, dass ein für die Informatik konstitutives Element, nämlich die eigene Modellierung durch die Schüler, nicht ausgewiesen ist. Im Ergebnis führt dies zu der Situation, dass die Strukturwissenschaft Mathema-

[13] ... erstellen Modelle zu gegebenen Sachverhalten

tik als Schulfach Routinen ausbildet, die darin bestehen, »Pläne abzuarbeiten« nicht aber selbstständig Pläne zu erstellen.

Das Schulfach Informatik kann nur in wenigen Teilbereichen auf Elemente der mathematischen Modellierung aufsetzen. Die aus Sicht der Didaktik der Informatik und der Bildungsstandards Informatik notwendige eigene informatische Modellierung geht wesentlich über die durch den Mathematikunterricht eingebrachten Voraussetzungen hinaus.

Im Gegenzug muss die Frage gestellt werden, ob nicht die Mathematik und ihre Didaktik von einem Pflichtschulfach Informatik erwarten könnte, dass grundlegende informatische Modellierungskompetenzen bei den Schülerinnen und Schüler so ausgebildet sind, dass die Nutzung von Informatiksystemen auf einer fachlich soliden Basis vorgenommen werden können.

Jede Nutzung von Informatiksystemen setzt ein mentales Modell des jeweiligen Systems voraus, das informatischer Grundlagen bedarf, wenn es über die **Bedienung** hinaus zielführend eingesetzt werden soll – so gesehen ist die Sicht der Mathematik und anderer Schulfächer auf Informatiksysteme und ihre Bedienung im Unterrichtseinsatz nicht zu verantworten, ohne dass fachliche Grundlagen der Informatik vorgängig vorhanden sind.

1.6 Beschreibung einer Informatikunterrichtsreihe mit dem Schwerpunkt Unterrichtspraxis

Im Folgenden wird die unterrichtspraktische Durchführung einer Unterrichtsreihe im Schulfach Informatik beschrieben. Sie ist auf den Anfangsunterricht in der Einführungsphase der gymnasialen Oberstufe ausgerichtet. Die Unterrichtsreihe und das zugrunde liegende didaktische Konzept wurde im Zusammenhang mit einer Zweiten Staatsarbeit für das Lehramt an Gymnasien und Gesamtschulen entwickelt. Innerhalb dieser Staatsarbeit werden neben der Beschreibung der Durchführung zudem das entwickelte didaktische Konzept sowie fachdidaktische Begründungen für zentrale konzeptionelle Entscheidungen ausführlich dargestellt. Daher sei an dieser Stelle für eine tiefer gehende Beschäftigung auf diese Zweite Staatsarbeit (Kuhaupt, 2012) verwiesen, die zudem die zentrale Quelle dieses Teils des Beitrages darstellt.

Ausgangspunkt der Reihe: eine Situationsbeschreibung bzw. ein Szenario

Den Ausgangspunkt der Unterrichtsreihe »Cafeteria« bildet eine textuelle Situationsbeschreibung, die einen Ausschnitt aus Lebenswelt der Schülerinnen und Schüler darstellt. Diese Situationsbeschreibung stellt den Problembereich dar, für den die Schülerinnen und Schüler ein objektorientiertes Modell erarbeiten sowie später dieses Modell in einer objektorientierten Programmiersprache implementieren.

Situationsbeschreibung:

> Tim und Lisa hatten in der zweiten großen Pause noch eine Besprechung und kommen deshalb erst gegen Ende der Pause in die Cafeteria »Futterkrippe«. Sie hoffen, dass sie noch etwas zu essen vorfinden und haben Glück, denn auf dem Tresen liegen noch ein Salami- und ein Käsebrötchen für je 1,50 Euro sowie ein Schnitzelbrötchen für 2 Euro. Tim hat 1,60 Euro dabei, Lisa noch 2,30 Euro. Tim nimmt sich das Salamibrötchen und Lisa nimmt sich das Schnitzelbrötchen. Beide gehen zur Verkäuferin Frau Müller, die ihnen die entsprechenden Preise nennt, und bezahlen. Frau Müller nimmt das Geld dankend entgegen.

Phase 1: Modellierung der Objekte

Im Rahmen der objektorientierten Modellierung der Situationsbeschreibung steht in der ersten Modellierungsphase die Identifikation der in der Situationsbeschreibung vorkommenden Objekte im Vordergrund. Den Schülerinnen und Schülern wird als methodische Hilfe dazu das Verfahren nach Abbott an die Hand gegeben, mit dessen Hilfe sie Kandidaten für Objekte identifizieren sowie die Objekte durch Attribute und Methoden genauer beschreiben. Ziel dieses Schrittes ist die Modellierung der in der Realsituation vorkommenden Objekte als informatische Objekte, die in Form von Objektkarten dargestellt werden.

Bereits an dieser frühen Stelle kann es zu unterschiedlichen Modellierungen kommen, die als Lerngelegenheit im Unterricht und Ausgangspunkt für fachliche Diskussionen über verschiedene Modellierungsmöglichkeiten mit ihren Vor- und Nachteilen genutzt werden sollten. Die Objektkarte in Abbildung 1.13 (S. 52) zeigt ein vorangestelltes Ausrufezeichen bzw. ein Fragezeichen. Die Bedeutung dieser

Bezeichnung wird im Folgenden kurz genannt.

Ausrufezeichen kennzeichnet einen *Auftrag* an das Objekt, den das Objekt ausführen soll. In dem Beispiel erhält die Verkäuferin den *Auftrag*, einen Geldbetrag zu nehmen, der dann durch die Veränderung von »geldInKasse« sichtbar wird.

Fragezeichen kennzeichnet eine *Anfrage* an das Objekt, bestimmte Informationen oder ein Objekt zu liefern. In dem Beispiel erhält die Verkäuferin den *Anfrage*, den Preis eines Produktes zu einem Produktnamen zu nennen.

Die Verkäuferin kennt ihre Produkte und den Kassenstand und kann daher die Anfragen beantworten und Aufträge ausführen.

Für die mathematische Beschreibung dieser Aspekte wird auf die Definition 3.5.2 (S. 106) verwiesen.

Phase 2: Modellierung der Beziehungen

Nach der Identifikation der Objekte liegen diese nun isoliert und ohne Beziehungen zueinander als Objektkarten vor. In der nächsten Modellierungsphase werden daher die Beziehungen zwischen den Objekten modelliert, die für jede objektorientierte Modellierung von zentraler Bedeutung sind. Dazu erarbeiten die Schülerinnen und Schüler ein Objektdiagramm, in dem alle Objekte sowie ihre Beziehungen untereinander modelliert und dargestellt werden. Die Schülerinnen und Schüler müssen sich dafür überlegen, welche Objekte miteinander kommunizieren bzw. Nachrichten austauschen und sich daher kennen müssen. Diese Beziehungen werden durch Beziehungsattribute, die mit den jeweiligen Bezugsobjekten verbunden werden, bei den Objektkarten modelliert. Diese Modellierung stellt damit eine erste statische Modellierung des Szenarios dar.

Phase 3: Validierung und Optimierung sowie Ausführung des Modells

Das erarbeitete Objektdiagramm – als statisches Modell – sollte nun von den Schülerinnen und Schülern auf seine Tragfähigkeit überprüft werden. Dazu bietet sich das Objektspiel als handlungsorientierte Methode an. Beim Objektspiel

wird jedes benötigte Objekt von einem Schüler gespielt, der dafür mit der entsprechenden Objektkarte des Objekts ausgestattet wird. Die Objektkarten werden entsprechend ihrer Beziehungen im Objektdiagramm mit Schnüren verbunden. Dazu klebt man eine Schnur mit einem Klebepunkt neben das ausgehende Beziehungsattribut und klebt das andere Ende der Schnur auf die Rückseite des Zielobjekts. Generell befinden sich auf der Rückseite einer Objektkarte, die für die anderen Spieler bzw. Objekte sichtbar ist, die angebotenen Methoden des Objekts. Eine Schülerin (als Objektkartenträgerin) kann nur dann eine andere Schülerin »anfragen« oder ihr einen »Auftrag geben«, wenn es eine Verbindung zwischen den beiden Objektkarten gibt. Die Schülerinnen und Schüler, die kein Objekt spielen, werden als Protokollanten des Objektspiels eingesetzt. Sie protokollieren den Ablauf des Objektspiels, indem sie Methodenaufrufe mit Ausgangs- und Zielobjekt sowie Attributveränderungen der Objekte festhalten. Die Protokolle werden später beim Übergang zur dynamischen Modellierung und für die Erarbeitung der Methodeninhalte benutzt.

Die Schülerinnen und Schüler können nach dieser Vorbereitung das Objektspiel durchführen. Die Lehrperson nimmt einen Stift (token), der als Kennzeichen für das aktive Objekt dient, und ruft bei einem Objekt die erste Methode auf. Sie gibt den Stift dann an das entsprechende Objekt weiter, das nun aktiv wird und seinerseits eine Methode bei einem anderen Objekt aufruft. Beim Objektspiel wird relativ schnell deutlich, wenn Attribute, Methoden, Parameter und Beziehungen fehlen oder ungünstig modelliert werden, da das Objektspiel dann einfach stoppt und es nicht weiter geht. Im Rahmen des Objektspiels können viele Konzepte wie z. B. Parameterübergabe und Methodenrückgaben anschaulich und direkt erfahrbar für die Schülerinnen und Schüler verdeutlicht werden. Das Objektspiel hat einen sehr hohen Lernertrag, ist aber bei erstmaliger Durchführung durchaus sehr anspruchsvoll und fordert auch von Seiten der Lehrperson sowie den Protokollanten eine hohe Konzentration. Es hat sich zudem bewährt, einen Schüler oder eine Schülerin mit einem speziellen Auftrag zur Überarbeitung bzw. Aktualisierung des Objektdiagramms festzulegen. So können Änderungen und Ergänzungen eingepflegt werden und werden nicht vergessen. Am Ende des Objektspiels sollte ein Objektdiagramm vorliegen, das sich durch die Simulation als tragfähig erwiesen hat. Das Objektspiel ermöglicht den Schülerinnen und Schülern so erstmals das erarbeitete Modell auch auszuführen.

Phase 4: ein dynamisches Modell durch Sequenzdiagramme

Das Objektdiagramm und die Protokolle sind ein wichtiger Ausgangspunkt für die dynamische Modellierung des Szenarios durch ein Sequenzdiagramm. Die Schülerinen und Schüler nutzen beide Produkte, um die Interaktion der Objekte durch ein Sequenzdiagramm zu modellieren. Die Kommunikation der Objekte wird so übersichtlich und kompakt dargestellt.

Phase 5: Modellierung der Methoden auf Objektebene

Als nächste Phase erfolgt die Modellierung der einzelnen Methoden. Ziel dieses Schrittes ist die Formalisierung der einzelnen Methoden in Form von Struktogrammen, die später implementiert werden.

Im Rahmen der oben angesprochenen Staatsarbeit wird hier ein didaktisches Konzept verfolgt, das sowohl die Modellierung als auch später die Implementierung der Methode auf Objektebene vorsieht. Durch den Verbleib auf der Objektebene wird einerseits die Komplexität reduziert und andererseits eine Vermischung von Objekt- und Klassenebene vermieden.

Die Vorgehensweise zur Erarbeitung der Methodeninhalte stützt sich auf den ablauforientierten Ansatz für Algorithmenentwurf nach DESEL und VON KLENZE (Desel & Klenze, 2010). Dazu müssen die Schülerinnen und Schüler zwei zentrale Arbeitsschritte durchführen. Zunächst steht die Formalisierung der im Objektspiel beobachteten und dokumentierten (durch Sequenzdiagramm und Protokolle) Methodenabläufe im Vordergrund. Die Schülerinnen und Schüler erstellen für jede Methode eines Objektes eine Methodenablaufübersicht. Die Methodenablaufübersicht verwendet bereits Struktogramme als Darstellungform, beinhaltet allerdings zunächst keine Strukturelemente zur Ablaufsteuerung wie Verzweigungen und Zyklen. Es werden einfache Anweisungen (Methodenaufrufe, Wertzuweisungen etc.) im Struktogramm festgehalten. Die Schülerinnen und Schüler greifen für diesen Arbeitsschritt auf das erarbeitete Sequenzdiagramm und auf die Protokolle zurück. Die Protokolle sind hier von entscheidener Bedeutung, da sie z. B. Attributveränderungen dokumentierten, die nicht aus einem Sequenzdiagramm ersichtlich werden. Daher müssen die Ablaufübersichten aus beiden Arbeitsergebnissen synthetisiert werden.

Da der Ablauf einigen Methoden z. B. von Parameterwerten beim Aufruf oder von Attributwerten des Objekts bedingt wird, gibt es mehrere unterschiedliche

Abläufe einer Methode, die so auch von den Schülerinnen und Schülern durch mehrere Struktogramme formalisiert und festgehalten werden müssen. Gerade die unterschiedlichen Abläufe der gleichen Methoden sind der Ausgangspunkt für den nächsten Abstraktionsschritt bei der Modellierung der Methoden.

Um die verschiedenen exemplarischen Methodenabläufe einer Methode nun auf eine abstraktere Beschreibung und Formalisierung der Methode zu übertragen, betrachten die Schülerinnen und Schüler die unterschiedlichen Abläufe und vergleichen diese miteinander. Beim ablauforientierten Ansatz bezieht sich dies auf den horizontalen Vergleich der unterschiedlichen Abläufe. Dazu vergleichen die Schülerinnen und Schüler die unterschiedlichen Struktogramme Block für Block. Wenn z. B. einem Attribut anhand eines Parameterwertes ein neuer Wert zugewiesen wird, so muss dies nun von speziellen und konkreten Anweisungen wie z. B. »geld ← 3« und »geld ← 2« auf eine abstrakte Anweisung mit Parameter wie z. B. »geld ← pGeld« abstrahiert werden. Auf diese Weise können auch Abhängigkeiten von Parameterwerten oder eigenen Attributwerten durch komplexere Strukturelemente wie z. B. eine Verzweigung abstrahiert werden.

Damit die Erarbeitung der Methodeninhalte durch den ablauforientierten Ansatz gut funktioniert, müssen verschiedene Abläufe einer Methode vorliegen. Sollte dies nicht bereits durch die Rahmenbedingungen des Szenarios gegeben sein, muss die Lehrperson evtl. ein neues Objekt mit anderen Attributwerten einführen, das andere Abläufe einer oder mehrerer Methode zur Folge hat.

Am Ende dieser Phase haben die Schülerinnen und Schüler die Methoden der Objekte abstrakt – aber immer noch auf Objektebene – beschrieben. Diese Methodenbeschreibungen können in Form von Dienstkarten für jedes Objekt kompakt gesammelt werden (siehe Abbildung 1.16 und 1.17, S.55-56 einer Dienstkarte für das Objekt *derTresen*).

Eine Dienstkarte ist eine Übersicht aller Methoden und deren Inhalte für ein bestimmtes Objekt. Diese Dienstkarten können dann z. B. durch ein Objektspiel auf ihre Tragfähigkeit überprüft werden.

Durch die abstrakten Beschreibungen der Methoden muss sich der selbe Verlauf ergeben wie vorher. Probleme bei der abstrakten Modellierung der Methoden tauchen an dieser Stelle wieder auf und können zum Stocken des Objektspiels oder falschen Abläufen führen, die den Schülerinen und Schülern aber direkt beim Objektspiel auffallen und dann verbessert werden können.

Phase 6: Implementierung der modellierten Methoden auf Objektebene

In der nächsten Phasen wird die Implementierung der Methoden vorgezogen und in den Modellierungsprozess eingebunden. Bisher wurde ausschließlich auf Objektebene modelliert, um so die Komplexität zu reduzieren. Dies soll nun auch auf die Implementierung übertragen werden. Dazu implementieren die Schülerinnen und Schüler die erarbeiteten Methodenrümpfe bzw. Methodeninhalte. Der Methodenkopf bleibt zunächst unberührt und wird später nach der Einführung des Klassenbegriffs implementiert. Als provisorischer Methodenkopf kann die Methodensignatur aus dem Objektdiagramm wie z. B. »? gibBroetchen (pBroetchen)« dienen. Die Implementierung soll hier im Prinzip nur eine Codierung der erarbeiteten Modellierung bzgl. der Methodenrümpfe darstellen.

Um die Schülerinnen und Schüler mit der Programmiersprache Java vertraut zu machen und zentrale programmiersprachliche Konzepte zu erarbeiten, wird eine Vorgehensweise aus den Fremdsprachen adaptiert, die u. a. von DIETHELM auf den Erwerb von Programmiersprachen übertragen wurde (vgl. (Diethelm, 2007)). Dazu erhalten die Schülerinnen und Schülern einige von der Lehrkraft implementierte einfache Methoden, die zentrale Konstrukte wie Punktnotation, Wertzuweisung etc. enthalten. Die Schülerinnen und Schüler können nun durch die vergleichende Betrachtung der Struktogrammdarstellung und der konkreten Implementierung einer Methode die Bedeutung der programmiersprachlichen Konstrukte aus dem Kontext selbstständig erarbeiten. In einer Übungsphase können nun weitere Methoden von den Schülerinnen und Schülern mit den erworbenen Programmierkenntnissen umgesetzt werden.

Anschließend kann die Lehrperson auf die gleiche Art und Weise komplexere Programmierkonstrukte wie die Verzweigung einführen. Es schließt sich eine weitere Phase zur Einübung der Konstrukte an, in der die Schülerinnen und Schüler die Verzweigung aus den Struktogrammen in weiteren Methoden durch die Verwendung der jeweiligen programmiersprachlichen Schlüsselworte und ihrer syntaktischen Ausprägung umsetzen[14]. Es ist besonders hervorzuheben, dass die Implementierung zunächst auf Papier erfolgt.

[14] in Java oder Python: »if« und »else«

Phase 7 (optional): Ausführung der Implementierung mit dem didaktischen Werkzeug »JooPu«

Um den Schülerinnen und Schülern die Möglichkeit zu geben, ihr erarbeitetes Modell und dessen Implementierung auszuprobieren sowie zu prüfen, kann ein Werkzeug benutzt werden, das im Rahmen der oben angesprochen Staatsarbeit als ein Prototyp entwickelt wurde. Das Programm »JooPu« (»Java object only Programmierumgebung«, siehe Abbildung 1.18, S. 57) hat die Aufgabe, die Klassenebene zu verstecken und die Eingabe von implementierten Methodenrümpfen für exemplarisch ausgewählte Methoden zu gestatten. Die Lehrperson implementiert dafür als Vorbereitung zunächst das erarbeitete Modell in Java und wählt z. B. fünf Methoden unterschiedlicher Komplexität aus. Für diese Methoden wird die Implementierung bis auf den Klassenkopf offen gelassen. Im Programm können später die Schülerinnen und Schüler ihren Quelltext in vorgegebene Eingabebereiche eintragen und so den Quelltext vervollständigen. Man kann dann mit JooPu den (nun vollständigen) Quelltext an den Java Compiler weiterleiten und später den entstandenen Bytecode interpretieren. Somit können die Schülerinnen und Schüler ihr erarbeitetes Modell und Implementierung bereits zu diesem Zeitpunkt ohne Kenntnisse des Klassenbegriffs ausführen. Der Sinn von JooPu besteht nicht darin, eine vollständige Entwicklungsumgebung zu sein. JooPu soll den Schülerinnen und Schülern einfach eine frühe Möglichkeit geben, mit Informatiksystemen kleinere Methoden zu implementieren und die Ausführbarkeit von Modell und Implementierung zu zeigen.

Phase 8: Abstraktion des Modells auf Klassenebene

Nach diesem Implementierungseinschub muss ein letzter Abstraktionsschritt bei der Modellierung vollzogen werden: Die Abstraktion von der Objektebene auf die Klassenebene. Dazu muss die Lehrperson den Klassenbegriff einführen. Die Schülerinnen und Schüler sollten gerade bei der Modellierung der Methoden erfahren haben, dass einige Objekt ähnlich sind und sich ähnlich verhalten. Man kann dies nun aufgreifen und den Klassenbegriff als Bauplan für Objekte oder zur Klassifizierung von Objekten einführen. Um eine möglichst trennscharfe Unterscheidung von Objekt und Klasse zu erreichen, bietet sich meiner Meinung nach zunächst die Bauplan-Variante bei der Einführung des Klassenbegriffs an. Der Klassifizierungs- und Gruppierungsaspekt kann später noch ergänzt werden.

Als nächster Arbeitsschritt steht nun die Abstraktion vom Objektdiagramm zum Klassendiagramm an. Die Schülerinnen und Schüler entwickeln das Klassendiagramm und haben somit ein vollständiges objektorientiertes Modell des Problembereichs erarbeitet.

Phase 9: Implementierung der Klassenkarten und Fertigstellung

In der letzten Phase werden die Klassen implementiert. Da die Methodenrümpfe bereits implementiert worden, müssen lediglich die Klassenkarten bzw. Klassenrümpfe (Methodenköpfe und Attribute) und natürlich der Konstruktor implementiert werden. An dieser Stelle muss die Lehrperson das Variablenkonzept von Java einführen und vor allem den Konstruktor intensiv behandeln. Zur selbstständigen Erarbeitung der Implementierung von Attributen und Methodenköpfen empfiehlt sich wieder ein dekonstruktives Vorgehen, bei dem die Lehrperson eine fertig implementierte einfache Klasse in Java vorgibt und von den Schülerinnen und Schüler durch den Vergleich mit der Klassenkarte selbstständig erschließen lässt. Der Konstruktor sollte separat und intensiv thematisiert werden, da er ein sehr komplexes Konstrukt darstellt, was sich nicht direkt aus dem erarbeiteten Modell ableiten lässt. Am Ende dieser Phase steht eine komplette Implementierung, die nun mit einem beliebigen Editor oder einer didaktischen Entwicklungsumgebung wie »BlueJ« von den Schülerinnen und Schülern mit Informatiksystemen implementiert und ausgeführt werden kann.

1.7 Zusammenfassende Thesen – Ausblick

1. Informatische Modellierung stellt – neben der mathematischen Modellierung – eine Strukturicht auf Gegenstandsbereiche dar. Die Aspektverschiebungen der informatischen Modellierung gegenüber der mathematischen Modellierung sind augenfällig:

   (a) Informatische Modellierung bearbeitet reale Probleme und Anwendungsfälle und führt zu einem Modellierungsergebnis der Schülerinnen und Schüler, mit dem das Problem gelöst wird. Auf dem Weg der Bearbeitung sind die Fachkonzepte der Bezugswissenschaft Informatik notwendiger und integrativer Bestandteil dieses problemlösenden Ansatzes.

(b) Dynamisches Verhalten soll abgebildet werden, dies lässt sich gut mit Hilfe der objektorientierten Modellierung dokumentieren und unterrichtlich umsetzen.

(c) Informatische Modellierung führt zu ablauffähigen Ergebnissen.

2. Soll die informatische Modellierung für das Schulfach Mathematik (und andere Schulfächer) ihre Qualität erweisen, so muss sie im Rahmen eines eigenständigen und von anderen Fächern unabhängigen Pflichtbereichs Informatik – bereits im Kindergarten und in der Primarstufe – fachlich sorgfältig vorbereitet und angelegt werden. Der Einsatz von Informatiksystemen in diesen Bereichen ist dabei kein notwendiger Bestandteil einer verantwortlichen Ausgestaltung der informatischen Bildung.

Die Einordnung der informatischen Modellierung macht deutlich, dass nur ein eigenständiges Schulfach Informatik, das von informatisch und fachdidaktisch qualifizierten Lehrkräften unterrichtet wird, die formulierten Anforderungen lerngruppenbezogen und mit fachlich korrekten Gestaltungselementen umsetzen kann.

Situationsbeschreibung

Auf Annes Mobiltelefon befinden sich jede Menge Dateien. Es handelt sich um Fotos, Videos, Musikstücke und SMS/MMS-Nachrichten. Der Speicher auf dem Telefon ist inzwischen knapp geworden. Sie entschließt sich, einige Dateien zu löschen, andere auf die Speicherkarte zu verschieben, um Platz für neue Dateien zu schaffen.
Damit sie zukünftig etwas mehr Ordnung hat, legt Anne zunächst das Verzeichnis
`e:/Archiv/2012/`
an. Anne verschiebt die Dateien
`c:/Data/Images/Screenshots/scrn43.png`
und
`c:/Data/Videos/Baum1.mp4`
in das neu angelegte Verzeichnis.
Die Bilddatei
`c:/Data/Images/Pictures/Foto-234.jpg`
und den Klingelton
`c:/Data/Sounds/Digital/Blackbird.aac`
löscht sie.
Die Präsentation
`c:/Data/Documents/INFOS.pdf`
sendet sie per Bluetooth an das Mobiltelefon ihrer Freundin.

Arbeitsauftrag:

Erstellen Sie aus dem obigen Text mit Hilfe der Methode von Abbott ein Objektdiagramm.

Abbildung 1.7: »Aufräumen« im Dateisystem – Arbeitsblatt

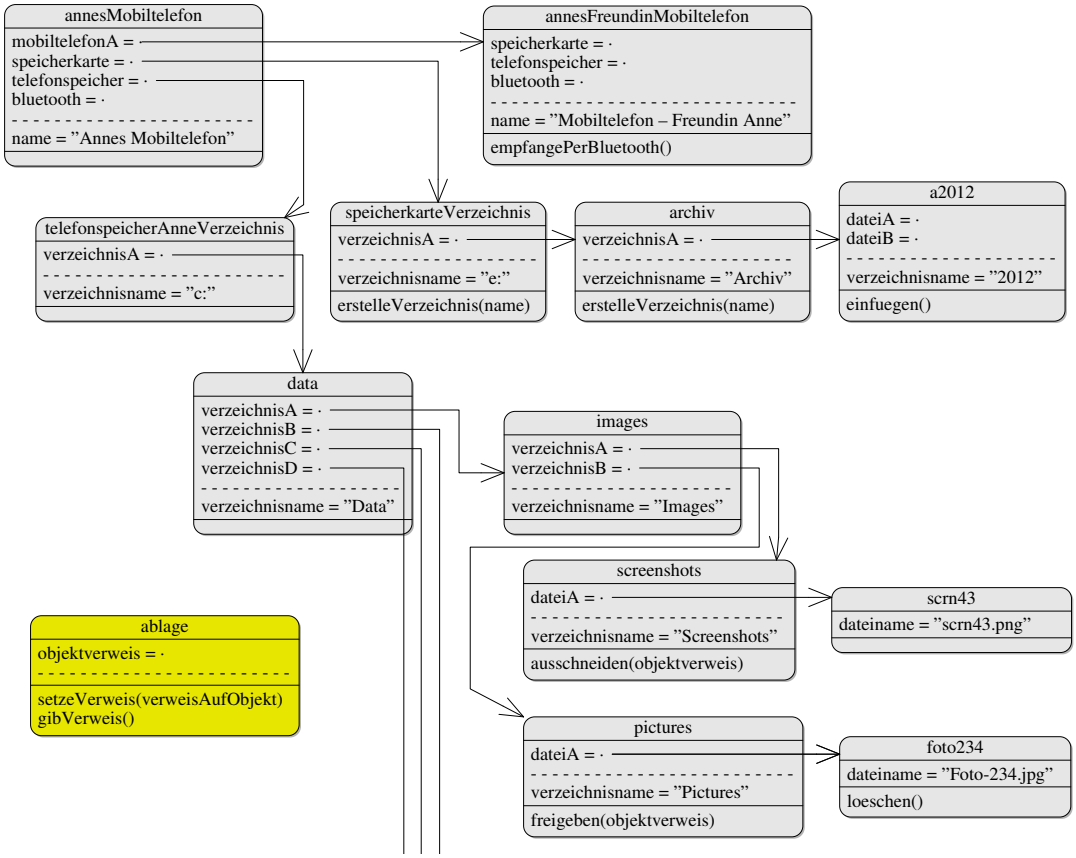


Abbildung 1.8: Objektdiagrammausschnitt – vor dem »Aufräumen«

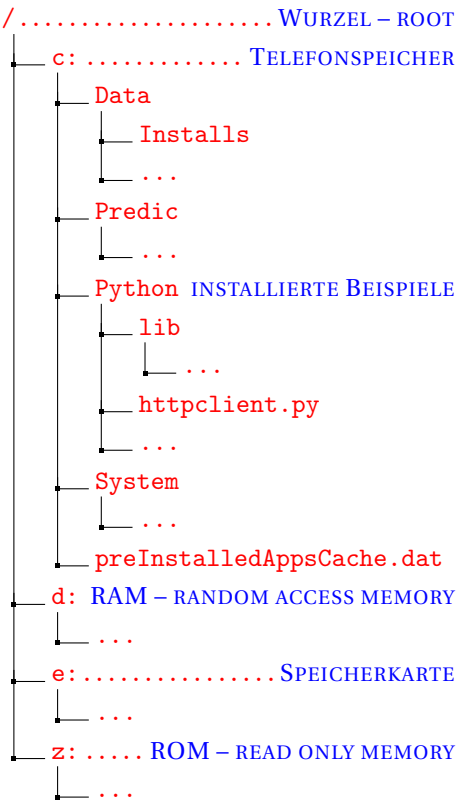
Betriebssystem

Jedes Informatiksystem verfügt über ein Betriebssystem…
Die erste Aufgabe eines Betriebssystems besteht nach dem Einschalten darin, …weitere Funktionen des Betriebssystems: …
Moderne Benutzungskonzepte nehmen der Benutzerin häufig die Entscheidung darüber ab, was mit einer Datei geschieht, wenn sie aktiviert wird. Dies ist aber, wenn programmiert werden soll, nicht gewollt.

Verzeichnisse (engl. Directory)

…grundlegende Strukturen zeigen zu können, die auf Informatiksystemen verfügbar sind. Dazu zählt beispielsweise, dass Sie die Struktur des Verzeichnisbaums inspizieren und verändern können: Sie können Dateien kopieren, löschen, verschieben, versenden, …

Wenn Sie sich das Dateisystem auf dem Mobiltelefon ansehen, werden Sie die abgebildete Verzeichnisstruktur wiedererkennen (die …deuten an, dass hier nur ein Teil der Dateien/Verzeichnisse angegeben sind).



Mobil Programmieren

…

Abbildung 1.9: Verzeichnisstruktur – Ausschnitt Informationsblatt

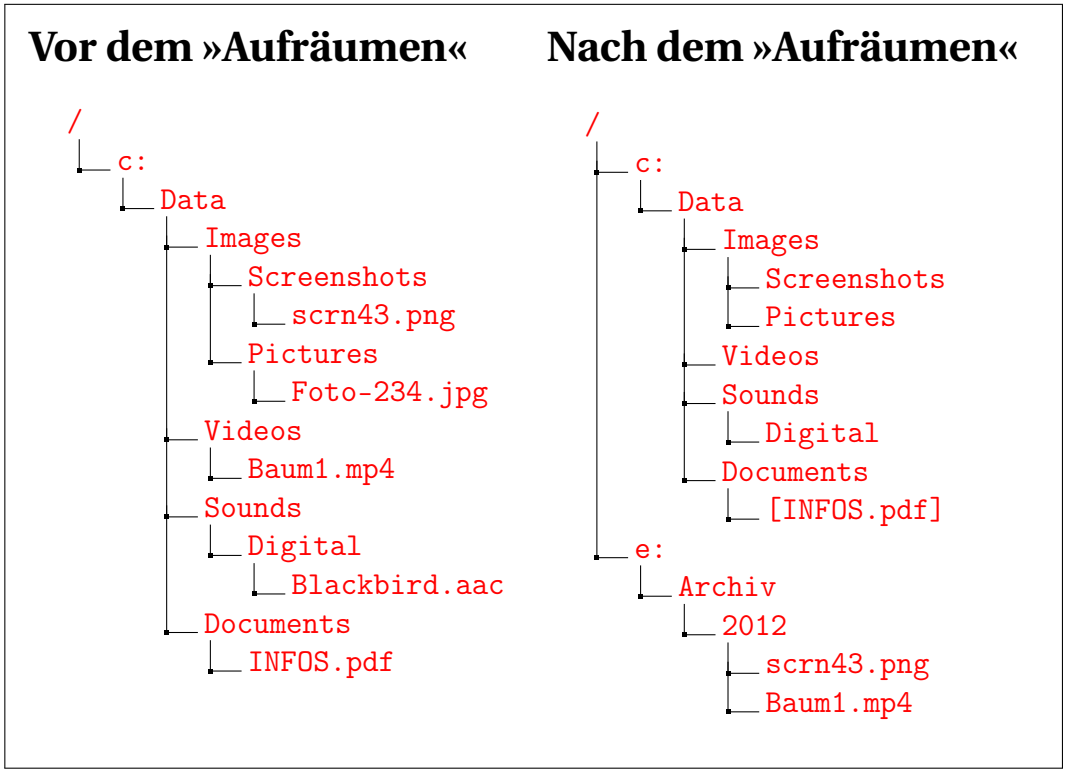


Abbildung 1.10: Verzeichnisstruktur: vorher – nachher

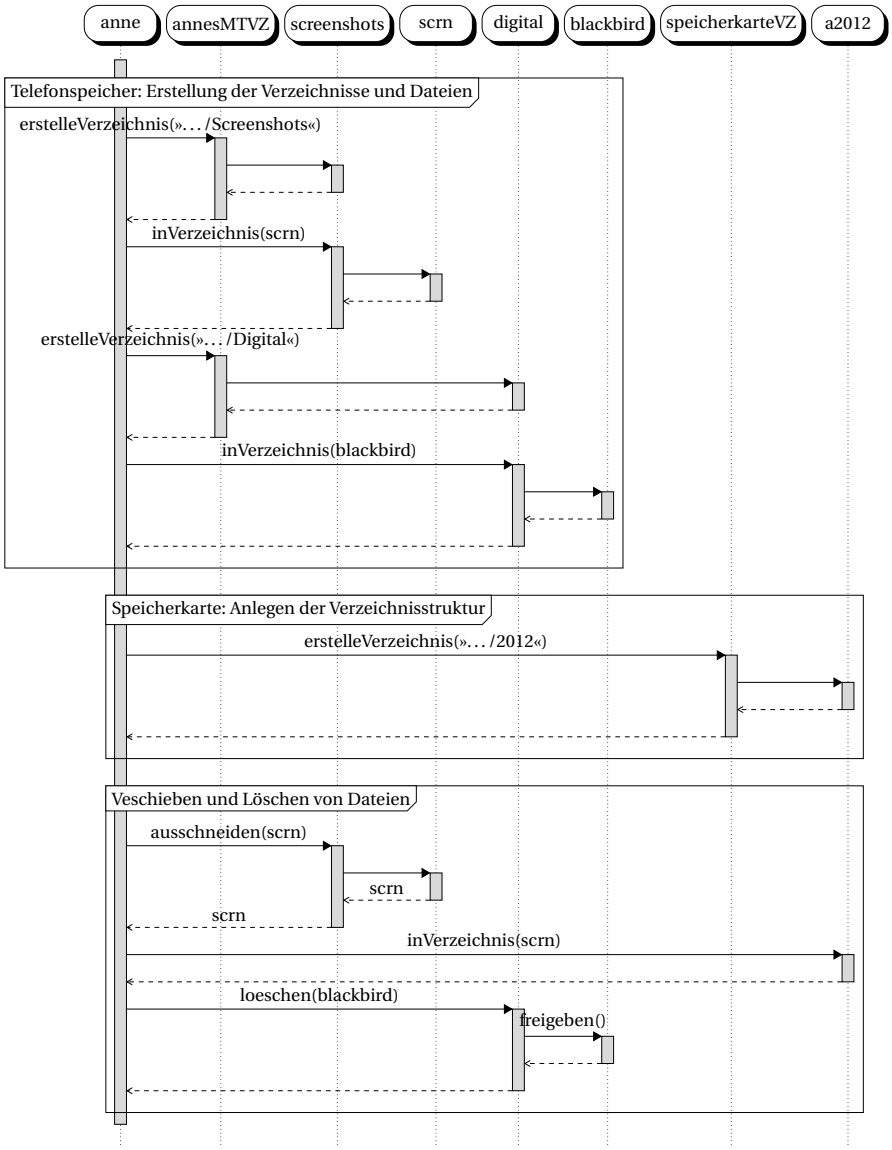


Abbildung 1.11: Sequenzdiagramm

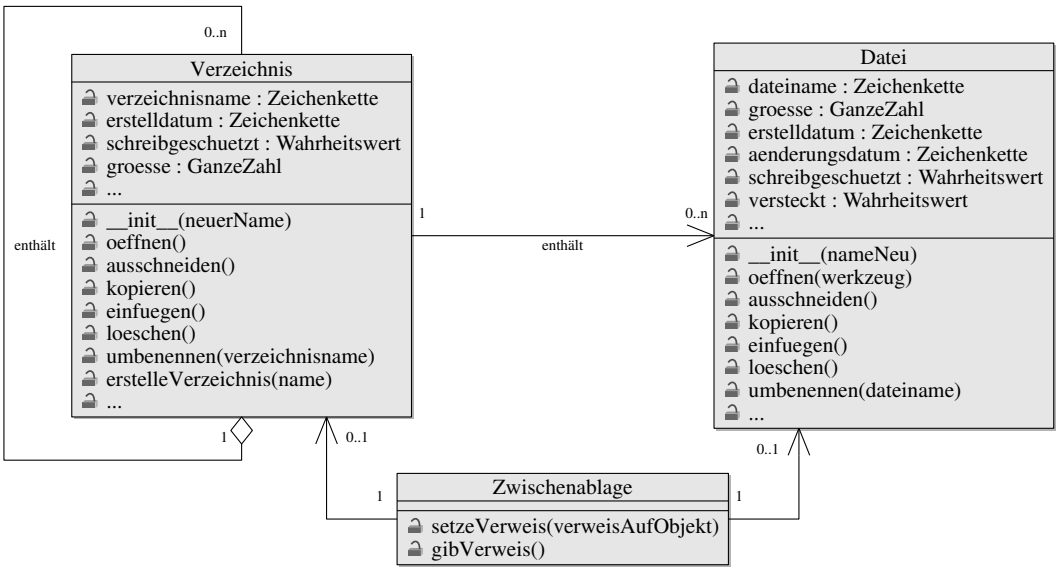


Abbildung 1.12: Klassendiagramm – Verzeichnis und Datei

dieVerkaeuferin
produkte name=”Frau Müller” geldInKasse=0.0
? broetchenGeben(pProduktname) ! geldAnnehmen(pGeldbetrag) ? preisNennen(pProduktname)

Abbildung 1.13: Beispiel einer Objektkarte

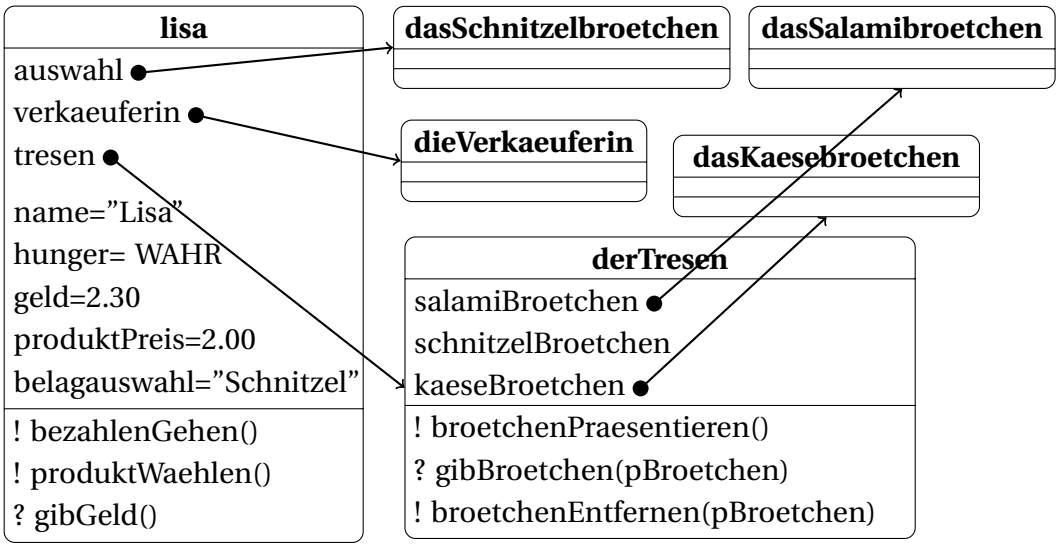
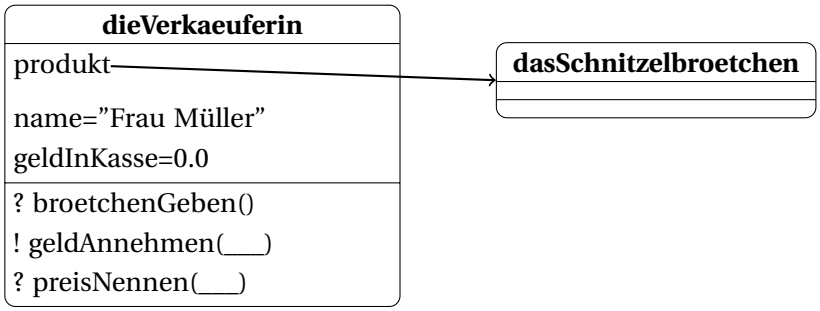


Abbildung 1.14: Ausschnitt aus einem Objektdiagramm

Ablauf 1: dieVerkaeuferin.geldAnnehmen(2.0)
Vor dem Methodenaufruf:



Methodenablauf: **! geldAnnehmen(2.0)**

erhöhe den aktuellen Wert des Attributs geldInKasse um 2.0

Nach dem Methodenaufruf:

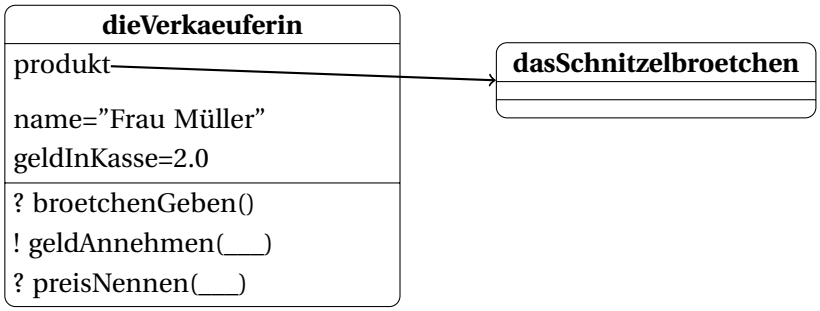


Abbildung 1.15: Eine Ablaufbeschreibung für eine Methode

! broetchenPraesentieren()

erzeuge das Objekt *dasSchnitzelbroetchen* und binde es an das Beziehungsattribut *schnitzelBroetchen*
erzeuge das Objekt *dasSalamibroetchen* und binde es an das Beziehungsattribut *salamiBroetchen*
erzeuge das Objekt *dasKaesebroetchen* und binde es an das Beziehungsattribut *kaeseBroetchen*

! broetchenEntfernen(pBroetchen)

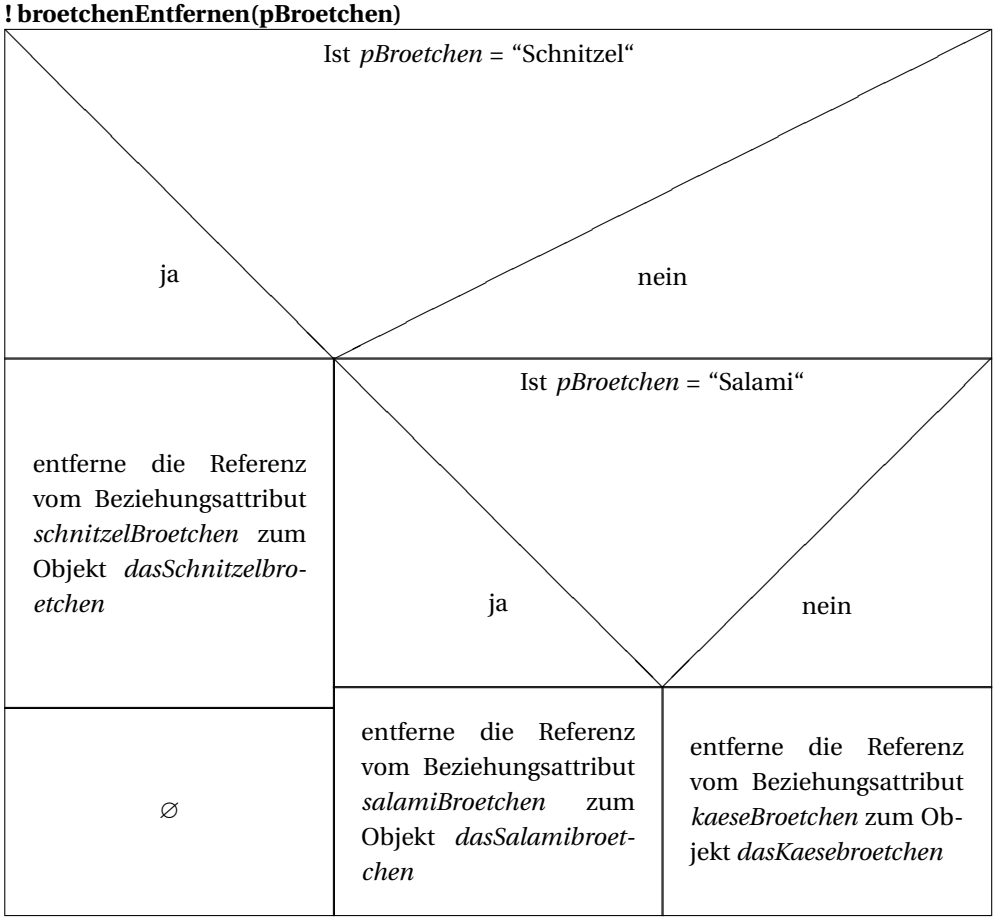


Abbildung 1.16: Eine Dienstkarte für *derTresen* - Teil 1

? gibBroetchen(pBroetchen)

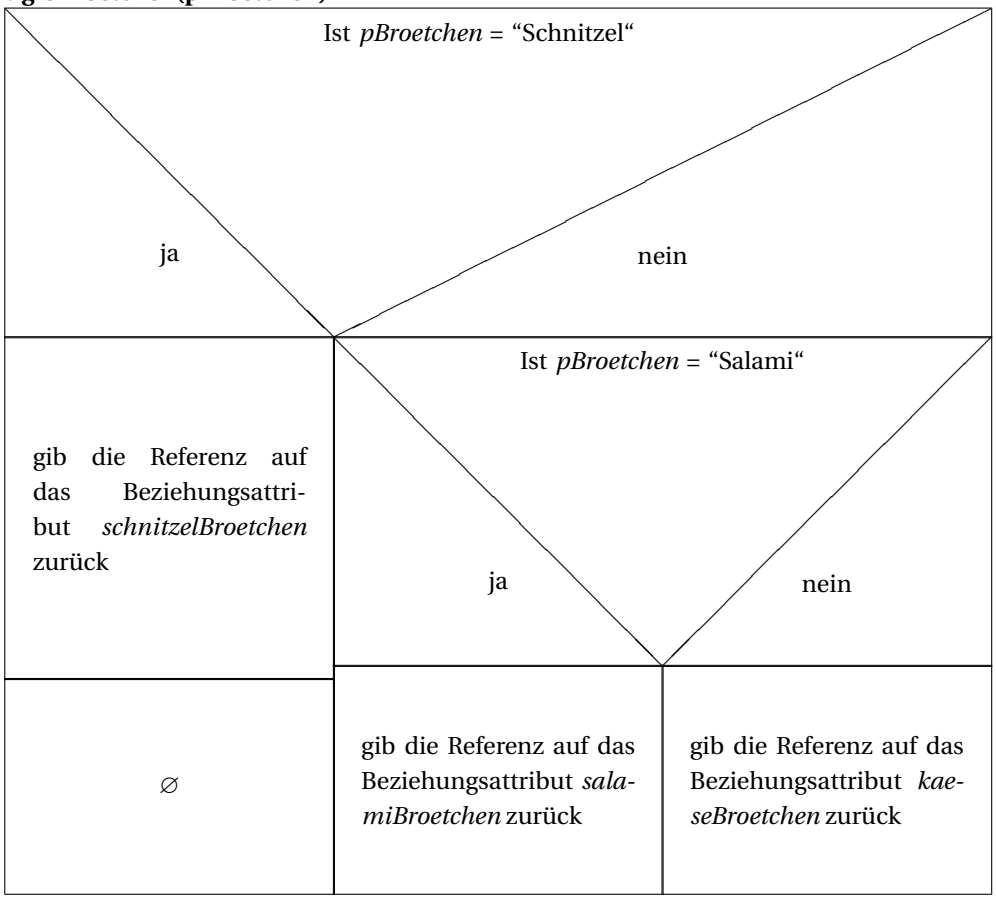


Abbildung 1.17: Eine Dienstkarte für *derTresen* Teil 2

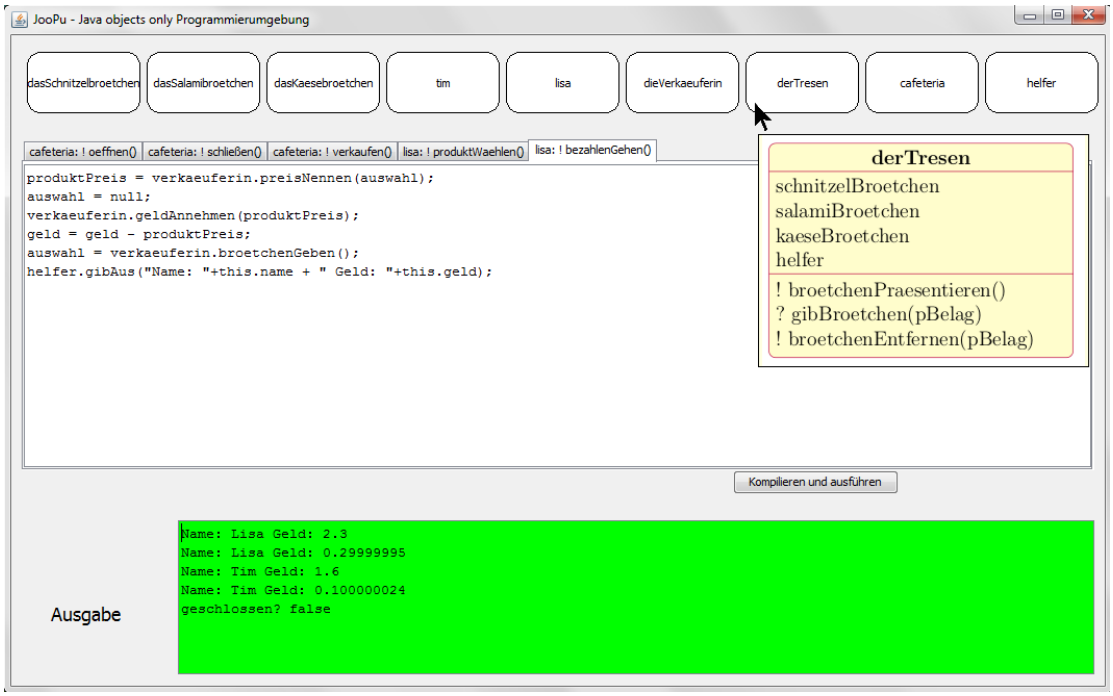


Abbildung 1.18: Screenshot von JooPu

Kapitel 2

Objektorientierte Modellierung in der 3D-Computergrafik

In der dreidimensionalen Computergrafik werden sichtbare Objekte durch ihre „Geometrie“ (z. B. Koordinaten und Abmessungen geometrischer Körper) sowie durch Material- und Oberflächeneigenschaften wie Farbe, Glanz, Transparenz und Brechungseigenschaften beschrieben. Zu den Methoden, die auf Objekte angewendet werden können, zählen geometrische Transformationen.

Die wichtigsten Beleuchtungsmodelle, welche den in der Computergrafik verwendeten Oberflächenattributen zu Grunde liegen, ergeben sich durch die mathematische Modellierung der Lichtausbreitung in der Realität – mathematische Modellierung und objektorientierte Modellierung aus Sicht der Informatik gehen in der 3D-Computergrafik also „Hand in Hand“.

Praktisch nachvollziehbar wird objektorientierte Modellierung in der 3D-Computergrafik durch die Erstellung von 3D-Grafiken mithilfe einer Grafiksoftware, die mit Szenenbeschreibungen arbeitet. Eine sehr gut geeignete Software dieser Art ist POV-Ray.[1] Nach einem einführenden Beispiel einer in POV-Ray erstellten Grafik werden durch die Betrachtung zusätzlicher Objekte, Attribute und Methoden sowie zu Grunde liegender mathematischer Modelle zunehmend größere grafische Gestaltungsmöglichkeiten erschlossen.

Prof. Dr. Andreas Filler
Humboldt-Universität zu Berlin
filler@math.hu-berlin.de

[1] POV-Ray kann (kostenlos) unter `http://www.povray.org` heruntergeladen werden.

2.1 Ein einführendes Beispiel

Der 3D-Computergrafik liegt die Vorstellung einer virtuellen Kamera zugrunde, durch die ein Ausschnitt des dreidimensionalen Raumes „fotografiert“ wird. Das in Abb. 2.1 (S. 59) dargestellte Bild wurde mit der Software POV-Ray durch die Eingabe folgender Szenenbeschreibung generiert:[2]

```
background {color rgb <1,1,1>}
camera{ location <0,8,-20>   angle 12   look_at <0,-0.5,0> }
light_source{ <-10,30,-10>   color rgb <1,1,1>}
sphere{ <-0.3,-0.5,1.5>, 0.5  texture{ pigment{ color rgb <0,0,1>} }
       }
cone{ <1.2,-1,0.5>, 0.5, <1.2,0.8,0.5>, 0
      texture{ pigment{ color rgb <1,0,0>} }
     }
torus{ 0.9, 0.15
       texture{ pigment {color rgb <0,1,0>} }
       translate <-0.6,-0.85,-0.6>
      }
cylinder{ <0,-1.1,0>, <0,-1,0>, 2
          texture{ pigment{ color rgb <0.5,0.5,0.5> }
                   finish{ reflection 0.2 }          }
         }
```

Alle Koordinatenangaben beziehen sich auf ein linkshändiges Koordinatensystem; die y–Achse wird als vertikale Achse abgebildet (siehe Abb. 2.2). In der Computergrafik ist diese Wahl des Koordinatensystems recht gebräuchlich.

[2] Auf der Internetseite `http://www.afiller.de/oom` stehen die POV-Ray-Quelldateien für alle in diesem Kapitel abgebildeten Grafiken zur Verfügung.

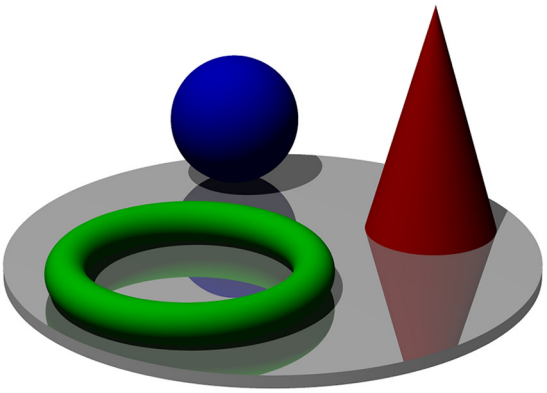


Abbildung 2.1: Beispiel einer dreidimensional generierten Computergrafik

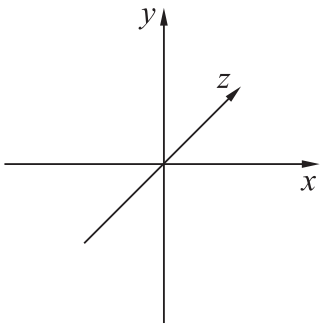


Abbildung 2.2: Linkshändiges Koordinatensystem

Das dargestellte Beispiel enthält alle Elemente, die für die Erzeugung einer dreidimensionalen computergrafischen Darstellung notwendig sind:

- `background` legt die Farbe des Hintergrundes fest (`color rgb <1,1,1>` beschreibt die Farbe Weiß im RGB-Farbmodell (siehe S. 81).
- Hinter `camera` werden die Koordinaten einer Kamera (`location`), ihr Öffnungswinkel (`angle`) sowie die Koordinaten eines Punktes, auf den diese Kamera gerichtet ist (`look_at`), angegeben. Die Kamera „fotografiert“ die Szene, letztendlich erfolgt dadurch eine Zentralprojektion.
- Die Anweisung `light_source` beschreibt eine Lichtquelle durch ihre Position im Raum und die Farbe des abgegebenen Lichtes. Da sich die Bildberechnung in der fotorealistischen Computergrafik an Gesetze der Optik anlehnt, müssen Objekte beleuchtet werden, um sichtbar zu sein.
- Die mit `sphere`, `cone`, `torus` und `cylinder` beginnenden Zeilen erzeugen geometrische Körper mit den dahinter angegebenen Abmessungen und positionieren sie (mit Ausnahme des Torus) durch Koordinaten bestimmender Punkte im Raum. Eine Kugel wird z. B. durch die Koordinaten ihres Mittelpunktes und den Radius beschrieben; zu den bestimmenden Punkten und Größen von Zylindern, Kegeln und Tori siehe die Abbildungen 2.3 auf S. 62 und 2.4 auf S. 63.
- Sichtbaren Objekten sind Oberflächeneigenschaften (*Texturen*) zuzuweisen. In POV-Ray geschieht dies durch `texture`. Im einfachsten Falle enthalten Texturdefinitionen Festlegungen von Farben. Zusätzlich lassen sich weitere Oberflächenmerkmale wie z. B. Rauheit und Reflexionsvermögen festlegen. So weist die Festlegung `reflection 0.2` dem Zylinder eine spiegelnde Oberfläche zu. Die Beschreibung optischer Eigenschaften von Oberflächen und die dem zugrunde liegenden mathematischen Modelle sind Gegenstände des Abschnitts 2.3.2.
- Nicht alle Objekte können bereits bei ihrer Erzeugung an den beabsichtigten Stellen positioniert werden. So erzeugt POV-Ray einen Torus generell durch Rotation eines Kreises um die y-Achse und mit dem Zentrum im Koordinatenursprung. Durch die Anweisung `translate` wird der Torus an die vorgesehene Stelle verschoben. Oft sind auch Drehungen, Skalierungen und andere Transformationen auf Objekte anzuwenden.

Wie anhand des Beispiels deutlich wird, sind für die Erstellung von 3D-Computergrafiken die folgenden grundsätzlichen Aspekte maßgeblich:

- *Geometrische Modellierung*: Repräsentation, Transformation und ggf. Animation[3] von Körpern oder Flächen des Raumes durch Koordinaten;
- *Festlegung von Oberflächeneigenschaften* der Objekte (Texturen);
- *Beleuchtung der Szene* (Definition von Lichtquellen);
- *Projektion der Szene in eine Ebene*, von der ein Ausschnitt auf dem Bildschirm dargestellt wird – dazu sind Kamerakoordinaten festzulegen.

2.2 Modellierung geometrischer Objekte

2.2.1 Attribute und Methoden geometrischer Objekte

Wir blicken zurück auf das auf S. 59 beschriebene Beispiel. Die Gemeinsamkeit aller Klassen sichtbarer Objekte kann durch das Schema einer abstrakten Klasse „Geometrisches Objekt“ ausgedrückt werden, siehe Abb. 2.3 auf S. 62. Geometrische Objekte lassen sich durch die Koordinaten spezieller Punkte und charakterisierende Abmessungen oder durch Gleichungen beschreiben. Die spezifischen Attribute hängen von den jeweiligen Körpern bzw. Flächen ab, weshalb als ein Attribut hier der sehr allgemeine Block „Geometrie“ auftritt, der für konkrete Unterklassen (wie Kugeln, Zylinder, Quader oder auch durch Gleichungen beschriebene Flächen usw.) spezifisch zu konkretisieren ist (siehe Abschnitt 2.2.2).

Neben der Form bzw. „Geometrie“ zählt die Oberflächenerscheinung eines sichtbaren Objekts zu seinen wichtigsten Eigenschaften in der Computergrafik; daher ist „Textur“ in der Abbildung als zweiter Block von Attributen angegeben. Dazu ist anzumerken, dass die Einordnung von Texturmerkmalen als *Attribute* sichtbarer Objekte aus Beobachtersicht erfolgt, der z. B. Farbe, Reflexionsvermögen oder Rauheit als Eigenschaften von Oberflächen wahrnimmt. Vom Standpunkt der Bilderzeugung in der 3D-Computergrafik lassen sich Texturmerkmale auch als *Methoden* ansehen, denn sie beschreiben die Interaktion von Objektoberflächen mit Lichtstrahlen (siehe Abschnitt 2.3.2). Schließlich ließen sich Texturen – aus Sicht der Informatik, nicht jedoch der Geometrie – als eigenständige Objekte betrachten und Texturklassen als Unterklassen (z. B. Metalle, Holz, Plastik)

[3] Unter Animation wird die zeitabhängige Veränderung von Szenen verstanden, wodurch Bildsequenzen bzw. Videos entstehen. Zeitabhängige Veränderungen können durch die Darstellung von Koordinaten, Größen und nicht geometrischen Werten (z. B. Farben) als Funktionen eines Zeitparameters erreicht werden, siehe z. B. (Filler, 2007a), S. 151ff. und (Filler, 2007b).

Abstrakte Klasse „Geometrisches Objekt“

Geometrisches Objekt	⟵ **Klasse**
Geometrie Textur	⟵ Attribute
Transformation	⟵ Methoden

Unterklassen (spezielle Arten geometrischer Objekte)

Kugel (`sphere`)	**Kegelstumpf** (`cone`)	**Zylinder** (`cylinder`)	**Torus** (`torus`)
Mittelpunkt Radius Textur	Mittelpunkt Grundfl. Radius Grundfläche Mittelpunkt Deckfl. Radius Deckfläche Textur	Mittelpunkt Grundfl. Mittelpunkt Deckfl. Radius Textur	großer Radius kleiner Radius Textur
Transformation	Transformation	Transformation	Transformation

Instanzen (in dem Beispiel auf S. 59 konkret vorhandene Objekte)

Kugel 1	**Kegel 1**	**Tischplatte**	**Torus 1**
`sphere`	`cone`	`cylinder`	`torus`

`<-0.3,-0.5,1.5>`
`0.5`
 	
`texture { ...`
siehe S. 59 }
	`<1.2,-1,0.5>`
	`0.5`
	`<1.2,0.8,0.5>`
	`0`
	`texture { ...`
	siehe S. 59 }
		`<0,-1.1,0>`
		`<0,-1,0>`
		`2`
		
		`texture { ...`
		siehe S. 59 }
			`0.9`
			`0.15`
			
			`texture { ...`
			siehe S. 59 }

			`translate` `<-0.6,-0.85,-0.6>`

Abbildung 2.3: Klassen und Instanzen in dem Beispiel auf S. 59

einer abstrakten Klasse „Texturen“ betrachten, obwohl Texturen nur nach Zuordnung zu Körperoberflächen „sichtbar“ sind.[4] Die Aufzählung von Attributen

[4]Die Stellung von Texturmerkmalen wie `reflection` als Attribute geometrischer Objekte, als Methoden der Interaktion von Objekten mit Lichtstrahlen oder als „selbstständige“ Objekte verdeutlicht, dass die Aufstellung von Objektmodellen keineswegs eindeutig ist, sondern von der Sichtweise des Betrachters abhängt. Dies entspricht einem Grundprinzip der allgemeinen Modelltheorie, wonach Modelle „nicht nur Modelle von etwas, sondern auch Modelle für jemanden; ... zu einem bestimmten Zweck“ sind (STACHOWIAK, (Stachowiak, 1973), S. 131f). Während hier zunächst die „Benutzersicht“, speziell die Erstellung von Grafiken mithilfe der Szenenbeschreibungssprache von POV-Ray, im Vordergrund steht (aus der heraus Texturmerkmale Attribute von Objekten sind), wird in dem Abschnitt 2.3.2 die Interaktion von Oberflächen mit Lichtstrahlen betrachtet.

in Abb. 2.3 beschränkt sich auf die unverzichtbaren Attribute sichtbarer Objekte. Darüber ist es in 3D-Grafiksoftware wie POV-Ray möglich, bei Bedarf eine Reihe weiterer Attribute zuzuweisen.[5]

Als *Methoden*, die auf Objekte angewendet werden können, sind in dem Schema in Abb. 2.3 *geometrische Transformationen* angeführt; dazu gehören Verschiebungen, Drehungen und Skalierungen. In dem auf S. 59 vorgestellten Beispiel wurde lediglich auf den Torus eine Transformation angewendet, da Tori nur in spezieller Lage erzeugt werden können – während sich z. B. Kugeln, Zylinder und Kegelstümpfe in der POV-Ray-Szenenbeschreibungssprache durch ihre Koordinatenbeschreibung beliebig positionieren lassen, so dass hierbei zunächst keine Transformationen erforderlich waren. In anderen Szenen oder mit komplexeren Objekten ist es allerdings oft sinnvoll, Transformationen zu nutzen, weshalb darauf in dem Abschnitt 2.2.3 noch eingegangen wird.

2.2.2 Koordinatenbeschreibungen geometrischer Körper

In dem zurückliegenden Abschnitt wurde deutlich, dass die Attribute einfacher Grundkörper davon abhängen, wie sich diese durch Koordinaten und Abmessungen beschreiben lassen. Die Attribute von Kugeln, Zylindern, Kegelstümpfen und Tori finden sich in den Klassenschemata auf S. 62 (siehe auch Abb. 2.4), die jeweils entsprechende POV-Ray-Syntax enthält das Beispiel auf S. 59.[6]

[5] Zu diesen optionalen Attributen zählt z. B. `no_shadow`, das die Berechnung von Schatten (die durch fotorealistische Grafiksoftware standardmäßig erzeugt werden) für diejenigen Objekte verhindert, denen dieses Attribut zugewiesen wird.

[6] Der in Abb. 2.1 dargestellte Kegel wurde als spezieller Kegelstumpf (mit dem Radius 0 der Deckfläche) beschrieben. Ebenso hätte das `cone`-Objekt genutzt werden können, um den Zylinder zu generieren (wobei dann die beiden Radien gleich sein müssten) – da allerdings für Zylinder in POV-Ray eine spezielle Objektklasse zur Verfügung steht, wurde diese genutzt.

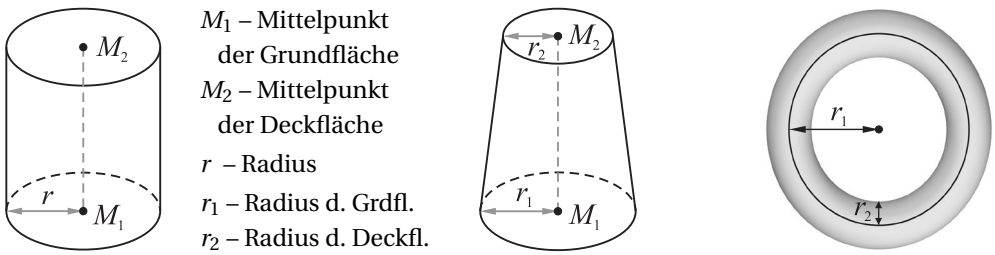
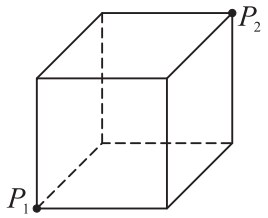


Abbildung 2.4:
Bestimmende Punkte und Größen von Zylindern, Kegelstümpfen und Tori



Quader (box)
Eckpunkt P_1 Eckpunkt P_2 Textur
Transformation

Abbildung 2.5: Skizze und Klassenschema für Quader

Quader lassen sich mithilfe von POV-Ray (ähnlich wie Tori) nur in spezieller Lage erzeugen, ihre Kanten sind stets zu den Koordinatenachsen parallel. Dadurch genügt zur eindeutigen Beschreibung eines derartigen Quaders die Angabe der Koordinaten zweier Eckpunkte, die Endpunkte einer Raumdiagonalen sind, siehe Abb.2.5. Dem entspricht die folgende POV-Ray-Syntax:

```
box{ <x1,y1,z1>, <x2,y2,z2>  texture{...} }
```

Da Quader nur in spezieller Lage erzeugt werden können, müssen für ihre Positionierung innerhalb von Szenen häufig Transformationen, vor allem Drehungen, angewendet werden (siehe Abschnitt 2.2.3).

Etwas allgemeinere Klassen geometrischer Objekte benötigen außer Koordinaten und Abmessungen weitere Attribute. So ist es bei der Objektklasse *Prisma* erforderlich, zunächst die Anzahl der Eckpunkte der Grundfläche (zuzüglich 1) anzugeben, bevor die Koordinaten der einzelnen Eckpunkte angegeben werden können. Das in Abb. 2.6 dargestellte Prisma wurde folgendermaßen erzeugt:

```
prism{ 0, // Höhe der Grundfläche
       3, // Höhe der Deckfläche
       6, // Zahl der Eckpunkte plus 1
       <3,5>, <-3,5>, <-5,0>, <-3,-5>, <3, -5>, <5,0>
       texture{ pigment{ color rgb<0.8,0.8,0.8> } }    }
```

Prisma (prism)
y_1 – Höhe (y-Koordinate) der Grundfläche y_2 – Höhe (y-Koordinate) der Deckfläche $n+1$ – Anzahl der Eckpunkte + 1 (x_1,z_1) – Koord. des Eckpunktes P_1 in der x-z-Ebene ⋮ (x_n,z_n) – Koord. des Eckpunktes P_n in der x-z-Ebene *quadratic_spline* / *cubic_spline* / *bezier_spline* Textur
Transformation

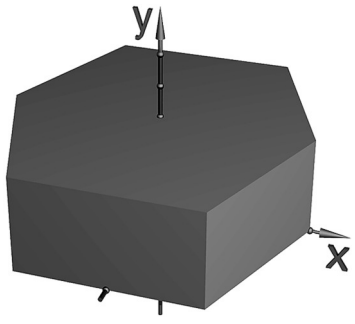


Abbildung 2.6: Skizze und Klassenschema für Prismen
Die kursiv gesetzten Attribute sind optional und können daher weggelassen werden.

Das Klassenschema in der Abbildung 2.6 enthält einige kursiv gesetzte Attribute, die optional sind.[7] So lassen sich anstelle üblicher Prismen durch die Attribute `quadratic_spline`, `cubic_spline` oder `bezier_spline` auch Körper erzeugen, deren Grund- und Deckflächen von „weichen“ Kurven (quadratischen oder kubischen Spline- oder Bézierkurven[8]) begrenzt sind. So wurde der in der Abb. 2.7 dargestellte Körper als „Prisma“[9] mit einer durch eine kubische Splinekurve begrenzten Grundfläche modelliert:

```
prism{ cubic_spline 0, 0.3, 17,
       <-2.5,0> <-2,0>     <-1.8,-0.5> <-1,-1>   <0,-1>  <1,-0.8>
       <2,-0.5> <2.5,-0.2> <3,0>       <2.5,0.2> <2,0.5> <1,0.8>
       <0,1>    <-1,1>     <-1.8,0.5>  <-2,0>    <-2.5,0>
       texture {...}
     }
```

Der so modellierte Körper wurde (in mehreren Instanzen) für Blätter von Blumen verwendet (siehe Abb. 2.10, S. 68), dazu sind allerdings Transformationen erforderlich, da Prismen in POV-Ray nur in spezieller Lage erzeugt werden können (Grund- und Deckfläche liegen in zur x-z-Ebene parallelen Ebenen).

Es ist im Rahmen dieses Kapitels nicht möglich, auf die Vielfalt weiterer Möglichkeiten der Modellierung geometrischer Objekte einzugehen, wozu u. a. Rotationskörper, durch Gleichungen oder Parameterdarstellungen beschriebene Flächen sowie Freiformflächen (Bézier- und Splineflächen) gehören. Ein Überblick hierzu findet sich in (Filler, 2007a) (Abschnitt 2.3), umfangreichere Darstellungen zu dieser Thematik enthalten u. a. (Brüderlin & Meier, 2001), (Foley, Van Dam, Feiner, Hughes & Phillips, 1994), (Goldman, 2009) und (Shirley, 2005).

[7] Auch diese Aufzählung ist nicht vollständig, es lassen sich für Prismen in POV-Ray verschiedene zusätzliche Attribute setzen (siehe `http://www.povray.org/documentation`). So bewirkt `open`, dass ein offenes Prisma (ohne Grund- und Deckfläche) erzeugt wird. Wird das Attribut `conic_sweep` gesetzt, so entsteht statt eines Prismas eine Pyramide.

[8] Es handelt sich hierbei um verschiedene, durch einzelne Punkte festgelegte „Freiformkurven“, siehe hierzu z. B. (Filler, 2007a), S. 98ff. oder (ausführlicher) (Brüderlin & Meier, 2001), S. 209ff. sowie (Goldman, 2009), S. 379ff.

[9] Aus mathematischer Sicht handelt es sich nicht um ein Prisma, sondern um einen Zylinder über einer kubischen Splinekurve.

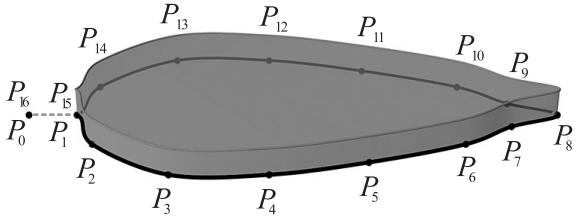


Abbildung 2.7: „Prisma“ mit einer durch eine kubische Splinekurve begrenzten Grundfläche

2.2.3 Geometrische Transformationen als Methoden

Die Klassenschemata in den Abbildungen 2.3, 2.5 und 2.6 zeigen, dass geometrische Transformationen Methoden für alle sichtbaren Objekte in der Computergrafik sind. Bislang wurde nur auf den Torus in dem Beispiel auf S. 59 eine Transformation (Verschiebung) angewendet, da sich Tori (in POV-Ray) durch die verfügbaren Attribute nur in spezieller Lage generieren lassen. Da für viele Objektklassen (wie z. B. für die in dem vorangegangenen Abschnitt betrachteten Quader und Prismen) Objekte nur in spezieller Lage erzeugt werden können, sind geometrische Transformationen für die Erzeugung komplexerer Szenen unabdingbar. Elementare geometrische Transformationen sind[10]

- *Verschiebungen* mit Translationsvektoren $\begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$; in POV-Ray wird dafür die folgende Syntax verwendet: `translate` <t_x, t_y, t_z>;
- *Drehungen* mit den Drehwinkeln φ_x, φ_y, φ_z um die Koordinatenachsen: `rotate` <φ_x,φ_y,φ_z>;
- *Skalierungen* mit den Skalierungsfaktoren k_x, k_y und k_z in den Richtungen der Koordinatenachsen: `scale` <k_x, k_y, k_z>.
  Sind alle drei Skalierungsfaktoren gleich ($k_x = k_y = k_z =: k$), so handelt es sich hierbei um zentrische Streckungen mit dem Streckfaktor k, wobei das Streckzentrum stets im Koordinatenursprung liegt. Hierfür lässt sich eine verkürzte Schreibweise nutzen: `scale k`.

Als Beispiel für die Nacheinanderausführung einer zentrischen Streckung, einer Drehung und einer anschließenden Verschiebung zeigt Abb. 2.8 die Wirkung dieser Transformationen anhand eines Torus.

[10] Auf den ersten Blick könnte man hier die Spiegelungen vermissen. Diese lassen sich aber als Skalierungen mit dem Skalierungsfakor -1 entlang einer Koordinatenachse auffassen. So beschreibt beispielsweise `scale <-1,1,1>` eine Spiegelung an der y-z-Ebene.

```
torus{ 1, 0.2
       scale 1.5
       rotate <60,30,0>
       translate
         <-0.5,0.5,0.5>
     }
TEST
```

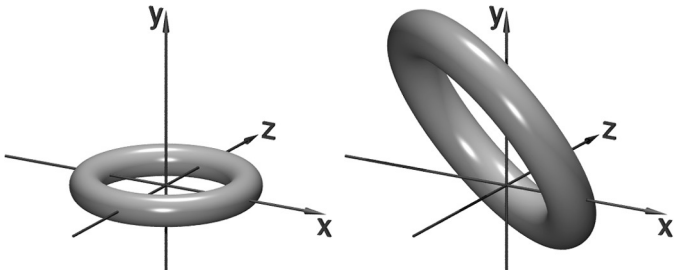


Abbildung 2.8: Transformation eines Torus

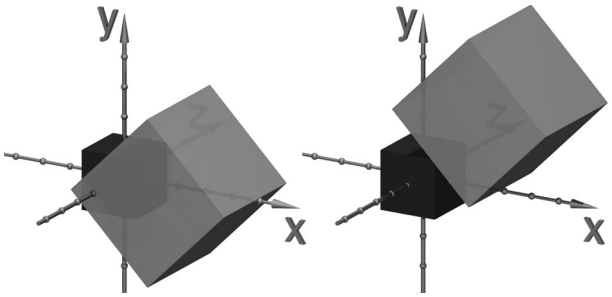


Abbildung 2.9: Beispiel für die Nichtkommutativität der Nacheinanderausführung geometrischer Transformationen

Die Nacheinanderausführung geometrischer Transformationen ist i. Allg. *nicht kommutativ*; die Vertauschung der Reihenfolge von Transformationen führt zu anderen Positionen von Bildobjekten. Die Abbildung 2.9 verdeutlicht dies anhand eines Würfels, auf den folgende Transformationen angewendet werden:

links: `scale 2  rotate <0,0,45>  translate <2,0,0>`
rechts: `translate <2,0,0>  scale 2  rotate <0,0,45>`

Im linken Teil von Abb. 2.9 wird der Würfel zunächst vergrößert (mit dem Streckfaktor 2 und dem Streckzentrum im Koordinatenursprung), danach um 45° um die z-Achse gedreht und anschließend um 2 Einheiten in x-Richtung verschoben. Rechts hingegen erfolgt zunächst die Verschiebung, sodass die danach erfolgende zentrische Streckung aufgrund der Lage des Streckzentrums im Ursprung zu einer weiteren Entfernung des Würfels vom Koordinatenursprung führt. Durch die anschließende Drehung (mit der z-Achse als Drehachse) wird der Würfel aus der x-z-Ebene „heraus gedreht“.[11]

Mitunter sollen mehrere gleichartige Objekte in einer Szene vorhanden sein und evtl. unterschiedliche Ausrichtungen und Größen haben. In diesem Falle ist es sinnvoll, einmalig eine Objektklasse zu erstellen und dann entsprechende Ob-

[11] Man stellt hierbei fest, dass Drehungen und Skalierungen von Objekten leichter zu „planen“ sind, wenn sich z. B. der Mittelpunkt des Objekts bzw. ein anderer charakteristischer Punkt im Koordinatenursprung befindet und Symmetrieachsen geometrischer Objekte (falls diese existieren) mit Koordinatenachsen übereinstimmen. Bei Nutzung von POV-Ray ist es daher sinnvoll, Objekte zunächst in einer derartigen Lage zu modellieren, ggf. zu drehen und zu skalieren und sie anschließend an die „gewünschte“ Stelle innerhalb einer Szene zu verschieben – dieses Vorgehen wurde u. a. bei der Verwendung des in Abb. 2.7 generierten Blattes für die in Abb. 2.10 dargestellten Blumen praktiziert.

Mitunter werden in 3D-Grafiksoftware auch „individuelle“ Koordinatensysteme für einzelne Objekte (so genannte Objekt- bzw. lokale Koordinatensysteme, siehe (Foley et al., 1994), S. 269ff.) genutzt, bezüglich derer Drehungen und Skalierungen erfolgen. Vor der für die Bildberechnung erforderlichen „Zusammensetzung“ einer Szene müssen die lokalen Koordinaten mittels Koordinatentransformation in Szenen- bzw. „Weltkoordinaten“ überführt werden.

Abbildung 2.10: „Blumenstrauß“

Diese Grafik wurde von einer Studentin an der Pädagogischen Hochschule Heidelberg im Rahmen einer Hausarbeit angefertigt. Neben „Prismen“ (mit durch kubische Splinekurven begrenzten Grundflächen, siehe Abb. 2.7 auf S. 65) verwendete sie zur Modellierung hauptsächlich Rotationskörper, siehe hierzu (Filler, 2007a), S. 104f.

jekte als Instanzen zu erzeugen sowie zu verschieben, zu drehen und ggf. zu skalieren. Diese Arbeitsweise kam bei der Erstellung der zahlreichen Blätter in der Abbildung 2.10 zum Einsatz. So wurde für die beiden Blätter der linken Blume zunächst ein Modell (als „Prisma“ mit einer durch eine kubische Splinekurve begrenzten Grundfläche, siehe S. 65) erstellt und verkleinert:

```
#declare Blatt = prism{ – siehe S. 65 –  scale 0.15 }
```

Durch diese Deklaration wurde gewissermaßen eine Klasse „Blatt“ (bei der es sich um eine Unterklasse der Klasse „Prisma“ handelt) erstellt. Anschließend wurden damit zwei Blätter generiert und unterschiedlich gedreht sowie schließlich so verschoben, dass sie sich am Stiel der Blume zu befinden scheinen:

```
object{ Blatt  rotate <-50,0,70>   translate <-0.7,4,0> }
object{ Blatt  rotate <40,180,-60> translate <-1.1,5,0> }
```

Neben der elementargeometrischen Beschreibung von Transformationen durch Anweisungen wie `translate`, `rotate`, `scale` ist in POV-Ray eine analytische Beschreibung durch Transformationsmatrizen möglich, wodurch sich beliebige affine Abbildungen (einschließlich Scherungen) realisieren lassen. Intern arbeitet Grafiksoftware in jedem Falle mit der Matrizendarstellung, um Koordinaten von Punkten transformierter Objekte zu berechnen, auch wenn diese durch Benutzer in der einfacheren Form `rotate` usw. angegeben werden.[12]

[12] Da die analytische Beschreibung geometrischer Transformationen den Rahmen des vorliegenden Kapitels überschreiten würde (und da sich die meisten für die Erstellung von Grafiken notwendigen Transformationen in der hier verwendeten einfachen Weise beschreiben lassen) sei hierzu lediglich auf einen Überblick in (Filler, 2007a) (Abschnitt 2.4) sowie auf umfangreichere Darstellungen in (Brüderlin & Meier, 2001), S. 41ff., (Foley et al., 1994), S. 194ff., (Goldman, 2009), S. 163ff. und (Shirley, 2005), S. 135ff. verwiesen.

2.2.4 Weitere Methoden zur Modellierung von Formen: Boolesche Operationen

Durch die Anwendung Boolescher Operationen (Mengenoperationen: Vereinigung, Differenz und Durchschnitt) auf geometrische Körper lassen sich auf recht einfache und intuitive Weise interessante Formen modellieren. Dieses Verfahren der Modellierung wird oft als *Constructive Solid Geometry* bezeichnet, weshalb wir die entsprechende Klasse mit CSG bezeichnen. Sie hat keine eigenen geometrischen Attribute sondern nur „optische“ Attribute (wie Textur).

CSG (Constructive Solid Geometry)
Textur
Vereinigung – `union` Differenz – `difference` Durchschnitt – `intersection` Transformation

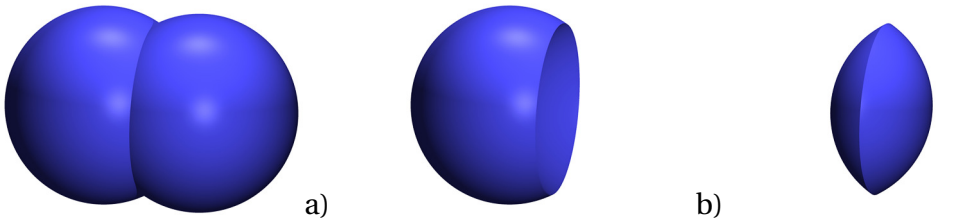
Abbildung 2.11: Die Klasse CSG

Die Methoden Vereinigung, Differenz und Durchschnitt verarbeiten jeweils zwei geometrische Objekte. Die Syntax dieser Methoden in POV-Ray wird durch die folgenden einfachen Beispiele anhand zweier Kugeln verdeutlicht.

a) `union{ sphere{<-1,0,0>, 2} sphere{<1,0,0>, 2} }`

b) `difference{ sphere{<-1,0,0>, 2} sphere{<1,0,0>, 2} }`

c) `intersection{ sphere{<-1,0,0>, 2} sphere{<1,0,0>, 2} }`

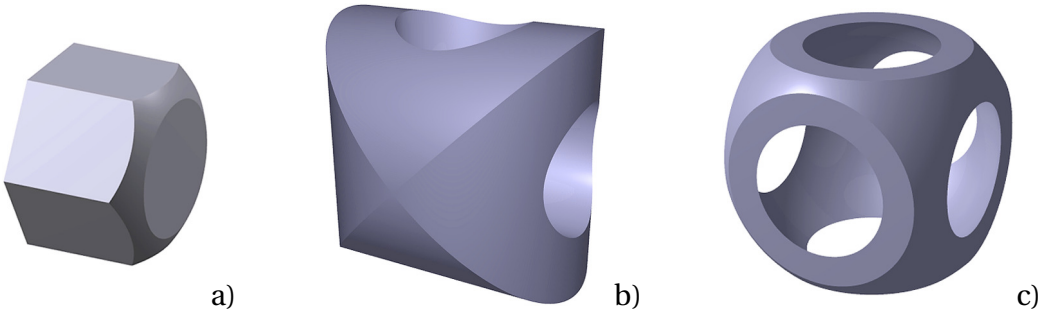


a) b) c)

Abbildung 2.12: Vereinigung, Differenz und Durchschnitt zweier Kugeln

Als ein weiteres Beispiel für den Durchschnitt zweier Objekte, welches ansatzweise aufzeigt, dass CSG-Objekte im Bereich des CAD (Computer Aided Design) von Bedeutung sind, zeigt Abb. 2.13 a) einen als Durchschnitt eines Prismas und einer Kugel modellierten Kopf einer Sechskantschraube.

Die Methoden Vereinigung, Differenz und Durchschnitt lassen sich auch auf CSG-Objekte anwenden (was der „Klassenlogik“ enstpricht, da die CSG-Klasse eine Unterklasse der Klasse „Geometrisches Objekt“ ist). Als Beispiel zeigt Abb. 2.13 b)



a) b) c)

Abbildung 2.13: Boolesche Operationen (CSG)

die Differenz aus dem Durchschnitt zweier Zylinder und der Vereinigung zweier Zylinder (kleineren Durchmessers):

```
difference{ intersection{ cylinder{ <-1,0,0>, <1,0,0>, 1}
                          cylinder{ <0,-1,0>, <0,1,0>, 1}  }
            union{ cylinder{ <-1,0,0>, <1,0,0>, 0.5}
                   cylinder{ <0,-1,0>, <0,1,0>, 0.5}  }
          }
```

Als weiteres Beispiel ist in Abb. 2.13 c) das Ergebnis der Subtraktion einer Kugel vom Durchschnitt einer Kugel und eines Würfels dargestellt:

```
difference{ intersection{ sphere{ <0,0,0>, 4}
                          box{ <-3,-3,-3>, <3,3,3>}  }
            sphere{ <0,0,0>, 3.5}
          }
```

2.2.5 Ein komplexeres Beispiel: Klasse „schneekugel“

Die in den vorherigen Abschnitten beschriebenen Konzepte wurden von einem Schüler genutzt, der für eine komplexere Szene eine Klasse `schneekugel` erstellte und mehrere Instanzen dieser Klasse in die Szene einfügte.[13] Er modellierte zunächst als Vereinigung von 23 Einzelobjekten einen Schneemann und schloss diesen in eine teiltransparente Kugel ein (siehe Abb. 2.14):[14]

[13] Die Arbeit entstand im Zusammenhang mit einer Unterrichtsreihe im Mathematikunterricht einer 12. Klasse, in der POV-Ray für Visualisierungen in der Analytischen Geometrie genutzt wurde, siehe (Filler, 2007a), S. 314ff. Als Einstiegsbeispiel für Koordinatenbeschreibungen einfacher geometrischer Objekte modellierten die Schüler Schneemänner. Dies bildete die Grundlage für die Modellierung einer komplexeren Szene durch einen der beteiligten Schüler (der einen fakultativen Informatikkurs besuchte und entsprechendes Vorwissen einbrachte).

[14] Als Texturen wurden hauptsächlich vordefinierte Texturen (wie `mattweiss`) genutzt. Für die teiltransparente Kugel erstellte der Schüler selbst eine Textur (`schneekugeltextur`) mithilfe von Parametern, auf die in dem Abschnitt 2.3.2 eingegangen wird.

```
#declare schneekugel =
union {   // ------- Kugel mit Ständer: ---------
     cone{ <0,-15,0>, 10, <0,-11.5,0>, 7 texture{mattweiss} }
     sphere{ <0, 3.5, 0> 15 texture{schneekugeltextur} scale <0,0,0>}
     cylinder{ <0,-11.5,0>, <0,-9.8,0>, 7 texture{mattweiss} }
          // ------- Eigentlicher Schneemann: ---
     sphere{ <0,-6,0> 6 texture{mattweiss} }     // Körper (unten)
     sphere{ <0, 2,0> 4 texture{mattweiss} }     // Körper (Mitte)
     sphere{ <0, 7,0> 3 texture{mattweiss} }     // Körper (oben)
     cone{ <0,7,-3>, 0.5, <0,7,-7>, 0.0 texture{rot_matt} } // Nase
     cylinder{ <0,2,0>,  <-8,2.5,0>,  0.08 }       // rechter Arm
                 ... Weitere 18 Objekte ...
      }
```

Anschließend wurden vier Instanzen der so eingeführten Klasse „schneekugel“ in einer komplexeren Szene verwendet:

```
object{ schneekugel rotate <0,-40,0> scale <0.4,0.4,0.4>    //Drei
                     translate <-30.7,27.5,1>            }  //kleine
object{ schneekugel rotate <0,-90,0> scale <0.4,0.4,0.4>    //Schnee-
                     translate <-31.1,27.5,22>           }  //männer
object{ schneekugel rotate <0, 40,0> scale <0.4,0.4,0.4>    //auf dem
                     translate <-31,27.5,-26>            }  //Regal
object { schneekugel rotate <0,-50,0>}           // Großer Schneemann
```

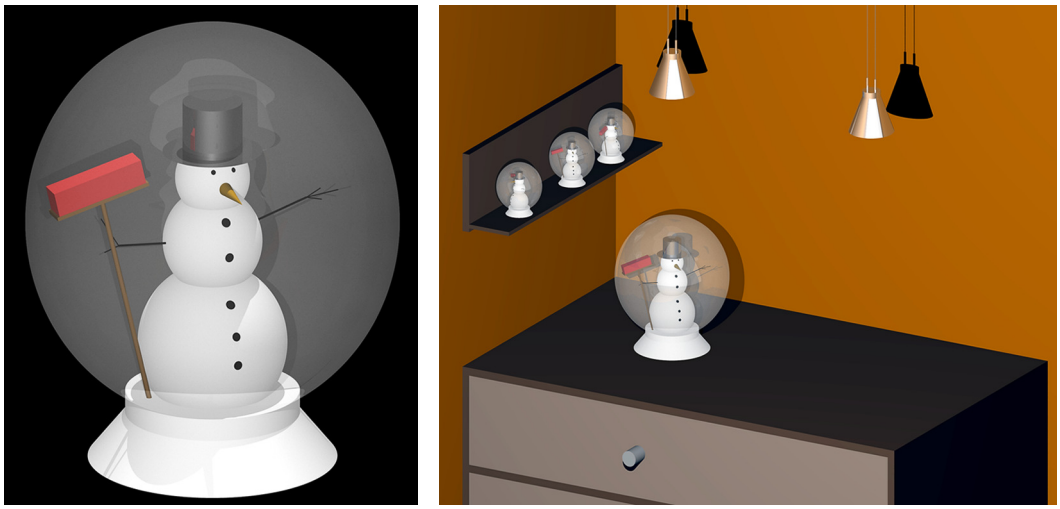


Abbildung 2.14: Instanzen einer Klasse „`schneekugel`“ (Arbeit eines Schülers)

Die zugehörige POV-Ray-Datei steht unter `http://www.afiller.de/oom` zur Verfügung.

2.3 Fotorealistische Bilderzeugung

Nachdem eine dreidimensionale Szene (bestehend aus sichtbaren Objekten sowie einer Kamera und mindestens einer Lichtquelle) beschrieben wurde, kann die Berechnung eines Bildes erfolgen – die Szene wird gewissermaßen „fotografiert“. Für die dabei erfolgenden Bildberechnungen (Rendering) wurden unterschiedliche Verfahren entwickelt; in der fotorealistischen Computergrafik hat das Raytracing-Verfahren (Strahlverfolgung) besondere Bedeutung erlangt.

2.3.1 Das Raytracing-Verfahren

In der Realität nimmt das Auge Lichtstrahlen wahr, die von Lichtquellen ausgehen und das Auge meist auf indirektem Wege erreichen. Auf ihrem Wege werden die Strahlen oft an verschiedenen Objekten reflektiert. Grafiksoftware kann nicht die Verläufe aller von den Lichtquellen ausgehenden Strahlen berechnen – das würde extrem lange dauern und die meisten dieser Strahlen erreichen das Auge nie, haben also keinen Einfluss auf das Bild. Nach einem Gesetz der Optik sind jedoch Verläufe von Lichtstrahlen umkehrbar. Diese Tatsache wird bei dem Raytracing-Verfahren genutzt. Vom „Auge des Betrachters“ bzw. von den festgelegten Kamerakoordinaten aus werden die Wege von Lichtstrahlen – oftmals über mehrfache Reflexionen und Lichtbrechungen hinweg – verfolgt (siehe Abb. 2.15). Raytracing-Programme berechnen dadurch automatisch Schatten und Spiegelungen.

Das Raytracing-Verfahren beinhaltet eine Zentralprojektion, da ausgehend vom Projektionszentrum Z (Kamera) nach Objekten „gesucht“ wird, die auf den betrachteten Ausschnitt der Projektionsebene abgebildet werden. Für alle Punkte des zu berechnenden Bildes sind Helligkeits- bzw. Farbwerte[15] zu berechnen und dazu die Verläufe von Sehstrahlen zu verfolgen. Die Anzahl der zu verfolgenden Strahlen ist von der gewünschten Auflösung des Bildes abhängig, bei $800 \cdot 600$ Bildpunkten (Pixeln) müssen z. B. 480 000 Strahlen verfolgt werden. Jeder dieser Strahlen kann durch Spiegelungen und Transparenz mehrfach verzweigen, wie bereits das einfache Beispiel in der Abbildung 2.15 zeigt. Für jeden zu verfolgenden Lichtstrahl sind folgende Schritte durchzuführen:[16]

[15] Beliebige Farben werden nach dem in der Computergrafik verwendeten rgb-Farbmodell durch Helligkeitswerte dreier Grundfarben (rot, grün und blau) gebildet, siehe S. 81.

[16] Detaillierter werden die hier skizzierten Schritte einschließlich der Ermittlung der Projektionsebene sowie der Schnittpunktbestimmungen von Lichtstrahlen mit Objektoberflächen u. a. in (Haenselmann, 1996) beschrieben; zur programmiertechnischen Umsetzung siehe z. B. (Shirley & Morley, 2000).

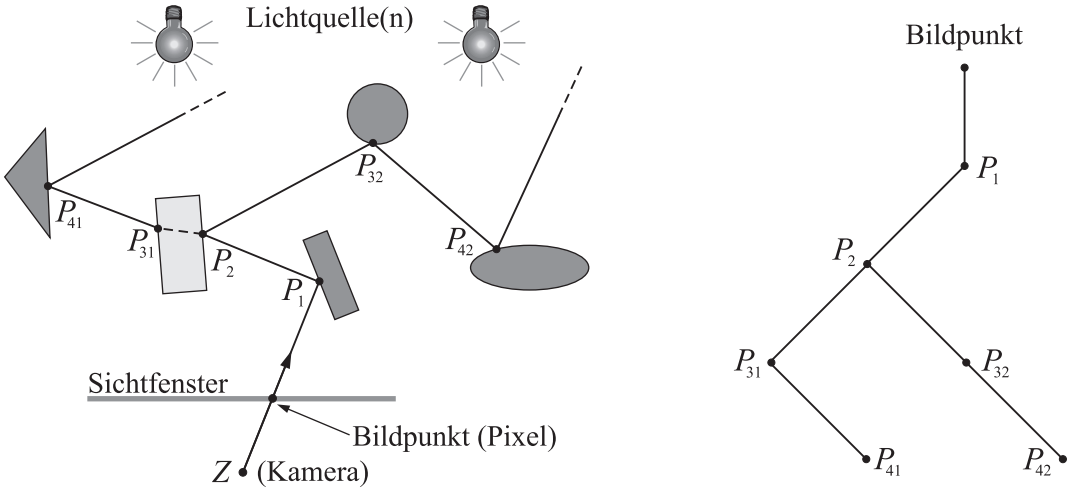


Abbildung 2.15: Strahlverfolgung bei einer einfachen Szene, Sichtstrahlenbaum

1. Bestimmung des vom Sichtfenster aus nächstgelegenen Schnittpunktes des Strahls mit einer Objektoberfläche. Ein derartiger Schnittpunkt wird Knoten des Sichtstrahlenbaums genannt. In dem Knoten muss eine Flächennormale berechnet werden.

2. Für den Lichtstrahl werden in dem unter 1. ermittelten Schnittpunkt unter Berücksichtigung der geometrischen und optischen Eigenschaften des Objekts die Richtungen sowie farbabhängigen Intensitäten reflektierter und bei transparenten Objekten transmittierter (hindurchgelassener) und dabei evtl. gebrochener Lichtstrahlen berechnet.

3. Für jeden reflektierten oder transmittierten Lichtstrahl werden erneut die Schritte 1. und 2. durchgeführt, wodurch neue Knoten und Lichtstrahlen entstehen, für die wiederum 1. und 2. auszuführen sind. Abbildung 2.15 zeigt die von einem einzigen Bildpunkt ausgehenden Strahlen und Knoten, wobei die Strahlverfolgung in diesem Beispiel nach spätestens 4 Reflexionen oder Durchdringungen abbricht.

4. Für jeden Knoten des Sichtstrahlenbaumes wird ein Helligkeitswert errechnet, der sich aus dahinter liegenden Reflexionen bzw. Durchdringungen und der unmittelbaren Beleuchtung durch die Lichtquellen ergibt. Im Sichtstrahlenbaum weiter hinten liegende Knoten haben Einfluss auf das Erscheinungsbild weiter vorn liegender Knoten, da diese das Licht der hinten liegenden Knoten zur Kamera hin reflektieren bzw. brechen. Jedem Knoten wird (für jede der Grundfarben rot, grün und blau) ein Helligkeitswert zu-

geordnet, der sich aus

- seiner Beleuchtung durch Lichtquellen (nach den in dem folgenden Abschnitt behandelten Beleuchtungsmodellen) sowie
- den Helligkeitswerten der in Reflexions- bzw. Lichtbrechungsrichtung unmittelbar hinter ihm liegenden Knoten (gewichtet mit Faktoren, welche die Abschwächung der Helligkeit durch Reflexion bzw. Durchdringung beschreiben)

ergibt.

5. Die Helligkeits- bzw. Farbwerte der von der Kamera aus ersten Knoten werden den Pixeln des Bildes zugeordnet.

Raytracing ist, wie aus diesen Schritten deutlich wird, ein rekursives Verfahren, das sich in jedem Knoten des Sichtstrahlenbaumes selbst aufruft. In komplexen Szenen sind die Schritte 1., 2. und 4 sehr oft auszuführen, da sehr viele Reflexionen und evtl. auch Durchdringungen erfolgen – auch auf schnellen Computern kann Raytracing daher ein beachtliches Maß an Rechenzeit erfordern. Da Lichtstrahlen oftmals auch nach beliebig vielen Reflexionen nicht „ins Leere“ laufen, sondern immer wieder reflektiert werden, ist es durchaus möglich, dass der Raytracing-Algorithmus unendlich lange dauert. Deshalb wird er nach einer gewissen Zahl von Reflexionen abgebrochen. Diese Zahl wird *Strahlverfolgungstiefe* genannt; sie hat einen wesentlichen Einfluss auf die Detailtreue der Reflexionen in einem Bild, siehe die Abbildungen 2.16 und 2.17.[17]

[17] Bei Verwendung von POV-Ray lässt sich die Strahlverfolgungstiefe einer Szene mittels `global_settings {max_trace_level` *n*`}` auf *n* einstellen.

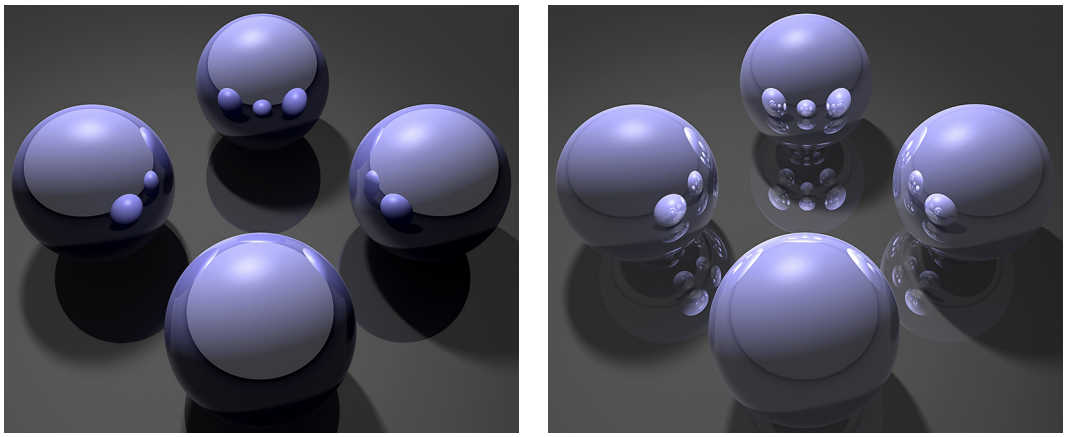


Abbildung 2.16: Einfluss der Strahlverfolgungstiefe (links: 2; rechts 20)

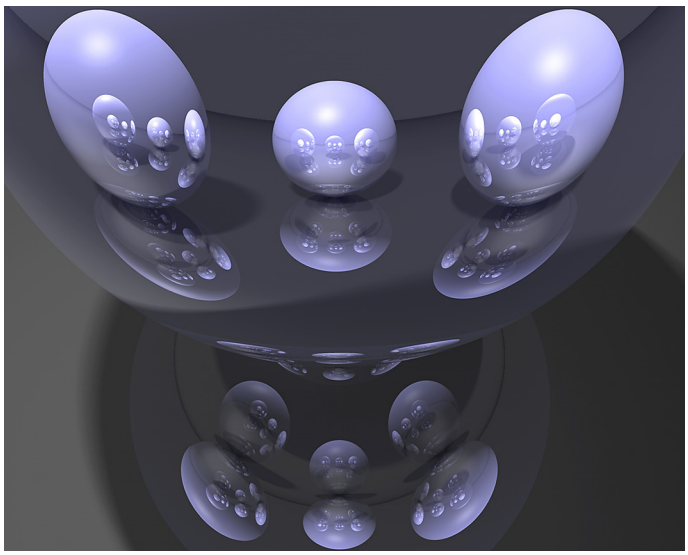


Abbildung 2.17: Vergrößerter Ausschnitt des mit der Strahlverfolgungstiefe 20 erzeugten Bildes in Abb. 2.16

2.3.2 Lokale Beleuchtungsmodelle

Um verschiedenartige Oberflächen zu simulieren, sind unterschiedliche Arten der Wechselwirkung von Objektoberflächen mit auftreffenden Lichtstrahlen zu betrachten und ihre Anteile zu wichten. Wird der Raytracing-Algorithmus durchlaufen, so muss dies für jeden Knoten des Sichtstrahlenbaums (Abb. 2.15 auf S.73) erfolgen. Dabei wird zwischen Wechselwirkungen

1. mit Lichtstrahlen, die direkt von Lichtquellen ausgehen, sowie
2. mit Lichtstrahlen, die von anderen Objektoberflächen reflektiert werden,

unterschieden. Wechselwirkungen des ersten Typs lassen sich durch *lokale Beleuchtungsmodelle* beschreiben, von denen im Folgenden die wichtigsten behandelt werden (ambiente und diffuse Beleuchtung, Entstehung von Glanzpunkten). Die Wechselwirkung mit Lichtstrahlen, die von anderen Objektoberflächen reflektiert werden, beschreibt das Modell der direkten Reflexion. Dieses nimmt daher im Vergleich zu den anderen im Folgenden beschriebenen Modellen eine Sonderstellung ein. Aus Anwendersicht (z. B. für die Festlegung von Texturattributen in POV-Ray[18] oder anderer 3D-Grafiksoftware) ist es jedoch notwendig, Wechselwirkungen beider Arten im Zusammenhang zu betrachten und zu wichten, um gewünschte Oberflächenerscheinungen zu erzielen.

[18]Die direkte Reflexionsfähigkeit wurde in dem Beispiel auf S. 59 für die Tischplatte mittels `reflection` eingestellt. Die Anteile anderer Beleuchtungskomponenten sind in POV-Ray-Szenenbeschreibungen ebenfalls innerhalb von `texture{finish{ ... }}` festzulegen.

2.3.2.1 Ambiente Beleuchtung (Umgebungslicht)

Die ambiente (richtungsunabhängige) Beleuchtung wird in der Natur durch Lichtstreuungen an Teilchen in der Atmosphäre hervorgerufen und tritt vor allem bei Nebel stark auf. Die Intensität I der ambienten Beleuchtung hängt nur von der globalen Intensität I_a des ambienten Lichtes und einem Faktor k_a (in POV-Ray: `ambient`) ab, der angibt, wie stark die Oberfläche eines Körpers ambientes Licht wiedergibt:

$$I = I_a \cdot k_a \quad \text{mit} \quad 0 \leq k_a \leq 1\,. \tag{2.1}$$

Die ambiente Beleuchtung ist somit unabhängig von der gegenseitigen Lage von Objekt, Kamera und Lichtquellen. Ein räumlicher Eindruck entsteht durch ambiente Beleuchtung nicht, eine Kugel erscheint wie in Abbildung 2.18.

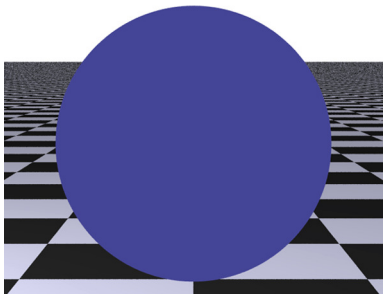


Abbildung 2.18: Ausschließlich ambiente Beleuchtung einer Kugel

2.3.2.2 Diffuse Reflexion (Streulicht)

Oberflächen wie weißes Papier, die nicht sehr glatt sind, reflektieren Licht unabhängig vom Einfallswinkel in alle Richtungen. Dies kommt dadurch zustande, dass die Lichtstrahlen unterschiedlich tief in unregelmäßig geformte Oberflächen eindringen (siehe Abbildung 2.19) und in verschiedene Richtungen zurückgeworfen werden. Je rauer eine Oberfläche ist, desto unterschiedlicher werden die Strahlen reflektiert bzw. gestreut.

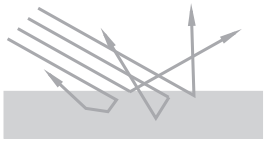


Abbildung 2.19: Diffuse Lichtstreuung

Um die so entstehende diffuse Reflexion zu beschreiben, wird die Leuchtdichte I_j des von einer Lichtquelle ausgehenden Lichtes betrachtet. Die Leuchtdichte beschreibt, wie viel Lichtleistung eine Lichtquelle pro Flächeneinheit abgibt. Tritt das Licht in einem flachen Winkel auf eine Oberfläche auf, so wird die Lichtleistung auf eine recht große Fläche verteilt und die Intensität der Beleuchtung

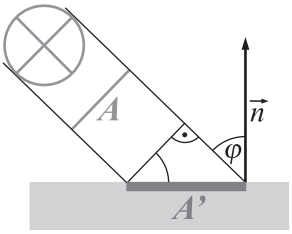


Abbildung 2.20:
Auf ein Flächenstück treffendes Licht

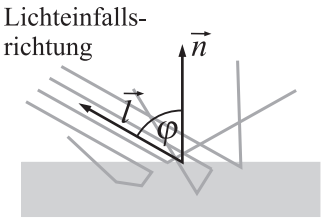


Abbildung 2.21:
Lichteinfalls- und Normaleneinheitsvektor

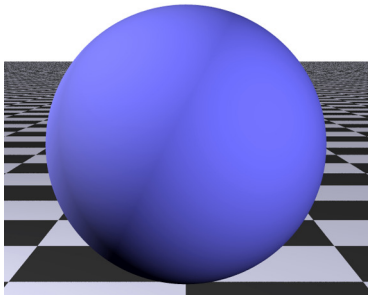


Abbildung 2.22:
Diffuse Beleuchtung einer Kugel

dadurch geringer. Wenn Licht, das durch ein Flächenstück A senkrecht zur Ausbreitungsrichtung scheint, auf ein Flächenstück A' der Materialoberfläche verteilt wird (siehe Abbildung 2.20), so gilt für die Beleuchtungsdichte I der Oberfläche:

$$I = \frac{A}{A'} \cdot I_j .$$

Das Verhältnis der Flächeninhalte A und A' lässt sich durch den Einfallswinkel φ ausdrücken:

$$A = A' \cdot \cos\varphi ,$$

wobei für φ Werte von 0 bis 90° betrachtet werden.[19] Für die Beleuchtungsintensität I, die das Licht auf der Oberfläche hervorruft, gilt:

$$I = \frac{A}{A'} \cdot I_j = I_j \cdot \cos\varphi .$$

Der Einfallswinkel kann als Skalarprodukt des normierten Lichteinfallsvektors $\vec{l}$ mit dem Normaleneinheitsvektor $\vec{n}$ im betrachteten Punkt angegeben werden:

$$\cos\varphi = \langle \vec{n}, \vec{l} \rangle .$$

Damit lässt sich die Beleuchtungsintensität, die durch diffuse Reflexion des Lichtes einer Lichtquelle mit der Lichtintensität I_j in einem Punkt verursacht wird, durch das LAMBERT*sche Kosinusgesetz* ausdrücken:[20]

$$\begin{aligned} I &= I_j \cdot k_d \cdot \max\big(0, \cos\varphi\big) \\ &= \begin{cases} I_j \cdot k_d \cdot \langle \vec{n}, \vec{l} \rangle & \text{falls } \langle \vec{n}, \vec{l} \rangle > 0 \\ 0 & \text{falls } \langle \vec{n}, \vec{l} \rangle \leq 0 . \end{cases} \end{aligned} \tag{2.2}$$

[19] Für Licht, das auf die Rückseite auftrifft ($\varphi > 90°$), würden sich negative Werte ergeben, was natürlich nicht sinnvoll ist.

[20] Durch die Betrachtung von $\max\big(0, \cos\varphi\big)$ bzw. die Fallunterscheidung in (2.2) wird vermieden, dass rückseitig einfallendes Licht negative Beleuchtungsintensitäten hervorruft.

Dabei ist k_d (mit $0 \leq k_d \leq 1$) eine Materialkonstante (in POV-Ray: `diffuse`), die angibt, wie stark eine Oberfläche Licht diffus reflektiert. Der diffuse Anteil des von einer Lichtquelle auf eine Oberfläche fallenden Lichtes ist somit abhängig von der Lichteinfallsrichtung, aber unabhängig von der Richtung zum Betrachter (siehe Abb. 2.22; es ist deutlich erkennbar, dass in der abgebildeten Szene zwei Lichtquellen vorhanden sind).

2.3.2.3 Direkte (spiegelnde) Reflexion

Bei direkten Spiegelungen liegen nach dem *Reflexionsgesetz* der einfallende Lichtstrahl $\vec{l}$, der reflektierte Strahl $\vec{b}$ und die Senkrechte $\vec{n}$ im betrachteten Flächenpunkt in einer Ebene und der Winkel zwischen der Senkrechten und dem Einfallsstrahl ist gleich dem Winkel zwischen der Senkrechten und dem reflektierten Strahl (siehe Abb. 2.23).

Vektoriell kann das Reflexionsgesetz folgendermaßen ausgedrückt werden: Ist $\vec{n}$ Normaleneinheitsvektor einer spiegelnden Oberfläche und sind $\vec{l}$ und $\vec{b}$ die normierten Richtungsvektoren des einfallenden bzw. des reflektierten Lichtstrahls, so sind $\vec{n}$, $\vec{l}$ und $\vec{b}$ komplanar und es gilt $\langle \vec{l}, \vec{n} \rangle = \langle \vec{b}, \vec{n} \rangle$.

Die Intensität I, mit der in einem Punkt eintreffendes Licht der Intensität I_{ein} zu einem Beobachter reflektiert wird, kann somit durch

$$I = \begin{cases} I_{ein} \cdot k_s & \text{falls } \langle \vec{l}, \vec{n} \rangle = \langle \vec{b}, \vec{n} \rangle \\ 0 & \text{sonst} \end{cases} \quad \text{mit} \quad 0 \leq k_s \leq 1 \tag{2.3}$$

angegeben werden, wobei $\vec{l}$ und $\vec{b}$ die normierten Verbindungsvektoren des betrachteten Punktes mit der Lichtquelle bzw. der Kamera sind. Die Konstante k_s gibt an, wie stark die Oberfläche eines Materials Lichtstrahlen direkt reflektiert, für ideale Spiegel gilt $k_s = 1$. In POV-Ray kann diese Konstante (wie in dem Beispiel auf S. 59) durch `reflection` für jedes Objekt festgelegt werden.

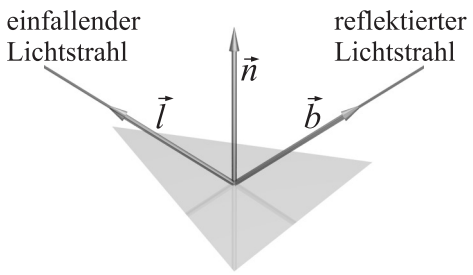


Abbildung 2.23: Reflexionsgesetz

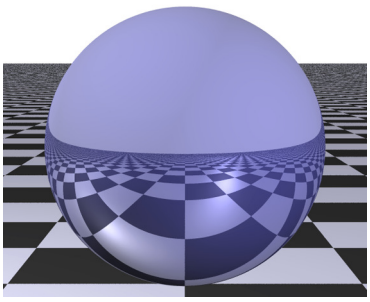


Abbildung 2.24: Direkte Reflexion

Im Gegensatz zur ambienten, diffusen und auch der PHONGschen Reflexion werden für die Berechnung der direkten Reflexion nicht nur Strahlen berücksichtigt, die unmittelbar von Lichtquellen ausgehen, sondern auch von anderen Objekten der Szene reflektierte Strahlen – dies wurde bereits anhand der Spiegelungen der Kugeln in den Abbildungen 2.16 und 2.17 sichtbar.

2.3.2.4 Entstehung von Glanzpunkten (spekulare Highlights)

Da reale Oberflächen keine idealen Spiegel sind, beschreibt das Reflexionsgesetz die Realität nur sehr unvollständig. Dennoch tritt auch bei nicht spiegelnden Oberflächen eine als *spekular* bezeichnete Beleuchtungskomponente auf, die vom Blickwinkel des Betrachters abhängt und somit nicht durch diffuse Reflexion erklärt werden kann. Diese Komponente ist durch Leuchtflecken („Highlights“) auf Körperoberflächen wahrzunehmen und z. B. auf Billardkugeln gut sichtbar. Da Lichtquellen meist als punktförmig angenommen werden, entstehen derartige Leuchtflecken durch direkte Reflexion nicht. In einem von PHONG BUI TUONG 1975 entwickelten Beleuchtungsmodell werden nun mehr oder weniger scharf abgegrenzte Leuchtbereiche in der Nähe der Punkte, in denen sich die Lichtquellen spiegeln, hinzugefügt.

Um Glanzpunkte zu generieren, wird ein *Highlight-Vektor* $\vec{h}$ betrachtet (siehe Abbildung 2.25), der die Richtung der Winkelhalbierenden der Verbindungsvektoren des Oberflächenpunktes zur Lichtquelle und zur Kamera angibt:

$$\vec{h} = \frac{\vec{b}+\vec{l}}{|\vec{b}+\vec{l}|}.$$

Wenn der Highlight-Vektor mit dem Normaleneinheitsvektor im betrachteten Flächenpunkt zusammenfällt, wird das Licht der Lichtquelle direkt zur Kamera reflektiert. Das Skalarprodukt der beiden Einheitsvektoren ist dann Eins. Je größer der Winkel zwischen Highlight-Vektor und Normaleneinheitsvektor ist, desto weniger beleuchtet die Lichtquelle den betrachteten Punkt der Oberfläche, und das Skalarprodukt wird immer kleiner. Die Intensität der PHONGschen Beleuch-

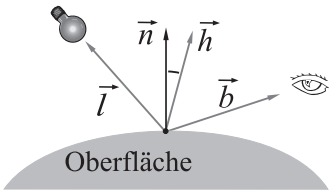


Abbildung 2.25: Highlight-Vektor

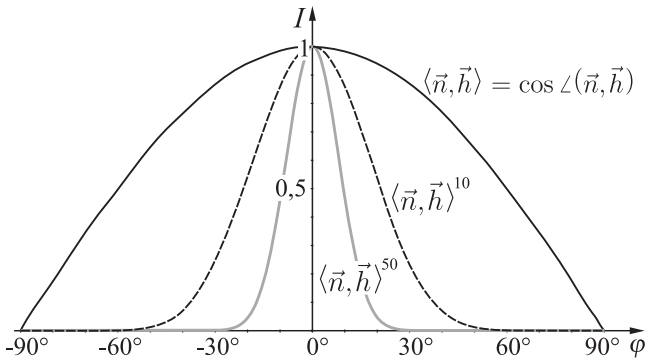


Abbildung 2.26: Einfluss des Rauigkeitsindex auf den Verlauf von Glanzpunkten

tung durch eine Lichtquelle mit der Lichtintensität I_j wird durch die folgende Gleichung beschrieben:[21]

$$I = \begin{cases} I_j \cdot k_p \cdot \langle \vec{n}, \vec{h} \rangle^m & \text{falls } \langle \vec{n}, \vec{l} \rangle > 0 \\ 0 & \text{falls } \langle \vec{n}, \vec{l} \rangle \leq 0\,. \end{cases} \tag{2.4}$$

Dabei ist k_p (mit $0 \leq k_p \leq 1$) eine Materialkonstante, die angibt, wie stark Glanzpunkte hervortreten. Der Exponent m (in POV-Ray: `phong_size`) wird als *Rauigkeitsindex* bezeichnet. Bei großen Rauigkeitsindizes fällt die Helligkeit bei wachsendem Winkel zwischen $\vec{n}$ und $\vec{h}$ schneller ab (siehe Abb. 2.26); es entstehen somit kleinere und schärfer abgegrenzte Glanzpunkte. Oberflächen aus Metall weisen kleine und stark hervortretende Glanzpunkte auf; sie sind somit durch hohe Werte des Rauigkeitsindex m gekennzeichnet. In der Abbildung 2.27 sind Beispiele für $m = 10$ (links) und $m = 50$ (Mitte und rechts) dargestellt, wobei der metallische Anschein der rechten Kugel durch die Kombination kleiner Glanzpunkte mit direkter Reflexion hervorgerufen wird.

[21] Die Fallunterscheidung in (2.4) ist – wie bereits in der Gleichung (2.2) der diffusen Lichtstreuung – notwendig, um zu vermeiden, dass rückseitig einfallendes Licht (mit $\angle(\vec{n}, \vec{l}) > 90°$, d. h. $\langle \vec{n}, \vec{l} \rangle < 0$) negative Beleuchtungsintensitäten hervorruft.

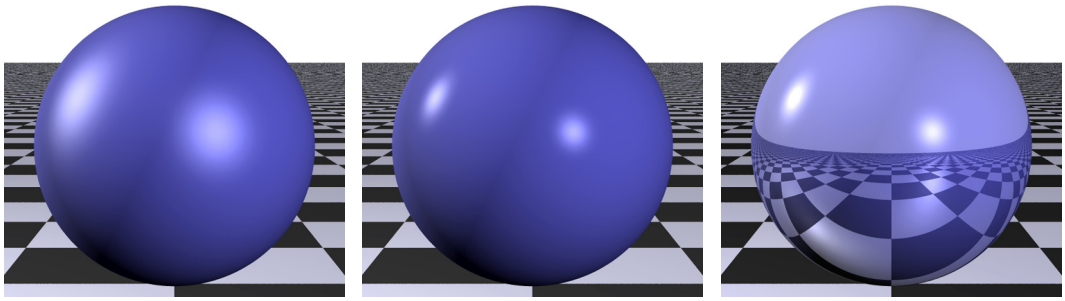


Abbildung 2.27: Spekulare Highlights nach dem PHONGschen Modell

2.3.2.5 Zusammenfassung und Erweiterungen (Ausblick)

Die Abbildungen 2.18, 2.22, 2.24 und 2.27 unterscheiden sich ausschließlich durch unterschiedliche Koeffizienten k_a, k_d, k_p, m für die ambiente, diffuse und PHONG-sche Beleuchtungskomponente sowie k_s für die direkte Reflexion, siehe die Gleichungen (2.1)-(2.4). Damit zeigt sich, dass diese fünf Texturattribute bereits umfangreiche Gestaltungsmöglichkeiten für Objektoberflächen gewähren. Bei Verwendung von POV-Ray wird dazu dem entsprechenden Objekt durch den folgenden Anweisungsblock eine Textur zugewiesen:[22]

```
texture{ pigment{ color rgb <r,g,b> }
         finish{ ambient k_a
                 diffuse k_d
                 phong k_p phong_size m
                 reflection k_s          } }
```

Dabei sind für k_a, k_d, k_p und k_s Werte von 0 bis 1 und für m beliebige positive (üblicherweise ganzzahlige) Werte zu setzen. Auch r, g und b sind Werte von 0 bis 1; aus diesen Werten werden im RGB-Farbmodell Farben durch rot-, grün- und blau-Komponenten gebildet; `<0,0,0>` entspricht Schwarz, `<1,1,1>` Weiß, `<1,0,0>` Rot, `<0,1,0>` Grün, `<1,1,0>` Gelb, `<0.5,0.5,1>` einem recht hellen Blau usw., siehe u. a. (Filler, 2007a), S. 74ff. und (Watt, 2002), S. 463ff.

Das aus den hier beschriebenen Komponenten bestehende Modell wurde in dieser Kombination 1975 von PHONG vorgeschlagen. Es erfuhr zahlreiche Erweiterungen (siehe z. B. (Watt, 2002), S. 242ff.). So verfeinerte BLINN 1977 das Modell der spekularen Highlights. Mit der spektralen Zusammensetzung von Glanzlichtern in Abhängigkeit vom Lichteinfallswinkel und vom Material beschäftigten sich COOK und TORRENCE 1982 – im Modell von PHONG haben Glanzlichter die Farbe der Lichtquelle. Andere Untersuchungen betreffen die Mikrogeometrie von Oberflächen und ermöglichen die überzeugende Simulation von Rauigkeit durch unterschiedliche Ausrichtungen von Mikrofacetten. In POV-Ray steht diese Möglichkeit durch Einfügen der Anweisung `crand` (mit einem Wert von 0 bis 1) zur Verfügung. Andere Erweiterungen der hier beschriebenen Beleuchtungsmodelle können durch Parameter wie `specular`, `roughness`, `metallic` u. a. genutzt werden[23] und treten mit teilweise anderen Bezeichnungen auch in den Texturierungsmodulen anderer Grafikprogramme auf.

[22] Unter `http://www.afiller.de/oom` steht eine POV-Ray-Szenenbeschreibung zur Verfügung, anhand derer leicht verschiedene Kombinationen der Werte getestet werden können.

[23] Siehe hierzu die POV-Ray-Hilfe oder `http://www.povray.org/documentation`.

2.3.3 Transparenz und Brechung

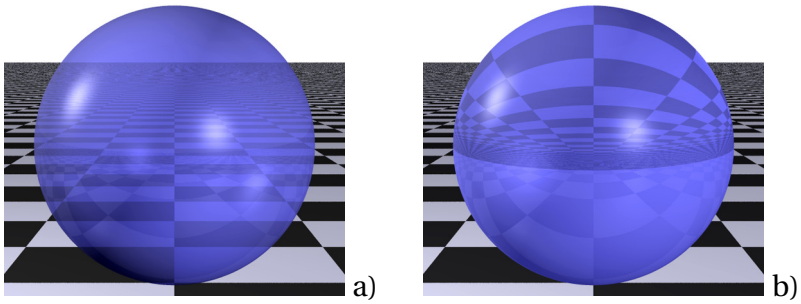
Transparenz von Objekten lässt sich bei der Bildberechnung durch Raytracing leicht realisieren, indem Lichtstrahlen Objektoberflächen (mit Abschwächung) durchdringen können.[24] Abb. 2.28 a) zeigt eine teilweise transparente (und zugleich leicht reflektierende) Kugel, die jedoch eher den Eindruck einer Seifenblase hinterlässt. Die Ursache hierfür besteht darin, dass die Richtungsänderung von Lichtstrahlen durch Brechung unberücksichtigt blieb.

Aufgrund der Tatsache, dass sich die Geschwindigkeit des Lichtes beim Übergang von einem Medium in ein anderes ändert, erfolgt an der Grenzfläche eine Richtungsänderung von Lichtstrahlen, die als Refraktion oder Brechung bezeichnet wird. Der Brechungsindex n eines durchsichtigen Materials ist der Quotient der Lichtgeschwindigkeit im Vakuum und der in dem betrachteten Material. Luft hat einen Brechungsindex von ca. 1,00029; der Lichtgeschwindigkeitsunterschied zwischen Luft und Vakuum ist also sehr gering. Demgegenüber beträgt der Brechungsindex von Wasser ca. 1,33, womit sich Licht in Wasser nur etwa 3/4 so schnell ausbreitet wie im Vakuum oder in der Luft. Für verschiedene Arten von Glas beträgt der Brechungsindex 1,5 bis 1,9.

Das *Brechungsgesetz* beinhaltet zwei Aussagen:

- Einfallender und gebrochener Lichtstrahl in einem Oberflächenpunkt sowie das Einfallslot in diesem Punkt liegen in einer Ebene.
- Für die Winkel α_1 und α_2 zwischen dem Einfallslot und dem einfallenden bzw. dem gebrochenen Lichtstrahl (siehe Abbildung 2.29) sowie die

[24] Bei Verwendung von POV-Ray lässt sich dazu die Farbgebung von Texturen (siehe S. 81) durch bis zu zwei Komponenten ergänzen: `pigment{ color rgbft <`r, g, b, k_f, k_t`> }`. Dabei steht f für gefilterte und t für transmittierende Transparenz. Bei der gefilterten Transparenz wird die Farbe dahinter liegender Objekte durch die Farbe des durchdrungenen Objekts gefiltert, was u. a. bei farbigem Glas der Fall ist. Transmittierende Transparenz bewirkt keine Farbfilterung; diese Transparenzart tritt z. B. bei sehr dünnen Textilien auf.



a) b)

Abbildung 2.28: Transparenz
a) ohne Brechung
b) mit Brechung

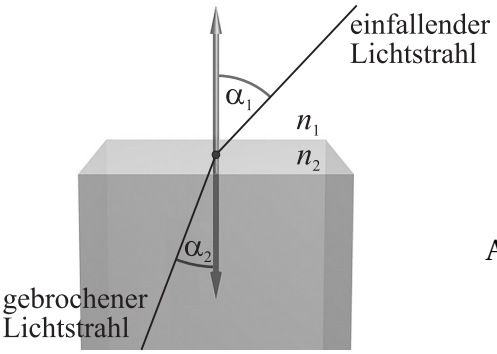


Abbildung 2.29: Brechungsgesetz

Brechungsindizes n_1 des Materials, aus dem der Strahl kommt, und n_2 des Materials, in das er übergeht, gilt

$$\frac{\sin\alpha_1}{\sin\alpha_2} = \frac{n_2}{n_1}. \tag{2.5}$$

Vektoriell kann das Brechungsgesetz in der Form

$$n_1 \cdot (\vec{n} \times \vec{l}) = n_2 \cdot (-\vec{n} \times \vec{b}) \tag{2.6}$$

geschrieben werden, wobei $\vec{n}$ der Normaleneinheitsvektor der Grenzfläche im Auftreffpunkt des einfallenden Lichtstrahls ist sowie $\vec{l}$ und $\vec{b}$ die normierten Richtungsvektoren des einfallenden bzw. des gebrochenen Lichtstrahls sind. Da das Vektorprodukt zweier Vektoren zu beiden der Vektoren, aus denen es gebildet wird, orthogonal ist, beinhaltet die Gleichung (2.6) auch die Bedingung, dass die Vektoren $\vec{n}$, $\vec{l}$ und $\vec{b}$ komplanar sind.

In POV-Ray lässt sich Lichtbrechung durch Einfügen der Anweisung

`interior { ior` *Brechungsindex* `}`

in die Objektbeschreibung simulieren;[25] für Abb. 2.28 b) und Abb. 2.31 (siehe S. 85) wurde `ior 1,5` gesetzt. Als weiteres Beispiel zeigt Abb. 2.30 die Wirkung einer sphärischen Linse, die als Durchschnitt zweier Kugeln (siehe Abschnitt 2.2.4) modelliert und ebenfalls mit dem Brechungsindex 1,5 versehen wurde.

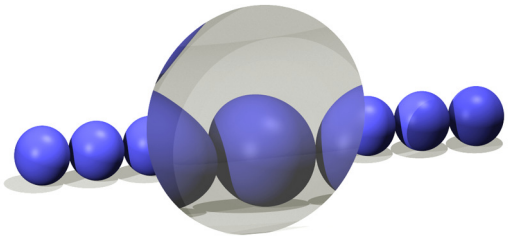


Abbildung 2.30: Lichtbrechung an einer optischen Linse

[25] Die Bezeichnung `interior` gibt an, dass es sich nicht (wie bei `texture`) um eine Oberflächen-, sondern um eine Materialeigenschaft handelt; `ior` steht für Index Of Refraction.

2.3.4 Zusammenfassung der berücksichtigten Wechselwirkungen; Grenzen des Raytracing-Verfahrens

In den vorangegangenen Abschnitten wurden zwei verschiedene Wechselwirkungen zwischen Objektoberflächen und Lichtstrahlen betrachtet:

1. Wirkungen von Lichtstrahlen, die unmittelbar von Lichtquellen ausgehen, auf Objektoberflächen werden durch lokale Beleuchtungsmodelle beschrieben (ambiente und diffuse Beleuchtung, Glanzpunkte).
2. Bedingt durch die Funktionsweise von Raytracing werden auf Objekten durch Reflexion und Brechung andere Objekte einer Szene sichtbar.

Es ist also zwischen *Wirkungen von Lichtquellen auf Objektoberflächen* und *Wechselwirkungen zwischen Objektoberflächen* zu unterscheiden, wenngleich diese ebenfalls durch Lichtstrahlen vermittelt werden. Damit im Zusammenhang steht eine grundsätzliche Grenze des Raytracing-Prinzips: Auch wenn Körper sehr hell erleuchtet sind, können sie nahe liegende Körper nicht „beleuchten“, sondern haben nur dann einen Einfluss, wenn sie sich direkt in ihnen spiegeln. Auch Erweiterungen lokaler Beleuchtungsmodelle (siehe S. 81) können somit die folgenden prinzipbedingten Beschränkungen nicht aufheben:

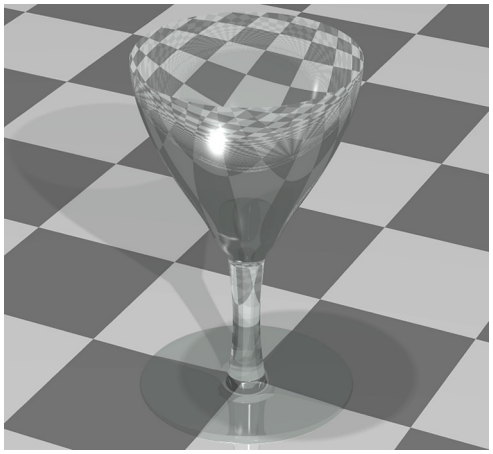
- Die Umkehrung der Strahlenrichtung ermöglicht zwar die exakte Modellierung von Reflexionen und Lichtbrechungen. Lichtstrahlen werden jedoch nur berücksichtigt, wenn sie den Betrachter erreichen. Aufgrund der dadurch bedingten „passiven Rolle“ der Lichtquellen erfolgt keine Darstellung von Lichtbündelungen (Kaustiken), die u. a. durch transparente Körper sowie Hohlspiegel hervorgerufen werden (Abb. 2.31 b).
- Farben von Objekten erkennt der Beobachter nach dem Prinzip des Raytracing nur dann in anderen Objekten wieder, wenn direkte Reflexion erfolgt. In der Realität entsteht jedoch ein großer Teil der Beleuchtung durch indirekte (diffuse) Mehrfachreflexionen, wodurch Objekte durch das von anderen Objekten reflektierte Licht sichtbar werden.[26]

Um diese Limitationen aufzuheben, wurde Raytracing als (nach wie vor dominierendes Verfahren der fotorealistischen Computergrafik) in den zurückliegenden beiden Jahrzehnten um zusätzliche Bildberechnungsverfahren ergänzt:

[26] So erfolgt eine Beleuchtung von Gegenständen in einem Raum durch das von matten Wänden reflektierte Licht. Umgekehrt hat z. B. eine weiße Wand in der Nähe eines großen roten Gegenstandes einen rötlichen „Schimmer“, denn Körper strahlen einen Teil des empfangenen Lichtes in diffuser Weise wieder ab und beleuchten dabei indirekt ihre Umgebung.

- Durch *Radiosity-Verfahren* werden Mehrfachreflexionen von Licht zwischen verschiedenen Flächen und die dadurch hervorgerufene Beleuchtung der Flächen beschrieben. Eine Grundlage dafür bilden Modelle der Strahlungsübertragung von Wärme.
- Eine physikalisch korrekte Simulation sowohl von Kaustiken als auch diffuser Reflexionen ist durch die Kombination von Raytracing mit *Photon-Tracing* bzw. „Vorwärts-Raytracing“ möglich. Dabei werden von Lichtquellen emittierte „Photonen“ verfolgt. Diese können von Objekten reflektiert, absorbiert oder hindurchgelassen und dabei eventuell gebrochen werden. Den von Photonen „getroffenen“ Punkten geometrischer Objekte werden Strahlungsenergien zugeordnet, und in jedem Knoten des Sichtstrahlenbaums wird die Dichte und Energie der Photonen in der Nähe des dem Knoten entsprechenden Punktes ermittelt. Nach dem Photon-Tracing erfolgt die Bildberechnung durch Raytracing. Hierbei werden nun die durch Photon-Tracing ermittelten Strahlungsintensitäten zu den durch Raytracing berechneten Lichtintensitäten addiert.

Sowohl Radiosity als auch Photon-Tracing und weitere Verfahren der globalen Illumination (siehe u. a. (Shirley, 2005), S. 547ff. und (Watt, 2002), S. 309ff.) werden zunehmend (mit der Verfügbarkeit schneller Computer bzw. „Renderfarmen“) seit der Jahrtausendwende ergänzend zu Raytracing eingesetzt. Die dadurch erreichten Verbesserungen der Realitätstreue sind teilweise recht subtil, erfordern aber oft einen hohen Rechenaufwand – so benötigt z. B. die Bildberechnung für Abb. 2.31 b) fast die dreifache Rechenzeit gegenüber Abb. 2.31 a).


a)

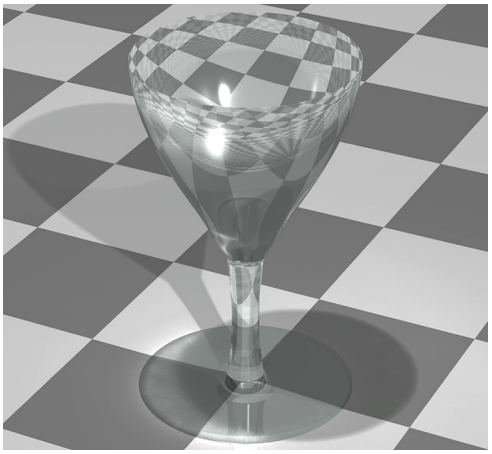

b)

Abbildung 2.31: a) Raytracing b) Raytracing kombiniert mit Photon-Tracing

2.4 Einbeziehung in den Mathematik- und Informatikunterricht

Die vorangegangenen Abschnitte haben deutlich gemacht, dass Modellierung und Bildberechnung in der 3D-Computergrafik enge Bezüge zu Unterrichtsinhalten des Mathematikunterrichts (insbesondere des Stoffgebietes Analytische Geometrie in der Sekundarstufe II) und zugleich zu Kernkonzepten der Informatik (objektorientierte Modellierung, Algorithmen) aufweisen. Es hat sich zudem gezeigt, dass die Erstellung fotorealistischer Computergrafiken mithilfe einer Software wie POV-Ray für Schüler eine motivierende Herausforderung darstellt. Eine Einbeziehung von Elementen der 3D-Computergrafik sowohl in den Mathematik- als auch in den Informatikunterricht erscheint somit sinnvoll. Diesbezügliche Konzepte wurden entwickelt und in Unterrichtsversuchen erprobt. Ihre umfassende Darstellung würde den Rahmen dieses Kapitels überschreiten, weshalb auf die folgenden (über `http://www.afiller.de/oom` erreichbaren) Arbeiten verwiesen wird.

- In (Filler, 2007a) werden Unterrichtsvorschläge zur Einbeziehung von Elementen der 3D-Computergrafik in das Stoffgebiet Analytische Geometrie dargestellt sowie Erfahrungen aus entsprechenden Unterrichtsreihen und Projekten geschildert. Dabei werden zwei Aspekte berücksichtigt:
  - Dreidimensionale Grafiksoftware (wie POV-Ray) kann im Unterricht als *Werkzeug* zum Einsatz kommen, um Objekte und Relationen in der Analytischen Geometrie zu visualisieren.
  - Zentrale mathematische Grundlagen der 3D-Computergrafik beruhen auf Unterrichtsinhalten der Analytischen Geometrie (Skalarprodukt, Normalenvektoren, siehe Abschnitt 2.3.2) und können deren Behandlung motivieren. Elemente der 3D-Computergrafik bieten sich insofern als *Gegenstände des Unterrichts* an.
- Mit dem Einsatz von POV-Ray in Informatik-Arbeitsgemeinschaften in der Sekundarstufe I befasst sich die Seminar-/Praktikumsarbeit (Catalina, 2006). Mit Schülern der Klassenstufen 8-10 wurde POV-Ray verwendet, um Grafiken zu generieren und dabei grundlegende Überlegungen zur objektorientierten Modellierung anzustellen (Attribute und Methoden von 3D-Objekten, Kompositionsmöglichkeiten für Operationen und Objekte, siehe dazu auch Abschnitt 2.2). Entsprechende Herangehensweisen und Erfahrungen werden in (Catalina, 2006) beschrieben.

Kapitel 3

Objektorientierte Schreibweisen in der mathematischen Modellbildung

Prof. Dr. Engelbert Niehaus
Fachgebiet Didaktik der Mathematik
Universität Koblenz-Landau
niehaus@uni-landau.de

Dieses Teildokument steht unter der »Creative Commons Lizenz« BY-NC-ND. Dies bedeutet: bei weiterer Verwendung des Textes sind die Namen der Autoren zu nennen, die Weiternutzung darf ausschließlich nicht kommerziell erfolgen, das Dokument darf nicht bearbeitet werden. Weitere Details zu den Creative Commons Lizenzen finden sich unter: `http://creativecommons.org/licenses/?lang=de`

3.1 Einleitung zur mathematischen Formalisierung der Objektorientierung

Wie in Abschnitt 1.3 (S. 26 ff.) gezeigt gibt es zahlreiche Zugänge, mit denen man einem Objektbegriff kommen kann und letztlich auf der Basis der möglichen Paradigmen ein objektorientiertes Modell zu erzeugen.

Aus mathematikdidaktischer ist dabei die genannte Abbildtheorie von Bedeutung, bei dem der Modellierende ein Abbild der Realität schafft und der Lehrende aus dem Modellierungsprodukt die analytischen Kompetenzen und den Wahrnehmungsfähigkeiten des modellierenden Lernenden ableiten kann. Die

diagnostische Strategien der Lehrenden beschränken sich dabei aber nicht notwendigerweise aus das Modellierungsprodukt allein, denn auch die sprachlichen Erläuterungen zum Modellierungsprodukt stellen einen wesentlichen Bestandteil für die Analyse der Modellierungskompetenz dar.

In diesem Abschnitt wird mathematische Formalisierung erarbeitet, die in Analogie zur Syntax einer Programmiersprache die Implementierung im Sinne des objektorientierten Modells eine Ausformulierung des Modells in mathematischer Syntax ermöglicht. Wie bei Programmiersprachen auch kann das Grundkonzept der Objektorientierung mit unterschiedlicher Qualität durch syntaktischen Rahmenbedingung der Syntax unterstützt werden. So kann eine Programmiersprachensyntax einerseits eine sehr transparente Implementierung der objektorientierten Ideen ermöglichen oder andererseits gibt es auch Progrmmiersprachen, deren Syntax eine nur unvollständige Unterstützung des objektorientierten Ansatzes bietet.

Die Mathematik hat bisher keine eigenständige syntaktische Strukturen erarbeitet, die sich als mathematische Formalisierung für objektorientierte Modelle eignen. Sicherlich können die objektorientierten Ergebnisse eines Modellierungsprozesses auch funktional äquivalent über Mengen und Abbildungen realisiert werden. Betrachtet man die objektorientierte Struktur des Modellierungsergebnisses auch selbst als didaktisch wertvolles Resultat, dann kann man daraus die Zielsetzung ableiten, auch diese Struktur in der mathematisch-formalen Beschreibung sichtbar zu machen und in einer zyklischen Optimierung eine Modells in der mathematischen Implementation sichtbar zu machen.

Wenn sich die Mathematik Computern in Numerik, Algebra und Geometire bedient, um mathematische Modelle in einer Softwareumgebung zu untersuchen, wird neben der Formalisierung in eine mathematischen Sprachen auch wieder die Implementation von objektorientierten Strukturen in Programmier- oder Skriptsprache notwendig. Als Beispiel für diese Implementierung ist aber nicht nur die in Kapitel 2 (S. 58 ff.) gezeigte Anwendung von 3D-Software oder Dynamische Geometriesoftware (wie z.B. Geogebra) zu nennen, sondern auch Computer-Algebra-System CAS (wie z.B. Maxima) oder Numeriksoftware (wie z.B. Octave).

Zunächst einmal scheint es daher sinnvoll, für die Einführung der objektorientierten Modellierung auch im mathematischen Kontext mit Programmier- oder Skriptsprachen einzusetzen, die das Konzept der Objektorientierung besonders deutlich machen.

Die oben genannten Softwaretypen für die Mathematik unterstützen objektori-

entierte Strukturen im Kern auf der syntaktischen Ebene nicht. Im schulischen Kontext kann in Analogie zu konkreten Softwareprojekten auch nicht notwendig davon ausgehen, dass eine Programmier- oder Skriptsprache frei wählbar ist (z.B. bei der Erweiterung eines bestehenden Programmpaketes).

Das hier nun vorgestellte mathematisches Modells der Objektorientierung bietet die Möglichkeit, objektorientierte Strukturen in einer mathematischen programmiersprachenunabhängigen Weise formal mit mathematischer Syntax zu beschreiben, bevor diese mit einer im Sinne der Objektorientierung ungeeigneten Programmier- oder Skriptsprachensyntax implementiert und aus einem mathematischen Blickwickel die Modellierungsergebnisse untersucht werden können. Auf diese Weise kann man zusätzlich auch ungeeigneten Programmiersprachen einen objektorientierten SStempel“ aufzudrücken. Am Beispiel von JavaScript, dessen Syntax keine Vererbung unterstützt, wird die Umsetzung objektorientierter Ideen veranschaulicht.

Bei dieser hier vorgeschlagenen Formalisierung der Objektorientierung (im Folgenden abgekürzt “OO“) geht es in erster Linie um eine maschinen- und programmiersprachenunabhängige Darstellung der Grundstrukturen der OO. Dies umfasst aus mathematischer Sicht eine klare Trennung der im Rahmen der objektorientierten Modellierung gegebenen Grundstrukturen. Diese sind:

- **Trennung von Objekt und Klasse** Objekt als Element einer Klasse, eine Klasse beschreibt Gemeinsamkeiten aller Objekte in einer Klasse)
- **Trennung von Prozessen und Methoden** Prozesse laufen in Objekten ab, eine Methode definiert in einer Klasse die Gemeinsamkeiten nach denen bestimmte Prozesse in aller Objekte einer Klasse ablaufen.
- **Lebenszyklus von Objekten:** Objekte unterliegen einem Lebenszyklus. Sie werden als Bestandteil eine dynamischen Systems betrachtet. Dieses Laufzeitsystem für den Lebenszyklus von Objekte wird als zeitabhängige Struktur getrennt von statischen Beziehungen zwischen Objekten oder zwischen Objekt und Klasse betrachtet. Beispielsweise würde man Albert Einstein immer noch zu Klasse Mensch rechnen, auch wenn sein Lebenzyklus bereits beendet ist.

Normalerweise ist eine formale Darstellung eines Gebietes im wesentlichen *bottom-up* aufgebaut, damit die Teilstrukturen exakt definiert sind, aus denen komplexe-

re formale Strukturen sich zusammensetzen. Für den Leser ist es aber schwierig, sich insbesondere formale Strukturen äuf Vorratänzueignen, um erst am Ende der Ausführungen zu wissen, wofür diese formalen Strukturen notwendig waren. Die formale Entwicklung der mathematischen Beschreibungssprache ist daher *top-down* strukturiert, wodurch bereits zu Beginn Begriffe verwendet werden, die erst im weiteren Verlauf präzisiert werden. Dadurch treten Inhalte mehrfach in unterschiedlicher Komplexität auf.

Innerhalb des Textes werden Fragen formuliert, die die Motivation für die formale mathematische Modellierung verdeutlichen sollen.

3.2 Klassen und Objekte

Frage I : *Wie unterscheiden sich die Begriffe Objekt und Klasse voneinander bzw. dürfen diese Begriffe sogar synonym verwendet werden?*

In der folgende Tabelle werden Bezüge zur Mengentheorie, der Variablendeklaration und zu technischen Konstruktionen hergestellt, die zunächst einmal grob die Unterschiede zwischen Objekt und Klasse veranschaulichen sollen. Diese zeigen aber nur eine kleinen Ausschnitt der möglichen Sichtweisen, woher die Objekte stammen (siehe Abschnitt 1.3 (S. 26 ff.) Für die mathematische Formalisierung ist aber im weiteren Verlauf des Kapitels allein die mengentheoretische Beschreibung von Bedeutung.

Bereits bei diesen Bezügen wird deutlich, dass Objekt und Klasse auf keinen Fall synonym verwendet werden dürfen.

Mengen	die Klasse K wird als Zusammenfassung von Objekten $x_1, x_2, x_3, \ldots$ zu einer Menge K aufgefasst. $K = \{x_1, x_2, x_3, \ldots\}$
Deklaration	die Klasse K wird als Datentyp aufgefasst und die Objekte $x_1, x_2, x_3, \ldots$ der Klasse besitzen den Typ K. z.B. durch die `x:INTEGER;` wird das Objekt `x` als Variable vom Typ K =`INTEGER` deklariert. Die Deklaration legt fest, welche Inhalte (hier ganzen Zahlen) durch Objekte der Klasse dargestellt werden können. *Erst Objekten können konkrete Variableninhalte zugewiesen werden, Klassen dagegen nie!*
Konstruktion	Klassen entsprechen einer Konstruktionsbeschreibung oder auch einem "Bauplan“ für Objekte, der Eigenschaften festlegt, die alle Objekte x_i der Klasse K besitzen – z.B. entspricht der Bauplan einer Klasse `kHaus`, nach dem verschiedene Objekte `xHaus1, xHaus2, xHaus3` gebaut werden.

Ferner haben Klassen bzw. Objekte folgende grundlegende Eigenschaften:

- Klassen definieren Datenstrukturen und Prozesse (Zustände, Methoden), die in einem Objekt dieser Klasse ablaufen können.

- Objekte sind dynamische, (lauf-)zeitabhängige Gebilde. Sie durchlaufen also einen Lebenszyklusünd existieren daher nur für einen beschränkten Zeitraum.

In der hier vorgestellten mathematischen Formalisierung der OO (=ObjektOrientierung) werden *Objekte* als Elemente von *Klassen* beschrieben, deren Existenz an einen *Adressraum* und die *Zeit* gebunden ist. In einer Klasse werden Objekte zusammengefasst, die eine gewisse *Minimalanforderung* in der *Zustandsverwaltung* und der *Funktionalität* erfüllen.

3.2.1 Beziehung zwischen Objekt und Klasse

Eine Klasse ist eine Zusammenfassung von Objekten zu einer Menge. Die Mengenbildung erfolgt bezüglich Eigenschaften, die alle Objekte (Elemente) der Klasse besitzen. Ein Objekt kann gleichzeitig zu mehreren Klassen gehören (wie z.B. $x \in \mathbb{N}$ und $x \in \mathbb{R}$). Jedes Objekt wird aber nur als Objekt einer ganz bestimmten Klasse erzeugt (x wird als natürliche Zahl erzeugt, damit gehört x aber auch zu der Oberklasse $\mathbb{R}$).

Die folgende Abbildung zeigt die *Elementbeziehung* zwischen Objekten und Klassen.

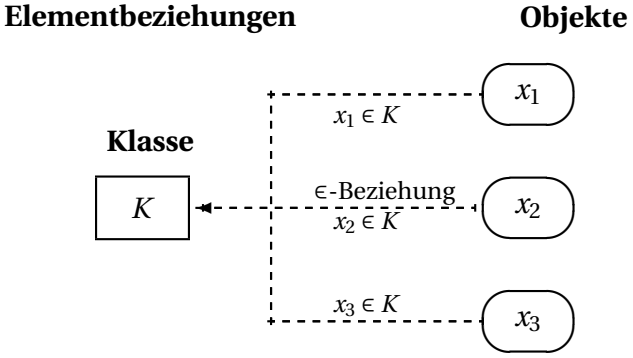


Abbildung 3.1: Elementbeziehung

3.2.2 Beziehung zwischen Klassen - Vererbung

In der Klasse K_0 sind Objekte zusammengefasst, die eine gewisse *Minimalanforderung* in der *Zustandsverwaltung* und der *Funktionalität* erfüllen. Durch die Vererbungsbeziehung ψ zu der Klasse K besitzt jedes Element direkt alle Zustände und die gesamte Funktionalität, die auch Objekte der Klasse K_0 besitzen (d.h. K erbt von K_0)

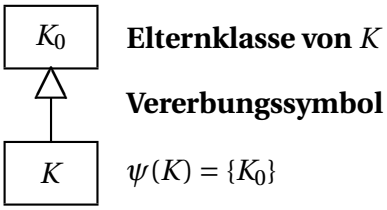


Abbildung 3.2: Vererbung

Durch die Vererbungsbeziehung erhält die *abgeleitete* Klasse K alle Eigenschaften (Zustände und Funktionalität) der *Elternklasse* K_0. Die abgeleitete Klasse stellt somit eine Erweiterung der Elternklasse dar. Daraus ergibt sich die Teilmengenbeziehung $K_0 \supset K$. K hat nun eine erweiterte Funktionalität bzw. eine komplexere Zustandsverwaltung, aber es gibt auch weniger Objekte, die diese erweiterte Minimalanforderung an Zustandsverwaltung und Funktionalität erfüllen.

3.2.3 Beziehungen zwischen Objekten

Weiterhin gibt es neben der Elementbeziehung zwischen Klasse und Objekt und der Vererbungsbeziehungen zwischen Klassen auch Beziehungen zwischen Objekten (*Teilobjektbeziehung* =Aggregation , *Kenntbeziehung* =Assoziation). Diese Beziehungen sind immer *(lauf-)zeitabhängig*. Bei einer Teilobjektbeziehung kann der Besitzer wechseln, Teilobjekt werden erzeugt bzw. freigegeben und Kenntbeziehungen können aufgebaut und wieder abgebaut werden.

- die Assoziation ist die Grundlage dafür, dass Nachrichten zwischen zwei Objekten ausgetauscht werden können.
- die Aggregation beschreibt die Beziehung zwischen dem *Ganzen* und seinen *Teilen*. Die Aggregation ist eine besondere Assoziation, bei der die Existenz der Teile an die Existenz des Ganzen gebunden ist.

In einem Diagramm werden die Beziehungen wie folgt veranschaulicht: Assoziationen

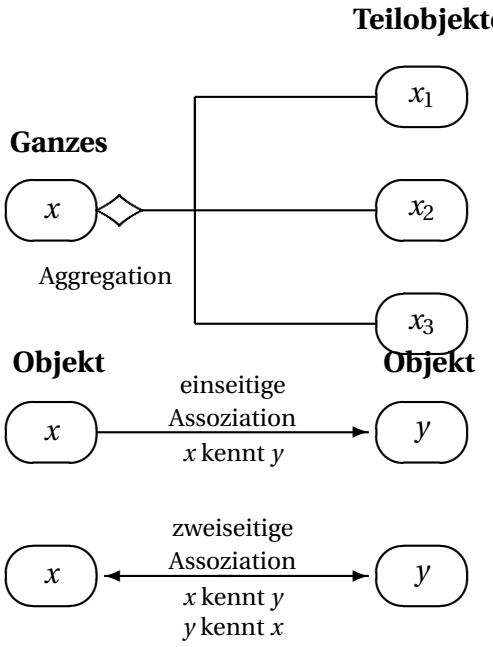


Abbildung 3.3: Assoziation und Aggregation

müssen *nicht* notwendig *zweiseitig* sein. *Paul kennt die Ministerpräsidentin* aber *die Ministerpräsidentin muss nicht unbedingt Paul kennen.*

Aggregation und Assoziation sind Beziehungen zwischen Objekten und nicht zwischen Klassen!

Im Folgenden werden zunächst Klassen und die Vererbungsrelation als zentrale Beziehung zwischen Klassen dargestellt.

3.3 Klassen

Eine Klasse K wird durch $\mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$ definiert, wobei $\psi(K)$ die Menge der *Elternklassen*, $\mathcal{Z}(K)$ die Menge der *Zustände* und $\mathcal{M}(K)$ die Menge der *Methoden* festlegt. $\mathcal{Z}(K)$ und $\mathcal{M}(K)$ enthalten nur die Zustände und Methoden, die *zusätzlich* zu den von $\psi(K)$ geerbten Zuständen und Methoden definiert werden.

Frage II : *Welcher Unterschied besteht zwischen der Klasse K und der Menge von Objekten $\mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$, die K definiert?*

$\mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$ fasst alle Elemente zusammen, die als Objekt der Klasse K erzeugt wurden. Die Klasse K umfasst u.U. aber noch weitere Elemente, die als Objekte von abgeleiteten Klassen $K_{sup} \subset K$ erzeugt wurden. Auch die Objekte $x \in K_{sup}$ erfüllen dann die Minimalanforderung der Klasse K. Ein $x \in K_{sup}$ besitzt aber u.U. noch weitere Zustände und eine erweiterte Funktionalität, die über die Minimalanforderung an die Klasse K hinausgehen.

3.3.1 Graphische Darstellung und Bezeichnung

Um Klassen- und Objektbezeichnungen zu trennen, werden in den folgenden Beispielen

- Klassen durch ein vorangestelltes k (z.B. `kAuto` ist die Klasse Auto) und
- Objekte durch ein vorangestellte x (z.B. `xPetersAuto`)

gekennzeichnet. Darüber hinaus werden Zustände und Methoden durch ein vorangestelltes z bzw. m unterscheidbar.

Die Klassendarstellung kann entweder vereinfacht (links) oder mit Zustandsvariablen und Methoden (rechts) dargestellt werden. Aus der erweiterten Klassen-

Klassendarstellungen

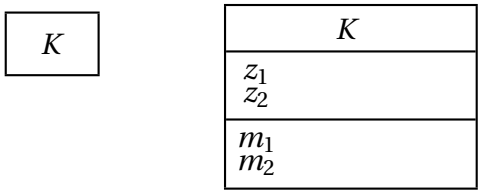


Abbildung 3.4: Darstellung von Klassen

darstellung (rechts) kann man also ablesen, dass alle Objekte der Klasse K die Zustände z_1, z_2 und die Methoden m_1 und m_2 besitzen. Weil es in einer Klasse K unterschiedliche Objekte x_1, x_2, x_3, x_4 ... geben kann und die Variableninhalte der Zustände bei jedem Objekt verschieden sein können, bezeichnet $x_4.z_2$ die

Zustandsvariable z_2 des Objektes x_4. Allgemein wird im Folgenden die nachstehende Notation verwendet:

- ⟨Klasse⟩.⟨Zustand⟩ bezeichnet den Typ“ der Zustandsvariablen ⟨Zustand⟩, der in der Klasse ⟨Klasse⟩ definiert ist (oder streng objektorientiert bezeichnet ⟨Klasse⟩.⟨Zustand⟩ die Klasse des Zustandes ⟨Zustand⟩).
- ⟨Objekt⟩.⟨Zustand⟩ bezeichnet die Zustandsvariable ⟨Zustand⟩ des Objektes ⟨Objekt⟩. Eine Zustandsvariable eines Objektes kann einen konkreten Wert (=Zustand) besitzen. Damit gilt für $x \in K$ auch $x.z_1 \in K.z_1$.
- ⟨Klasse⟩.⟨Methode⟩ bezeichnet die Definition der Methode ⟨Methode⟩ in der Klasse ⟨Klasse⟩.
- ⟨Objekt⟩.⟨Methode⟩ ist ein Methodenaufruf der Methode ⟨Methode⟩, der an das Objekt ⟨Objekt⟩ gesendet wird. Diesen Methodenaufruf bezeichnet man auch als *Nachricht* an das Objekt ⟨Objekt⟩.

Beispiel 3.3.1: *(Klasse)* Die Klasse kLebewesen, die in der nachstehenden Abbildung als Beispiel angeführt ist, ist eine sehr einfache Klasse. Alle Objekte dieser

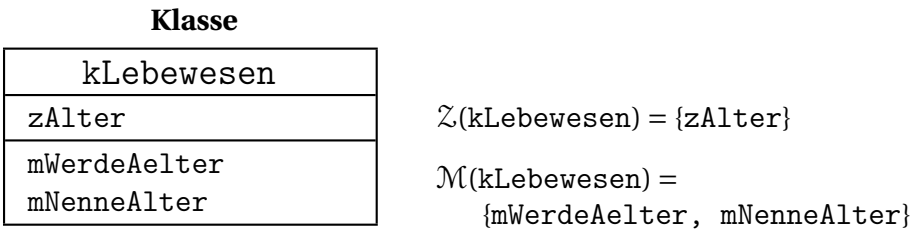


Abbildung 3.5: Beispiel für eine Klasse

Klasse besitzen ein Alter zAlter, das natürlich bei jedem Objekt der Klasse verschieden sein kann, z.B. xMeinHamster[1] und xPaulsPferd seien beides Objekte der Klasse kLebewesen, mit xMeinHamster.zAlter=12 und xPaulsPferd.zAlter=7. Ferner besitzen alle Objekte der Klasse kLebewesen jeweils die Methoden mWerdeAelter, die das Alter um 1 erhöht und mNenneAlter die Möglichkeit gibt, das Alter des jeweiligen Objektes der Klasse auszugeben.

[1] an dieser Stelle ist zu erkennen, dass objektorientierte Hamster sehr viel älter als die Hamster in freier Wildbahn werden können. Dieser Vorteil kann allerdings nicht allein der Objektorientierung zugeschrieben werden. :–)

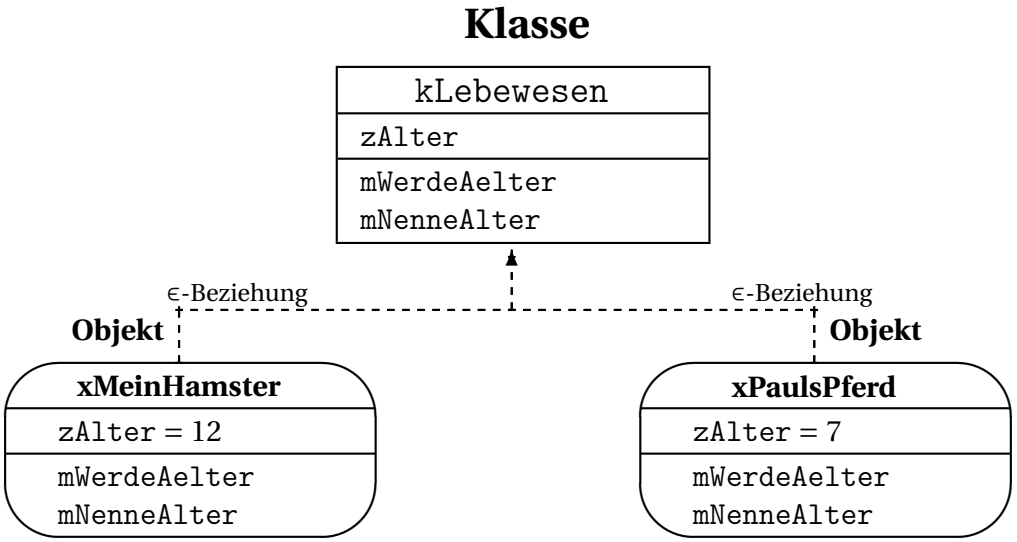


Abbildung 3.6: Methoden einer Klasse

Frage III : *Welcher Unterschied besteht zwischen einem Methodenaufruf in der OO und einem Prozedur- oder Funktionsaufruf im herkömmlichen Sinn?*

Im Gegensatz zur prozeduralen Denkweise sind bei der OO Methodenaufrufe immer an ein bestimmtes Objekt geknüpft[2] . Im obigen Beispiel ist die in der Klasse `kLebewesen` definierte Methode `mWerdeAelter` keine *globale Prozedur*, die einen Zustand `zAlter` erhöht, sondern die Verbindung mit einem Objekt macht den Methodenaufruf erst vollständig, d.h.

- `xMeinHamster.mWerdeAelter` erhöht den Zustand `xMeinHamster.zAlter` um 1 auf 13, während
- `xPaulsPferd.mWerdeAelter` den Zustand `xPaulsPferd.zAlter` auf 8 erhöht.

Damit wird deutlich, dass Methodenaufrufe immer auch zugleich Operatoren auf dem jeweiligen Zustandsraum der Objekte sind. Dies ist bei einem Prozeduraufruf nicht der Fall.

[2] In vielen Programmiersprachen kann der Objektbezeichner entfallen, wenn klar ist, dass sich der Methodenaufruf auf die klasseneigene Methode bezieht.

3.3.2 Elternklassen

$\psi(K)$ ist eine Menge von Klassen. In den meisten Fällen enthält die Menge der *Elternklasse* $\psi(K) = \{K_0\}$ genau ein Element, nämlich die *Oberklasse* (hier die Klasse K_0), aus der die Klasse K abgeleitet wurde. Damit beschreibt ψ die *Vererbungsbeziehung* zwischen den Klassen und es gilt die Teilmengenbeziehung $K \subset K_0$.

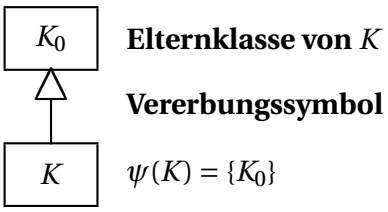


Abbildung 3.7: Vererbungsbeziehung

Beispiel 3.3.2: *(Vererbung von Zuständen und Methoden)* Wählt man beispielsweise für Klasse K_0 die oben angegebene Klasse `kLebewesen`, so erbt auch K die Zustände und Methoden von `kLebewesen`. Damit besitzen auch Objekte x der Klasse K ein Alter und die Objekte können auch älter werden und ihr Alter ausgeben.

Die folgende Abbildung zeigt Vererbungsbeziehungen zwischen den Klassen K_0 bis K_6. Jede der in der Abbildung angegebenen Klassen hat höchstens eine Elternklasse. Nur K_0 hat keine Elternklasse.

3.3.3 Basisklassen

Wegen $\psi(K_0) = \varnothing$ (=leere Menge) besitzt die Klasse K_0 keine Elternklasse. Klassen der Form $K_0 = (\varnothing, Z_0, M_0)$ heißen *Basisklassen*. Durch ψ entsteht für jede Klasse K in der Regel innerhalb einer Baumstruktur eine statische Verweiskette, die bei einer[3] Basisklasse endet.

[3] Bei Mehrfachvererbung entsteht u.U. ausgehend von der Klasse K ebenfalls eine Baumstruktur, die bei *mehreren Basisklassen* endet.

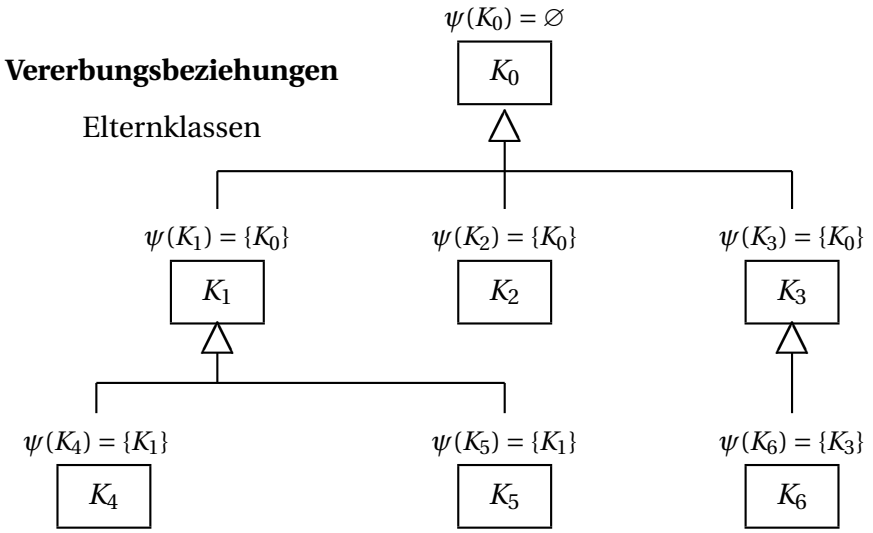


Abbildung 3.8: Vererbungsbaum

3.3.4 Mehrfachvererbung

Die für den Compilerbau nicht ganz unproblematische Eigenschaft der *Mehrfachvererbung* (d.h. eine Klasse wird von mehreren Oberklassen zugleich abgeleitet) äußert sich dadurch, dass $\psi(K)$ mehrere Elemente enthält

Beispiel 3.3.3: *(Mehrfachvererbung)* $\psi(K_3) = \{K_1, K_2\}$. Die folgende Abbildung zeigt, wie die formale Schreibweise von "Die Klasse K_3 hat zwei Elternklassen K_1 und K_2" als Diagramm dargestellt wird. Probleme können bei Zulässigkeit der

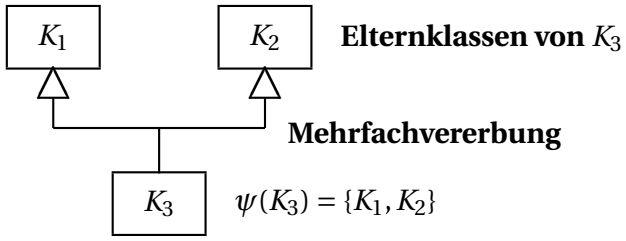


Abbildung 3.9: Mehrfachvererbung

Mehrfachvererbung entstehen, falls z.B. eine Klasse K_3 von zwei Oberklassen K_1 und K_2 die Methoden erbt, von denen sowohl $\mathcal{M}(K_1)$ als auch $\mathcal{M}(K_2)$ eine Methode mit der gleichem Namen m besitzen ($m \in \mathcal{M}(K_1) \cap \mathcal{M}(K_2) \neq \varnothing$). In diesem

Fall kann ein Objekt $x \in K_3$ eine Nachricht $x.m$ nicht eindeutig einer Methodedefinition $K_1.m$ bzw. $K_2.m$ zuordnen, welche für die Behandlung der Nachricht verwendet werden soll.

Lösungsmöglichkeiten:

- *(Mehrfachvererbung als sequentiellen Prozess)*
  Dabei liest man obiges Diagramm als Hintereinanderausführung der Vererbung von K_1 und anschließend von K_2. Tritt eine Methodendeklaration m sowohl in K_1 als auch in K_2 auf, überschreibt $K_2.m$ die Methode $K_1.m$.

- *(Spezifikation der Vererbungsklasse)*
  Ein Compiler kann bei der Übersetzung eines Programmes feststellen, ob bei der Vererbung $\psi(K_3) = \{K_1, K_2\}$ Zustände und Methoden innerhalb der Verweiskette im Vererbungsbaum identische Bezeichner generiert werden. Wird in K_3 nun auf eine nicht eindeutig zuordbaren Bezeichner verwendet, kann der Compiler eine syntaktische Spezifikation der Methoden- bzw. Zustandszugehörigkeit zu K_1 bzw. K_2 verlangen.

3.3.5 Vererbungsabbildung

Sei $\mathbb{IK} := \{K_0, K_1, \ldots K_m\}$ mit $m \in \mathbb{N}$ eine Menge von Klassen und

$$\begin{aligned} \psi: \ \mathbb{IK} &\longrightarrow \wp(\mathbb{IK}) \\ K &\longmapsto \psi(K) \end{aligned}$$

eine Abbildung, die jeder Klasse $K \in \mathbb{IK}$ eine (in der Regel einelementige) Teilmenge aus $\mathbb{IK}$ zuordnet ($\wp(\mathbb{IK})$ Potenzmenge von $\mathbb{IK}$). Durch ψ wird eine Vererbungsbeziehung auf $\mathbb{IK}$ definiert, die rekursiv auf eine Vererbungsbeziehung über mehrere Generationen von Klassen erweitert werden kann:

$$\psi^0(K) := \{K\} \quad \text{und} \quad \psi^{n+1}(K) := \bigcup_{K_{sup} \in \psi^n(K)} \psi(K_{sup}) \text{ mit } K \in \mathbb{IK}.$$

ψ heißt *Vererbungsabbildung*, falls für alle $n \in \mathbb{N}$ auch $K \notin \psi^n(K)$ gilt.

Bemerkung 3.3.4: *(zyklenfreie Vererbung)* Für die Vererbungsabbildung wird also nur verlangt, dass keine Klasse K wieder in der eigenen statischen Verweiskette (-baum) auftritt (d.h. u.a. keine Klasse K kann aus sich selbst abgeleitet werden).

$\psi^0(K)$	die Klasse K selbst
$\psi^1(K)$	die Elternklassen von K
$\psi^2(K)$	die Großeltern-Klassen von K
$\psi^3(K)$	die Urgroßeltern-Klassen von K
$\psi^4(K)$	.....

Betrachtet man die obige rekursive Darstellung der Vererbungsabbildung ψ für $n = 2$, so bestehen die Urgroßelternklassen $\psi^3(K)$ aus allen Elternklassen von $K_{sup} \in \psi^2(K)$, wobei K_{sup} also eine Großelternklasse von K ist. Zusammenfassend sind $\psi^3(K)$ die Eltern der Großeltern $\psi^2(K)$.

3.3.6 Oberklassenmenge

In der Oberklassenmenge zu einer gegebenen Klasse K werden alle Klassen zusammengefasst, die eine Oberklasse von K sind.

Sei $\mathbb{IK} := \{K_0, K_1, \ldots K_m\}$ mit $m \in \mathbb{IN}$ eine Menge von Klassen und über die rekursive Erweiterung der Vererbungsabbildung ψ können zu jeder gegebenen Klasse K die *Oberklassenmenge* $\Psi(K)$ zu K bestimmt werden:

$$\Psi(K) := \{K_{sup} \in \mathbb{IK} \mid \text{ es gibt ein } n \in \mathbb{IN}_o \text{ mit } K_{sup} \in \psi^n(K)\}$$

Bemerkung 3.3.5: *(Oberklassenmengen und Klassenbeziehungen)* Die Menge $\Psi(K)$ ist eine Teilmenge von $\mathbb{IK}$ (d.h. $\Psi(K) \in \wp(\mathbb{IK})$), die K und alle Oberklassen von K enthält. Es erscheint zunächst sinnvoller zu sein, nur die echten Oberklassen hinzuzunehmen. Wenn man allerdings über die Oberklassenmenge *Klassenbeziehungen* beschreiben möchte, so ist die Hinzunahme der Klasse K selbst zur Oberklassenmenge sinnvoll. Oberklassenmengen von zwei Basisklassen sind z.B. andernfalls leer und unterscheiden sich somit nicht. Die Teilmengenbeziehung zwischen den Oberklassenmengen gilt dann, obwohl keine Vererbungsbeziehung zwischen Basisklassen bestehen kann (sonst wären es keine Basisklassen). Aus diesem Grund gehört zu jeder Oberklassenmenge auch die Klasse selbst.

Über die Oberklassenmengen ergeben sich die folgenden Definitionen der Begriffe *Oberklasse* bzw. *Unterklasse* unmittelbar.

Oberklasse: K_1 heißt Oberklasse von K_2, falls $\Psi(K_1) \subset \Psi(K_2)$ mit $K_1 \neq K_2$ gilt.

Unterklasse: K_1 heißt Unterklasse von K_2, falls $\Psi(K_1) \supset \Psi(K_2)$ mit $K_1 \neq K_2$ gilt.

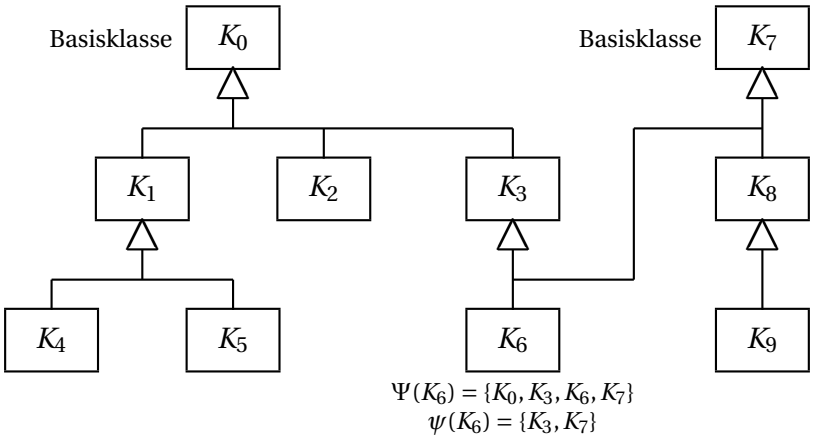


Abbildung 3.10: Baum mit Mehrfachvererbung

In der obigen Abbildung gelten folgende Beziehungen, die in der verbalen Beschreibung und der äquivalenten formalen Darstellung angegeben sind:

- K_0 ist *Oberklasse* von K_5, $\Longleftrightarrow K_0 \in \Psi(K_5)$,
  oder äquivalent dazu
  K_5 ist *Unterklasse* von K_0, $\Longleftrightarrow K_0 \in \Psi(K_5)$,
  oder äquivalent dazu
  K_5 *erbt* von der Klasse K_0, $\Longleftrightarrow K_0 \in \Psi(K_5)$,

- K_5 hat nur die Klasse K_1 als *Elternklasse,* $\Longleftrightarrow \psi(K_5) = \{K_1\}$,
  oder äquivalent dazu
  K_5 ist nur von der Klasse K_1 *abgeleitet* worden, $\Longleftrightarrow \psi(K_5) = \{K_1\}$,
- K_0 und K_7 sind *Basisklassen,* $\Longleftrightarrow \psi(K_0) = \varnothing$ und
  $\psi(K_7) = \varnothing$,
- K_6 besitzt genau *zwei Elternklassen* K_3 und K_7 $\Longleftrightarrow \psi(K_6) = \{K_3, K_7\}$.

In der folgenden Klassenhierachie sind jeweils an der Position der Klasse auch die Oberklassenmenge angegeben. Für eine Basisklasse (hier K_0) gilt $\Psi(K_0) = \{K_0\}$. Die Oberklassenmengen von Basisklassen sind immer einelementig und enthalten nur die Basisklasse selbst.

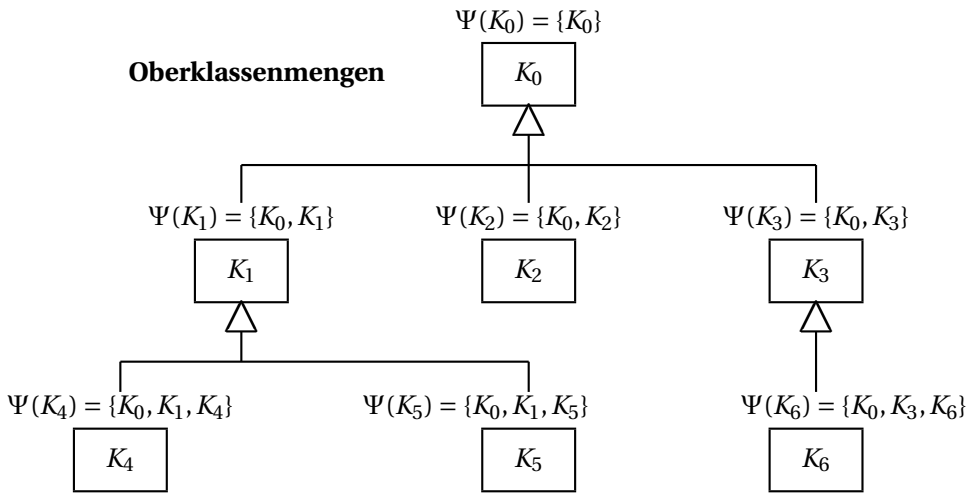


Abbildung 3.11: Oberklassenmengen

Frage IV : *Wie unterscheiden sich Elternklassen und Oberklassen?*

Für jede Klasse K gilt: $\psi(K) \subset \Psi(K)$, d.h., dass jede Elternklasse von K ist zugleich auch eine Oberklasse von K. Dies gilt aber *nicht* umgekehrt. Betrachtet man die obige Abbildung, so ist K_0 zwar eine Oberklasse von K_5 aber keine Elternklasse von K_5.

3.4 Zustände

Formal muss man zwischen drei Erscheinungsformen der Zustandsbezeichner z unterscheiden:

- z ist der Bezeichner (Variablenname) des Zustands.
- $K.z$ ist die definierte Klasse des Zustands z, die in der Regel nicht mit der Klasse K übereinstimmt.
- $x.z$ ist der Zugriff auf den Zustand z von einem Objekt x.

3.4.1 Zustandsabbildung

Für K mit $\mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$ ist $\mathcal{Z}(K)$ die Zustandsmenge der Klasse K. Die Zustandsabbildung $\mathcal{Z}$ ordnet der Klasse K die Menge der innerhalb der Klasse

definierten Zustände zu.

Beispiel 3.4.1: *(Zustandsmenge)* $\mathcal{Z}$(kLebewesen)= {zAlter}

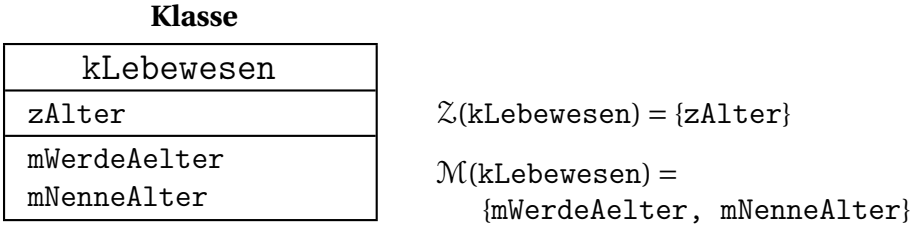


Abbildung 3.12: Zustandsmengen

3.4.2 Zustandsraum

Erzeugt man ein Objekt x der Klasse K, so verwaltet x nicht nur die Zustände, die innerhalb der Klasse definiert sind, sondern auch alle geerbten Zustände. Der Zustandsraum $\Phi(K)$ der Klasse K umfasst alle Zustände der Klasse selbst und sämtlicher Oberklassen. Formal ausgedrückt:

$$\Phi(K) := \prod_{K_{sup} \in \Psi(K)} \prod_{z \in \mathcal{Z}(K_{sup})} K_{sup}.z$$

$K_{sup}.z$ ist die Klasse des Zustands z. Durch $\prod_{z \in \mathcal{Z}(K_{sup})}$ wird das Kreuzprodukt über die in der Klasse K_{sup} definierten Zustände gebildet. $\prod_{K_{sup} \in \Psi(K)}$ liefert dann das Kreuzprodukt über alle Oberklassen von K (inklusive K selbst) gebildet. Insgesamt fasst $\Phi(K)$ alle in K definierten und durch Vererbung erhaltenen Zustandsräume $K_{sup}.z$ zu einem Kreuzprodukt zusammen.

Beispiel 3.4.2: *(Klassenhierachie)* In der folgenden Klassenhierachie ist `kLebewesen` die Basisklasse und `kSaeugetier` wird daraus abgeleitet. Von der Klasse `kSaeugetier` wird wiederum die Klasse `kPferd` abgeleitet. Rechts ist jeweils der Zustandsraum $\Phi(K)$ der jeweiligen Klasse K angegeben. In dem Diagramm wird deutlich, dass der Zustandsraum Φ der jeweiligen Klasse in Vererbungsrichtung immer mächtiger wird.

Frage V : *Wie unterscheidet sich der Zustandsraum $\Phi(K)$ einer Klasse K von der Definition eines Records?*

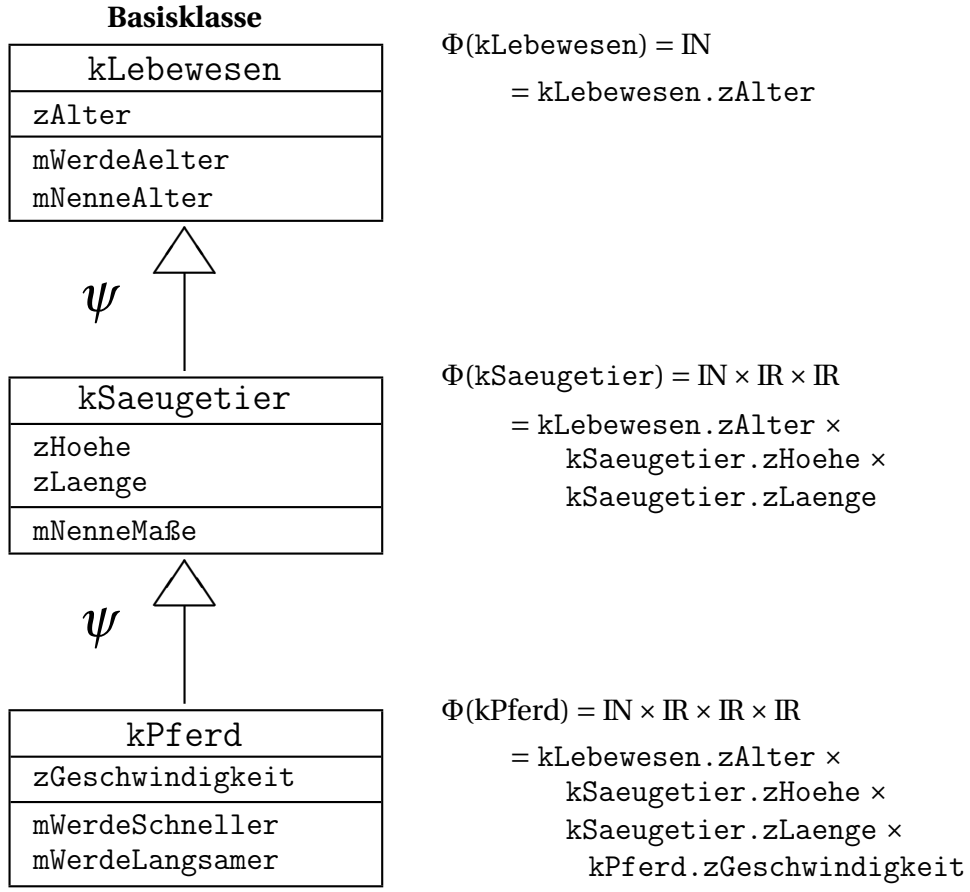


Abbildung 3.13: Klassenhierarchie

(bzw. im obigen Beispiel: Welche Zustände muss ein Objekt der Klasse `kPferd` *verwalten?)*

In der obigen Abbildung wächst der Zustandsraum Φ genau um die in der Klasse definierten Zustände. Der für ein Objekt reservierte (Speicher-)Platz hängt damit in entscheidender Weise von den geerbten Zuständen ab und nicht allein von den in der Klasse definierten Zuständen (wie z.B. bei der Definition eines *Records*).

3.5 Methoden

Sei K eine Klasse mit $\mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$. Die Abbildung $\mathcal{M}$ gibt zu der Klasse K die in der Klasse definierten Methodenbezeichner an (ohne die geerbten Methoden). Damit fasst $\mathcal{M}(K) := \{m_1, m_2, \ldots\}$ also die Menge der neu definierten bzw. überschriebenen Methoden in K zusammen.

Formal muss man zwischen drei Erscheinungsformen der Methodenbezeichner m unterscheiden:

- m ist der Bezeichner der Methode.
- $K.m$ ist die Definition der Methode m in der Klasse K.
- $x.m$ ist der Aufruf der Methode m von einem Objekt x.

Beispiel 3.5.1: *Menge der Methodenbezeichner* $\mathcal{M}$(`kLebewesen`)= {`mWerdeAelter`, `mNenneAlter`}

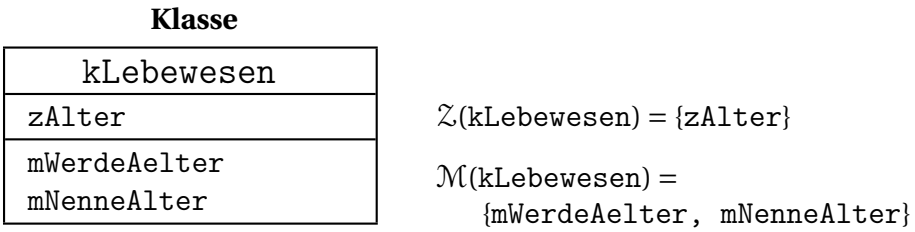


Abbildung 3.14: Methodenbezeichner

Bislang ist der Begriff Methode bereits verwendet worden, ohne die Methode als Abbildung genauer zu definieren.

Definition 3.5.2 *Sei K mit $\mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$ eine Klasse mit $\psi(K)$ als Menge der Elternklassen, $\mathcal{Z}(K)$ als Menge der Zustände und $\mathcal{M}(K)$ als die Menge der Methoden. Für jede Methode $m \in \mathcal{M}(K)$ ist eine Eingabemenge $\mathbb{IM}_{in}(m)$ und eine Ausgabemenge $\mathbb{IM}_{out}(m)$ gegeben. Die Methode m ist dann eine Abbildung der Form:*

$$m : \Phi(K) \times \mathbb{IM}_{in}(m) \longrightarrow \Phi(K) \times \mathbb{IM}_{out}(m)$$

Da jede Methode $m \in M$ ohne Ein- und Ausgabekomponenten zugleich auch ein Operator auf dem Zustandsraum $\Phi(K)$ der Klasse K ist, vereinbart man für die obige Abbildung folgende vereinfachte Notation:

$$K.m : \mathbb{IM}_{in}(m) \longrightarrow \mathbb{IM}_{out}(m)$$

Ist $\mathbb{IM}_{out}(m) = \varnothing$, *so heißt* m Auftrag.
Ist $\mathbb{IM}_{out}(m) \neq \varnothing$, *so heißt* m Anfrage.

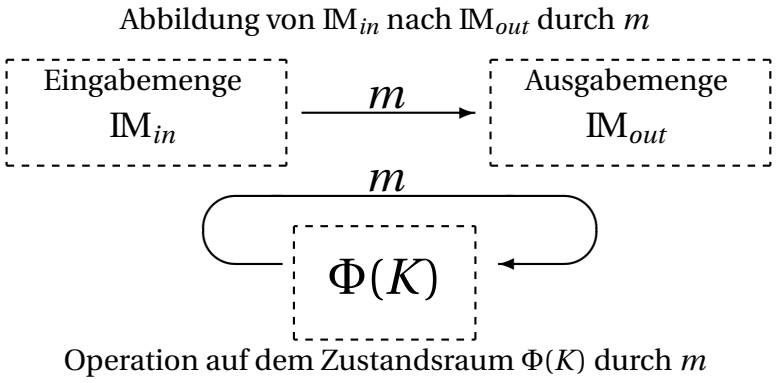


Abbildung 3.15: Ein- und Ausgaben mit Zustandsraum

Bezugnehmend auf die bereits genannte

> **Frage I:** *Welcher Unterschied besteht zwischen einem Methodenaufruf in der OO und einem Prozedur- oder Funktionsaufruf im herkömmlichen Sinn?*

kann an dieser Stelle noch einmal darauf hingewiesen werden, dass Methoden $m \in \mathcal{M}(K)$ auch ohne Ein- bzw. Ausgabekomponenten immer *Operatoren* auf dem Zustandsraum $\Phi(K)$ sind, d.h. m ist eine Abbildung der Form:

$$m : \Phi(K) \longrightarrow \Phi(K) \qquad m \text{ operiert auf der Menge } \Phi(K).$$

Allgemein heißt das: Jeder Methodenaufruf von einem Objekt (= Nachricht an das Objekt) kann beliebige Zustände aus dem Zustandsraum des Objektes modifizieren. Dies schließt natürlich (nach Definition) eine Modifikation der *geerbten* Zustände mit ein.

- Eine *Anfrage* der Klasse K ist eine *Funktion*, die zusätzlich auf dem Zustandsraum $\Phi(K)$ der Klasse K *operiert*.

$$m : \Phi(K) \times \mathbb{M}_{in} \longrightarrow \Phi(K) \times \mathbb{M}_{out} \quad \text{bzw.} \quad K.m : \mathbb{M}_{in} \longrightarrow \mathbb{M}_{out}$$

  Erfordert eine Anfrage keine Parameter zur Eingabe aus $\mathbb{M}_{in}$, gilt:

$$m : \Phi(K) \longrightarrow \Phi(K) \times \mathbb{M}_{out} \quad \text{bzw.} \quad K.m : \quad \longrightarrow \mathbb{M}_{out}$$

- Analog ist ein *Auftrag* der Klasse K eine *Prozedur*, die zusätzlich auf dem Zustandsraum $\Phi(K)$ der Klasse K *operiert*.

$$m : \Phi(K) \times \mathbb{M}_{in} \longrightarrow \Phi(K) \quad \text{bzw.} \quad K.m : \mathbb{M}_{in} \longrightarrow$$

  Erfordert ein Auftrag keine Parameter zur Eingabe aus $\mathbb{M}_{in}$, gilt:

$$m : \Phi(K) \longrightarrow \Phi(K) \quad \text{bzw.} \quad K.m : \quad \longrightarrow$$

Auf dem Zustandsraum zu operieren, bedeutet, dass jede Anfrage bzw. jeder Auftrag an ein Objekt der Klasse auch die zugehörigen Zustandsvariablen von $\Phi(K)$ modifizieren kann.

Beispiel 3.5.3: *(Auftrag und Anfrage an Objekte)* In der folgenden Abbildung sind zwei Objekte der Klasse `kLebewesen` angegeben. Die Methode `mWerdeAelter` ist eine *Auftrag*, die die Zustandsvariable `zAlter` um 1 erhöht mit $\mathbb{M}_{in}$(`mWerdeAelter`) = $\varnothing$ und $\mathbb{M}_{out}$(`mWerdeAelter`) = $\varnothing$. Die Methode `mNenneAlter` ist eine *Anfrage*, die das Alter des Objektes ausgibt und deren Eingaberaum bzw. Ausgaberaum wie folgt definiert ist:

$$(\mathbb{M}_{in}(\texttt{mNenneAlter}) = \varnothing \text{ und } \mathbb{M}_{out}(\texttt{mNenneAlter}) = \mathbb{N}.$$

Bemerkung 3.5.4: *D* ie Methoden einer Klasse $\mathcal{M}(K)$ umfassen nur die innerhalb der Klassendefinitionen deklarierten Methoden (*Aufträge* bzw. *Anfragen* an Objekte der Klasse). Aufträge und Anfragen sind die zwei verschiedene Arten von *Nachrichten*, die einem Objekt gesendet werden können. Das Objekt `xMeinHamster` versteht z.B. die Nachricht `mNenneAlter`. Durch die Nachricht wird in dem Objekt `xMeinHamster` ein Prozess ausgelöst, der zur Ausgabe des Alters. Wie der Prozess abläuft, ist durch die zugehörige Methode der jeweiligen Klasse definiert. An dieser Methodendefinition ist auch zu erkennen, dass eine Operation

Abbildung 3.16: Anfragen und Aufträge

auf dem Zustandsraum nicht notwendig eine Veränderung von Zustandsvariableninhalten nach sich zieht. Dennoch ist die mathematische Operation auf dem Zustandsraum eine fundamentale Eigenschaft einer Methodendeklaration in der OO.

Methoden sind immer auch Operatoren auf dem Zustandsraum.

3.5.1 Protokoll einer Klasse

Für eine Klasse K ist $\mathcal{P}(K) := \mathcal{M}(\Psi(K))) := \bigcup_{K_{sup} \in \Psi(K)} \mathcal{M}(K_{sup})$ das Protokoll der Klasse K. Für jede Oberklasse K_1 von K_2 gilt $\mathcal{P}(K_1) \subseteq \mathcal{P}(K_2)$.

Bemerkung 3.5.5: *Oberklassen und Überschreiben von Methoden* Auch für eine echte Oberklasse K_1 von K_2 kann $\mathcal{P}(K_1) = \mathcal{P}(K_2)$ gelten, wenn z.B. K_2 aus K_1 nur durch *Überschreiben einer Methode* hervorgegangen ist. Um diese mögliche Mengengleichheit herrauszustellen, wurde in der Definition '$\subseteq$' verwendet

Bemerkung 3.5.6: *Protokoll und Nachrichten* Das Protokoll $\mathcal{P}(K)$ einer Klasse K umfasst alle Nachrichten, die ein Objekt der Klasse K versteht. Die Definition beschreibt dies als Vereinigung über alle Methodenbezeichner in der Oberklassemenge $\Psi(K)$. Wegen $K \in \Psi(K)$ enthält das Protokoll auch die in K definierten Methodenbezeichner.

Die Methodenbezeichner $\mathcal{M}(K)$ einer Klasse K legen also *nicht allein* das Protokoll $\mathcal{P}(K)$ fest, sondern $\mathcal{P}(K)$ enthält ebenfalls die Methodenbezeichner aller Oberklassen $K_{sup} \in \Psi(K)$.

Beispiel 3.5.7: *Protokollerweiterung durch Vererbung* In dem folgenden Beispiel wird deutlich, wie durch die Vererbung das Protokoll einer Klasse immer mäch-

tiger[4] wird. Jedes Objekt der Klasse `kPferd` kann also über die geerbte Methode

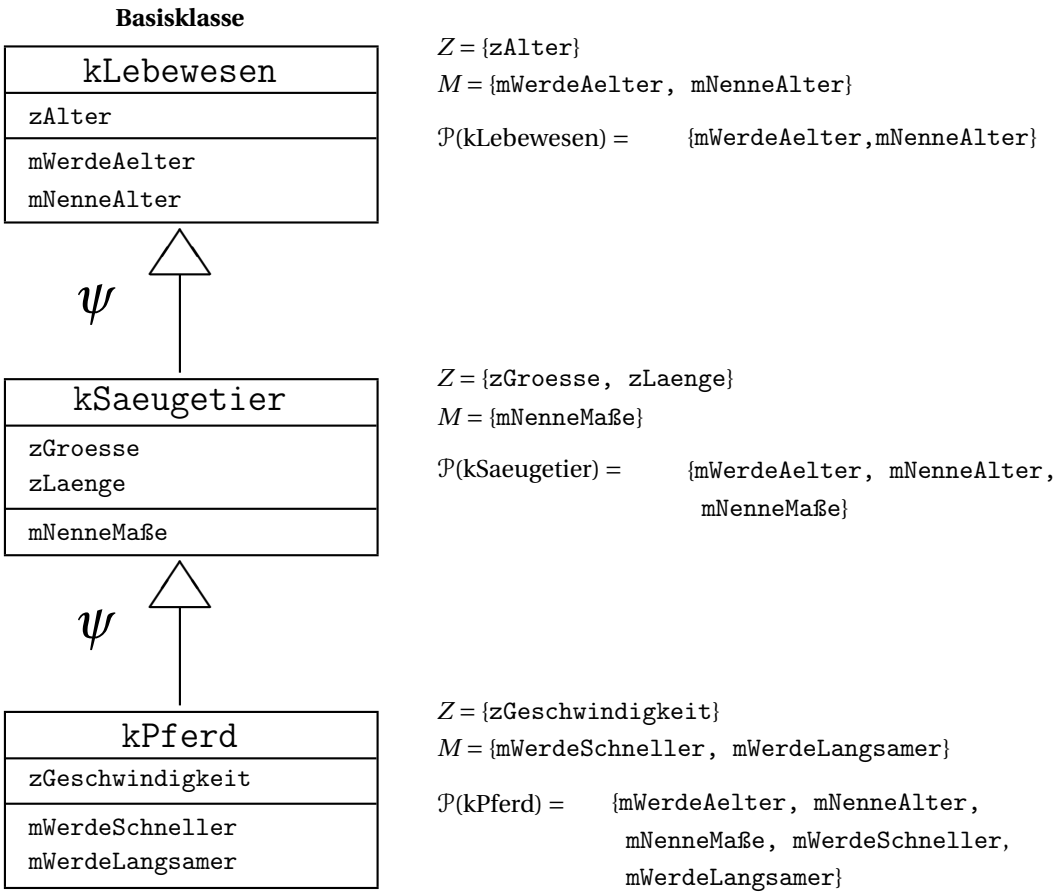


Abbildung 3.17: Protokollerweiterung durch Vererbung

`mWerdeAelter` auch älter werden. Durch die Vererbung der Zustände (siehe Definition des Zustandsraums Φ) verwaltet jedes Objekt der Klasse `kPferd` auch ein eigenes Alter `zAlter`, das durch den Aufruf der Methode `mWerdeAelter` erhöht wird.

[4] *mächtig* wird hier im mathematischen Sinn als Mächtigkeit von Mengen gebraucht (= Anzahl der Elemente einer Menge).

3.5.2 Überschreiben von Methoden

Betrachtet man eine gegebene Klasse K, so kann eine bereits in einer Elternklasse $K_{sup} \in \Psi(K)$ definierte Methode m in K erneut definiert werden.

Ein Überschreiben der Methode erfolgt formal unter folgenden Bedingungen:

$$m \in \mathcal{M}(K), \qquad m \in \mathcal{P}(\psi(K))$$

Die obigen formalen Beschreibung besagt, dass die Methode m in der Klasse K definiert ($m \in \mathcal{M}(K)$) und zugleich bereits im Protokoll der Elternklassen $\psi(K)$. Da das Protkoll $\mathcal{P}(\psi(K))$ auch alle geerbten Methodenbezeichner enthält, existiert mindestens eine[5] Klasse $K_{sup} \supset K$ mit $m \in \mathcal{M}(K_{sup})$.

Ist x ein Objekt der Klasse K_{sup}, so erzeugt die Nachricht $x.m$ einen Aufruf der Methode $K_{sup}.m$. Ist x ein Objekt der Klasse K, so erzeugt die Nachricht $x.m$ einen Aufruf der Methode $K.m$. *Ohne Überschreiben* der Methode m in K würde für $x.m$ z.B. weiterhin die Methode $K_{sup}.m$ aufgerufen.

3.5.3 Klassenbibliothek

Sei $\mathbb{IK} := \{K_0, K_1, \ldots K_m\}$ mit $m \in \mathbb{IN}$ eine Menge von Klassen und $\mathbb{IK}_{basis} \subset \mathbb{IK}$ die Menge aller Basisklassen in $\mathbb{IK}$. Die Klassenmenge $\mathbb{IK}$ heißt Klassenbibliothek, wenn eine Vererbungsabbildung

$$\psi : \mathbb{IK} \longrightarrow \wp(\mathbb{IK})$$

existiert.

Bemerkung 3.5.8: *(Vollständigkeit einer Klassenbibliothek)* Eine Klassenbibliothek muss also im Gegensatz zu einem Programmbibliotheken zu jeder Klasse K auch *alle zugehörigen Oberklassen* enthalten.

Da Klassenbibliotheken endliche Mengen[6] sind und die Vererbungsabblidung keine Zyklen enthalten darf ($K \notin \psi^n(K)$ für alle $n \in \mathbb{IN}$), ergibt sich unmittelbar:

[5] $K.m$ kann bei Mehrfachvererbung gleichzeitig mehr als eine Methodendeklaration ($K_1.m$ und $K_2.m$) überschreiben.

[6] Beispiel: $\mathbb{IK} := \{\mathbb{IN}, \mathbb{Q}, \mathbb{IR}\}$ ist eine endliche Menge mit 3 Elementen. Die Elemente der Menge sind selbst wieder Mengen, die unendlich viele Elemente besitzen.

(i) Jede Klassenbibliothek IK enthält zumindest eine Basisklasse[7], d.h. $IK_{basis} \neq \varnothing$) und

(ii) alle Klassen $K \in IK$ sind entweder Basisklassen oder sie können über maximal $m-1$ Vererbungsschritte mit ψ aus Basisklassen abgeleitet werden.

3.5.4 Zusammenfassung Klassen und Objekt

Jede maschinenunabhängige Formulierung einer Klasse K mit $\mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$ ist durch die Menge der Elternklassen $\psi(K)$, die Menge der *Zustände* $\mathcal{Z}(K)$ und die Menge der *Methoden* $\mathcal{M}(K)$ bestimmt.

In einer Klasse K werden Objekte zusammengefasst, die eine gewisse Minimalanforderung im Bereich der *Zustandsverwaltung* und der *Funktionalität* erfüllen.

Diese Minimalanforderung im Bereich der *Zustandsverwaltung* wird durch die klasseneigene Menge der Zustände $\mathcal{Z}(K)$ und durch die über die Elternklassen $\psi(K)$ geerbten Zustandsmengen festgelegt. Die klasseneigenen und die geerbten Zustände werden in dem *Zustandsraum* $\Phi(K)$ zusammengefasst.

Analog wird die Minimalanforderung im Bereich der *Funktionalität* durch die klasseneigene Menge der Methoden $\mathcal{M}(K)$ und durch die über die Elternklassen $\psi(K)$ geerbten Methoden festgelegt. Die klasseneigenen und die geerbten Methoden werden durch das *Protokoll* $\mathcal{P}(K)$ zusammengefasst, die alle Nachrichten enthält, die Objekte $x \in K$ „verstehen“ können.

3.6 Lebenszyklus von Objekten

Realisierungen von Klassen heißen *Objekte* (= Instanzen, Exemplare). Die Existenz von Objekten ist von einem Adressraum /A abhängig. Der Adressraum dient dazu, die verschiedenen Objekte in einer Klasse unterscheidbar zu machen. Dies ist eine grundlegende Voraussetzung dafür, dass einzelnen Objekten eine Nachricht gesendet werden kann.

Nur existierende Objekte können Nachrichten versenden

[7] Basisklassen K_0 haben die Eigenschaft $\psi(K_0) = \varnothing$.

Beispiel 3.6.1: *(Lebenszyklus und Kommunikation zwischen Objekten)* Ein Lebenszyklus von Objekten wird als zeitabhängige dynamische Veränderung des Zustandsraumes durch Nachrichten zwischen Objekten betrachtet. Betrachtet man Albert Einstein als Element der Klasse Mensch, so kann eine Kommunikation mit anderen Objekten nur dann einem objektorientierten Modell realisiert werden, wenn das Objekt zu einem Zeitpunkt existiert bzw. der Lebenzyklus noch nicht beendet ist.

Das folgende Beispiel zeigt, dass der Begriff des *Adressraumes* dabei nicht allein im Kontext Hardware zu verstehen ist.

Beispiel 3.6.2: *(Haus)* Fasst man eine Klasse K wie eine Konstruktionsbeschreibung oder auch als "Bauplan“ für Objekte auf, der Eigenschaften festlegt, die alle Objekte x_i der Klasse K besitzen, so kann man z.B. für die nach der Klasse `kHaus` „gebauten“ Objekte `xHaus1, xHaus2, xHaus3` die Menge von Hausnummer als Adressraum auffassen. Auch die Objektbezeichnungen `xHaus1, xHaus2, xHaus3` oder auch zusammengesetzte Merkmale der Form:

(Land, PLZ, Straße, Hausnummer)

bilden Adressen, die die Objekte der Klasse unterscheidbar machen.

3.6.1 Objekte im Adressraum

Objekte können verschiedene Bezeichnungen haben und dennoch auf das gleiche Objekt verweisen (z.B. $x = 5$ und $y = 5$). Die Unterscheidbarkeit von Objekten wird in der OO formal durch eine Adresse $\alpha(x)$ eines Objektes x gewährleistet. Bei einem Objekt $x \in K$ muss man daher zwischen x und seiner Adresse $\alpha(x)$ unterscheiden. Jeder Objektvariablen x ist *genau eine* Adresse $\alpha(x)$ zugeordnet, wenn das Objekt X existiert. Falls ein Objekt nicht existiert, gilt $\alpha(x) = \varnothing$. Das Symbol $\varnothing$ ist dabei in Analogie zur Potenzmenge als Element einer Menge zu verstehen. Diese Menge $I\!A \cup \{\varnothing\}$ ist der Wertebereich der Abbildung α, der aus einem Adressraum $I\!A$ und dem Element $\varnothing$ besteht.

$$\begin{array}{rcl} \alpha\ V & \longrightarrow & I\!A \cup \{\varnothing\} \\ x & \longmapsto & \alpha(x) \end{array}$$

V ist dabei Raum von Objektvariablen, die als Definitionsbereich von α verwendet wird.

Bemerkung 3.6.3: *(Undefinierte Variablen)* $\alpha(x) = \varnothing$ findet sich in Programmiersprachen z.B. als `x=NIL` oder `x=null`.

Beispiel 3.6.4: *(Unterscheidbarkeit von Objekten)* Erst durch eine Adresse werden Objekte unterscheidbar. `xMeinHaus` und `xBertsHaus` verweisen auf die gleiche Adresse, wenn

$$\alpha(\texttt{xMeinHaus}) = \alpha(\texttt{xBertsHaus})$$

gilt. Besitzen die Bezeichner die gleiche Adresse, dann verweisen die Objektvariablen `xMeinHaus` und `xBertsHaus` das gleiche Haus. Unterscheiden sich die Adressen, dann handelt es sich um unterschiedliche Objekte. In einem solchen Fall können dennoch alle Variableninhalte der Zustandsvariablen von beiden Objekten identisch sein.

Bemerkung 3.6.5: *(Adressraum als Menge)* Ein Adressraum, aus dem die Adressen $\alpha(x)$ stammen, ist zunächst einmal nicht eingeschränkt. $\alpha(x)$ könnte eine Adresse eines Hauses oder auch eine Adresse im Hauptspeicher eines Rechners sein. In der folgenden Definition können die Klassen auf einen bestimmten Adressraum eingeschränkt werden.

Definition 3.6.6 (Objekte im Adressraum) *Sei* $\mathbb{K}$ *eine Klassenbibliothek,* $K \in \mathbb{K}$ *eine Klasse und* $\mathbb{A}$ *ein Adressraum, dann definiert* $K(\mathbb{A})$ *alle in* $\mathbb{A}$ *realisierten Objekte der Klasse* K*.*

$$K(\mathbb{A}) := \{x \in K \mid \alpha(x) \in \mathbb{A}\}$$

$K(\mathbb{A})$ schränkt die Objekte von K auf den Adressraum $\mathbb{A}$ ein. Daher gilt $K(\mathbb{A}) \subset K$ für alle $K \in \mathbb{K}$. Ferner darf man den Adressraum nur bedingt als Speicheradresse in einem Hauptspeicher verstehen. Im Hauptspeicher kann man den freie Speicherplatz wieder mit einem anderen Objekt belegen und damit wären unterrschiedliche Objekt durch Wiederverwertung von Speicheradresse über die Adresse selbst nicht mehr unterscheidbar. Diese Unterscheidung ist aber für Identifikation von Objekten entlang der Zeitachse T zwingend notwendig, denn alle andere Zustandsvariablen im Zustandsraum können sich in Zeit verändern.

Damit erhält man für Oberklassen $K_{sup} \supset K$ von K auch die Teilmengenbeziehung $K_{sup}(\mathbb{A}) \supset K(\mathbb{A})$. Das bedeutet also: Je spezialisierter die Klasse ist, desto

weniger Elemente (= Objekte) enthält sie.

3.6.2 Zeitabhängige Existenz von Objekten

Der Philosoph *Sokrates* gehört sicherlich zu der Klasse `kLebewesen`. Zum jetzigen Zeitpunkt lebt Sokrates allerdings nicht mehr („ein Objekt existiert nicht mehr"). Diese Zeitabhängigkeit der Existenz von Objekten wird nun formalisiert.

3.6.2.1 Zeitmenge T

Um die Zeitabhängigkeit auch formal beschreiben zu können, wird eine Zeitmenge T benötigt. Da in der Objektorientierung jede „Objektaktivität" (Kommunikation) als Abfolge von einzelnen Nachrichten aufgefasst wird, ist eine *Diskretisierung* der Zeitmenge auf $T = \mathbb{N}_0$ eine sinvolle Vereinfachung.

3.6.2.2 Laufzeitklassen – Objekte zum Zeitpunkt $t \in T$

Über die Zeitmenge T wird formal zwischen einer allgemeinen Klasse K und den zum Zeitpunkt $t \in T$ existierenden Objekten dieser Klasse K unterschieden (Notation: K_t). K_t wird als *Laufzeitklasse* von K bezeichnet.

$K_t(\mathbb{A})$ bezeichnet dann alle zum Zeitpunkt $t \in T$ realisierten bzw. existierende Objekte im Adressraum $\mathbb{A}$. Da Objekte dynamisch erzeugt und wieder freigegeben werden, ist diese Laufzeitklasse von dem Parameter t abhängig. Die Abhängigkeit der Veränderung von einzelnen *Nachrichten* lässt eine diskrete Zeitmenge $\mathbb{N}_0$ zu. $t = 0$ bestimmt z.B. den Start eines Programmes. Ist $x \in K_t(\mathbb{A})$, so ist also x ein Objekt der Klasse K im Adressraum $\mathbb{A}$, das zum Zeitpunkt t existiert. K_t bzw. $K_t(\mathbb{A})$ sind Mengen von Objekten, deren Zustandsraum sich während des Programmablaufes verändert. Ferner besitzten Laufzeitklassen im Gegensatz zu Klassen immer nur *endlich viele Elemente*.

Definition 3.6.7 (Laufzeitklasse) *Sei $\mathbb{A} \neq \emptyset$ ein Adressraum. $K_t(\mathbb{A})$ heißt Laufzeitklasse in der Zeitmenge T, falls $K_t(\mathbb{A}) \subset K$, $|K_t(\mathbb{A})| < \infty$ und $t \in T$ erfüllt ist.*

Ist der Adressraum $\mathbb{A}$ für die Laufzeitklasse $K_t(\mathbb{A})$ geklärt kann die Bezeichnung ggf. auch K_t ersetzt werden.

Um auch alle dynamischen Aspekte in den Objekten formal darstellen zu können, müssen auch die Zuständsveränderung in Zeit $t \in T$ festgehalten werden.

Definition 3.6.8 (Objekthistorie von Objekten) *Sei $\mathbb{A} \neq \emptyset$ ein gegebener Adressraum und $K_t(\mathbb{A})$ die Laufzeitklassen in der Zeitmenge T. $(x_t)_{t\in T}$ heißt Objekthistorie, wenn für alle $t, \widehat{t} \in T$ gilt:*

$$\alpha(x_t) \neq \emptyset \quad \alpha(x_{\widehat{t}}) \neq \emptyset \Longrightarrow \alpha(x_t) = \alpha(x_{\widehat{t}}),$$

Die obige Definition verlangt also von einer Zustandsraumhistorie, dass existierende Objekt x_t aus $(x_t)_{t\in T}$ in einer Objekthistorie die gleiche Adresse besitzen müssen. Damit ist es möglich, die dynamischen Veränderungen durch $(x_t)_{t\in T}$ in der Zeit formal zu beschreiben.

Im Folgenden werden die Laufzeitklassen ausschließlich dazu verwendet die Prozesse der Erzeugung und der Freigabe (Löschen) von Objekten formal zu beschreiben. $K_t(\mathbb{A})$ sind also alle zum Zeitpunkt t existierenden Objekte der Klasse K in dem Adressraum $\mathbb{A}$.

Bemerkung 3.6.9: *(Garbage Collection - Müllsammlung für Objekte)* Beim Verlassen eines Programms ist es in der Regel[8] notwendig, alle während des Programmablaufs erzeugten Objekte wieder freizugeben. Laufzeitklassen können dazu verwendet werden, alle erzeugten Objekte automatisiert wieder freizugeben. Aber auch eine automatisierte Freigabe von Objekten, an die keine Nachrichten mehr versendet werden können, benötigen Laufzeitklassen zur Überprüfung, ob ein bestimmtes Objekt $x \in K_t(\mathbb{A})$ von keinem anderen Objekt zu Zeitpunkt $t \in T$ mehr referenziert wird. Die Garbage Collection in JAVA realisiert z.B. diese automatisierten *Aufräumarbeiten* im Speicher. Die Behandlung von Laufzeitklassen bietet daher einen Ansatzpunkt, um sich mit Mechanismen dieser Aufräumarbeiten auseinanderzusetzen *(Anküpfungspunkt: Laufzeitsystem)*.

> **Frage VI :** *Wenn in der OO alle Prozesse in Objekten ablaufen, lassen sich dann Befehle wie* `new` ***kClass()*** *nicht mehr in das Grundkonzept der OO einordnen?*

Für die Erzeugung eines Objektes ist das noch kein Problem, wenn man davon ausgeht, dass die Erzeugung immer von einem existierenden Objekt ausgelöst

[8] Ausnahme sind speicherresistente Komponenten eines Programms

wird und jedem Objekt die Möglichkeit durch Vererbung auf den Adressraum zugreifen zu können. Für die Freigabe von Objekten ist das allerdings ein logisches Problem, denn die Freigabe ist ein Prozess, den ein gelöschtes Objekt nicht mehr die eigene Freigabe beenden kann. Wenn es existiert kann sichdiese Aufgabe der nicht an ein anderes Objekt

Der Objektmanager ist ein Objekt der einen bestimmten Adressraum /A verwaltet und für die Erzeugung und die Freigabe von Objekten zuständig ist. Objektorientiert kann man einen Objektmanager als Teilobjekt eines Laufzeitsystems betrachten, deren Formalisierung für eine exakte Beschreibung von Veränderungen im Bereich der Existenz von Objekten notwendig ist. Für die Modellbildung hat

3.6.2.3 Objektmanager

Um einen Programmablauf zu regeln, muss es mindestens ein Objekt geben, das die *„freien Bauplätze“* und die *„Baupläne“* der Klassen $K \in \mathbb{K}$ kennt. Ein solches Objekt wird im Folgenden *Objektmanager* genannt. Ein Objektmanager hat die Aufgabe, Objekte einer bestimmten Klasse zu erzeugen und diese bei Bedarf wieder freizugeben.

- *Erzeugung* eines Objektes $x \in K$ zum Zeitpunkt t_o bedeutet für eine Zustandshistorie $(x_t)_{t \in T}$:

  $$x_t = \varnothing \text{ für } t < t_o \text{ und } x_t \in K_t \text{ für } t + s > t \geq t_o$$

  s ist dabei die Lebensspanne des Objektes.

- *Freigabe* (Löschen) eines Objektes $x \in K$ zum Zeitpunkt t bedeutet für eine Zustandshistorie $(x_t)_{t \in T}$:

  $$x_t \in K_t \text{ für } t - s \geq t < t_o \text{ und } x = \varnothing \text{ für } t \geq t_o x \in K_t \text{ und } x \notin K_{t+1}$$

  s ist dabei wieder die Lebensspanne des Objektes.

Bemerkung 3.6.10: *(Javascript)* In dem JavaScript-Beispiel im Abschnitt 3.7 (siehe 122ff.) wird gezeigt, wie ein Bauplan in einer objektbasierten Programmiersprache wie JavaScript realisiert werden kann und eine Klassendefinition mit Vererbung auch in JavaScript möglich wird.

Definition 3.6.11 (Konstruktor) *Sei* $\mathbb{K}$ *eine Klassenbibliothek, dann ist ein Konstruktor eine Methode des Objektmanagers der Form:*

$$\begin{array}{rcl} \texttt{kObjektManager.mKonstruktor} : \mathbb{K} & \longrightarrow & \bigcup_{K_{sup} \in \mathbb{K}} K_{sup} \\ K & \longmapsto & x \text{ mit } x \in \mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K)) \end{array}$$

Für alle Aufrufe `xObjektManager.mKonstruktor`(K)*, die ein* $x \in K$ *zu einem Zeitpunkt* $t \in T$ *erzeugen, gilt:* $x \notin K_t$ *und* $x \in K_{t+1}$

In der Definition steht die Bedingung $x \in \mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$ und nicht $x \in K$, da mit $x \in K$ auch ein Objekt einer abgeleiteten Klasse $K_{sup} \subset K$ erzeugt werden kann, das einen größeren Zustandsraum bzw. eine größere Funktionalität als die durch $\mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$ definierten Eigenschaften besitzt.

Ein Objektmanager kann neben der Objekterzeugung auch eine Methode zur Freigabe zur Verfügung stellen. Falls eine Objekt nicht mehr gebraucht wird, kann der Objektmanager dann auch den belegten Platz wieder freigeben. Nach dem Aufruf der Methode existiert das Objekt in dem jeweiligen Adressraum nicht mehr.

Definition 3.6.12 (Destruktor) *Sei* $\mathbb{K}$ *eine Klassenbibliothek, dann ist ein Destruktor eine Methode des Objektmanagers der Form:*

$$\begin{array}{rcl} \texttt{kObjektmanager.mDestruktor} : \bigcup_{K \in \mathbb{K}} K & \longrightarrow & \varnothing \\ x & \longmapsto & \end{array}$$

Für alle Aufrufe `xObjektManager.mDestruktor`(x) *mit* $x \in \mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$ *zu einem beliebigen Zeitpunkt* $t \in T$ *gilt:* $x \in K_t$ *und* $x \notin K_{t+1}$

Die Abbildungsdefinition der Methode `mDestruktor` erscheint ungewöhnlich zu sein, aber auch hier muss wieder berücksichtigt werden, dass jede Methode zugleich auch ein Operator auf dem Zustandsraum Φ der Klasse `kObjektmanager` ist. Also bezeichnet `mDestruktor` eigentlich vollständig formuliert folgende Abbildung:

$$\begin{array}{rcl} \texttt{kObjektmanager.mDestruktor} : \Phi(\texttt{kObjektmanager}) \times \bigcup_{K \in \mathbb{K}} K & \longrightarrow & \Phi(\texttt{kObjektmanager}) \\ \Phi(\texttt{xObjektmanager}) \times x & \longmapsto & \Phi(\texttt{xObjektmanager}) \end{array}$$

Das gleiche gilt natürlich auch für die Methode `mKonstruktor`.

Jedes Objekt x kann einem Objektmanager den Auftrag geben, ein neues Objekt der Klasse K zu erzeugen (Konstruktor), bzw. den durch ein Objekt

$$x \in \mathcal{K}(\psi(K), \mathcal{Z}(K), \mathcal{M}(K))$$

belegten (Speicher-)Platz wieder freizugeben (Destruktor).

Bemerkung 3.6.13: *(Freigabe und Wirkung auf Teilobjekte)* Objekte $x \in K$ besitzen u.U. eigene Freigabemethoden (z.B *x.mDestruktor*), die bezogen auf Teilobjekte von x zunächst die Freigabemethoden für die Teilobjekte aufruft und erst anschließend für die Freigabe von sich selbst sorgt. Die Aufräumarbeiten selbst kann x aber nicht selbst übernehmen, da x nach seiner Freigabe nicht mehr existiert. Daher unterscheiden sich die in Programmiersprachen auftretenden Destruktoren von dem Destruktor eines Objektmanagers. Die Aufräumarbeiten können nämlich von *x.mDestruktor* nicht durchgeführt werden, da es nach der Freigabe nicht mehr existiert. *x.mDestruktor* erteilt also dem Objektmanger den Auftrag zur Freigabe des eigenen Objektes x.

Aufruf: `xObjektmanager.mDestruktor(`x`)`

Der Objektmanager muss dann lediglich warten, bis alle in x ablaufenden Methoden beendet sind. Aber auch die Objekterzeugung kann genauso wie die Freigabe nicht von dem Objekt x selbst übernommen werden, denn ein Methodenaufruf *x.mKonstruktor* kann nicht ausgeführt werden, da x zum Zeitpunkt des Aufrufs noch nicht existiert.

Bemerkung 3.6.14: *(Laufzeitsystem)* Bei der Behandlung des Objektmanagers bietet sich an, die Bezüge zu den Aufgaben eines Laufzeitsystems herzustellen.

Bemerkung 3.6.15: *(Hierarchie von Objektmanagern)* Ein Objektmanager xOM_1 ist ein Objekt einer bestimmten Klasse kOM_1. Auch dieses Objekt muss erzeugt werden. Technisch können Objektmanager xOM_1 wieder von einem umfassenderen Objektmanager xOM_2 erzeugt werden, der xOM_1 mit der Verwaltung eines Teiladreßraumes $\mathbb{A}_1$ beauftragt hat. xOM_2 verwaltet dann z.B. einen größeren Adressraum $\mathbb{A}_2 \supset \mathbb{A}_1$. Dadurch entsteht eine Hierarchie von Objektmanagern, bei der man in letzter Konsequenz nach dem Objektmanager des Universums fragen müsste. Diese metaphysischen Aspekte sind aber *nicht* Bestandteil der OO, denn die OO ist zwar eine Problemlösestrategie, aber sie kann nicht jedes Problem lösen! :-)

3.6.2.4 Aggregation - Teilobjektbeziehung

Jedes Objekt besitzt die *Verantwortung* für die Objekte, die es erzeugt. Zwischen dem Objekterzeuger und dem erzeugten Objekt besteht dann eine *Teilobjektbeziehung* . Damit ein Objekt x mit seinen Teilen kommunizieren kann, ist es notwendig, dass x Zustandsvariablen besitzt, die auf die Teilobjekte veweisen. Auf die folgende Abbildung bezogen heißt das: Es existieren Zustandsvariablen $x.z_1, x.z_2, x.z_3$ von x mit $\alpha(x.z_i) = \alpha(x_i)$ für $i = 1,2,3$. Ferner ist jedem erzeugten

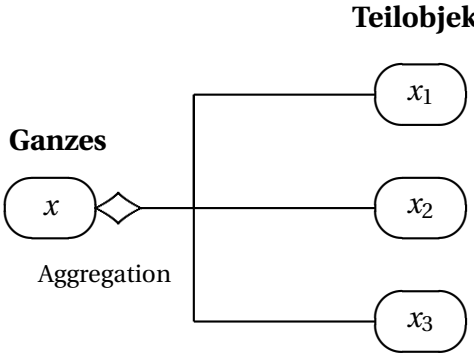


Abbildung 3.18: Aggregation

Objekte x damit auch eine *Laufzeitklasse* $K_t(x) = \{x_1, x_2, x_3\}$ zugeordnet, die alle Teilobjekte von x verwaltet. Wird das Objekt $x \in \mathbb{K}$ wieder freigegeben, so muss x dafür sorgen, dass seine Teilobjekte ebenfalls freigegeben werden.

Bemerkung 3.6.16: *(Laufzeitklasse)* Die Laufzeitklasse $K_t(x)$ muss nicht *als Menge* explizit im Programmcode implementiert sein, sondern in der Regel sind diese Beziehungen in den Freigabemethoden versteckt, sofern keine automatisierten Freigaberoutinen im Laufzeitsystem implementiert wurden (Garbage Collection). Gibt es mehrere Besitzer (y_1 und y_2), so teilen sich die Besitzer eine Laufzeitklasse $K_t(y_1, y_2) = \{x_1, x_2, x_3\}$, die alle Objekt verwaltet, die zugleich zu y_1 und y_2 gehören. Zu jedem Zeitpunkt $t \in T$ können

- weitere Besitzer hinzugefügt $K_{t+1}(y_1, y_2, y_3) = \{x_1, x_2, x_3\}$,
- ein Besitzer entfernt $K_{t+1}(y_2) = \{x_1, x_2, x_3\}$,
- weitere Teilobjekte hinzugefügt $K_{t+1}(y_1, y_2) = \{x_1, x_2, x_3, x_4\}$ oder
- ein Teilobjekt entfernt $K_{t+1}(y_1, y_2) = \{x_1, x_3\}$ werden.

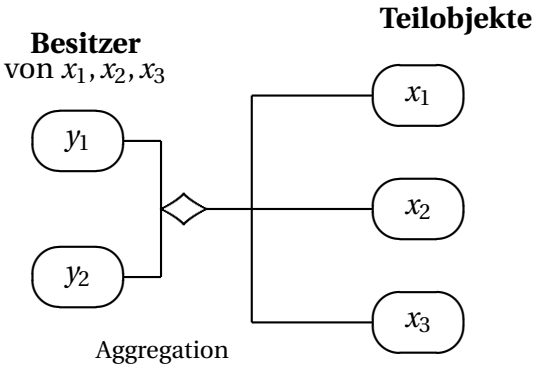


Abbildung 3.19: Aggregation mit zwei Besitzern

Erst wenn es keine Besitzer mehr gibt, können alle Teilobjekte x_1, x_2, x_3 entfernt werden. Daher gilt $K_t() = \varnothing$ für alle $t \in T$.

3.6.2.5 Assoziation

Die Assoziation als Beziehung zwischen Objekten dient dazu, dass zwischen x und y Nachrichten ausgetauscht werden können. Ist diese *„Kenntbeziehung“* (=Assoziation)

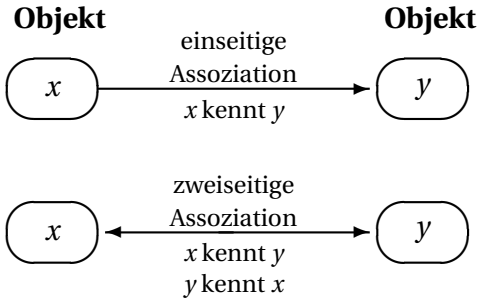


Abbildung 3.20: Assoziation

zwischen x und y nur einseitig (siehe Abbildung), so kann nur x eine Nachricht an y versenden, aber nicht umgekehrt. Ein Objekt x „kennt“ ein Objekt y (=ist assoziiert mit), wenn es einen Zustand $x.z$ besitzt, mit $\alpha(x.z) = \alpha(y)$.

Eine Freigabe eines der beiden assoziierten Objekte x bzw. y hat im Gegensatz zur Teilobjektbeziehung keinen Einfluss auf die Existenz des assoziierten Objektes.

Es ist lediglich auf die Konsistenz der Objektadressen zu achten, denn nach einer

Freigabe von y muss auch $\alpha(x.z) = \varnothing$ gelten.

Bemerkung 3.6.17: *(Assoziation und Aggregation)* Es gibt ausgearbeitetere objektorientierte Modelle, bei denen z.B. für eine Assoziation noch ein Verknüpfungsobjekt zwischen den assoziierten Objekten zu finden ist. Diese Verfeinerungen sind aber lediglich *Spezialfälle* der hier vorgestellten formalen Konzeption. Wesentlich ist dabei lediglich das (lauf-)zeitabhängige Verständnis dieser *Objekt*-Beziehungen über Laufzeitklassen und Adressen.

3.7 Objektorientierte Programmierung mit JavaScript

Überträgt man die obigen formalen Beschreibungen in eine konkrete Programmiersprache, so kann man die Modellbildung auch innerhalb von Sprachen verwenden, die dieses objektorientierte Modell nicht vollständig unterstützten. In JavaScript können zunächst zwar Objekte erzeugt und Methoden von Klassen definiert werden. Die Vererbung von Klassen wird aber von der Syntax nicht unterstützt.

Die *Wiederverwertung* und *Erweiterung* bestehender Klassen ist aber ein wesentlicher Vorteil der objektorientierten Modellbildung. Die reine Verwendung einer objektorientierten Programmiersprache garantiert dabei natürlich noch nicht die Umsetzung der objektorientierten Grundidee (Hauptfehler: „Eine Klasse mit allen Methoden und Zuständen“).

3.7.1 Zustände und Methoden in JavaScript

Die folgende Datei ist eine JavaScript-Datei, die in der Regel durch einen Tag der Form

```
<script language=javascript src="klebewesen.js"></script>
```

in eine HTML-Datei eingebunden wird. Nach der Einbindung stehen die Funktionen und objektorientierten Definitionen zur Verfügung.

```
//=============================================================================
//
//  Klasse: kLebewesen         Datei: klebewesen.js
//  ======
```

```
//
//  Erzeugung:                  xPaulsPferd        = new kLebewesen();
//  Zugriff auf einen Zustand: xPaulsPferd.zAlter = 5;
//  Aufruf einer Methode:      xPaulsPferd.mWerdeAelter();
//                              xPaulsPferd.mNenneAelter();
//                              xPaulsPferd.mSetzeAlter(18);
//=============================================================================
// Konstruktor der Klasse kLebewesen
//=============================================================================

function kLebewesen() {
  //---Zustaende (Attribute)----
        this.zAlter = 0;
  //---Methoden-----------------
        this.mWerdeAelter = mWerdeAelter_kLebewesen;
        this.mNenneAlter  = mNenneAlter_kLebewesen;;
        this.mSetzeAlter  = mSetzeAlter_kLebewesen;;
  //---RETURN---neues Objekt----
        return this
}

//=============================================================================
// Implementation der Methoden der Klasse kLebewesen
//=============================================================================
function mWerdeAelter_kLebewesen;() {
        this.zAlter++;
}
//=============================================================================
function mNenneAlter_kLebewesen;() {
        return this.zAlter;
}
//=============================================================================
function mSetzeAlter_kLebewesen;(pAlter) {
        this.zAlter = pAlter;
}
//=============================================================================
```

3.7.2 Integration der Vererbung

Nun kann auf dem oben gezeigten Weg keine Vererbung über syntaktische Möglichkeiten direkt realisiert werden, da die Konstruktoren der Klassen eine feste Anzahl von Methoden und Zuständen definieren. Da Konstruktoren in Javascript

Funktionsdefinitionen zur Objekterzeugung darstellen, können diese sowohl als Funktion als auch als Variable angesprochen werden.

Wenn `x` ein Objekt der Klasse `kLebewesen` ist, dann wird

- ohne Klammern `x.mWerdeAelter` als Variable angesprochen und
- mit Klammern `x.mWerdeAelter()` wird die Methode `mWerdeAelter` in dem Objekt `x` aufgerufen

Beide Aspekte der Nutzung der Funktionen sind notwendig, um in den Konstruktoren die Vererbung aus den Elternklasse verschachtelt zu realisieren. Eine mögliche Realisierung, die Vererbung in die JavaScript-Syntax zu integrieren, zeigt das folgende Beispiel. Dabei wird der Konstruktor der Oberklasse zunächst als Variable interpretiert und der Variable `Super` zugewiesen und anschließend wird dieser Konstruktor der Objektklasse `Super()` ausgeführt. Durch diese Ausführung werden Methoden und Zustände der Objektklasse vererbt. Wenn eine neue Klasse `kLebewesenMitNamen` von einer Oberklasse `KLebewesen` abgeleitet werden soll und damit die Eigenschaften von der Oberklasse erbt, dann wird in dem Konstruktor in JavaScript

- ohne Klammern `this.Super` als Variable angesprochen. Die folgende Zeile

  ```
  this.Super = KLebewesen
  ```

  definiert die Klasse `KLebewesen` als Oberklasse und
- mit Klammern `this.Super()` wird der Konstruktor der Oberklasse `KLebewesen` in dem Konstruktor von `KLebewesenMitNamen` aufgerufen. Erst dieser Aufruf sorgt für die Vererbung von Eigenschaften.

3.7.3 Überschreiben von Methoden

Dieses Vorgehen bei der Umsetzung der Vererbung in JavaScript liefert dadurch unmittelbar die Möglichkeit des Überschreibens von Methoden, da die Methoden wieder duch den Variablenaspekt durch eine andere Funktion ersetzt werden kann, da in dem Konstruktor ein Verweis auf die zugehörige `function` verändert wird. Damit wird eine Methode nicht nur in einem Konstruktor überschreibbar, sondern zu jeder Zeit im Programmablauf kann man individuell für einzelne

Objekte den Verweis eines Methodenbezeichners auf eine Methodedefinition dynamisch ändern.

Das Überschreiben von Methoden ist in JavaScript sogar nach Erzeugung eines Objektes möglich (Dateiende im Beispiel). Die angegebene HTML-Datei ist eine lauffähige JavaScript-Version, die bei der Interpretation durch einen Browser die Vererbungsmöglichkeit in JavaScript veranschaulicht.

3.7.4 Namenskonventionen für die Objektorientierung

Folgende syntaktisch in JavaScript nicht notwendigen Namenskonventionen werden im dem nachfolgenden Programmbeispiel verwendet. Diese Namenskonventionen dienen der verbesserten Lesbarkeit. Diese sind insbesondere dann sinnvoll, wenn objektorientierte Modelle z.B. in Skriptsprachen von Mathematiksoftware (z.B. Octave, wxMaxima, ...), die keine syntaktischen Strukturen für die Unterstützung objektorientierter Modelle besitzen. Ferner kann man

3.7.4.1 Klassenbezeichnungen

Im nachfolgenden Programmbeispiel in Javascript sieht man, dass der Konstruktor in der Syntax nicht von einer Funktion allgemein auf den ersten Blick zu unterscheiden ist.

Die Funktion `KLebewesen()` setzt den Konstruktionsaspekt eines Objektmanagers um. Allein diese Ergänzung des Buchstabens `K` im Funktionsnamen kann die Definition einer Klasse sichtbarer machen. im JavaScript-Code kann als Namenskonvention anzeigen, dass es sich um eine Klassendefinition handelt, die bei der Erzeugung eines Objektes aufgerufen wird.

Allgemein können daher die Klassendefinitionen durch ein führendes `K` vor der Klassenbezeichnung durch

```
function KMeineKlasse()
```

gekennzeichnet werden. Je nach gewählter Syntax gibt es für den Lernenden auch noch weitere Indikatoren für die Klassendefinition (z.B. in JavaScript `return this;` am Ende des Konstruktors). Die hier genannte Namenskonvention sollte programmiersprachenunabhängig die Erkennung von objektorientierten Strukturen erleichtern.

3.7.4.2 Bezeichnungen für Zustände

Zustände werden in dem nachfolgenden Beispiel durch ein führendes z vor der Zustandsbezeichnung gekennzeichnet. Die Zustandsvariablen für das Alter lauten dann z.B. zAlter.

Im Javascript-Code wurde dieses für das Alter der Lebewesen verwendet.

```
this.zAlter
```

Das führende this. im Konstruktor sorgt dafür, dass zAlter als Zustandsvariable für die Klasse interpretiert wird und nicht als lokale Variable einer Funktion KLebewesen.
Wenn man in einem erzeugten Objekt xLebewesen auf die Zustandsvariable zAlter zugreifen möchte, kann das in der syntaktisch in vielen Programmiersprachen verwendeten objektorientierten Notation erfolgen:

```
xLebewesen = new KLebewesen();\\
xLebewesen.zAlter = 5;
```

Im obigen Beispiel wird xLebewesen als Objekt der Klasse KLebewesen erzeugt und dann das Alter auf 5 gesetzt.

3.7.4.3 Bezeichnungen für Methoden

Methoden werden in dem nachfolgenden Beispiel durch ein führendes m vor der Methodenbezeichnung gekennzeichnet. Die Methodenvariablen für das Älterwerden lauten dann z.B. mWerdeAelter.
Wenn man in einem erzeugten Objekt xLebewesen einen Prozess des Älterwerdens um ein Jahr starten möchte, wird durch die Klammern mWerdeAelter() angedeutet, dass der Prozess nach der Methodendefinition aus dem Konstruktor ausgeführt werden soll.

```
function KLebewesen() {
     //---Konstruktor der Klasse KLebewesen---
     ....
     this.mWerdeAelter = mWerdeAelter_KLebewesen;
     ....
}

....
```

```
....

function mWerdeAelter_KLebewesen() {
    //---Methodendefinition
  ....
}
```

In den obigen Zeilen die Methodvariable `mWerdeAelter` auf eine Methodendefinition gesetzt, die weiteren Verlauf des Quelltextes definiert sein muss.

```
  xLebewesen = new KLebewesen();
  xLebewesen.mWerdeAelter();
```

Sprachlich wird auch hier zwischen Prozessen, die in Objekten ablaufen und den Methoden, die in Klassen definiert werden, unterschieden. Die eigentlich Definition der Methode ist aus Gründen der Übersichtlichkeit nicht in den Konstruktor der Klasse intergriert. Um die Methodendefinitionen beim Überschreiben auch unterscheiden zu können, wird bei den Methodendefinition immer die Klassenbezeichnung angehaengt, damit Namengleichheit bei den Methodenbezeichnungen nicht zu einem Namenskonflikt führt. In dem nachfolgenden Programmbeispiel ist die für das Überschreiben der Methode `mZeigeAlter` veranschaulicht worden.

```
   function mZeigeAlter_KLebewesen() {
   ...
   function mZeigeAlter_KLebewesenMitNamen() {
    ....
```

3.7.4.4 Bezeichnungen für Parameter von Methoden und Konstruktoren

Parameter werden in dem nachfolgenden Beispiel durch ein führendes `p` vor der Parameterbezeichnung gekennzeichnet. Die Parametervariablen für einen Parameter, der das Alter setzt, lauten dann z.B. `pAlter`.
Dies ist z.B. bei der Methode `mSetzeAlter` für die Klasse `KLebewesen` veranschaulicht worden

```
function mSetzeAlter_KLebewesen(pAlter) {
        this.zAlter = pAlter;
}
```

Aber auch Konstruktoren können Parameter enthalten. Wenn man z.B. im Konstruktor bereits das Alter initialisieren möchte, kann man das in der Definition der Klasse wie folgt realisieren.

```
function KLebewesenInit(pAlter) {
  //---Zustaende----
        this.zAlter = pAlter;
        .....
```

Methoden und Konstruktoren können natürlich auch mehrere Parameter enthalten, die durch Komma getrennt werden.

```
function KLebewesenInit(pAlter,pName) {
  //---Zustaende----
        this.zAlter  = pAlter;
        this.zName = pName;
        .....
```

3.7.5 Programmbeispiel mit Vererbung in JavaScript

Die folgenden Programmzeilen stellen eine vollständige HTML-Datei dar, die jeweils für einzelne Erläuterungen zu den Programmzeilen unterbrochen wurde. Ziel ist es, eine erste in einem Browser nutzbare HTML-Datei mit eingebettetem JavaScript-Code zu erzeugen.

3.7.5.1 HTML-Kopf

Zunächst wird die Datei mit einem HTML-Kopf eröffnet, der den Titel der HTML-Datei enthält und mit der letzten Zeile den Beginn des JavaScript-Bereichs definiert.

```
<HTML>
<HEAD>
   <TITLE>
      Vererbung in JavaScript
   </TITLE>
<SCRIPT LANGUAGE="JavaScript">
```

3.7.5.2 Konstruktor der Klasse KLebewesen

Die folgenden Programmzeilen definieren die Klasse KLebenwesen . Der Konstruktor legt die Zustände und Methoden der Klasse KLebewesen fest. Die Klasse KLebewesen ist eine Basisklasse, die keine Oberklasse besitzt (d.h. es fehlt die Definition und Ausführung von Super())

```
//=============================================================================
// Konstruktor der Klasse KLebewesen
//=============================================================================

function KLebewesen() {
  //---Zustaende----
        this.zAlter = 0;
  //---Methoden-----------------
        this.mWerdeAelter = mWerdeAelter_KLebewesen;
        this.mZeigeAlter  = mZeigeAlter_KLebewesen;
        this.mSetzeAlter  = mSetzeAlter_KLebewesen;
  //---RETURN---neues Objekt----
        return this
}
```

3.7.5.3 Implementation der Methoden von der Klasse KLebewesen

Allgemein wurden die Programmzeilen durch Trennlinien und Kommentare strukturiert, die jeweils mit // beginnen. Diese Kommentare können gegebenenfalls durch ausführlichere Kommentare zu den Eigenschaften und Methodendefinitionen ersetzen.

Die Methodendefinitionen als Funktionen werden nicht direkt aufgerufen. Die Bezeichnung ist aus Gründen einer konsistenten nachvollziehbaren Darstellung gewählt. Die Methodenbezeichnung in der Definition hat keinen Einfluss darauf, wie ein Prozess aufgerufen wird, Methode aufgerufen wird. In dem Konstruktor der Klasse in Abschnitt 3.7.5.2 wird durch die Zeile

```
this.mWerdeAelter = mWerdeAelter_KLebewesen;
```

Die Verbindung zwischen dem Bezeichner für die Methode und der Implementation der Methode hergestellt. Die folgenden Programmzeilen in JavaScript zeigen die Implementationen der Methoden für die Klasse KLebewesen, die das Alter um 1 erhöhen, das Alter ausgeben können und das Alter eines Objekts der Klasse KLebewesen setzen können.

```
//=============================================================================
// Implementation der Methoden von der Klasse KLebewesen
//=============================================================================
function mWerdeAelter_KLebewesen() {
        this.zAlter++;
}
//=============================================================================
function mZeigeAlter_KLebewesen() {
        alert("Ich bin "+this.zAlter+" Jahre alt.");
}
//=============================================================================
function mSetzeAlter_KLebewesen(pAlter) {
        this.zAlter = pAlter;
}
//=============================================================================
```

3.7.5.4 Konstruktor der Klasse `KLebewesenMitNamen`

Die folgenden Programmzeilen definieren die Klasse `KLebenwesenMitNamen` . Der Konstruktor legt zunächt die Oberklasse `KLebewesen` fest, von der die Klasse `KLebewesenMitNamen` die Zustands- und Methodendefinitionen erbt. Die Ausführung von `Super())` führt zur Vererbung. Danach beherrscht die Klasse z.B. die Methode `mZeigeAlter` und führt dann ohne weitere Ergänzungen in der Klasse die Methodendefinition `mZeigeAlter_KLebewesen()` aus. Diese wird allerdings im weiteren Verlauf der Klassendefinition durch die Methodendefinition `mZeigeAlter_KLebewesenMitNamen()` überschrieben.

```
//=============================================================================
// KLASSE: KLebewesenMitNamen
//=============================================================================
function KLebewesenMitNamen() {
  //---Oberklasse-------------
        this.Super = KLebewesen;
        this.Super();
  //---Zustaende--------------
        this.zName = "";
  //---Methoden-----------------------
  //------Methoden ueberschreiben--------
       this.mZeigeAlter  = mZeigeAlter_KLebewesenMitNamen;
  //------Neue Methoden-----------------
       this.mZeigeNamen  = mZeigeNamen_KLebewesenMitNamen;
```

```
  //---RETURN---neues Objekt------------
        return this
}
```

3.7.5.5 Implementation der Methoden von der Klasse `KLebewesenMItNamen`

Die Methodendefinitionen, die zusätzlich in der Klasse `KLebenwesenMitNamen` als Funktionen definiert werden, sind in den folgenden Programmzeilen aufgeführt. Die Funktion `alert()` ist eine Standardfunktion in JavaScript, mit der man eine kleines Nachrichtenfenster öffnen kann, das der Nutzer lesen und mit OK bestätigen muss. Dann wird der Programmablauf weiter fortgesetzt. Diese verwendete Funktion ist keine Methode eine Klasse, sondern klassisch funktional als Bestandteil des Befehlumfangs von JavaScript allgemein bereitgestellt wird. Die Bezeichnung für die überschriebene Methode `mZeigeAlter` muss sich zur Vermeidung von Namenskonflikten in der Implementation von

```
mZeigeAlter_KLebewesen()
```

unterscheiden.
Die hier verwendeten Namenskonventionen führen zu der folgenden Bezeichnung:

```
function mZeigeAlter_KLebewesenMitNamen() {
  ....
}
```

An der Implementation der Klassen kann man nicht erkennen, dass eine bestimmte Methode überschrieben wurde. Daher kann man zur Sichtbarkeit einen Kommentar ergänzen.

```
//=============================================================================
// Implementation der Methoden der Klasse KLebewesenMitNamen
//=============================================================================
function mZeigeNamen_KLebewesenMitNamen() {
        //--- ueberschreibt mZeigeAlter() aus der Klasse KLebewesen ----
        alert("Mein Name ist "+this.zName);
}

function mZeigeAlter_KLebewesenMitNamen() {
        alert("Das Lebewesen "+this.zName+ " ist "+ this.zAlter+" Jahre alt.");
```

```
}
```

3.7.5.6 Objekte erzeugen und Prozesse in Objekten starten

Es wird ein Objekt der Klasse `KLebewesenMitNamen` erzeugt, der Name auf `Pferd` gewählt und das Alter auf 12 gesetzt. Es werden zwei Methoden aufgerufen und

```
//=========================================================================
// OBJEKT: "xLebewesen" der Klasse "KLebewesenMitNamen" erzeugen
//=========================================================================
var xLebewesen = new KLebewesenMitNamen();

xLebewesen.zName='Pferd';
xLebewesen.mZeigeNamen();
xLebewesen.mSetzeAlter(12);
xLebewesen.mWerdeAelter();
xLebewesen.mZeigeNamen();
//=========================================================================
//---- dynamisches Ueberschreiben von Methoden ------------------------
//=========================================================================

function zeigeNamenMitBesitzer(pBesitzer) {
   alert("Dieses " + this.zName + " gehoert dem Besitzer "+ pBesitzer + ".")
}
//=========================================================================
xLebewesen.mZeigeNamen = zeigeNamenMitBesitzer;
xLebewesen.mZeigeNamen("Bert");
//=========================================================================
```

3.7.5.7 Objektinformationen in den HTML-Text schreiben

Zunächst wird die JavaScript-Umgebung und der Kopf der HTML-Datei geschlossen. Mit dem BODY-Tag beginnt der eigentliche HTML-Text. Mitten im Quelltext werden dann noch einmal JavaScript-Umgebungen geöffnet. In diesen Umgebungen mit `document.write(...)` Variableninhalte aus den JavaScript-Umgebung der HTML-Seite ausgegeben werden. Im unten angegeben Beispiel werden zunächst der Name und dann das Alter von dem Objekt `xLebewesen` ausgegeben.

```
</SCRIPT>
</HEAD>
```

```
<BODY>
Demo: Vererbung von Klassen realisiert mit JavaScript
Im HTML-Text wird nun der Name von einem Objekt dargestellt.
Das Objekt hat den Namen
<SCRIPT LANGUAGE="JavaScript">
   document.write(xLebewesen.zName);
</SCRIPT>
und es ist
<SCRIPT LANGUAGE="JavaScript">
   document.write(xLebewesen.zAlter);
</SCRIPT>
Jahre alt.
```

3.7.5.8 HTML-Formulare und Interaktion mit Objekten

Die folgenden Programmzeilen zeigen, wie man durch Formularelemente Informationen aus Objekten auslesen kann bzw. einen Methodenaufruf starten über Formularelemente starten kann.

```
Wenn Sie auf den Knopf mit der Aufschrift <i>Alter</i> klicken, wird das
aktuelle Alter von <TT>xLebewesen</TT> in dem folgende Textfeld angezeigt.

<FORM NAME="fOOM">
  <INPUT TYPE="text" NAME="tAlter" VALUE="0" SIZE="3">
  <INPUT TYPE="button"
         ONCLICK="xLebewesen.mWerdeAelter()"
         VALUE="altern um 1 Jahr">
 <INPUT TYPE="button"
         ONCLICK="document.fOOM.tAlter.value=xLebewesen.zAlter"
         VALUE="Alter">
</FORM>
Wenn Sie auf den Knopf mit der Aufschrift <i>altern um 1 Jahr</i> klicken, wird das
aktuelle Alter von xLebewesen um 1 erh&ouml;ht.
</BODY>
</HTML>
```

Die gesamte Datei kann unter unter

`http://math.uni-landau.de/download/niehaus/oom/oomdemo.html`

heruntergeladen werden.

3.8 Zusammenfassung: Mathematische Objektorientierung

Für die objektorientierte Modellbildung gibt es unterschiedliche Ausdrucksmöglichkeiten, das Modellierungsprodukt in einer Sprache zu beschreiben. Eine mathematisch-formale Beschreibung stellt dabei eine Verbindung zwischen den Konzepten der Objektorientierung aus der Informatik und mathematischen Modellbeschreibungen dar. Diese bedient sich den Grundelementen einer jeden mathematischen Formalisierung durch Mengen und Abbildungen. Diese mathematische Formalisierung ist dabei als Beschreibungsoption für Modellierungsergebnisse ebenfalls programmiersprachenunabhängig.

Betrachtet die Objektorientierung aus dem Blickwinkel Abbildtheorie (siehe Abschnitt 1.3, S. 26 ff.), dann müssen alle in einem (realen) System vorhandenen Objekte bestimmt werden, die für die mathematische oder informatische Modellbildung relevant sind. Gemeinsame Eigenschaften dieser Objekte führen zur Klassifizierung der Objekte.

Ziel dabei ist eine Strukturäquivalenz zwischen den als relevant charakterisiert Objekten und deren Eigenschaften. Das Modellierungsergebnis sollte sich dabei soweit wie möglich an die Strukturierung des modellierten realen Systems anzunähern, d.h. real existierende Objekte müssen nach Möglichkeit ein (bezogen auf Zustandsraum und Funktionalität) korrespondierendes Objekt im Modell besitzen. Diese strenge Orientiertung am realen System hat vor allem für die Wartbarkeit und die Tests des Modells große Bedeutung.

Diese Strukturäquivalenz ist einer mathematischen Beschreibung zunächst einmal nicht durch die gebräuchlichen mathematischen Formalismen direkt gegeben. Diese Strukturerhaltung für die Objektorientierung auch auf die mathematische Notation zu übertragen, wurde mit diesem Kapitel 3 erreicht.

Ein formaleres mathematisches Verständnis der Objektorientierung kann aber auch auf die informatische Umsetzung Impulse liefern, um in Programmiersprachen ohne volle Unterstützung der OO, dennoch Teilaspekte umzusetzen sind, die die Syntax der Sprache eigentlich nicht unterstützt.

Das Beispiel „JavaScript“, dem syntaktisch zunächst die Möglichkeit der Vererbung fehlte, verdeutlicht diesen Ansatz durch die Implementation der Vererbung.

Insgesamt stellt dieses Kapitel nur eine Darstellungsebene der Objektorientierung auf formal mathematischer Ebene vor.

Kapitel 4

Objektorientierte Modellierung und Spracherwerb

Prof. Dr. Ludger Humbert
Fachgebiet Didaktik der Informatik
Bergische Universität Wuppertal
humbert@uni-wuppertal.de

Martin Weise
Reinoldus- und Schiller-Gymnasium, Dortmund
martin.weise@sophoi.de

Prof. Dr. Engelbert Niehaus
Fachgebiet Didaktik der Mathematik
Universität Koblenz-Landau
niehaus@uni-landau.de

Dieses Teildokument steht unter der „Creative Commons Lizenz“ BY-NC-ND. Dies bedeutet: bei weiterer Verwendung des Textes sind die Namen der Autoren zu nennen, die Weiternutzung darf ausschließlich nicht kommerziell erfolgen, das Dokument darf nicht bearbeitet werden. Weitere Details zu den Creative Commons Lizenzen finden sich unter: `http://creativecommons.org/licenses/?lang=de`

Didaktische Ansätze der objektorientierten Modellierung – OOM fordern im Bereich der informatischen Modellierung den expliziten Einsatz künstlicher Spra-

chen, wie didaktisch gestaltete Elemente der UML[1], Struktogramme und formale Programmiersprachen. Die unterschiedlichen Ausdrucksformen dienen dabei der Darstellung und zur Umsetzung von Modellierungsergebnissen. Damit fällt der Nutzung von künstlichen Sprachen im Zusammenhang mit dem Informatikunterricht eine wichtige Rolle zu. Formale Programmiersprachen stehen in einem Lehrprozess am Ende der Sprachentwicklung und der Darstellungskompetenzen für die Modellierungsergebnisse.

Analoge Strukturen findet man in der Mathematik, wenn Modellierungsergebnisse in einer formalen mathematischen Sprache ausgedrückt werden (siehe Kapitel 3, S.87 ff.). Auch ikonische und enaktive Repräsentationen von Modellierungsergebnissen stellen Ausdrucksmöglichkeiten für Modellierungsergebnisse dar.

In diesem Kapitel 4 stehen im Gegensatz zu Kapitel 3 (S. 87 ff.) nicht die formalen künstlichen Sprachen im Vordergrund, sondern der Anfang der Sprachentwicklung zur objektorientierten Modellierung wird aus dem Blickwinkel des Spracherwerbs betrachtet. An dieser Schnittstelle der Objektorientierung zu den Sprachwissenschaften sind daher die ersten Schritte von einer umgangssprachlichen Beschreibung in eine fachsprachliche Strukturbildung von Interesse.

Mit einer unterrichtspraktische Zielsetzung kann diese frühe Phase der objektorientierten Strukturbildung wie folgt beschreiben:

> Lehrpersonen in die Lage versetzen, in umgangssprachlichen Beschreibungen der Schülerinnen und Schüler *objektorientierte Strukturen entdecken* zu können, um daraus Unterstützungsmaßnahmen und Hilfen für den Lernprozess abzuleiten.

Sprachbildung und Sprachenlernen für natürliche Sprachen sind Forschungsbereiche, deren Ergebnisse in den Didaktiken der sprachbezogenen Fächer eine zentrale Rolle zukommen. In diesem Kapitel wird die Anwendung ausgewählter Forschungsergebnisse aus den Sprachwissenschaften diskutiert, um Hinweise zum Sprachenlernen von Novizen bezogen auf künstliche, formal wohldefinierte Sprachen im Zusammenhang mit der objektorientierten Modellierung zu erhalten und diesen Forschungsbereich für die Fachdidaktik zu erschließen.

[1] UML=Unified Modelling Language

4.1 Spracherwerbstheorien und OOM

Werden zentrale Theorien des menschlichen Spracherwerbs auf die OOM bezogen, so können aus den grundlegenden – sich in Teilen ausschließenden – Sichten erste Ideen für den Erwerb einer objektorientierten Sicht expliziert werden. Zusammenfassend stellt z. B. (Volmert, 2000, S. 210–219) (ohne den kognitiv-linguistischen Ansatz) verschiedene Sichten dar, die wir auf die „Welt der Objekte“ beziehen:

Lerntheoretischer Ansatz Der Spracherwerb ist ein Konditionierungsvorgang, der nach dem Black-Box-Modell und dem Reiz-Reaktions-Schema wahrgenommene Objekte durch Üben und Wiederholen informatischen Objektbeschreibungen zuordnet (B. F. SKINNER).

Nativistischer Ansatz Die Kompetenz, Objekte in der Welt wahrzunehmen und von der Umgebung abzugrenzen, ist dem Menschen angeboren und damit wäre die Grundstruktur einer „informatischen Sprache“ (im Sinne der Universalgrammatik von (Chomsky, 1965)) für die Modellierung an sich bereits im menschlichen Gehirn angelegt (N. CHOMSKY).

Kognitivistischer Ansatz Der Spracherwerb basiert auf kognitiven Fähigkeiten, die zeitgleich erworben werden; das Erlernen der Sprache ist dem Erlernen von Denkprozessen in Objektnetzen analog (J. PIAGET).

Kommunikativer Ansatz Sprache wird aus der Interaktion, d. h. dem Handeln mit Objekten, gelernt; dies gilt sowohl für semantische Einheiten wie Objekte und Prozesse als auch für Elemente, die keine eigene Bedeutung tragen (synsemantisch), wie Attribute mit Objektbeziehungen (J. BRUNER).

Kognitiv-linguistischer Ansatz Kognitivistischer Ansatz, gepaart mit einer Fokussierung des symbolischen Charakters von Sprache. D. h. Sprachentwicklung ist hauptsächlich mit semantisch-formeller Kategorisierung bzw. Konzeptualisierung gleichzusetzen (A. E. GOLDBERG, M. TOMASELLO).

4.2 Phasen des Spracherwerbs und der Wortschatzentwicklung – Analogiebetrachtungen zur Objektorientierung

Wird die Spracherwerbsforschung im Kontext des Erlernens künstlicher Sprachen betrachtet, so stellt sich die Frage, ob Analogien zwischen der Sprachentwicklung einer natürlichen und dem Erlernen einer künstlichen Sprache bestehen könnten.

Legt man zunächst die Muttersprache (wird in der Spracherwerbsforschung mit *L1* bezeichnet) zum Vergleich zugrunde, wird offensichtlich, dass der Erwerb einer künstlichen Sprache eher ein Zweitspracherwerb (Spracherwerbsforschung: *L2*) ist, bei dem einige der Phasen wegfallen bzw. anders ausgestaltet sind. Wesentlich geeigneter ist hier eine Orientierung an den den Phasen, die Stephen KRASHEN im Rahmen seines „aqcuisition orders“, wie sie z. B. von KRASHEN und TERRELL im Rahmen ihres „natural approach“ formuliert wurden (vgl. (Krashen & Terrell, 1983)).

Für künstliche Sprachen der informatischen Modellierung wird die Analogiebildung zu einer Begriffsbildung am Beispiel der OOM dargestellt und am Beispiel der Begriffe: Objekt, Prozess, Zustand, Klasse, Methode, Objekt- und Klassenbeziehungen erläutert.

Die Analogiebildung betrifft den Spracherwerb: von einer unscharfen Systemwahrnehmung, die zu einer immer präziser werdenden Systembeschreibung führt, an deren Ende die Verwendung von künstlichen Sprachen steht. Diese Präzisierung der Systembeschreibung ist als Lernprozess Gegenstand der fachdidaktischen Analyse. Die Unschärfe von Sprache auf der einen Seite und die Anforderungen an die Rigidität in der Anwendung künstlicher Sprachen auf der anderen Seite führt in weiteren Verlauf zur Nutzung von Fuzzy-Logik zur formalen Beschreibung dieser sprachlichen Präzisierung.

4.2.1 Phasierung des L2-Erwerbs

Für die Analogiebildung betrachten wir die von der Sprachforschung vorgenommene Phasenstruktur mit fünf Entwicklungsschritten nach KRASHEN und TERRELL. Den Phasen des Spracherwerbs werden bestimmte Lebensmonate bzw. Lebensjahre zugeordnet, denen idealtypisch Fähigkeiten und Ausdrucksmöglich-

keiten im Umgang mit der Sprache entsprechen.

Nr	Phase	Merkmale	Wortschatz
1	Prä-Produktion (pre production)	Stille Phase (fast ausschließlich rezeptiv)	~500 Wörter
2	Frühe Produktion (early production)	Nutzung einzelner Wörter oder Phrasen, vorwiegend reaktiv	~1.000 Wörter
3	Sprachemergenz (speech emergence)	Formulierung ganzer Sätze, anfangs noch reproduktiv	~3.000–7.000 Wörter
4	Mittlere Sprachkompetenz (intermediate fluency)	mühelose grundlegende soziale Kommunikation (BICS – Basic interpersonal communication skills), produktiv	~6.000–12.000 Wörter
5	Fortgeschrittene Sprachkompetenz (advanced fluency)	Kognitiv- akademische Sprachkompetenz (CALP – cogintive academic language proficiency), Sprachbewusstheit	

Tabelle 4.1: Phasen des Spracherwerbs und Wortschatz – nach (Krashen & Terrell, 1983))

Wir werden in der folgenden Aufzählung einzelnen Phasen des Spracherwerbs Elemente der objektorientierten „Sprachentwicklung“ zuordnen. Diese Darstellung ist ein erster Versuch, um Verbindungen zwischen dem Spracherwerb und der OOM herzustellen.

Aufbauend auf den Phasen des Spracherwerbs wurde ein didaktisches Modell für das Erlernen einer objektorientierten Modellierungssprache thematisiert, an dessen Ende z.B. die Fähigkeit steht, ein Modellierungsergebnis in Fachsprache auszusdrücken. Schwerpunkt ist hier allerdings der Anfang der objektorientierten Sprachentwicklung aus einer Alltagssprache heraus ohne fachspezifische Modellierungskenntnisse.

1. **Prä-Produktion (pre production)**

Merkmale/Eigenschaften der Phase Stille Phase (fast ausschließlich rezeptiv)

Bezug zum informatischen Spracherwerb Syntaktische Elemente werden identifiziert, z. B.: Objektbezeichner beginnen immer mit einem Kleinbuchstaben; die Camelcase-Schreibweise wird durch Mustererkennung entdeckt; in der graphischen Beschreibung werden abgerundete Ecken als Identifikation Objekten zugeordnet.

Die Zuordnung der Objekt- und Prozessbezeichnungen ist nicht stabil. Systemgrenzen sind unscharf ausgebildet und es werden Objekte und Prozesse außerhalb des betrachteten Systems mit Objekten und Prozessen innerhalb des betrachten Systems vermischt.

Sprachliche Beispiele „Ich weiß nicht genau, ob das jetzt dazu gehört oder nicht."

Erläuterungen der Beispiele Die unspezifische, eher reproduktive Phase ist im Spracherwerb einer künstlichen Sprache analog einer unspezifische Sammlung von assoziativen Gedanken zu einem betrachteten System für die OOM.

2. **Frühe Produktion (early production)**

Merkmale/Eigenschaften der Phase Nutzung einzelner Wörter oder Phrasen, vorwiegend reaktiv

Bezug zum informatischen Spracherwerb es sind keine objektabgrenzenden Sprachstrukturen der OOM zu erkennen, bzw. es können nur sehr schwer wahrgenommene Objekte informatischen Objektbeschreibungen zugeordnet werden. Es gibt ein eingeschränkt nutzbares Inventar einer Objektsprache und Prozesssprache.

Erste abgegrenzte Wörter/Bezeichnungen einer Objektsprache und einer Sprache, um Prozesse zu beschreiben, entstehen.

Sprachliches Beispiel „Dort sieht man etwas."

Erläuterungen des Beispiels Entspricht einer ersten Abgrenzung eines im Fokus liegenden Wahrnehmungsgegenstands und dem unwichtigeren Hintergrund. Die Sprachwahl „Dort sieht man etwas" zeigt dabei keine Unterscheidung von Prozess und Objekt (Fokus-Hintergrund-Wahrnehmung).

3. **Sprachemergenz (speech emergence)**

   Merkmale/Eigenschaften der Phase Formulierung ganzer Sätze, anfangs noch reproduktiv

   Bezug zum informatischen Spracherwerb Ausbau von objektabgrenzenden Sprachstrukturen; Aktivitätsbeschreibungen von betrachteten Objekten werden analog wiedergegeben; erkannte objektorientierte Strukturen werden übergeneralisiert, z. B. alle Objekte können bestimmte Operationen durchführen (in SuM[2]: jedes Objekt kann sich zeichnen, steht in Beziehung zu einem Objekt der Klasse Stift).

   Sprachliches Beispiel „Dort geschieht etwas." – „Dort liegt ein . . . "

   Erläuterungen des Beispiels „Dort geschieht etwas" kennzeichnet das Wahrgenommene bereits als Prozess. „Dort liegt ein . . . " kennzeichnet ein Objekt und bezeichnet das Objekt mit einem Begriff. Selbst wenn bei den Lernenden mit der Sprachwahl bereits eine propädeutische Prozess-Objektunterscheidung erkennbar ist, müssen die Begriffe „Prozess" und „Objekt" aus informatikdidaktischer Seite als Begriffe noch nicht unterschieden werden.

   Man kann in dem Sprachgebrauch die Identifikation von Objekten erkennen und auch Prozesse werden unterschieden und zutreffend erkannt. Ein direkter Zusammenhang zwischen den Prozessen und Objekten ist sprachlich noch nicht hergestellt. Ferner werden Klassenhierarchien nicht berücksichtigt, da z. B. LKW nicht als eine Unterklasse der Klasse Fahrzeuge wahrgenommen wird. Diese Klassenbeziehung ist auf der Objektebene prinzipiell nicht erkennbar.

4. **Mittlere Sprachkompetenz (intermediate fluency)**

   Merkmale/Eigenschaften der Phase mühelose grundlegende soziale Kommunikation (BICS), produktiv

   Bezug zum informatischen Spracherwerb Bildung von Objektbezeichnungen, Sprachverständnis für objektorientierte Beschreibungen; Aufbau des Wortschatzes für wahrgenommene Objekte und Prozesse; Beschreibung von Abläufen auf Objektebene.

   Sprachliches Beispiel: „Man sieht dort Fahrzeuge, Autos, LKW und Fahrräder – sie fahren oder parken"

[2] SuM: Stifte und Mäuse

Erläuterungen des Beispiels Beliebige, komplexe Problemstellungen können objektorientiert analysiert werden – Methoden zur Darstellung der Objekte und ihrer Beziehungen führen zu Objektdiagrammen. Die Lösung des Problems wird über Sequenzdiagramme dargestellt. Exemplarisch ist auch eine Klassifikation der Objekte möglich, die über Ähnlichkeit zu bekannten Problemen und ihren Lösungen hergestellt wird. Beginn der Nutzung von Entwurfsmustern.

5. **Fortgeschrittene Sprachkompetenz (advanced fluency)**

   Merkmale/Eigenschaften der Phase Kognitiv-akademische Sprachkompetenz (CALP), Sprachbewusstheit

   Bezug zum informatischen Spracherwerb Die vollständige objektorientierte Analyse von Problemstellungen wird realisiert und wird fach- und sachgerecht umgesetzt – vom ersten identifizierten Objekt bis hin zu Klassen, ihren Beziehungen und dem korrekten Beschreiben von Methoden. Es wird systematisch in existierenden Bibliotheken nach Funktionalität für Teillösungen gesucht – vorhandenes informatisches Wissen wird genutzt. Anwendungsfälle werden korrekt abgebildet; Testfälle werden vor der Implementierung der Klassen festgelegt und im Sinne der iterativen Entwicklung des Gesamtsystems als Prüfsteine für die Realisierung von Problemlösungen systematisch geplant.

6. **Verb-Nomen-Konstruktion**

   Merkmale/Eigenschaften der Phase Syntaxverständnis einfacher Sätze vorhanden; Bildung von Sätzen mit zwei Bestandteilen (Balla habe); Bildung einer Angelpunkt-(Pivot-)Grammatik, Wortschatzspurt

   Bezug zum informatischen Spracherwerb Wörter/Bezeichnungen aus einer Objektsprache werden mit einer Prozesssprache in Beziehung gesetzt, Zwei-Wort-Sätze (z.B. ‘‘`Ball rollt`‘‘ bzw. ‘‘`Ball.rollt()`‘‘ in einer formaleren Notation) entstehen aus Objektbezeichnungen zusammen mit Verben, die eine Tätigkeit oder einen Prozess beschreiben (Verb-Nomen-Konstruktion).

7. **Attributierung von Verb-Nomen-Konstruktionen**

Merkmale/Eigenschaften der Phase Bildung von Sätzen mit drei und mehr Bestandteilen; Ausbau der Morphosyntax; intensiver Ausbau des Wortschatzes: Substantive, Verben, Funktionswörter

Bezug zum informatischen Spracherwerb Wörter/Bezeichnungen aus einer Objektsprache erhalten Eigenschaften und die Sprache, um Prozesse zu beschreiben, wird unter vereinzelter Verwendung von identifizierten Teilprozessen beschrieben, Drei-Wort-Sätze und Mehr-Wort-Sätze entstehen aus Objektbezeichnungen (erweiterte Verb-Nomen-Konstruktion). Dabei wird von den Lernenden objektorientiert die beiden folgenden Fällen der unterschieden

- **Attributierung eine Objektes mit einer Zustandsvariable:** Der Ball mit der Oberfläche von $50cm^2$ rollt.

  ‘‘`Ball.rollt()`‘‘ und ‘‘`Ball.zOberflaeche = 50`‘‘
- **Attributierung durch Parametrisierung von Prozessen:** Der Ball rollt $50cm$.

  ‘‘`Ball.rollt(50)`‘‘

8. **Entwicklung Objektsprache**

   Merkmale/Eigenschaften der Phase Bildung komplexer Sätze; Zugang zu mentalen und epistemischen Prädikaten (denken, hoffen, wissen usw.)

   Bezug zum informatischen Spracherwerb Entwicklung einer Objektsprache, die Zustände und Prozesse klar unterscheidet

9. **Entwicklung einer Klassensprache**

   Merkmale/Eigenschaften der Phase Abschluss der elementaren Sprachentwicklung; alle wesentlichen Elemente des Sprachsystems werden in den Grundzügen beherrscht: Phoneme, Lexeme, Morphosyntax

   Bezug zum informatischen Spracherwerb Entwicklung einer Objektsprache, in der Objekte nach gemeinsamenen Eigenschaften klassifiziert werden, die Zustände und Prozesse klar unterscheidet

4.2.2 Voraussetzungen der Sprachentwicklung

Betrachtet man die Voraussetzungen für eine normale Sprachentwicklung, so sind viele Aspekte von Bedeutung, die oft nur indirekt mit der Sprache in Verbindung stehen. Kinder erlernen die Muttersprache und das Sprechen in der

Muttersprache nicht durch explizite Nennung von Regelsystemen über das Erlernen grammatikalischer Strukturen. Bei einer Fremdsprache können erst bekannte Regelsysteme der Grammatik mit der Grammatik der Fremdsprache in Beziehung gesetzt werden und durch Analogiebildung abstraktere Strukturen beim Spracherwerb genutzt werden. Ersetzt man die Rolle der Fremdsprache durch die Muttersprache, so besitzt die Muttersprache oder Alltagssprache Unschärfen, die erst in Lernprozess zu einer rigiden Nutzung von künstlichen Sprachen entwickelt werden muss.

Bezogen auf die Informatikdidaktik können zwei Arten des Spracherwerbs betrachtet werden: die künstliche Sprache in Analogie zur **Muttersprache** bzw. zur **Fremdsprache** mit einem Fokus auf grammatikalischen Strukturen und den spracherzeugenden Regelsystemen.

Verfolgt man bei der Analyse den Spracherwerb einer künstlichen Sprache im Sinne der Muttersprache, so sind zwei Ebenen der Lernvoraussetzungen für die Lernenden zu unterscheiden:

- sich prinzipiell in der Muttersprache zu einem analysierte System äußern,
- sich in einer künstlichen Sprache zu einem analysierten System ausdrücken.

Wenn Kinder mit allen Sinnen lernen, wird die Sprache als Kontext zu Erfahrungen aus der unmittelbaren Umwelt erlebt. In der Informatikdidaktik ist dies mit einer aktiven Auseinandersetzung mit dem analysierten System zu vergleichen, bei dem Handlungserfahrung bzw. Erfahrungen zu Systemzuständen zu Zustandsübergängen später durch den Sprachgebrauch einer künstlichen Sprache verbalisiert werden können.

Beim Erwerb von Fremdsprachen wird durch diesen enaktiven Zugang Sprache mit Handlungen assoziiert, weil Sprache im Kontext von Handlungen erfahren wird. Erst nach dieser Vorerfahrung kann Sprache auch Handlungen induzieren und Bedürfnisse ausgedrückt werden.

Betrachtet man diese Voraussetzungen für den Spracherwerb einer Fremdsprache, so können im Sinne der Sprachentwicklung drei Ebenen von Lernumgebungen unterschieden werden:

1. Handlungen mit Sprache assoziieren (Bereich: Spracherwerb)
2. in der Sprache Handlungen induzieren (Bereich: handelnder Sprachgebrauch)

3. in der erlernten Sprache über Inhalte kommunizieren

Diese Ebenen sind allerdings für das Erlernen von künstlichen Sprachen im Kontext der objektorientierten Modellierung zu grob. Überträgt man diese Lernvoraussetzungen wieder auf die Informatikdidaktik, so ist der Spracherwerb einer künstlichen Sprache zunächst an eine *deskriptive* Nutzung gekoppelt, bei der die künstliche Sprache im Kontext von Handlungserfahrungen mit einem informatischen oder allgemeiner einem analysierten System auftritt. Erst wenn die Semantik der Sprache präzisiert wurde, kann man mit ihr auch *präskriptiv* arbeiten, um der objektorientierte Modellierungssprache auch nicht existierende Objekte zu verwenden, die zum Zeitpunkt der Modellierung noch nicht existieren und somit auch nicht als Lernvoraussetzung die Möglichkeit besteht, Handlungserfahrungen zu sammeln.

Rollenspiele und Objektspiele, die diesen handelnden Aspekt in den Vordergrund stellen, wurden z. B. in den Staatsarbeiten (Kuhaupt, 2012), (Koehne, 2011), (?, ?) bearbeitet. Das Erleben, Hineinversetzen und Nachspielen von oft unsichtbaren informatischen Akteuren und deren Zusammenspiel in einem informatischen System macht nicht nur unsichtbare Prozesse erlebbar, sondern betrachtet eine Rolle in einem Rollenspiel als semantische Einheit, die auf die eigenen Zustände, den Ablauf der eigenen Prozesse nach eine festgelegten Methode und die Kommunikation mit anderen Rollen achten muss. An diesem Beispiel wird deutlich, dass gerade die enaktive Auseinandersetzung z. B. im Kontext der Sprachentwicklung auch eine Präzisierung der Begriffe Prozess und Methode nach sich zieht. Die soll kurz am Beispiel von zwei Objekten x, y, die als Instanzen der Klasse K erzeugt wurden, dargestellt werden:[3]

- Die *Methode* $m()$ der Klasse K (formal $K.m()$) definiert, wie Prozesse in Objekten abzulaufen haben, die aus der Klasse K instanziiert werden. Die Notation $K.m()$ für die Methode ist notwendig, damit man beim Überschreiben einer Methode in einer Unterklasse K_u die Methode $K_u.m()$ von $K.m()$ unterscheiden kann.

- *Prozesse* existieren im Unterschied zu Methoden nur im Raum der Objekte. Ein Prozess läuft in einem Objekt ab. $x.m()$ deutet an, dass dieser Prozess in dem Objekt x abläuft und $y.m()$ in y. Im Gegensatz zum Quellcode in einem Programm ist $x.m()$ bzw. $y.m()$ nicht als Aufruf der Methode oder

[3] Im Folgenden wird die Darstellung für Methoden verwendet, die mit () abschließt – auch wenn diese Syntax nicht in allen objektorientierten Programmiersprachen verwendet wird.

auslösendes Ereignis zu verstehen, sondern in einem handelnden Sinn als gesamter Prozess, der in einem Rollenspiel erlebt werden kann.

Im Kontext der OOM in der Informatik können dabei Analogien wie folgt abgeleitet werden.

1. **Sprache als Kontext zu Objekten und Handlungen** Sprache wird den Lernenden als Kontext zu Objekten und Handlungen/Prozessen präsentiert. Die Lernenden untersuchen oder arbeiten mit Objekten und/oder sind mit der Durchführungen von Handlungen beschäftigt und erfahren objektorientierte Sprache in einem zeitlichen und räumlichen Zusammenhang zu diesen Aktivitäten.

   > **Situative Beschreibung:** „Der rote Ball rollt 5 Sekunden“
   > **Beschreibung mit objektorientierten Kontextinformationen:** „Das *Objekt* 'roter Ball' rollt. Dieser *Prozess* dauert 5 Sekunden“.

2. **Wiedererkennung von Objekten und Prozesse:** Objekte verändern sich in der Zeit (siehe Abschnitt 3.6.2, S.115, ff.) Das Wiedererkennen von leicht modifizierten Objekten ist als Voraussetzung für das Nutzen von Begriffen für sich stetig verändernde Objekte wichtig und damit für den Erwerb der objektorientierten Modellierungssprache als eine Fremdsprache wesentlich. Dies liefert auch die Grundlagen, um bekannte Objekte eindeutig identifizieren bzw. Handlungen wahrzunehmen und reproduzieren zu können. Für diesen Aufbau der Fähigkeit müssen nicht notwendig auch die Bezeichnungen bzw. Begriffe für die Objekte bereits in der Modellierungssprache erlernt worden sein. Die Wiedererkennung von modifizierten Objekten ist ein Abstraktionsschritt, der die Klassifikation von Objekten vorbereitet (siehe Abschnitt 3.3 S.94 ff.). Lehrende

   > „Das Objekt habe ich bereits hier (Zeigen auf ein anderes Objekt) schon einmal gesehen!“

3. **Sprachliche Entwicklung objektorientierter Fähigkeiten:** Als Voraussetzung zum Erwerb der Fremdsprache ist Wahrnehmung von Gemeinsamkeiten für Abstraktionsprozesse und die Wahrnehmung individueller Unterschiede für die Präzisierung der Begriffsbildung wesentlich.

   - Objekte im analysierten Gegenstandsbereich unterscheiden

- Objekte sprachlich unterscheiden
- Beziehungen zwischen Objekten im Gegenstandsbereich herstellen
- Objekt (auch bei geringfügigen Veränderung) wiedererkennen
- Zuordnungen durchführen
- Gemeinsamkeiten von Objekten entdecken (Klassifikation)
- in Prozessen Teilprozesse entdecken
- Teilprozessen von Prozesse sprachlich identifizieren Abstrahieren usw. als geistige Fähigkeit sind eng mit dem Spracherwerb verknüpft.

Aus dieser Voraussetzung ergibt sich eine notwendige Trennschärfe zwischen den Begriffen Prozess und Methode. Ein Prozess läuft tatsächlich in einem Objekt ab. Eine Methode gehört zur Klassendefinition und legt fest (wie der Name bereits andeutet), nach welchem Prinzip die Prozesse in den Objekten ablaufen.

4.2.3 Sprachentwicklung anhand der Wortschatzentwicklung

Betrachtet man die informatischen Begriffe, wie Objekt, Klasse, Vererbung, Aggregation, Assoziation, ... und die Phasen, in denen beim Erwerb der Fremdsprache der Wortschatzumfang wächst, so ist die prinzipielle Auseinandersetzung im strukturähnlichen Gegenstandsbereich notwendig, um in den Objekten und Prozessen die Gemeinsamkeiten zu entdecken und diese mit Begriffen belegen zu können.

Ein Blick auf (Rescorla, Mirak & Singh, 2000) und (Grimm, o. J.) zeigt die Entwicklung des Wortschatzes (vgl. Tabelle 4.1). Die Zeitangaben können zwar auf den Erwerb des objektorientierten Sprachsystems nicht direkt übertragen werden; dennoch stellt sich die Frage, wie viele Auseinandersetzungen und Zeit mit realen Gegenstandsbereichen Schülerinnen und Schüler benötigen, um einen objektorientierten Blickwickel auf die strukturgebenden Begriffsbildung der OOM zu entwickeln.

Wortkombination von Zwei-Wort-Sätzen (z. B. Auto fahren) entsprechen objektorientierten Schreibweisen (z. B. `diesesAuto.fahren()`) oder sprachliche Formulierungen von Klassenzordnungen (z. B. das ist ein Auto) müssen in Objekt-Klasse-Beziehungen umgesetzt werden (`diesesAuto` wird/wurde aus der Klasse `Auto`) erzeugt.

Die Anzahl der Begriffe in einem Wortschatz explodieren beim Erwerb der Muttersprache erst nach 36 Monaten, wenn die grammatikalischen Grundstrukturen der Sprache verstanden sind. Kann man ableiten, dass bei dem Spracherwerb für die Objektorientierte Modellierung ähnlich viel Erfahrung und Zeit für die ersten Wörter und die grundlegenden Begriffe notwendig sind, bis die Lernenden nach 36 Monaten die grammatikalischen Grundstrukturen der OOM anwenden können?

Von einem mengentheoretischen Blickwinkel muss man für eine Klassifikation zumindest die Objekte kennen, die man klassifizieren möchte. Den Zeitraum, den Lernende für diese Auseinandersetzung benötigen, sollte dabei als domänenspezifisch in die Planung von Lernprozessen zur objektorientierten Modellierung einfließen. Aus dieser grundlegenden sprachlichen Betrachtung kann man das Ziel ableiten, dass den Lehrenden ein diagnostisches Instrument für die objektorientierte Sprachentwicklung im Klassenraum zur Verfügung gestellt werden sollte, damit die sich die didaktischen Entscheidung für den Unterricht an der Sprachentwicklung der Lernenden orientieren kann.

4.2.4 Fuzzy-Theorie

Betrachtet man nun die Wortschatzentwicklung von einem informatischen Standpunkt, so kann man Begriffe als Bezeichner für eine Menge von Objekten aus dem Gegenstandsbereich auffassen, die diese gemeinsamen Eigenschaften besitzen. Geht man zu einem objektorientierten Lernprozess über, bei dem man im Sinne eines Spracherwerbs die Handlungserfahrungen mit dem Gegenstandsbereich in den Mittelpunkt stellt, so kann man nicht davon ausgehen, dass syntaktische und semantische Strukturen einer künstlichen Sprache *additiv* durch den *Erwerb weiterer Regeln* erlernt werden.

Lernende könnten z. B. das Ordnen von Karten als Prozess identifizieren, während die Aktivierung der Sortierfunktion in einer Tabellenkalkulation nicht als ein strukturgleicher informatischer Prozess identifiziert würde, weil dieser u.U. versteckt abläuft und ohne Zeitverzögerung das sortierte Endprodukt sichtbar ist.

- Diese Wahrnehmung von Gemeinsamkeiten bei unterschiedlichen *Objekten* führt zur *Klassifikation* (siehe 3.3 S.94 ff.)
- Diese Wahrnehmung von Gemeinsamkeiten bei unterschiedlichen *Prozes-*

sen führt zur Beschreibung durch *Methoden* (siehe 3.5 S.106 ff.)

Diese Wahrnehmung von Gemeinsamkeiten werden bei Handlungserfahrungen auf der Ebene von einzelnen Objekten mit unterschiedlicher Qualität und Grad mit einer Objekt-, Klassen-, Methoden- oder Zustandsbezeichung in Beziehung gesetzt. Damit ist die Zuordnung zu Objektbezeichnungen weniger eine Zusammenfassung von Objekten, die ein bestimmte Menge von Regeln erfüllen, als eine durch individuelle Bewertung gegebene graduelle Zuordnung. Das Erlernen einer künstlichen Sprache läuft somit nicht notwendigerweise durch eine regel- oder grammatikinduzierte scharfe Begriffsbildung ab. Dies führt zur Anwendung von Fuzzy-Theorie zur Beschreibung dieser Phänomene.

- **scharf** (crisp): Ist die Person A 14 Jahre? Ja=1, Nein=0)
- **unscharf** (fuzzy): Ist die Person A alt? Ja zu einem Grad von 0,2, wenn diese z. B. 14 Jahre ist.

Auf formaler Ebene unterscheidet man daher in der Fuzzy-Theorie scharfe und unscharfe Menge und die Gültigkeit einer bestimmten Eigenschaft eines linguistischen Wertes (z. B. alt) für eine Grundmenge von Objekten.

- scharfe Mengen S, bei denen man für jedes Element x aus eine Grundmenge Ω eindeutig entscheiden kann, ob dieses zur Menge gehört oder nicht,
- unscharfe Mengen U, bei denen für jedes Element $x \in \Omega$ mit einem Grad $f_U(x)$ mit $0 \leq f_U(x) \leq 1$ zur Menge gehört. Für $f_U(x) = 0$ gehört x in Analogie zu klassischen Mengentheorie nicht zur Menge und bei $f_U(x) = 1$ gehört x nicht zur Menge. f_U heißt Zugehörigkeitsfunktion der unscharfen Mengen U.

Scharfe Mengen können damit als Spezialfall von unscharfen Mengen aufgefasst werden, wobei entweder $f_S(x) = 1$ (d.h. $x \in S$) oder $f_S(x) = 0$ (d.h. $x \notin S$) Bezeichnet nun U formal die Zuordnung von Objekten x aus dem Gegenstandsbereich zu einem Begriff der Informatischen Modellbildung, dann können einzelne Objekte auch zu einem Grad $f_U(x) = 0,8$ zu einer Klasse gehören. In Hinblick auf eine Präzisierung eines objektorientierten Spracherwerbs sind Aussagen wie

„Diese Dinge können alle fahren.“

als eine Zuordnung der Methode `fahren()` zu einer Auswahl von Objekten verstanden werden. Die Zugehörigkeit der Methode m zu einem Objekt x kann damit durch eine Zugehörigkeit $f_m(x)$ für alle Objekt x aus dem Gegenstandsbereich beschrieben werden.

Welchen Abstraktionsgrad die Objektauswahl in Hinblick auf eine Klasse `Fahrzeug` oder `GegenstandMitRaedern` beinhaltet, kann natürlich noch nicht aus der oben genannten Aussage abgeleitet werden. Die Zugehörigkeit $f_m(x)$ beinhaltet lediglich die Grad der Verbindung von x zur Methode m. Dass dieser Grad der Verbindung unscharf sein kann, zeigt das folgende plakative Beispiel:

Eine Schüler wählt z.B. die Ski aus einer gegebenen Menge von Objekten nicht aus, wenn dieser nach Objekten gefragt wird, mit denen man fahren.

> „Man sagt ja 'Ski fahren', aber irgendwie passt es auch nicht, weil die Ski keine Räder haben. Also so richtig zuordnen kann man das zu 'fahren' nicht."

In einem solchen Fall würde man in einem regelorientierten Beschreibung eine Inkonsistenz zwischen der Anwendung von Sprache und einer Klassifikationsaufgabenstellung durch die Objektauswahl zur Methode `fahren()` feststellen.

Eine exakte Methodendefinition steht aber in einem frühen Stadium der Modellierung nicht notwendigerweise Zeitpunkt zur Verfügung, um eine klare Entscheidung zu treffen, ob `meineSki.fahren()` durch ein Verb-Nomen-Konstruktion dem Objekt `meineSki` zugeordnet wird.

Ferner wird deutlich, dass in einer unscharfen Beziehung zwischen Objekt und Methode die Sprachentwicklung unterschiedlich stark ausgeprägt sein kann.

- Beziehung (x_1, x_2) von `meineSki` zu `fahren()` zu einem Zugehörigkeitsgrad $f_U(x_1, x_2) = 0,9$ mit einem kleinen Zweifel durch die Frage selbst, weil man Ski fahren sagt.

- Beziehung (x_2, x_1) von `fahren()` zu `meineSki` zu einem Zugehörigkeitsgrad $f_U(x_1, x_2) = 0,1$ mit sehr großen Bedenken, weil Ski keine Räder besitzen.

Dies erweiterte Beispiel zeigt auf elementarer Ebene, wie Methoden in einer objektorientierten Modellierungssprache selbst bidirektionale Beziehungen unterschiedlich stark im Lernprozess etabliert sind. Diese Unschärfe in Modellierungsbeziehungen können aber auch für bereits präzise abgrenzbare Klassen für die

Lernenden auftreten. Selbst wenn die Lernende z.B. die Klasse `Anhaenger` treffsicher und präzise aus einer Ansammlung von Objekten crisp abgrenzen können, könnten die Zuordnung der Methode `fahren()` zu der Klasse `Anhaenger` unscharf ausgebildet sein. In einer regelorientierten Formulierung kann der Lernende also nicht sprachlich präsisieren, ob man mit Objekten `fahren()` kann, die keinen eigenständigen Antrieb besitzt.

Mit der fuzzy-logischen Betrachtung ist keine methodische Bewertung verbunden, ob in Analogie zu Spracherwerb einer natürlichen Sprache eher der handlungsorientierte *Fremdsprachenzugang* oder der grammatik- bzw. regelorientierte *Fremdsprachenzugang* für das Erlernen einer künstlichen Sprache geeignet ist. Ferner darf die Fuzzy-Theorie auf dieser Ebene der Analogiebildung zwischen Erwerb der Fremdsprache und dem Erwerb einer künstlichen Sprache nicht in einem *quantitativen* Sinne verstanden werden, da es nicht das Ziel ist, die Werte von Zugehörigkeitsfunktionen in empirischen Untersuchungen konkret zu ermitteln, d. h. es ist in diesem Kontext nicht von Bedeutung, ob $f_U(x_1, x_2) = 0,9$ oder eher $f_U(x_1, x_2) = 0,6$ in einer Grundgesamtheit von Lernenden gilt. Die fuzzy-logische Betrachtung auf den Spracherwerb soll *rein qualitativ* den Blick für unscharfe Beziehungen in Objekt- und Begriffsnetzen öffnen und für den Lernprozess einen Ausdruck `meinAuto.fahren()` in eine bidirektional unterschiedlich stark ausgebildete Verb-Nomen-Konstruktion aufzuschlüsseln. In Analogie zum Erwerb der Fremdsprache wird nun durch Handlungserfahrungen mit Objekten und Prozessen erst die Präzisierung einer syntaktischen oder semantischen Regel in der künstlichen Sprache vorbereitet.

Dennoch könnte aufbauend auf diese theoretische Analyse eine fuzzy-logische Bewertung durch die Lernenden selbst als diagnostisches Instrument für den aktuellen Modellierungsstand für die Lehrperson eine hilfreiche Information für individuelle Hilfen verwendet werden.

ZADEH als Vater der Fuzzy-Logik (Zadeh, 1965) hat den Übergang von der klassischen Logik zur Fuzzy-Logik entwickelt, weil fast jeder Sprachgebrauch im Alltag Unschärfen besitzt und diese mit der klassischen Logik nicht adäquat beschrieben werden können. Im Spracherwerb bricht man in die Rigidität auf, mit der Compiler syntaktische und überprüfbare semantische Korrektheit von Programmen in einer Quellsprache fordern. Diese rigiden Strukturen ersetzt man durch unscharf ausgebildete Strukturen

4.2.5 Schriftspracherwerb

In der Schrifterwerbsforschung geht man von verschiedenen Stufen des Erwerbs von Schriftbeherrschungskompetenz aus (nach (Volmert, 2000)):

1. Frühe Stufe: Werden bei der Schriftsprache Schriftbilder als logographische Repräsentanten Wörter betrachtet, so wird in einem ganzheitlichen Sinne eine Brücke zwischen der etablierten Verbindung von Objekten zu beschreibenden Begriffen mit einem geschriebenen Wort als symbolische Einheit hergestellt.

   Bei einer erlernten Verbindung (Fuzzy-Grad der Beziehung) werden Objekte als Vertreter einer Klasse sowohl mit dem Begriff für die Klasse als auch mit dem formalen symbolischen Beschreibung (Schrift) verbunden.

   Dabei müssen nicht notwendigerweise alle beschreibenden Eigenschaften vollständig geklärt sein und ein Anhänger mit 4 Rädern könnte fälschlicherweise der Klasse `Personenwagen` zugeordnet werden, weil die definierende Eigenschaften nur unvollständig und unscharf ausgebildet sind und als definierende Eigenschaft lediglich die Anzahl der Räder verwendet wird. Regelorientiert könnte man die definierenden Eigenschaften einer syntaktischen Struktur vorgeben und nennen. Allerdings würde die Lehrperson dann der eigentlichen Kompetenzentwicklung entgegengewirkt, nämlich der eigenständigen Entwicklung von individuellen nicht-normativen Modellierungsergebnissen .

   In einem ganzheitlichen Ansatz werden die Objekte und die klassifizierenden Eigenschaften von den Modellierenden festgelegt und das Modellierungsergebnis zeigt durch das objektorientierte Sprachverständnis, wie der Gegenstandsbereich und das analysierte Objektnetz wahrgenommen wird. Im informatikdidaktischen Kontext werden so die analysierten Objekte im Gegenstandsbereich nicht als gegebene Zusammensetzungen aus ihren einzelnen definierenden Eigenschaften gesehen.

2. Erst weitere analytische Tätigkeit im Objektnetz erfasst tiefere Teilstrukturen sowohl bei klassifizierenden Attributen als auch bei Teilprozessen. Beim Erwerb der Schriftsprache wird das Inventar der Buchstaben zunächst in seiner Bedeutung für Laute identifiziert und dieses unscharfe Regelsystem an bekannten Wörtern überprüft, um es später auch synthetisch für das Schreiben neuer auditiv wahrgenommenen neuen Wörtern im Schrift-

und Sprachsystems verwenden zu können. Übertragen auf die objektorientierte Modellbildung entsprechen Grapheme in der Schriftsprache ikonischen und/oder symbolischen Repräsentationen für Klassen, Objekte, Methoden, Prozesse, Beziehungen zwischen Objekt und Objekt, Beziehungen zwischen Objekt und Klasse und die Beziehung zwischen Klassen.

Problematisch für den Lernprozess ist allerdings dabei, dass UML diese Trennung von Objekt und Klasse in den einzelnen UML-Diagrammformen keinen zentralen Stellenwert in einem Erkenntnisprozess für die Objektorientierung einräumt, der bei einem informatikdidaktischen Blick auf den Spracherwerb notwendig wäre. Im Sinn der Fuzzy-Theorie muss in dem Lernprozess eine gewisse Trennschärfe für die charakterisierenden Objekteigenschaften erarbeitet werden, damit klassifizierende Eigenschaften erkannt und in der künstlichen Modellierungssprache verbalisiert werden können. Die Didaktisierung der Objektdarstellung in Diagrammen der Sprachfamilie UML unter der Maßgabe „Objects strictly first“ führt zu Entscheidungen bezüglich der Unterstützung dieser Lernprozesse:

(a) keine Angabe des Klassenbezeichners, da Klassen zu Beginn der didaktisch vorgenommenen Strukturierung nicht bekannt sind und

(b) Abrundung der Ecken, damit später Objekte und Klassen allein durch die graphische Darstellung unterschieden werden können. In allen anderen UML-Diagrammen werden Objekte folgend mit abgerundeten Ecken dargestellt;

(c) Sequenzdiagramme ausschließlich auf Objektebene;

(d) Aktivitätsdiagramme werden nicht benutzt, da für Abläufe (sowohl Prozesse wie auch später Methoden) die Beschreibung mittels Struktogramm erfolgen

Auch wenn den Spracherwerb in Lernumgebungen zur OOM nicht von den Zielen der OOM vollständig zu trennen sind, so wurde an dieser Stelle ein theoretischer Blickwinkel eingenommen, um für das Thema zu sensibilisieren.

4.2.6 Spracherwerb und OOM-Kompetenzen

An einem plakativen Beispiel sollte die Trennung von Spracherwerb und OOM-Kompetenzen verdeutlicht werden.

(a) Ein Lernender kann zufällig handeln und im Kontext seiner Handlungen zufällige Handlungssequenzen generieren, die mit sinnvollen sprachlichen Verbalisierungen deskriptiv belegt werden können (Sprache mit Handlungen assoziieren). Wenn der Lernede zufällig sinnvoll durch Sprache benennbare Handlungssequenzen erzeugt, so kann man damit keine direkte Lösungsabsicht für eine Problemstellung unterstellen. Dennoch erlernt er Begriffe im Kontext spezifischer Handlungssequenzen ohne eine übergeordnete Lösungsabsicht.

(b) Wenn Lernende in mehreren unterschiedlichen Objektszenarien in der Lage sind, Objekte nach unverbalisierten Ordnungsprinzipien zu Einheiten zusammenfassen, liegt die Einschätzung nache, dass diese Lernenden eine zentrale klassifizierende Eigenschaft immer wieder korrekte in konsistenter Weise identifzieren können. Dennoch ist es im Einzelfall ggf. für die Lernenden sehr schwierig, diese erkannten Strukturen in eigenen Muttersprache zu verbalisieren. Damit sind auch die Voraussetzung für die Übersetzung in eine („künstliche“) Modellierungssprache nicht gegeben.

4.3 Schlussfolgerungen – Thesen

In den vorausgegangenen Abschnitten wurden theoretisch Analogien zwischen dem Spracherwerb in der Angewandten Linguistik und dem Erlernen einer Modellierungssprachen gezogen. Diese Grundlagen beziehen sich explizit auf die Anfänge des Erwerbs einer Modellierungssprache, in der die Fuzzy-Logik als methodisches Werkzeug verstanden werden kann, um den Reifegrad (maturity) der Sprachentwicklung für die Modellentwicklung beschreibar zu machen.

(a) Betrachtet man UML als Sprachfamilie und das Objektorientierte Modell als Systembeschreibung, so muss der Lernende Systembeschreibungen aus einer Repräsentationsform (das kann auch ein Rollenspiel sein) in eine andere Repräsentationsform übertragen.

(b) Die Rigidität, mit der Compiler die syntaktische Korrektheit bis zu Lieferung der Übersetzung fordern, sind für die Informatische Modellierung in der Didaktik nicht besonders hilfreich. Viel wichtiger wäre es, unscharfe Beschreibungen (Babbling) in einer Adhoc-Systembeschreibung

in eine unscharfe Repräsentationsform in einer Zielsprache zu übersetzen und die Unschärfe für Modellbildner sichtbar zu machen.

(c) In einem intuitiven ersten Zugang spielt die Alltagssprache eine deutlich größere Rolle, als die noch zu entwickelnden Kompetenzen einer rigiden künstliche Sprache. Die Sprachübersetzung eines Compilers hat das Ziel, Programme einer Quellsprache in funktional äquivalente Programme einer Zielsprache zu übertragen. Mit der Alltagssprache oder auch Rollenspielen als Quellsprache muss man unscharfe Beschreibungen in funktional äquivalente Programme einer rigiden Zielsprache überführen. Dies ist informatisch nicht eindeutig lösbare Aufgabe für einen Compiler, aber die Aufgabe eines informatikdidaktisch gestalteten Lernprozesses.

(d) Mit Sprachlernen vs. Spracherwerb verhält es sich im Prinzip wie mit ICT-Vermittlung vs. Informatikunterricht (bzw. anwendungsbezogenem Wissen vs. informatischer Literalität).

   i. Sprachlernen = Lernen von festen Ausdrücken oder starren Regeln. Das führt maximal zu BICS[4] aber niemals zu CALP[5], die aber durch Sprachunterricht angestrebt wird. Kann es auch gar nicht, da die Regeln/Ausdrücke nicht das implizite (unbewusste) Regelsystem der Sprache eingehen.
   entspricht im Wesentlichen

   ii. ICT/anwendungsbezogenes Wissen = Lernen von festen Klickfolgen in Standard-Anwendungssoftware. Führt allenfalls zu einfacher Bedienfähigkeit, aber niemals zu Kompetenz (vgl. oben).
   Im Gegensatz dazu:

   iii. Sprachererwerb (führt im Gegensatz zu Sprachlernen zur langfristigen Entwicklung von CALP)
   entspricht im Wesentlichen

   iv. Aufbau von Modellierungskompetenz/informatischer Literalität durch objektorientierte Sprachkompetenz – analog zur CALP.

[4] BICS Basic Interpersonal Communicative Skills Manifestation von Sprache im unmittelbaren persönlichen Austausch â¼ Mündlichkeit

[5] CALP=Cognitive Academic Language Proficiency, bedeutet, dass Sprache in dekontextualisierten 'akademischen' Situationen als Schriftsprache verwendet werden kann.

Kapitel 5

Fazit, Grenzen und Ausblick

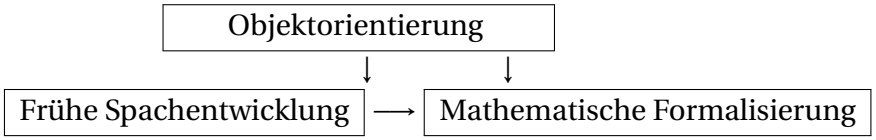
Die Objektorientierung als

- Modellierungsmethode,
- Analysemethode und
- Implementierungsstruktur in Programmiersprachen

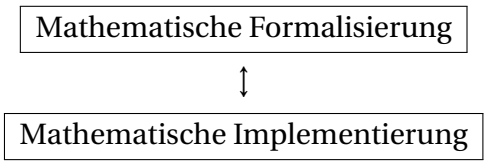
hat neben der fachlichen Bedeutung in der Informatik auch in der Didaktik der Informatik als Problemlösestrategie einen zentralen Platz. Aus den fachinformatischen Erwägungen, Software effizient zu entwickeln und gut wartbare Softwareprodukte durch die Objektorientierung zu erzeugen, ergibt sich auch eine notwendige Kompetenz der Softwareentwickler, diesen Anforderungen zu genügen. Ein wesentlicher Aspekt für effiziente Wartung ist es, Beziehungen zwischen dem Anwendungsgebiet der Software und der internen Struktur der Software bei Problemlösungen, Optimierungen und Weiterentwicklungen herzustellen (siehe Tabelle 1.3, S. 30 zu den Kosten der Fehlerbehebung).

Die in Abschnitt 1.1.3 (vgl. S. 16 ff.) genannte informatische Literalität beinhaltet im Rahmen der Kompetenzentwicklung auch das Bildungsziel, die lernenden Modellierenden in die Lage zu versetzen bidirektionale Beziehungen zwischen Realität und individueller Sichtweise des Lernenden auf das Modellierungsprodukt zu herzustellen. Die Lehrenden müssen in diesem Zusammenhang die Wahrnehmungsfähigkeiten der modellierenden Lernenden identifizieren können und individuelle Hilfen für die Unterstützung der analytischen und konstruktiven Kompetenzen in der Objektorien-

tierung aus den Lernvoraussetzungen ableiten. Die diagnostischen Strategien der Lehrenden beschränken sich dabei aber nicht notwendigerweise aus das Modellierungsprodukt allein, denn auch die sprachlichen Erläuterungen zum Modellierungsprodukt stellen einen wesentlichen Bestandteil für die Analyse der Modellierungskompetenz dar.



In dem Abschnitt 3 (vgl. S. 87 ff.) wird eine mathematische Formalisierung der Objektorientierung erarbeitet. Betrachtet man im Vergleich dazu das Kapitel 4 (vgl. S. 135 ff.) über die frühe sprachliche Entwicklung einer objektorientierten Sprache, so stellen die beiden Kapitel die jeweiligen Inhalte Extremstellen in der sprachlichen Entwicklung von künstlichen Sprachen dar. Wie Lehrende lernen können in geeigneter Weise in der Umgangssprache objektorientierte Strukturen zu entdecken und die Weiterentwicklung in Richtung eine objektorientierten Modellierungsprache zu fördern, bleibt eine in bisher unbeantwortet Frage in der Hochschuldidaktik an der Schnittstelle zwischen Mathematik und Informatik.



In Analogie zur Implementierung von objektorientierten Modellen in einer Programmiersprache sind auch in der Mathematik Implementierungen (z.B. in Computeralgebrasystemen CAS) wesentlich, um Modelle in Simulationen zu überpüfen oder in einem Anwendungskontext zu validieren.

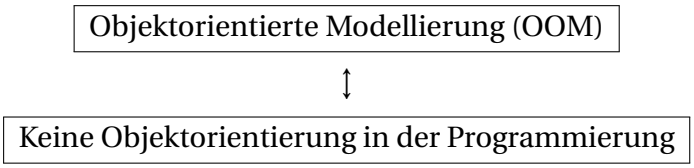
Eine solche Strukturäquivalenz zwischen Modellierung und Implementierung zeigt die Bedeutung der wissenschaftlichen Vorarbeiten in der Didaktik der Informatik selbst für didaktische Modellierungsansätze in der Mathematik. Funkional Ã¤quivalent zu dem objektorientierten Modell benötigt man eine Ausformulierung des mathematischen Modells als ausführbares Produkt in einer Programmier- bzw. Skriptsprache. Das Grundkonzept der Objektorientierung wird von Programmiersprachen mit unterschiedlicher Qualität unterstützt. Die syntaktischen Rahmenbedingung

von Skriptsprachen in gängiger in der Mathematik eingesetzter Software (z.B. Octave, Maxima, Geogebra, LibreOffice, ...) unterstützt die Objektorientierung nur unzureichend. Auch wenn die Softwarevoraussetzungen bzgl. der Programmiersprachensyntax keine sehr transparente Implementierung der objektorientierten Ideen ermöglicht, so kann man zum jetzigen Zeitpunkt mit diesem Defizit nur in einer Weise umgehen, dass man auch in einer solchen Skriptsprachen eine maximale Umsetzungen und Sichtbarkeit der objektorientierten Modellierungsergebnisse anstrebt. In einem ersten Schritt wurde in der Syntax von Javascript in Abschnitt 3.7 (vgl. S. 122 ff.) gezeigt, wie man diese unvollständige bzw. eine ungeeignete Unterstützung des objektorientierten Ansatzes erweitert kann. Die Nähe zu JAVA als objektorientierte Sprache macht diesen dargestellte Schritt etwas einfacher. Der Aufwand für Skriptsprachen in gängiger Mathematiksoftware ist dabei ungleich größer, aber im Sinn einer besseren Sichtbarkeit der Objektorientierung in dem Programmiercode lohnend, da man die in einem Modellierungszyklus die immer wiederkehrenden Beziehungen zwischen Modellierungsergebnis und Programmierzeilen in der Regel sehr oft herstellen muss.

Die Mathematik hat bisher keine eigenständige syntaktische Strukturen für die Objektorientierung erarbeitet, die sich als mathematische Formalisierung für objektorientierte Modelle eignen. Auf der Grundlage der 3D Modellierung 2 (vgl. S. 58 ff.), den Transformationen von Objekten im dreidimensionalen Raum zeigen die Schnittstellen zwischen der Modellierung mathematischer Objekte und deren Implementierung im einer 3D-Software.

Bezogen auf die mathematische Beschreibung können sicherlich auch die objektorientierten Ergebnisse eines Modellierungsprozesses auch funktional äquivalent über Mengen und Abbildungen realisiert werden. Betrachtet man die objektorientierte Struktur des Modellierungsergebnisses auch selbst als didaktisch wertvolles Resultat, dann kann man durch Analogiebildung zur Didaktik der Informatik daraus die Zielsetzung ableiten, auch diese objektorientierte Struktur in der mathematisch-formalen Beschreibung ebenfalls sichtbar zu machen. Dies gilt insbesondere in einer zyklischen Optimierung eines mathematischen Modells, das wiederkehrende Bezüge zwischen visueller Darstellung der 3D-Objekte und der mathematischen Implementation in einer Skriptsprache notwendig macht. Insgesamt

beschäftigen sich die Lehrenden und Lernenden mit didaktischen Fragestellungen zu Transformationsprozessen zwischen beschreibenden Sprachen für ein objektorientiert modelliertes System. Die Umsetzung der Objektorientierung in einer formalen Sprache stellt damit lediglich ein Wechsel eine Wechsel von einer sprachlichen Beschreibung in eine andere abstraktere Darstellungsform dar.



Wenn sich die Mathematik Computern in Numerik, Algebra und Geometrie bedient, um mathematische Modelle in einer Softwareumgebung zu untersuchen, wird neben der Formalisierung in einer mathematische Sprache auch wieder die Implementation von objektorientierten Strukturen in Programmier- oder Skriptsprache notwendig. Das dies natürlich auch ohne eine objektorientierte Programmiersprache möglich ist, haben die Beispiele aus Kapitel 2 (vgl. S. 58 ff.) gezeigt und die Anwendung von 3D-Software, Dynamische Geometriesoftware (wie z.B. Geogebra), auch Computer-Algebra- System CAS (wie z.B. Maxima) oder Numeriksoftware (wie z.B. Octave) können ebenfalls analog zu einem Compiler/Sprachübersetzer mit der Aufgabe verbunden werden, sprachliche Formulierungen aus einer Quellsprache und eine funktional äquivalente Formulierung in einer Zielsprache zu übertragen.

Zunächst einmal scheint es daher sinnvoll, für die Einführung der objektorientierten Modellierung auch im mathematischen Kontext mit Programmier- oder Skriptsprachen einzusetzen, die das Konzept der Objektorientierung besonders deutlich machen.

Da die oben genannten Softwaretypen für die Mathematik objektorientierte Strukturen im Kern auf der syntaktischen Ebene nicht unterstützen, muss man sich im schulischen Kontext mit den Skriptsprachen auseinander setzen, die verfügbar sind. In Analogie zu konkreten Softwareprojekten kann man auch im schulischen Kontext nicht notwendig davon ausgehen, dass eine Programmier- oder Skriptsprache frei wählbar ist (z.B. bei der Erweiterung eines bestehenden Programmpaketes).

Das hier nun vorgestellte mathematisches Modells der Objektorientierung bietet die Möglichkeit, objektorientierte Strukturen in einer mathematischen programmiersprachenunabhängigen Weise formal mit mathematischer Syntax zu beschreiben, bevor diese mit einer im Sinne der Objektorientierung ungeeigneten Programmier- oder Skriptsprachensyntax implementiert und aus einem mathematischen Blickwickel die Modellierungsergebnisse untersucht werden können. Auf diese Weise kann man zusätzlich auch ungeeigneten Programmiersprachen einen objektorientierten „Stempel“ aufzudrücken. Am Beispiel von JavaScript, dessen Syntax keine Vererbung unterstützt, wird die Umsetzung objektorientierter Ideen veranschaulicht. Dennoch bleibt die Aufgabe, zumindest für die OpenSource-Software im schulischen Kontext, objektorientiert syntaktische Hilfskonstruktion zu entwickeln, bis die Objektorientierung auch mit einer neue Softwareversion in diesen Skriptsprachen zur Verfügung steht.

Aber auch ohne diese programmiertechnischen Aspekte kann die mathematische Modellbildung von der objektorientierten Modellbildung von der Didaktik der Informatik profitieren.

Dieses gilt natürlich nur unter der Voraussetzung, dass nicht das normative Vorhandensein von mathematischen Objekten in eine Modellierungsprodukt im Mittelpunkt stehen, sondern die Strukturierung von Objekten zusammen mit der Beschreibung deren Interaktionen. An der Schnittstelle zwischen Mathematik und Informatik können diese entstandenen Objektbeschreibungen mit unterschiedlichen disziplinären formaleren Sprachebenen weiter präzisiert werden.

Die in diesem Buch behandelten Kapitel stellen an der Schnittstelle zwischen Didaktik der Informatik und Didaktik der Mathematik allerdings nur die ersten Schritte der Objektorientierung aus einigen ausgewählten Blickwinkeln dar. Insbesondere die Synergieeffekt abgestimmter Lernumgebungen zwischen Mathematik und Informatik bietet aus didaktischer Sicht sicherlich noch zahlreiche Möglichkeiten, in zyklischen Entwicklungs-, Validierungs- und Optimierungsschritten die Förderung objektorientierter Modellierungskompentenzen bei den Lernenden zu verbessern und weitere stoffdidaktische Erfahrungen zu sammeln.

Evolutionäre Publikationen und OpenContent können die Entwicklungszyklen selbst auf die Dokumentenentwicklung diese Bandes übertragen. In der Zukunft wird sich zeigen, ob sich dieses in der Einleitung genannte

Konzept für einzelne Kapitel oder sogar für den gesamten Band als tragfähigen Weg unter einer Creative Commons Lizenz erweisen wird.

Engelbert Niehaus

Literatur

Bauersfeld, H., Otte, M. & Steiner, H. G. (Hrsg.). (1977). *Informatik im Unterricht der Sekundarstufe II: Grundfragen, Probleme und Tendenzen mit Bezug auf allgemeinbildende und berufsqualifizierende Ausbildungsgänge* (Nr. 15 (Band I) und 16 (Band II)). Bielefeld: Universität. (Arbeitstagung: Bielefeld 12.–14. September 1977)

Beer, H. T. de. (2006). *The History of the ALGOL Effort.* Master's Thesis in Computer Science and Engineering – Technische Informatica, University of Technology, Eindhoven. Zugriff am 21st February 2012 auf `http://www.heerdebeer.org/ALGOL/The_History_of_ALGOL.pdf`

Boettcher, D., Grabowsky, A., Humbert, L., Poth, O., Pumplün, C. & Schulte, J. (2007, September). Ada – dieser Zug hat Verspätung. In S. Schubert (Hrsg.), *Informatik und Schule – Didaktik der Informatik in Theorie und Praxis – INFOS 2007 – 12. GI-Fachtagung 19.–21. September 2007, Siegen* (S. 217–228). Bonn: Gesellschaft für Informatik, Köllen Druck + Verlag GmbH.

Brauer, W. & Brauer, U. (1992). Wissenschaftliche Herausforderungen für die Informatik: Änderung von Forschungszielen und Denkgewohnheiten. In W. Langenheder, G. Müller & B. Schinzel (Hrsg.), *Informatik cui bono?* (S. 11–19). Berlin: Springer. Zugriff am 6. Februar 2012 auf `http://www.model.in.tum.de/um/courses/datamining/WS0405/Wissen.ps`

Brüderlin, B. & Meier, A. (2001). *Computergrafik und geometrisches modellieren.* Stuttgart: Teubner.

Catalina, M. (2006). *Entwicklung einer mittelstufen-ag informatik.* (Seminararbeit und Praktikumsbericht; verfügbar unter http://www.afiller.de/oom)

Chomsky, N. (1965). *Aspects of the Theory of Syntax.* Cambridge, MA: MIT Press.

Denning, P. J. (Hrsg.). (1989, December). *A debate on teaching computing science.* ACM. (Comm. ACM 32 (1989), Nr. 12)

Denning, P. J. (2003, November). Great Principles of Computing. *Comm. ACM*, *46* (11), 15–20. Zugriff am 5th February 2012 auf `http://cs.gmu.edu/cne/pjd/PUBS/CACMcols/cacmNov03.pdf`

Denning, P. J. (2005, April). Is Computer Science Science? *Comm. ACM*, *48* (4), 27–31. Zugriff am 5th February 2012 auf `http://cs.gmu.edu/cne/pjd/PUBS/CACMcols/cacmApr05.pdf`

Denning, P. J., Comer, D. E., Gries, D., Mulder, M. C., Tucker, A., Turner, A. J. et al. (1989, January). Computing as a discipline. *Comm. ACM*, *32* (1), 9–23. Zugriff am 5th February 2012 auf `http://cs.gmu.edu/cne/pjd/GP/CompDisc.pdf`

Desel, J. & Klenze, L. von. (2010). AMSEL – ein Lernsystem zum Algorithmenentwurf. In M. Kerres, N. Ojstersek, U. Schoeder & U. Hoppe (Hrsg.), *DeLFI 2010 – 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V. – 12.–15. September 2010 an der Universität Duisburg-Essen.* Bonn: Köllen Druck + Verlag GmbH.

Diethelm, I. (2007). *Strictly models and objects first – Unterrichtskonzept und -methodik für objektorientierte Modellierung im Informatikunterricht.* Dissertation (dr. rer. nat.), Universität – Fachbereich Elekktrotechnik/Informatik, Kassel. Zugriff am 27. Juni 2008 auf `http://deposit.d-nb.de/cgi-bin/dokserv?idn=98668760x&dok_var=d1&dok_ext=pdf&filename=98668760x.pdf`

Dijkstra, E. W. (1968, March). Go To Statement Considered Harmful. *Comm. ACM*, *11* (3), 147–148.

Filler, A. (2007a). *Einbeziehung von elementen der 3d-computergrafik in den mathematikunterricht der sekundarstufe ii im stoffgebiet analytische geometrie.* VDM Verlag. (Online-Ausgabe: edoc Dokumenten- und Publikationsserver der Humboldt-Universität: http://edoc.huberlin.de/)

Filler, A. (2007b, 2). Herausarbeiten funktionaler und dynamischer aspekte von parameterdarstellungen durch die erstellung von computeranimationen. *Mathematische Semesterberichte, 54*, 155-176.

Floyd, C., Mehl, W.-M., Reisin, F.-M., Schmidt, G. & Wolf, G. (1987). *SCANORAMA – Methoden, Konzepte, Realisierungsbedingungen und Ergebnisse von Initiativen alternativer Softwareentwicklung und -*

gestaltung in Skandinavien (MAGS, Hrsg.) (Nr. 30). MAGS – Ministerium für Arbeit, Gesundheit und Soziales der Landes Nordrhein-Westfalen. (Landesprogramm »Mensch + Technik – Sozialverträgliche Technikgestaltung«)

Foley, J. D., Van Dam, A., Feiner, S. K., Hughes, J. F. & Phillips, R. L. (1994). *Grundlagen der computergraphik*. Bonn: Addison-Wesley.

GI. (2008, April). *Grundsätze und Standards für die Informatik in der Schule – Bildungsstandards Informatik für die Sekundarstufe I* (Bd. 27) (Nr. 146/147). Zugriff am 5. Mai 2012 auf `http://fa-ibs.gi.de/fileadmin/gliederungen/fb-iad/fa-ibs/Empfehlungen/bildungsstandards_2008.pdf` (Erarbeitet vom Arbeitskreis »Bildungsstandards« – Beschluss des GI-Präsidiums vom 24. Januar 2008 – veröffentlicht als Beilage zur LOG IN 28 (2008) Heft 150/151)

Goldberg, A. & Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison Wesley. Zugriff am 2nd July 2011 auf `http://wiki.squeak.org/squeak/64`

Goldman, R. (2009). *An integrated introduction to computer graphics and geometric modeling*. CRC Press.

Grimm, H. (o. J.). *Sprachentwicklungstest fÃ¼r zweij ahrige Kinder (SETK-2). Diagnose rezeptiver und produktiver Sprachverarbeitungsf ahigkeiten*. Göttingen, year = 2000,.

Haenselmann, T. (1996). *Raytracing*. Bonn: Addison-Wesley.

Haug, R. & Holzäpfel, L. (Hrsg.). (2011). *Beiträge zum Mathematikunterricht (BzMU). Vorträge auf der 45. Tagung für Didaktik der Mathematik der Gesellschaft für Didaktik der Mathematik (GDM) vom 21.02.2011 bis 25.02.2011 in Freiburg*. Münster: Verlag für wissenschaftliche Texte und Medien (WTM). Verfügbar unter `http://is.gd/6CtHVe` (Band 1 und 2)

Hesse, W., Barkow, G., Braun, H. von, Kittlaus, H.-B. & Scheschonk, G. (1994, Februar, April). Terminologie der Softwaretechnik, Ein Begriffssystem für die Analyse und Modellierung von Anwendungssystemen, Teil 1: Begriffssystematik und Grundbegriffe, Teil 2: Tätigkeits- und ergebnisbezogene Elemente. *Informatik Spektrum, 17* (1, 2), 39–47, 96–105.

Hesse, W. & Braun, H. von. (2001, September). Wo kommen die Objek-

te her? Ontologisch-erkenntnistheoretische Zugänge zum Objektbegriff. Workshop »Erkenntnistheorie-Semiotik-Ontologie (ESO): Die Bedeutung philosophischer Disziplinen für die Softwaretechnik«. In K. Bauknecht, W. Brauer & T. Mück (Hrsg.), *Informatik 2001: Wirtschaft und Wissenschaft in der Network Economy? Visionen und Wirklichkeit – Tagungsband* (S. 776–781). Berlin: Springer. Zugriff am 28. November 2011 auf `http://is.gd/Bux56a` (2 Bde.)

Humbert, L. (2006). *Didaktik der Informatik – mit praxiserprobtem Unterrichtsmaterial* (2., überarbeitete und erweiterte Aufl. Aufl.). Wiesbaden: B.G. Teubner Verlag. Zugriff am 29. Januar 2012 auf `http://humbert.in.hagen.de/ddi/`

Kerres, M., Ojstersek, N., Schoeder, U. & Hoppe, U. (Hrsg.). (2010). *DeLFI 2010 – 8. Tagung der Fachgruppe E-Learning der Gesellschaft für Informatik e.V. – 12.–15. September 2010 an der Universität Duisburg-Essen* (Nr. P-169). Bonn: Köllen Druck + Verlag GmbH.

Klaeren, H. & Sperber, M. (2007). *Die Macht der Abstraktion. Einführung in die Programmierung* (1. Aufl. Aufl.). Wiesbaden: B. G. Teubner.

Koehne, P. (2011). *Objektrollenspiele und Parameter – Konzept zur Entwicklung fachlich korrekter Sichten.* Hausarbeit gemäß OVP, Studienseminar für Lehrämter an Schulen – Seminar für das Lehramt für Gymnasien/Gesamtschulen, Hamm. Zugriff am 15. Juni 2012 auf `http://www.ham.nw.schule.de/pub/bscw.cgi/2281263`

Krämer, S. (2011, Juli). Simulation und Erkenntnis. Über die Rolle computergenerierter Simulationen in den Wissenschaften. In T. Lengauer (Hrsg.), *Computermodelle in der Wissenschaft – zwischen Analyse, Vorhersage und Suggestion* (Bd. 110, S. 303–322). Stuttgart: Wissenschaftliche Verlagsgesellschaft mbH. Zugriff am 25^{th} February 2012 auf `http://www.leopoldina.org/NovaActa_Band_110/epaper/Band110_Nr_377_Computermodelle.pdf`

Krashen, S. D. & Terrell, T. D. (1983). *The natural approach: Language acquisition in the classroom.* The Alemany Press, P.O. Box 5265, San Francisco, CA 94101. Verfügbar unter `http://www.eric.ed.gov/ERICWebPortal/detail?accno=ED230069`

Kuhaupt, J. (2012). *Implementierung von Methoden als Bestandteil einer strikten objektorientierten Modellierung.* Hausarbeit gemäß OVP, Zentrum für schulpraktische Lehrerausbildung – Seminar für das Lehramt für Gymnasien Gesamtschulen, Hamm. Zu-

griff am 16. Juni 2012 auf http://www.ham.nw.schule.de/bscw/bscw.cgi/3122295

Lengauer, T. (Hrsg.). (2011, Juli). *Computermodelle in der Wissenschaft – zwischen Analyse, Vorhersage und Suggestion* (Bd. 110) (Nr. 377). Stuttgart: Wissenschaftliche Verlagsgesellschaft mbH. Zugriff am 25th February 2012 auf http://www.leopoldina.org/NovaActa_Band_110/epaper/Band110_Nr_377_Computermodelle.pdf

Maaß, K. (2005). Was sind Modellierungskompetenzen? In GDM (Hrsg.), *Beiträge zum Mathematikunterricht (BzMU) – Vorträge auf der 39. Tagung für Didaktik der Mathematik vom 28.2. bis 4.3.2005 in Bielefeld* (S. 364–367). Hildesheim und Berlin: Verlag Franzbecker. Zugriff am 22. Januar 2012 auf http://www.mathematik.tu-dortmund.de/ieem/cms/media/BzMU/BzMU2005/Beitraege/maass-gdm05.pdf (GDM – Gesellschaft für Didaktik der Mathematik)

Miller, J. A. (2004). *Promoting Computer Literacy Through Programming Python*. Dissertation, The University of Michigan. Zugriff am 7th February 2012 auf http://www.python.org/files/miller-dissertation.pdf

Möller, K.-H. & Paulish, D. J. (1993). *Software-Metriken in der Praxis*. Oldenbourg.

Neumann, I., Heinze, A., Ufer, S. & Neumann, K. (2011). Modellieren aus mathematischer und physikalischer Perspektive. In R. Haug & L. Holzäpfel (Hrsg.), *Beiträge zum Mathematikunterricht (BzMU). Vorträge auf der 45. Tagung für Didaktik der Mathematik der Gesellschaft für Didaktik der Mathematik (GDM) vom 21.02.2011 bis 25.02.2011 in Freiburg* (S. 603–606). Münster: Verlag für wissenschaftliche Texte und Medien (WTM). Verfügbar unter http://is.gd/6CtHVe (Band 1 und 2)

Parnas, D. L. (1972, December). On the Criteria To Be Used in Decomposing Systems into Modules. *Comm. ACM*, *15* (12), 1053–1058.

Pólya, G. (1945). *How to Solve It: A New Aspect of Mathematical Method*. Princeton, NJ: Princeton University Press.

Putnam, H. (1999). *Repräsentation und realität*. Frankfurt a. M.: Suhrkamp.

Rescorla, L., Mirak, J. & Singh, L. (2000). Vocabulary growth in late talkers: Lexical development from 2,0 to 3,0. *Journal of Child Language*, *27*,

293–311.

Rogerson, A. (Hrsg.). (2002, September). *Proceedings of the International Conference »The Humanistic Renaissance in Mathematics Education« – The Mathematics Education into the* 21^{st} *Century Project, Terrasini, Palermo, Italy.* Verfügbar unter `http://math.unipa.it/~grim/palermo2002.htm`

Royce, W. W. (1970, August). Managing the Development of Large Software Systems. In *Proceedings, wescon* (S. 1–9 (328–338)). TRW (The Institute of Electrical and Electronics Engineers Inc.—IEEE). Zugriff am 28^{th} February 2012 auf `http://facweb.cs.depaul.edu/jhuang/is553/Royce.pdf`

Shirley, P. (2005). *Fundamentals of computer graphics.* A K Peters.

Shirley, P. & Morley, R. K. (2000). *Realistic ray tracing.* A K Peters, Natick, Massachusetts.

Stachowiak, H. (1973). *Allgemeine modelltheorie.* Springer.

Thomas, M. (2002). *Informatische Modellbildung – Modellieren von Modellen als ein zentrales Element der Informatik für den allgemeinbildenden Schulunterricht.* Dissertation, Universität Potsdam Didaktik der Informatik. Zugriff am 4. Januar 2012 auf `http://ddi.cs.uni-potsdam.de/Personen/marco/Informatische\_Modellbildung\_Thomas_2002.pdf`

Volmert, J. (Hrsg.). (2000). *Grundkurs Sprachwissenschaft* (4. Auflage Aufl.). München.

Watt, A. (2002). *3d-computergrafik.* München: Pearson Education.

Wirth, N. (1971, April). Program Development by Stepwise Refinement. *Comm. ACM*, *14* (4), 221–227.

Wirth, N. (2002, June). Computer Science Education: The Road Not Taken. Opening address at ITiCSE conference, Aarhus, Denmark. *SIGCSE BULLETIN*, *34* (3), 1–3. Zugriff am 27^{th} February 2012 auf `http://www.softwareschule.ch/download/wirthcomputing_education.pdf` (ITiCSE – Innovation and Technology in Computer Science Education)

Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, *8*, 338–353.