Ein Objektmodell zur Repräsentation und Wiederverwendung verfahrenstechnischer Prozeßmodelle

Von der Fakultät für Maschinenwesen
der Rheinisch–Westfälischen Technischen Hochschule Aachen
zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
genehmigte Dissertation

vorgelegt von

Diplom–Informatiker
Markus Baumeister
aus Münster/Westf.

Berichter:
Universitätsprofessor Dr.-Ing. Wolfgang Marquardt
Universitätsprofessor Dr. Matthias Jarke

Tag der mündlichen Prüfung: 14.12.2000

Abstract

In der Verfahrenstechnik wird die Entwicklung neuer und die Verbesserung bekannter chemischer Prozesse häufig durch eine mathematische Modellierung derselben unterstützt. Das Erstellen dieser Modelle ist aufgrund der notwendigen Abstraktion selbst ein Entwurfsprozeß, dessen Artefakte zur Beschleunigung und Qualitätsverbesserung nach Möglichkeit wiederverwendet werden sollen. Bei der Wiederverwendung ergeben sich die bekannten Probleme des Suchens nach und der Adaption von Modellteilen. Letzteres wird in der Verfahrenstechnik dadurch verschärft, daß die den Prozessen zugrundeliegenden Anlagen häufig Einzelstücke sind. Vorhandene Ansätze zur Modellierung — sowohl inner- als auch außerhalb der Verfahrenstechnik — können beide Probleme nicht gleichzeitig lösen.

Die vorliegende Arbeit betrachtet darum mögliche Objektmodelle und darauf benötigte Operationen, um, ausgehend von der Modellierungssprache VeDa, ein Objektmodell zu entwickeln, das die Speicherung und Wiederverwendung verfahrenstechnischer Prozeßmodelle unterstützt. Es kann dabei gezeigt werden, daß sich die jeweils eines der obigen Probleme vereinfachenden klassenbasierten bzw. prototypbasierten Paradigmen nicht durch Kombination vereinigen lassen. Ihre Eigenschaften werden darum durch eine Emulation prototypbasierten Verhaltens auf Grundlage eines klassenbasierten Modells zusammengeführt. Hierbei wird die Zahl der entstehenden Vereinigungsklassen durch einen Mechanismus zur multiplen Instanziierung, Aspekte genannt, begrenzt. Um Metaklassen kohärent behandeln zu können, wird ein Verfahren vorgestellt, mit dem sich alle benötigten Funktionen der Metaebenen auf ein zwei–ebeneniges Objektmodell abbilden lassen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Entwurfsprozesse stellen wegen ihres kreativen, nicht-deterministischen Ablaufs, durch ihren kooperativen Charakter und die Notwendigkeit einer frühzeitige Fehlererkennung schon lange eine Herausforderung für Computerunterstützung und Automatisierung dar. Die Wiederverwendung sowohl der Ergebnisse eines Entwurfsprozesses als auch des Wissens, das der während Erstellung dieser Ergebnisse gewonnen wird, bildet das Ziel vieler Verbesserungsvorschläge zur Beschleunigung, Vereinfachung und Qualitätssteigerung des Entwicklungsprozesses.

So hebt [Sommerville 1995, Kap.20] die im Software–Engineering durch Wiederverwendung erzielbare gesteigerte Zuverlässigkeit hervor, und [Altmeyer et al. 1994] erwartet durch die Konzentration der Entwickler auf das Wesentliche eine Beschleunigung des Entwurfsprozeß von VLSI–Bausteinen.

Eine erfolgreiche Wiederverwendung setzt aber voraus, daß Wiederzuverwendendes in effizienter, den Randbedingungen des Entwurfsprozesses angemessener Form gespeichert ist. Aufgrund der Vorteile, die eine integrierte Datenverwaltung bei der Verwendung verschiedener Entwurfswerkzeuge bietet, sollte diese Speicherung vorzugsweise in Datenbanken oder Repositories erfolgen und nicht in Dateien [Ahmed et al. 1992; Bernstein und Dayal 1994]. Erfassung, Repräsentation und Aufbereitung der Wiederverwendungsbausteine spielen hierbei gleichwertige Rollen, sollen nicht mangelnde Weiterentwicklung des Datenbestandes oder Nichtakzeptanz durch die Benutzer zum Scheitern führen.

Auch bei der mathematischen Modellierung chemischer Prozesse, einem Teilgebiet der Verfahrenstechnik, treten diese Probleme auf und werden durch eine geringe Standardisierbarkeit der modellierten Prozesse sowie häufig exploratives Vorgehen verschärft. Das Ziel dieser Arbeit besteht daher in der Definition eines Datenmodells sowie der zugehörigen, von der Datenbank und der Entwicklungsumgebung anzubietenden Dienste, so daß Speicherung und Wiederverwendung der Ergebnisse des Modellierungsprozesses bei der Modellierung chemischer Prozesse ermöglicht werden.

1.1 Prozeßentwicklung in der Verfahrenstechnik

Hemming definiert die Verfahrenstechnik folgendermaßen [Hemming 1991]:

> *Die Verfahrenstechnik ist die Ingenieurwissenschaft, die sich mit allen Verfahren befaßt, die Stoffe hinsichtlich Zusammensetzung, Eigenschaften oder Stoffart gezielt verändern. Das Ziel der Verfahrenstechnik ist also die Stoffumwandlung. Ihre Aufgaben umfassen* [1] *die theoretische Erforschung der Stoffumwandlungsvorgänge,* [2] *die Entwicklung von Produktionsverfahren,* [3] *die Planung und Auslegung von Produktionsanlagen sowie* [4] *deren Betrieb und Überwachung.*

Um die bei der Stoffumwandlung auftretenden Vorgänge (oder auch “Prozesse“) zu verstehen, zu überprüfen und zu verbessern, muß ihr Verhalten analysiert werden. Neben Laborexperimenten und

Untersuchungen an Versuchsanlagen werden die Prozesse hierzu mathematisch modelliert. Da verfahrenstechnische Prozesse durch komplexe, nichtlineare Gleichungssysteme mit algebraischen und meist auch differentiellen Gleichungen beschrieben werden, können die mathematischen Prozeßmodelle meist nicht analytisch gelöst werden. Nur durch ihre numerische Lösung (Simulation) erhält man die gewünschten Ergebnisse.

Bild 1.1 zeigt eine sehr grobe Einordnung der Modellierung und Simulation in den **Entwicklungsprozeß**. Wesentlich detailliertere, teilweise den gesamten Geschäftsablauf umfassende Einordnungen enthalten beispielsweise [Smith 1995; Krieger 1995; Marquardt 1996; Nagl und Marquardt 1997]. Obwohl Modellierung und Simulation ein auf allen Ebenen evolutionärer Prozeß ist, wird hier ein dem Wasserfallmodell (siehe etwa [Sommerville 1995]) nachempfundener Ablauf gezeigt, der es ermöglicht, Unterschiede zwischen der Modellierung von Anlagen in der Entwurfsphase und in der Produktionsphase darzustellen. Die hier gezeigten Schritte werden für größere Modelle mehrfach durchlaufen, wobei sich der Detaillierungsgrad kontinuierlich erhöht. Der Gesamtprozeß entspricht also eher dem Spiralenmodell von Boehm [Boehm 1988]. Die obere Hälfte der Abbildung entspricht dem Vorgehen

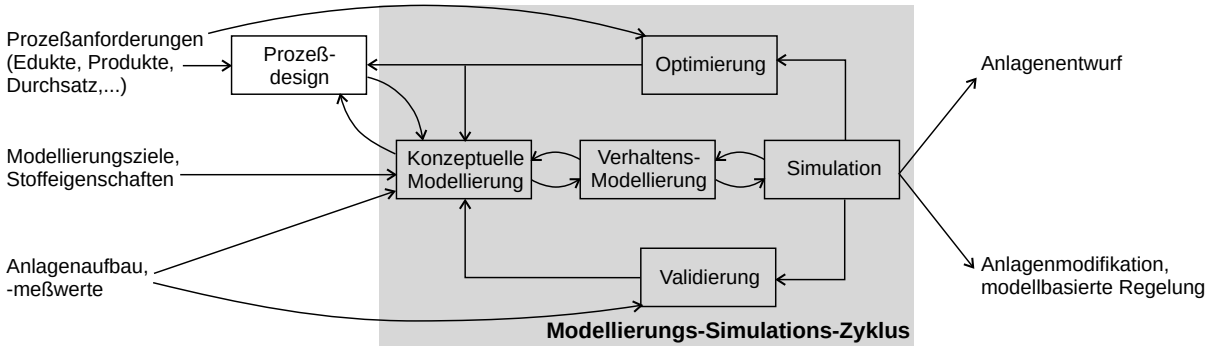


Abbildung 1.1: Prozeßentwurf und -modellierung

beim Entwurf eines neuen Prozesses, also dem zweiten oben von [Hemming 1991] erwähnten Aufgabengebiet, die untere dem beim Abwandeln oder Verbessern einer bestehenden Anlage, dem vierten Aufgabengebiet. Der obere Ablauf ist offensichtlich ein Entwicklungsprozeß, wobei Prozeßdesign und konzeptionelle Modellierung teilweise in einem Schritt vereinigt werden. Aber auch die Modellierung bestehender Anlagen stellt einen kreativen Prozeß dar, da der Übergang Realität → Modell von den durch den Modellierer zu wählenden Abstraktionen und Gewichtungen (wie z.B. Modelltyp, Detaillierungsgrad, zu vernachlässigende Effekte, vgl. [Marquardt 1997]) abhängt.

Entwurf und dauerhafte Nutzung von mathematischen Prozeßmodellen sind mit Schwierigkeiten verbunden, die im nächsten Abschnitt erläutert werden, bevor das Konzept integrierter Modellierungsumgebungen als Lösung vorgestellt wird.

1.1.1 Modellierung — die Praxis

Die sich durch die Globalisierung verschärfende Konkurrenz führt auch für die verfahrenstechnische Industrie zu einem verstärkten Preisdruck, erhöhten Qualitätsanforderungen und kürzeren Produktzyklen. Hinzu kommen sich verschärfende Sicherheits- und Umweltschutzanforderungen. Die für die notwendigen Prozeßverbesserungen erforderlichen Informationen lassen sich, wie oben bereits erwähnt, durch vermehrte Experimente oder Simulationen erreichen. Dem stehen jedoch Kosten und Zeitaufwand entgegen. Während sich diese im experimentellen Bereich nur begrenzt verringern lassen, kann computergestützte Modellierung den zur Erstellung und Validierung von Simulationsmodellen notwendigen Aufwand entscheidend verringern [Marquardt 1996; Foss et al. 1997].

Bei der computergestützten Modellierung existieren zwei grundsätzliche Klassen von Prozeßmodellen: blockorientierte und gleichungsorientierte. Beide zeigen Unzulänglichkeiten, wenn es um die schnelle Generierung detailierter Modelle geht. Die **blockorientierte Modellierung** setzt Modelle

aus vorgegebenen, parametrisierten Bausteinen zusammen. Die den entsprechenden Baustein beschreibenden Gleichungssysteme liegen hierbei in ausprogrammierter Form in einem Modul vor, das alle für die Simulation notwendigen Informationen und Prozeduren enthält. Für viele dynamische oder untypische Modellierungsprobleme fehlen aber Bibliotheken mit den entsprechenden Bausteinen [Foss et al. 1997], zumal der Auswahlraum durch die Kombinationsmöglichkeiten zwischen Apparaten und in ihnen berücksichtigten Phänomenen umfangreich ist [Bogusch et al. 1996].

Das **gleichungsorientierte Vorgehen** hingegen beschreibt den (grob strukturierten) Prozeß als eine Menge von differential–algebraischen Gleichungssystemen, die durch einen entsprechenden Simulator numerisch gelöst werden. Dies erlaubt die Modellierung jedes mathematisch beschreibbaren, verfahrenstechnischen Prozesses. Allerdings erfordert der Entwurf korrekter mathematischer Modelle sowohl ein gutes Verständnis der physikalischen Vorgänge als auch spezielle mathematische Fähigkeiten [Pantelides und Britt 1994], so daß nur wenige, erfahrene Ingenieure dies beherrschen [Jarke und Marquardt 1996]. Desweiteren liegt der Zeitbedarf über dem eines blockorientierten Ansatzes.

In [Foss et al. 1997] berichtet Foss, daß viele Benutzer eine **Unterstützung der Wiederverwendung** von Modellen wünschten, was Geschwindigkeit und Qualität der Modellerstellung verbessern würde. Bisherige Tools erfüllten diese Forderung so schlecht, daß alle Versuche scheiterten, eigene Modellbibliotheken zu erstellen. Neben der Neumodellierung von Grund auf würden darum bestenfalls Teilmodelle früherer, persönlicher Projekte per Hand modifiziert und wiederverwendet.

Auch die in [Pantelides und Britt 1994] von der Wiederverwendung zwischen Projekten abgegrenzte Wiederverwendung eines Modells innerhalb der verschiedenen Phasen/Anwendungen eines Projekts (beispielsweise Marktanalyse, Entwurf, Wärmeintegration, Materialflußoptimierung, Anlagenplanung, Anlagenregelung) leidet unter der Zersplitterung der entsprechenden Anwendungen auf verschiedene, nicht integrierte Programme.

Vorhandene Modellierungsparadigmen scheitern also an ihrer Komplexität (gleichungsorientierte Ansätze) oder an ihrer Unvollständigkeit (blockorientierte Ansätze). Die naheliegende Konsequenz, Kombination der Vorteile beider Paradigmen und Zusammenführung mit Konzepten zu Wiederverwendung und Integration, behandelt der nächste Abschnitt.

1.1.2 Modellierung — das Ziel

Aus den oben aufgeführten Problemen ergibt sich vereinfacht gesagt das Motto:

Schneller, Besser, Einfacher!

In kürzerer Zeit sollen auch weniger erfahrene Modellierer korrektere und genauere Modelle erstellen können. Auf einer groben Ebene kann dies durch eine durchgehende Unterstützung des Modellierungs- und Entwurfsprozesses mittels einer **integrierten Entwicklungsumgebung** erreicht werden, wie sie etwa in [Geoffrion 1989; Nagl und Marquardt 1997; Bañares-Alcántara 1995; n-dim Group 1995b] vorgeschlagen wird. Sie erlaubt durch ein gemeinsames Datenmodell und das Bereitstellen von Grundfunktionalitäten die kombinierte Verwendung verschiedener externer (siehe auch Abschnitt 1.1.4) und interner Werkzeuge.

Einen ersten Schritt in Richtung einer solchen integrierten Entwicklungsumgebung stellt eine **Modellierungsumgebung** dar, die sich auf die während Modellierung und Simulation anfallenden Aufgaben konzentriert. Eine vollständige Entwicklungsumgebung benötigte noch wesentlich weitergehende Funktionalität (vgl. Bild 1.2), doch wird sich diese Arbeit auf die Modellierungsproblematik konzentrieren. Teile einer Modellierungsumgebung für verfahrenstechnische Prozesse werden durch den Lehrstuhl für Prozeßtechnik (LPT) [Bogusch et al. 1996] und den Lehrstuhl für Informatik V (I5) entwickelt [Dömges et al. 1996]. Bild 1.3 zeigt einige in einer solchen Umgebung benötigte Werkzeuge in einer dem ECMA-Modell (vgl. [ECMA und NIST 1991]) nachempfundenen Architektur.

Entwicklungsumgebung	Modellierungsumgebung
• Anforderungsgewinnung • Simulationsexperimentverwaltung • CAD–Konstruktions- und Geometriemodellierung • Optimierungsverfahren • Personal- & Resourcenmanagement • ... + Modellierungsumgebung (siehe rechts)	• Erstellung des Modells aus gegebenen Anforderungen • Verifizierung des Modells • Simulation • Modellierungs–Simulations–Zyklus • Modellierungsalternativen • Verwaltung von (konzeptionellen und mathematischen) Modellen, Modellteilen und -bausteinen

Abbildung 1.2: Modellierungsumgebung als Teil einer Entwicklungsumgebung

Ähnliche Modellierungsumgebungen wie OmSim, n-dim, epee und KBMOSS und Datenrepräsentationssprachen wie Omola, Model.La und Ascend werden in Abschnitt 1.1.4 sowie genauer in Kapitel 4.3 vorgestellt.

Trotz der Begrenzung auf den Modellierungs–Simulations–Zyklus unterscheiden sich die Anforderungen an die Modellierungsumgebung nur wenig von denen an eine integrierte Entwicklungsumgebung. So werden die in [Bañares-Alcántara 1995] von Entwicklungsumgebungen zu unterstützenden Tätigkeiten (Exploration, Evolution, Kooperation, Integration und Automatisierung) auch in Modellierungsumgebungen benötigt. Allerdings lassen sie sich teilweise spezialisieren und um Anforderungen an die Modellierungssprache ergänzen. Sieht man beispielsweise die blockorientierte Modellierung als Grundvoraussetzung für Einfachheit und die gleichungsorientierte Modellierung als Voraussetzung für Genauigkeit an, so muß die zum Einsatz kommende Modellierungssprache eine Kombination der beiden Modellierungsparadigmen sein. Entsprechend ergeben sich die nachfolgenden **Anforderungen für die Modellierungsumgebung** und das in Ihr verwendete Datenmodell:

1. **Trennung des konzeptionellen und des mathematischen Modells:** Ohne die Gleichungen wie bei der blockorientierten Modellierung völlig zu verbergen, soll dennoch eine primär konzeptionelle Modellierung den Entwurf des Modells vereinfachen [Marquardt 1996; Jarke und Marquardt 1996]. Die einem Baustein zugrundeliegenden Gleichungen sollen allerdings auf Wunsch oder bei Notwendigkeit sicht- und editierbar sein [Bogusch et al. 1996].
2. **Konzeptionelles Prozeßmodell Anreichern:** Zusätzliche Informationen in den atomaren Komponenten des konzeptionellen Modell sollen es ermöglichen, die beschreibenden Gleichungen (halb-)automatisch zu erzeugen.

   Wenn zumindest erreicht werden kann, daß das System dem Benutzer konsistente Vorschläge für die einen Baustein beschreibenden differentiellen und algebraischen Gleichungen anbietet bzw. ihn bei der Wahl dieser Gleichungen anleitet, so wird die Definition der Gleichungen für neue Bausteine, das Hauptproblem der gleichungsorientierten Modellierung, auch für weniger erfahrene Benutzer durchführbar [Pantelides und Britt 1994, S.130f].
3. **Dekomposition der konzeptionellen Beschreibung des Prozesses:** Dekomposition ist eine Standardvorgehensweise zur Vereinfachung komplexer Konzepte. Die durch die Dekomposition gewonnene Verringerung der Komplexität darf allerdings nicht dadurch zunichte gemacht werden, daß der Modellierer bei der Aggregation der Gleichungen eingreifen muß. Die Dekompositionstiefe sollte prinzipiell unbeschränkt sein, um beliebige Verfeinerungen des Modells zuzulassen [Marquardt 1991].
4. **Verwaltung der untersuchten Varianten:** Suchverfahren zur Optimierung des Prozesses, die Berechnung von Initialisierungswerten oder auch die Experimentierfreude des Modellierers erzeugen verschiedene Varianten eines Modells für einen Prozeß [Bañares-Alcántara 1995; Marquardt 1996; Jarke und Marquardt 1996; Bogusch et al. 1996]. Zwischen diesen Varianten muß schnell gewechselt werden und ein Auswahl der entsprechenden Konfigurationen einfach gespeichert werden können.

5. **Ergänzung des Prozeßmodells um Dokumentationsinformationen:** Entsprechende Dokumentation, z.B. Anforderungen, Annahmen, Zweck und Beschränkungen von Bausteinen, erleichtert sowohl die Wiederverwendung als auch die Fehlererkennung durch Modellinspektion. Ebenso wird der Lerneffekt für Modellierungsanfänger durch das Studium abgeschlossener, gut dokumentierter Modelle gesteigert.[Bogusch et al. 1996; Marquardt 1996; Jarke und Marquardt 1996; Bañares-Alcántara 1995; MacFarlane et al. 1989]
6. **Frühzeitige Konsistenztests:** Die Zahl der Durchläufe des Modellierungs–Simulations–Zyklus (vgl. Bild 1.1) und damit die Gesamtentwicklungszeit läßt sich durch Fehlerkontrolle bereits in der Modellierungsphase verringern.
7. **Durch den Benutzer erweiterbare Bibliothek von Modellbausteinen:** Umfangreiche Bausteinbibliotheken verringern die Wahrscheinlichkeit, Modelle von Grund auf neu erstellen zu müssen. Erst durch die Erweiterbarkeit jedoch können sie vom Benutzer seinen Bedürfnisse und Erkenntnisse angepaßt werden. [Marquardt 1991; Jarke und Marquardt 1996; Bogusch et al. 1996]
8. **Unterstützung (auch unerfahrener) Modellierer:** Die Verfahren zur Definition neuer und zur Wiederverwendung und Änderung vorhandener Modellbausteine sowie die auf ihnen durchführbaren Operationen müssen intuitiv erfaßbar und dem bisherigen Vorgehen ähnlich sein, um Lernaufwand und Akzeptanzprobleme zu minimieren. (teilweise in [MacFarlane et al. 1989])
9. **Führung der Benutzer entlang bekannter Modellierungsteilstücke:** Modellierern sollten weitere, sinnvolle Arbeitsschritte vorgeschlagen und von unsinnigen abgeraten werden [Pohl 1995; Jarke und Marquardt 1996; Lohmann 1997]. Das Wissen über diese Arbeitsschritte kann entweder den Überlegungen eines Experten entspringen (*Metamodellierer*) oder durch Beobachtung häufig auftretender Vorgehensweisen gewonnen werden (*Auswertung der Aufzeichnungen früherer Modellierungsabläufe*). Ggf. können so wiederkehrende Arbeiten auch automatisiert werden.

Es fällt auf, daß die oben aufgezählten Eigenschaften **Kooperation und Integration** in den spezialisierten Anforderungen nur am Rande auftauchen. Solche Aspekte werden in dieser Arbeit weniger berücksichtigt. Integrationsfragestellungen werden etwa in [Barker et al. 1993] für den Regelungsentwurf in einer Architektur ähnlich zu der in [ECMA und NIST 1991] vorgestellten beantwortet. Dabei liegt eine Betonung auf den aus drei Ebenen verschiedener Abstraktion bestehenden Modellierungsdiensten. Mit der Datenintegration im Bereich der Verfahrenstechnik beschäftigen sich zudem mehrere Standardisierungsinitiativen unter dem STEP/EXPRESS–Dach [ISO 1994], für den Bereich der Prozeßmodelle etwa pdXi [ISO 1998]. Eingehender werden Integrationsaspekte in einem Sonderforschungsbereich (SFB 476, s. [Nagl und Marquardt 1997]) der RWTH Aachen behandelt.

Viele der aufgeführten Punkte dienen der Wiederverwendung von Wissen, allerdings auf verschiedene Arten desselben. Die ersten Punkte (1 bis 3, 5) umfassen Expertenwissen, das, einmalig gewonnen und sich nur langfristig ändernd, als **Metamodell** die Grobstrukturen vorgibt. Dieses Metamodell findet sich im in [Marquardt et al. 1992] entworfenen und am LPT weiterentwickelten "**Ve**rfahrenstechnischen **Da**tenmodells" (VeDa) wieder, das die Grundlage für die auf Seite 3 erwähnten Teilimplementierungen einer integrierten Modellierungsumgebung der Lehrstühle LPT und I5 darstellt. VeDa definiert Sprache und Metamodell für die Repräsentation des konzeptionellen und des mathematischen Modells. Eine genauere Beschreibung befindet sich in Kapitel 2.

Ein besonderer, weil **prozeduraler** Teil dieses Metawissens sind die in Anforderung 2 erwähnten Verfahren zur Gleichungsgenerierung für die atomaren, konzeptionellen Modellelemente. Hierbei muß aus abstrakten Beschreibungen der auftretenden Phänomene auf die in einem Variablen–Gleichungs–Netz repräsentierten Bilanzgleichungen und konstitutiven Gleichungen[1] geschlossen werden. Mit die-

[1] Bilanzgleichungen repräsentieren die aufgrund physikalischer Gesetzmäßigkeiten (etwa Energie-, Massen-, Impulserhaltungssatz) geltenden Zusammenhänge, konstitutive Gleichungen (etwa Transportgleichungen, Bestimmungsgleichungen für Transportkoeffizienten, Gleichgewichtsbedingungen und Schließbedingungen) bestimmen die in Bilanzgleichungen auftretenden Größen genauer durch Rückführung auf nicht bilanzierte Größen (wie etwa Temperatur oder Druck) und Verknüpfung mit anderen Bilanzgleichungen.

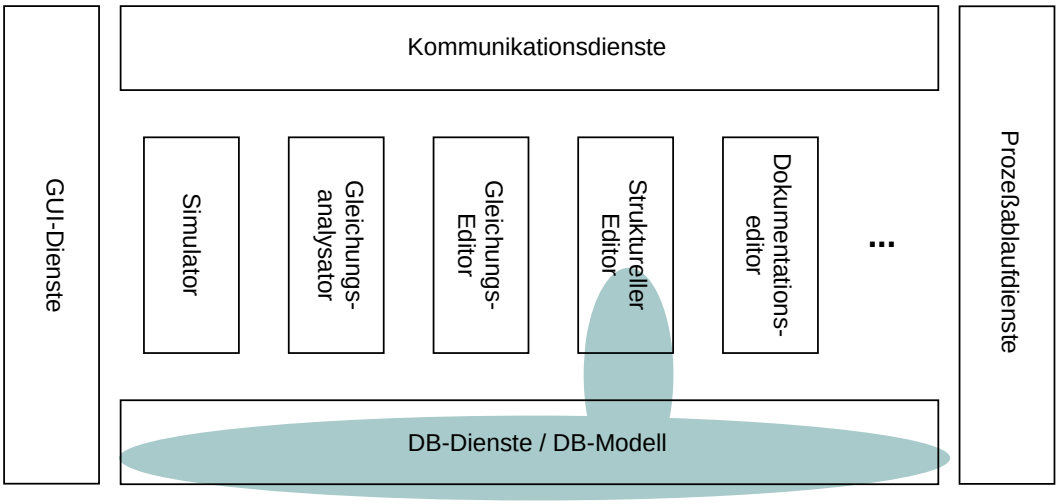


Abbildung 1.3: Grobarchitektur der Modellierungsumgebung (ähnlich [ECMA und NIST 1991])

sen Themen, der Gleichungs- und Variablenrepräsention und ihrer Generierung, beschäftigte sich R. Bogusch im Zusammenhang mit der Entwicklung der Modellierungsumgebung ModKit (s. etwa [Bogusch et al. 1996; Bogusch und Marquardt 1997]). Dies umfaßt in Abbildung 1.3 etwa die Bereiche "Gleichungseditor" und "Dokumentationseditor"

Die Führung des Benutzers durch komplexe Teilabläufe sowie die Anpaßbarkeit von Abläufe in den Werkzeugen (Anforderung 9 und Teile der Anforderung 8) umfassen das **Ablaufwissen** (im Gegensatz zum "Faktenwissen"). Es enthält beispielsweise Informationen über Kombinierbarkeit, Austauschbarkeit, Reihenfolge und vorgegebene Folgen von Modellierungsschritten sowie die Definition atomarer Schritte (vgl. [Lohmann 1997; Krobb 1997; Pohl 1995]). Im Unterschied zu den beiden vorherigen Wissensformen soll hierbei auch während des Modellierungsprozesses Information in Form von Ablaufaufzeichnungen gewonnen und später wiederverwendet werden. Mit der Syntax und Semantik der Modellierungsschritte, ihrer Beziehungen in der verfahrenstechnischen Prozeßmodellierung und ihrer Implementierung in ModKit beschäftigt sich B. Lohmann [Lohmann 1997]; die Aufzeichnung des Prozeßablaufs behandelt R. Doemges [Dömges 1999]. In Abbildung 1.3 betrifft dies etwa die "Prozeßablaufdienste".

Die Punkte "Erweiterbarkeit der Bausteinbibliothek" sowie "Intuitive Erzeugung und Wiederverwendung von Modellen" (Anforderungen 7 und 8) betreffen die Wiederverwendung der direkten Ergebnisse des Modellierungsprozesses. Der Repräsentation dieses **Produktwissens** in der Datenbank, seiner Aufbereitung und Organisation widmet sich diese Arbeit (grau hinterlegte Bereiche in Abbildung 1.3).

Bevor auf die notwendigen Eigenschaften dieser Datenbank und ihres Datenmodells eingegangen wird, enthalten die nächsten Abschnitte einen Vergleich der Entwurfsprozesse im Software–Engineering und bei der Modellierung verfahrenstechnischer Prozesse sowie einen kurzen Überblick über verschiedene Ansätze zur computergestützten verfahrenstechnischen Prozeßmodellierung.

1.1.3 Vergleich verfahrenstechnischer Prozeßmodellierung und Software–Entwurf

Entwurfsprozesse treten in verschiedenen Disziplinen auf und beinhalten regelmäßig die Erstellung von Modellen. Die Modellierung verfahrenstechnischer Prozesse, eigentlich also nur Teil des Entwurfsprozesses verfahrenstechnischer Anlagen, wurde auf Seite 2 selbst als Entwurfsprozeß identifiziert. Ein Vergleich mit den Problemen und Lösungen der in der Informatik auftretenden Ent-

wurfsprozesse (also dem Software–Engineering (SE)) erscheint darum sinnvoll. Da die Repräsentation der Prozeßmodelle in der Datenbank und die darauf aufbauenden Dienste im Vordergrund dieser Arbeit stehen, nehmen Repräsentationsfragestellungen einen Schwerpunkt dieses Vergleichs ein[2].

Als Vergleichsobjekte dienen der Modellierungs–Simulations–Zyklus (Bild 1.1) einer gleichungsorientierten Prozeßmodellierung und ein einfaches Modell des Software–Entwicklungsprozesses, bestehend aus Design, Implementierung und Anwendung. Die konzeptionelle Modellierung verfahrenstechnischer Prozesse beschreibt die Struktur der Prozesse sowie grob ihr Verhalten und erzeugt dabei u.U. verschiedene Lösungsalternativen. Sie fällt darum in der obigen Einteilung mit dem Software–Design zusammen, dessen Ziel ein konzeptionelles Modell des Software–Systems ("Architektur" genannt) sowie des zugehörigen Datenmodells ("Data dictionary" genannt) ist. Die Erstellung des Verhaltensmodells des verfahrenstechnischen Prozesses, meist eine vollständige, mathematische Beschreibung des Prozeßverhaltens, ermöglicht die 'Ausführung' des Modells. Sie entspricht also in etwa der Implementierung. Zumindest auf den ersten Blick erscheinen die jeweiligen Phasen also ähnlich.

Es besteht allerdings ein Unterschied darin, inwieweit Informationen aus dem jeweiligen konzeptionellen Modell die jeweilige Verhaltensbeschreibung bestimmen helfen. [Krueger 1992] spricht hier von der **kognitiven Distanz** als derjenigen intellektuelle Mühe, die ein Entwickler aufwenden muß, um ein System von einer Entwicklungsphase in die nächste zu bringen. So kann im SE die Architektur neben umgangssprachlichen Angaben zur Funktionalität von Modulen und Methoden zwar auch formalere Spezifikationen etwa in Form algebraischer oder modellbasierter Sprachen enthalten (vgl. [Sommerville 1995, Kapitel 9-11]). Diese sind jedoch für größere, interaktive System meist schlecht geeignet [Sommerville 1995, S.163]. Zudem sollen sie eher Präzision, Vollständigkeit und Konsistenz der Spezifikation sicherstellen als die kognitive Distanz zwischen Architektur und Implementierung zu verringern. In der Prozeßmodellierung wird bei einem Ansatz wie VeDa (siehe nächstes Kapitel) abstrakte, das Verhalten eines Bausteins beschreibende Information in den entsprechenden konzeptionellen Bausteinen eingefügt. Die Angabe der auftretenden Phänomene und Flüsse erlaubt die Generierung der notwendigen Bilanzgleichungen aus dem Gleichungs-/Variablennetz. Ihre Attribute erleichtern die weitere Navigation in diesem Netz zur Definition der konstitutiven Gleichungen [Marquardt 1996]. Durch diese Eigenschaft ähnelt die Modellierung verfahrenstechnischer Prozesse weniger dem Standard–SE, sondern eher der Softwareerstellung durch Anwendungsgeneratoren oder Transformationssysteme (vgl. [Krueger 1992; Shlear und Mellor 1988]), welche in [Krueger 1992] bereits als Wiederverwendungsmethoden klassifiziert werden. Sowohl im SE als auch in der Prozeßmodellierung sind Spezifikationen allerdings häufig unvollständig, so daß die kognitive Distanz zwischen konzeptionellem Modell und Verhaltensbeschreibung in der Realität jeweils größer ist.

Ein Grund für diesen Unterschied in der kognitiven Distanz könnten die verschiedenen **Abstraktionsniveaus** in der jeweiligen Verhaltensbeschreibung sein. Im SE bedingt die Implementierung je nach gewähltem Programmiersprachenparadigma eine mehr oder weniger intensive Beschäftigung mit Algorithmen und den für sie notwendigen Kontrollstrukturen. Dagegen müssen bei der gleichungsbasierten Prozeßmodellierung 'nur' das Verhalten beschreibende mathematische Gleichungen gefunden werden. Der für Algorithmen notwendige Aufwand wurde dabei in die zur Simulation des Verhaltens notwendige Simulatorsoftware verlagert. In der Prozeßmodellierung kann sozusagen auf einem höherem Abstraktionsniveau 'programmiert' werden.

Aufgrund obiger Unterschiede divergieren natürlich auch die Fragestellungen bezüglich der Repräsentation und der Wiederverwendung von Bausteinen. Im SE bereiten bereits die für die Repräsentation wiederverwendbarer Bausteine notwendige Abstraktion Schwierigkeiten [Krueger 1992, S.134ff]. Dagegen können im konzeptionellen Prozeßmodell — zumindest prinzipiell — durch die Wahl des Verfeinerungsgrads unterschiedliche Abstraktionsebenen weitgehend abgebildet werden. Auch ist das Hauptziel der Wiederverwendung in der Prozeßmodellierung die Wiederverwendung konzeptioneller Modelle (also von Designs), was zwar das SE auch angestrebt, aber bisher kaum verwirklicht hat

[2] Nicht behandelt werden andere Bereiche, wie etwa Anforderungsspezifikation, Kooperationsunterstützung, Werkzeugintegration oder Dokumentation, in denen man die vielen Gemeinsamkeiten findet.

[Sommerville 1995].

Nachteilig wirkt sich die in konzeptionellen Modellen enthaltene abstrakte Information über das Modellverhalten allerdings auf den Umfang derselben aus. So enthielt das Modell eines in fünf Stufen diskretisierten Blasensäulenreaktors in ModKit aufgrund der mitmodellierten Anrechenvarianten bereits 33.000 Objekte. Dies stellt natürlich besondere Anforderungen an Cache-, Cluster- und Zugriffsstrategien der Datenbank.

Im SE beschränkt sich die zielgerichtete Wiederverwendung meist auf speziell hierfür entwickelte Bausteine: Bibliotheken (z.B. Leda oder IMSL) und 'Component Objects' [Adler 1995] auf Implementierungsebene bzw. Architekturen [Shlear und Mellor 1988] oder Frameworks [Talingent 1993] als abstraktere Einheiten. [Sommerville 1995] begründet dies mit der mangelnden Generalität nicht spezifisch zur Wiederverwendung entwickelter Bausteine. Ähnlich sieht es in der Praxis der Prozeßmodellierung aus. Die **Wiederverwendung jeglicher Modellierungsergebnisse** verspräche jedoch eine einfachere Erweiterbarkeit sowie eine größere und damit passendere Auswahl an wiederzuverwendenden Komponenten. Aufgrund des hohen Abstraktionsniveaus sowie der klaren Schnittstellendefinition erscheint dieses Vorgehen in der verfahrenstechnischen Prozeßmodellierung eher möglich als im SE.

Also unterscheidet sich die Repräsentation und Wiederverwendung verfahrenstechnischer Modellbausteine von ähnlichen Problemen im Software–Engineering im Abstraktionsniveau von konzeptionellem Modell und Verhaltensbeschreibung sowie der kognitiven Distanz zwischen ihnen. Dadurch wird die Wiederverwendung von Bausteinen einerseits einfacher. Denn der Entwurfsprozeß spielt sich größtenteils in nur einer, noch relativ abstrakten Spezifikation ab und die verwendeten Bausteine besitzen dadurch eine hohe Informationsdichte. Andererseits kompliziert die große Zahl der Objekte und ihre Variabilität im Detail die Wiederverwendung. Aus diesen Gründen wurden nur generelle Wiederverwendungsansätze des SE untersucht, aber keine spezifischen Realisierungen.

1.1.4 Systeme zur Unterstützung der Modellierung verfahrenstechnischer Prozesse

Das Ziel, die Modellierung verfahrenstechnischer Prozesse zu vereinfachen, wird schon länger mit verschiedenen Ansätzen verfolgt. Überblicke bieten beispielsweise [Stephanopoulus und Han 1996] und [Marquardt 1992]. Die verschiedenen Ansätze lassen sich grob in drei Kategorien unterteilen: Teilautomatisierung von Modellierungsschritten, Simulationsexperiment- und Gleichungsanalyse sowie verschiedene Arten von Entwicklungsumgebungen.

Zur **Teilautomatisierung** lassen sich beispielsweise Expertensysteme [Marquardt 1992, S.37] und Ablauf- und Methodikenwiederverwendung [Lohmann 1997; Morie et al. 1993] zählen. Auch die auf Seite 4 erwähnte Gleichungsgenerierung gehört hierzu.

Expertensysteme beruhen auf qualitativem oder probabilistischem Schließen [Bibel et al. 1996] statt auf reinen numerischen oder strukturellen Informationen und versuchen ggf. unter Rückfragen eine Lösung für gegebene Anforderungen zu generieren. Setzt man sie zur Modellierung ein, ergeben sich schnell Probleme mit dem Umfang des benötigten Wissens sowie der zyklischen Gestalt des Entwurfsprozesses, der zudem meist mit unvollständigen Anforderungen beginnt. Erfolgreich sind solche Systeme allerdings, wenn sie auf eng begrenzte Gebiete im Prozeßdesign angewandt werden [Stephanopoulus und Han 1996, S.775]. Durch Integration mehrerer "Spezialisten"–Systeme in eine blockorientierte Entwurfsumgebung (z.B. [Schembecker et al. 1994]) kann dann eine umfassende Unterstützung des Anwenders geboten werden.

Die Proponenten der Methodikwiederverwendung gehen davon aus, daß nicht nur die Ergebnisse des Modellierungsprozesses, sondern auch die in ihm auftretende Abläufe wiederverwendet werden können. Hat man eine deklarative Repräsentation einzelner Prozeßschritte, wie z.B. in [Pohl 1995] und [Lohmann 1997] vorgestellt, so kann man nicht nur auf einfache Art verschiedene Abläufe vorgeben, sondern auch stattfindende Abläufe aufzeichen [Pohl et al. 1997], verallgemeinern und

wiederverwenden [Morie et al. 1993]. Besonders bei zahlreichen Unterschieden zwischen Modellierungsergebnissen aber ähnlichem Vorgehensweisen innerhalb des Modellierungsprozesses bringt die Methodikwiederverwendung zusätzlichen Nutzen.

An **Gleichungen und Simulationsexperimenten** setzen meist Verfahren an, die die Ausführung der eigentlichen Simulation beschleunigen sollen. Naheliegende Vorgehensweisen um Rechenzeit und Konvergenz zu beschleunigen, sind die Vereinfachung der Gleichungssysteme und die Verbesserung der numerischen Lösungsverfahren. Weiter gehen Verfahren, die den Benutzer bei Erstellung, Durchführung und Analyse von Simulationsexperimenten unterstützen, diese teilweise automatisieren, indem sie Informationen aus dem Modell erhalten oder aus dem Simulationsprozeß abstrahieren und sie zur Verbesserung und Dokumentation der Simulation und ihrer Ergebnisse einsetzen [Stephanopoulus und Han 1996, S.769ff].

Die Integration der verschiedenen Teillösungen zur Entwicklungsvereinfachung ist eines der Ziele von **Entwicklungsumgebungen** [Marquardt 1992; Bañares-Alcántara 1995; Stephanopoulus und Han 1996, S.776f]. Je nach ihrer Konzentration auf Kooperations-/Dokumentations- bzw. auf Repräsentationsaspekte werden wir sie im folgenden als Kooperationsumgebungen bzw. als Modellierungsprachen (mit ggf. zugehöriger Modellierungsumgebung) bezeichnen. Erstere beschäftigen sich also mehr mit generisch verwendbaren Lösungen, während letztere verfahrenstechnisches Domänenwissen zu nutzen und repräsentieren suchen. Die bedeutendsten Vertreter von Kooperationsumgebungen innerhalb der Verfahrenstechnik dürften n–dim [n-dim Group 1995b] und épée [Costello et al. 1996] sein. Épée versteht sich als Werkzeugintegrationsumgebung und n–dim als Prototyping–Umgebung für Entwicklungsumgebungen[3]. Beide stellen auf einem sehr flexiblen Datenmodell Versionierung und workspace-basierte Kooperation zur Verfügung.

Modellierungssprachzentrierte Systeme versuchen hauptsächlich die der verwendeten Modellierungssprache eigenen Repräsentationsmöglichkeiten auszunutzen. Diese Sprachen unterteilt [Marquardt 1996] nochmals in generelle Modellierungssprachen und in Prozeßmodellierungssprachen, was sich hauptsächlich in Art und Spezialisierung des (teilweise in die Sprache integrierten) Datenmodells widerspiegelt. Als Hauptvertreter beider Richtungen lassen sich Omola und die Modellierungssprache von Ascend als generelle Modellierungssprachen swoie Model.La und VeDa als Prozeßmodellierungssprachen nennen. Alle sind objektbasiert und enthalten Semantikerweiterungen, um das erfaßbare Wissen zu vergrößern[4] bzw. seine Erfassung zu vereinfachen[5]. Die meisten sprachzentrierten Systeme befassen sich nicht mit der Fragestellung der Persistenz der erzeugten Objekte, sondern speichern die entstandenen Modelle in separaten Dateien.

Kooperationsumgebungen und Modellierungssprachen zusammenzuführen gelingt beispielsweise durch die Integration von Ascend in n-dim sowie in épée, das auch ein einfaches Modell verfahrenstechnischer Prozesse enthält.

Alle vorgestellten Systeme tragen zur Beschleunigung oder Vereinfachung des Modellierungsprozesses bei. Wie bereits erwähnt, wird diese Arbeit aber nur die Speicherung und Wiederverwendung von Daten in Modellierungsumgebungen untersuchen, die Ansätze der Teilautomatisierung und der Gleichungsanalyse werden darum im folgenden nicht mehr betrachtet. Grundlage der Speicherung in einer Modellierungsumgebung muß immer eine Modellierungssprache sein, die es erlaubt, die für die verschiedenen Verfahren notwendigen Informationen zu repräsentieren. Kapitel 4.3 wird die verschiedenen oben erwähnten Modellierungsumgebungen und insbesondere ihre Modellierungssprachen genauer auf ihre Eignung analysieren.

[3] Also als Grundlage zur schnellen Realisierung neuer Ansätze für Entwicklungsumgebungen.
[4] Facetten in VeDa, spezielle Relationen wie ARE_ALIKE, is–attached–to oder AT in Ascend, Model.La und Omola
[5] lokale Klassenmodifikationen in Omola

1.2 Datenbankunterstützung für die Modellierung

Auf Seite 3 wurde eine Modellierungsumgebung für verfahrenstechnische Prozesse vorgestellt. Eine zentrale Position in dieser Umgebung nimmt die Datenbank ein, deren Notwendigkeit unumstritten ist [Marquardt 1992; Maffezzoni et al. 1994; Kemper und Moerkotte 1994]. Sie nimmt sämtliche Modellierungsartefakte auf, deren Existenzdauer die jeweilige Aktivierung des erzeugenden Programms überschreitet. Artefakte können die Ergebnisse verschiedener Phasen des Modellierungs-Zyklus sein, also etwa Anforderungen, Stoffdaten, Gleichungssysteme, Simulationsvorgaben und -ergebnisse aber auch Modellierungsabläufe.

Bei den Ergebnissen der frühen Phasen (also Anforderungen, Sammlungen physiko–chemischer Daten und Meßwerten) besteht das Hauptproblem in der Sammlung und Strukturierung der informalen oder in vielfältigen Formaten vorhandenen Information. Die späte Phase (also die Simulation) hingegen erzeugt Zeit-/Ortsreihen von Variablenwerten der Gleichungen. Die große Zahl hierfür zu speichernder einfacher Daten ergibt sich sowohl aus der Vielzahl möglicher Meßpunkte als auch aus der Zahl der meist leicht unterschiedlichen Simulationsexperimente.

In der Mitte zwischen diesen Phasen liegen die verschiedenen Modellierungsschritte, also etwa konzeptionelle Modellierung und Verhaltensmodellierung. Diese Arbeit beschränkt sich auf die **Ergebnisse der konzeptionellen Modellierung** (also die konzeptionellen Modelle). Konzeptionelle Modelle lassen sich strukturiert erfassen und vom Umfang beherrschen. Dadurch erscheint ihre Speicherung in einer Datenbank vergleichsweise einfach. Wie wir aber bereits gesehen haben, nehmen Wiederverwendbarkeit, Variationen im Verlauf des Modellierungs–Simulations–Zyklus sowie Verständlichkeit eine wichtige Rolle bei der Repräsentation konzeptioneller Modelle ein, denn ihre Erstellung hat den größten Anteil an der benötigten Zeit. Die sich hieraus für die Datenbank ergebenden Anforderungen werden jetzt beschrieben.

1.2.1 Wiederverwendung von Modellen

Die Notwendigkeit, Modelle oder deren Teile aus Geschwindigkeits-, Vereinfachungs- und Qualitätsgründen wiederzuverwenden, wurde bereits dargelegt. Es reicht hierfür nicht aus, daß die der Modellierungsumgebung zugrundeliegende Datenbank lediglich die Persistenz der Modelle sicherstellt. Vielmehr muß, um die Wiederverwendung zu ermöglichen, die Anpassung, Einordnung und Suche von Objekten/Konzepten unterstützt werden. Denn genau dies sind die im Verlauf des Wiederverwendungszykluses auftretenden Aktivitäten (Bild 1.4, ganz rechts). Die einfachste Form der

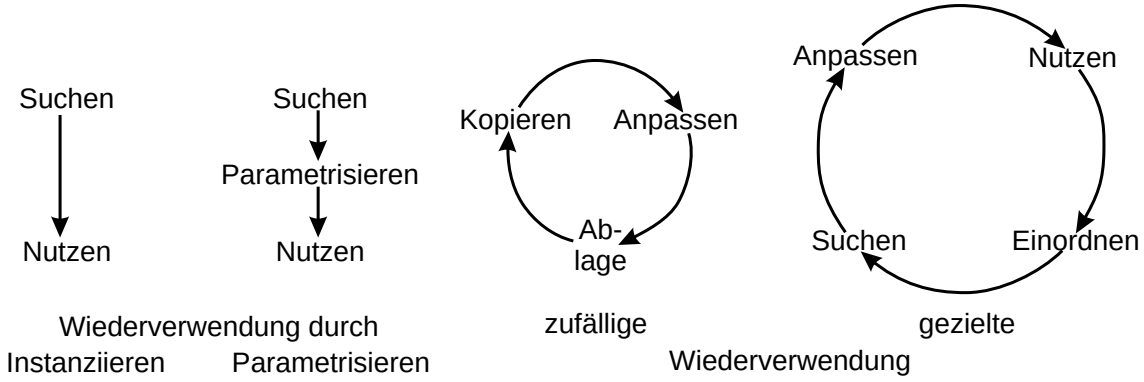


Abbildung 1.4: Verschiedene Arten von Wiederverwendung

Wiederverwendung, "Reuse by Instantiation" in [Altmeyer et al. 1994] (ganz links in Abb. 1.4), selektiert aus einer Menge fertiger, vorgegebener Komponenten die zu einem Anforderungsprofil passende und verwendet sie. Die Komponentensammlung kann eine Bibliothek mathematischer oder

betriebssystem–naher Funktionen sein oder Standardbausteine elektronischer Schaltungen, mechanischer Konstruktionen oder verfahrenstechnischer Prozesse enthalten. Grundvoraussetzung ist aber immer ein eng umrissenes, gut verstandenes Anwendungsgebiet [Krueger 1992, S.147].

Sind die instanziierten Komponenten parametrisiert, können sie an die genauen Anforderungen ihres Einsatzpunktes durch die Modifikation von Parametern angepaßt werden. Beispiele sind Abstrakte Datentypen mit Typvariablen, elektronische Bauteile unterschiedlicher Bitbreite sowie die komplette blockorientierte Vorgehensweise bei der Modellierung verfahrenstechnischer Prozesse. Allerdings müssen neue parametrisierbare Komponenten normalerweise speziell entworfen und implementiert werden, was Erweiterungen des Komponentenbestandes erschwert.

Beide obigen Ansätze gehen davon aus, daß wiederzuverwendende Komponenten speziell für die Wiederverwendung entwickelt und meist auch dem Benutzer präsentiert werden. Interessanter, da leichter erweiterbar, sind Ansätze die auch die Wiederverwendung von früheren Ergebnissen ermöglichen, bei denen sich also ein **Wiederverwendungszyklus** ergibt. Die beiden rechten Schemata in Abbildung 1.4 repräsentieren solche Ansätze. Zufällige, adaptierende Wiederverwendung, "Scavenging" in [Krueger 1992], modifiziert ein auf willkürlichem Weg (meist aufgrund des Erinnerungsvermögens des Modellierers) gefundenes Modell solange, bis es das gewünschte Verhalten zeigt. Nach Anpassung und Nutzung verschwindet die Komponente wieder in den gleichen undurchsichtigen Speicherorganisationen aus der ihr Vorgänger stammte[6].

Für die gezielte, adaptierende Wiederverwendung hingegen muß aus den in der Datenbank befindlichen Modellen und Modellteilen ein den Anforderungen des Benutzers möglichst gut entsprechendes Modell *gefunden* werden (Abbildung 1.4 ganz rechts). Daß der gefundene Baustein exakt den Wünschen des Suchenden entspricht, ist unabhängig von der Güte des Suchalgorithmus aufgrund der Variabilität der Modellbausteine unwahrscheinlich. Er muß darum den Anforderungen und Umgebungsbedingungen einfach *angepaßt* werden können. Die Anpassung geht hierbei über die Parametrisierung hinaus, da beliebige Änderungen möglich sein müssen. Die *Nutzung* des so gewonnenen Modells ist die Aufgabe der in der Modellierungsumgebung integrierten Werkzeuge und ist somit für diese Arbeit weniger interessant. Anschließend muß das neue Modell so in die Datenbank *eingeordnet* werden, daß spätere Suchoperationen vereinfacht oder erst ermöglicht werden[7].

Die weitestmögliche Wiederverwendung wird durch die gezielte, adaptierende Wiederverwendung erreicht. Bezüglich der verwendeten Modellierungssprache ergibt sich nun die Frage: Reicht die Verwendung eines objektorientierten Datenmodells aus? Oder sind domänenspezifische Relationen, die Benutzung von Suchverfahren oder die Erweiterungen des objektorientierten Sprachmodells selbst notwendig, um die Wiederverwendung verfahrenstechnischer Prozeßmodelle entscheidend zu vereinfachen.

1.2.2 Anforderungen an die Repräsentation von Modellen

Die meisten Autoren empfehlen für die Repräsentation (ingenieurwissenschaftlicher) Entwurfsmodelle ein objektorientiertes Datenmodell [Ahmed et al. 1992; Kemper und Moerkotte 1994; Loomis 1994]. Allgemeine Anforderungen an Datenbankmanagementsysteme (DBMS) für solche — im Gegensatz zu finanzwirtschaftlichen oder verwaltungstechnischen Datenbanken "fortgeschritten" genannten — Anwendungen postulieren verschiedene "Manifestos": [Atkinson et al. 1989] für objektorientierte sowie [Darwen und Date 1995] für objektorientierte und relationale Datenbanken. Sie fordern hauptsächlich grundsätzliche Eigenschaften der Datenmodelle und darauf aufbauenden Dienste, wie

[6] Äußerst erfolgreich ist diese Strategie, wenn Änderungen von Umfang oder Gebiet begrenzt und Ablageorte bekannt sind. So geht beispielsweise das LaTeX–Rahmenwerk dieser Arbeit über Modifikationen für diverse Berichte, Konferenzbeiträge und eine Diplomarbeit auf eine Seminarausarbeitung zurück.

[7] [Biggerstaff und Richter 1989] unterteilt die für Wiederverwendung im SE notwendigen Maßnahmen ähnlich in 'abstracting', 'selecting', 'specializing' und 'integrating'. Und auch im Case–based Reasoning werden die Schritte 'index', 'search' und 'adapt' unterschieden.

etwa nichtatomare Datentypen, Vererbung, Kapselung, deklarative Anfragesprache und Konsistenzerhaltung. Aus den in den Abschnitten 1.1.2 und 1.1.3 aufgeführten Anforderungen und Besonderheiten ergeben sich zusätzlich die folgenden **Eigenschaften von Datenmodell und DBMS** als notwendig für eine einer Modellierungsumgebung zugrundeliegende Datenbank:

Verständlichkeit: Um unerfahrenen Benutzern schnellen und einfachen Umgang mit den Modellen zu ermöglichen, müssen das Datenmodell sowie die darauf möglichen Operationen leicht zugänglich und die Modellierungsobjekte der Vorstellungswelt des Modellierers entsprechen. In [Krueger 1992] wird hierfür der Begriff der geringen "kognitiven Distanz" geprägt und als Haupttechnik, dieses Ziel zu erreichen, wird die im nächsten Punkt erläuterte Abstraktion herausgestellt.

Übersichtlichkeit/Abstraktion: Wenn Wiederverwendung und eine effektive Nutzung der in der Datenbank enthaltenen Information gewünscht ist, muß diese Information in schnell erfaßbarer Form zugreifbar und geordnet sein. Dies setzt Abstraktion voraus, also das Herausziehen der wichtigen Informationsteile aus der gesamten Informationsmenge. Auf dieser kleineren und leichter überschaubaren Menge kann durch eine oder mehrere partielle Ordnungsrelationen (etwa Instanziierung oder Spezialisierung) die Übersichtlichkeit weiter gesteigert werden.

Variierbarkeit: Da keine wirklichen Standardbausteine oberhalb einer sehr elementaren Ebene existieren, muß die Datenbank in der Lage sein, vielfältige, oft nur leicht variierte Modellbausteine effizient zu speichern. Diese Eigenschaft umfaßt Ausnahmen (z.B. der Pinguin als nicht fliegender Vogel) wie auch Varianten von Objekten, die sich nur in wenigen Punkten unterscheiden (das Automodell als Limousine, Coupé oder Kombi; der Reaktor mit Kühlschlange oder Kühlmantel).

Modifizierbarkeit: Wegen der im Modellierungs–Simulations–Zyklus auftretenden Änderungen und der bei der Wiederverwendung häufig notwendigen Adaptionen (vgl. letzten Abschnitt) muß die Datenbank die einfache Modifikation der Modelle unterstützen.

Versionierbarkeit: Ältere Versionen und alternative Lösungen von Modellen *und* Modellbausteinen, die bei Korrekturen oder explorativem Vorgehen entstehen, müssen zu Dokumentations- und Wiederverwendungszwecken auf Wunsch aufgehoben werden und der Zugriff auf sie ermöglicht werden

Skalierbarkeit: Die Datenbank muß auch auf die bei größeren Modellierungsprojekten zu erwartende Datenmenge (vgl. S.8) einen effizienten Zugriff bieten.

Die Zusammenhänge dieser Anforderungen an das Datenmodell mit den in Abschnitt 1.1.2 genannten Anforderungen an die Entwicklungsumgebung zeigt Bild 1.5.

1.2.3 Ziele dieser Arbeit

Wir haben gesehen, daß aufgrund der gestiegenen Anforderungen an Detailiertheit und Geschwindigkeit der Prozeßmodellierung Werkzeuge zur Computerunterstützung der Modellierung benötigt und von den Anwendern auch gewünscht werden. Diese Werkzeuge benötigen, insbesondere wenn sie gemeinsam eingesetzt werden sollen, eine Datenbank zur Speicherung ihrer (Zwischen)Ergebnisse. Das Datenmodell dieser Datenbank (d.h. sowohl ihre Datenmodellierungssprache als auch das logische Schema) sowie die vom DBMS angebotenen Dienste müssen dem Wunsch entsprechen, Modellobjekte unkompliziert, effizient und wiederauffindbar zu speichern. Hierbei müssen sowohl die sich aufgrund der angestrebten Wiederverwendung ergebenden als auch die aus dem Anwendungskontext stammenden Anforderungen erfüllt werden.

Abbildung 1.5 zeigt, wie sich die Anforderungen an Entwicklungsumgebungen auf die Anforderungen an das Datenmodell abbilden lassen und wie sich diese wiederum auf Repräsentation und Dienste zur Wiederverwendung auswirken. Die auf den verschiedenen Ebenen nicht weiter verbundenen Anforderungen fallen, wie erwähnt, in andere Teilbereiche.

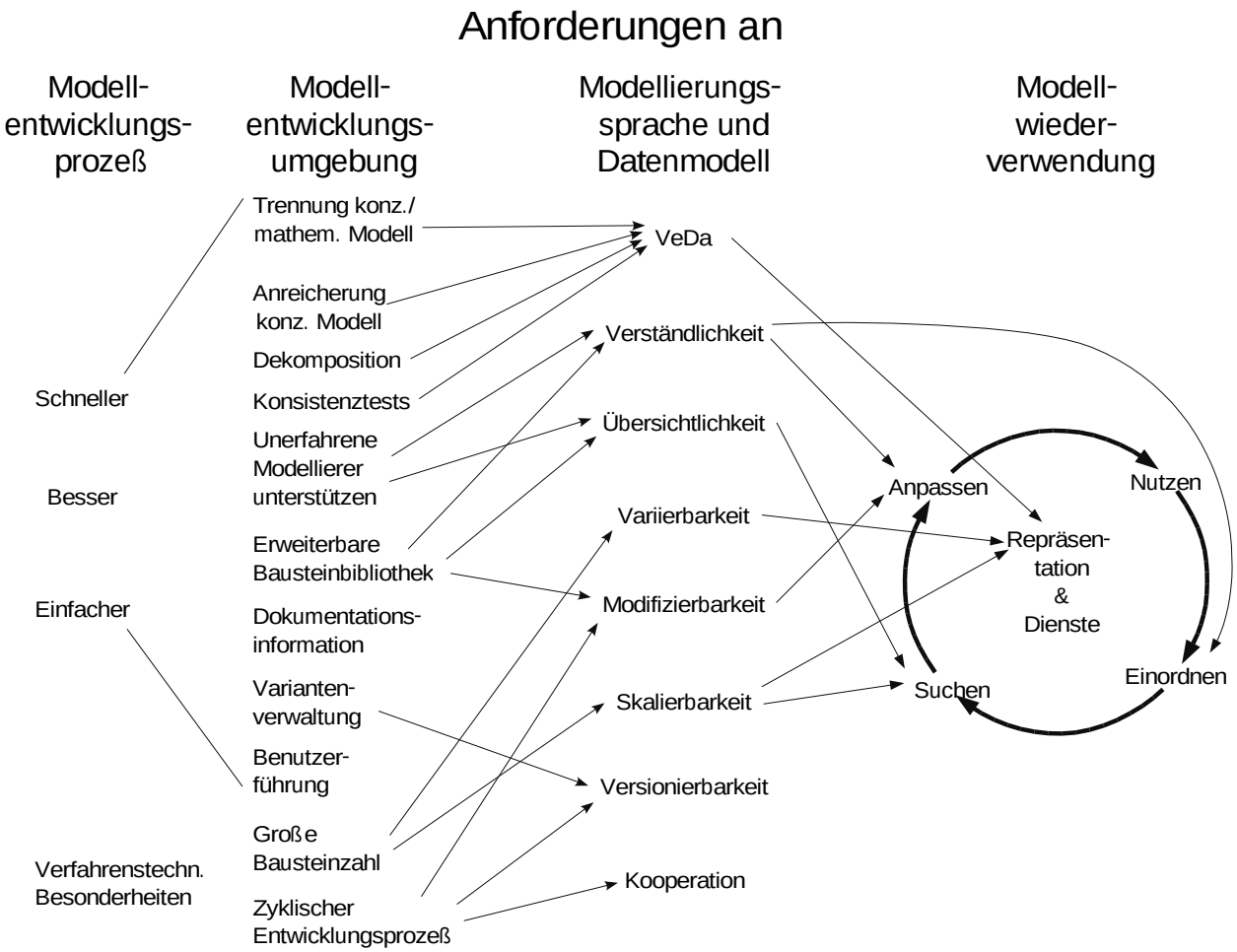


Abbildung 1.5: Zusammenhang der Anforderungen

Bei der Betrachtung dieser Anforderungen muß berücksichtigt werden, daß diese Arbeit im Rahmen mehrerer auf VeDa basierender Projekte stattfand, so daß sich Daten- und Objektmodell an VeDa orientieren müssen. Dies bedeutet unter anderem, daß statt des oben erwähnten objektorientierten Modells ein framebasiertes (vgl. [Bibel et al. 1996, Kap.2.5]) zum Einsatz kommt. Hauptunterschied ist die fehlende Verkapselung, was zwar für Programmiersprachen als gefährlich gilt [Nagel 1990], aber z.B. in [Stonebraker et al. 1990] für Datenbanken und in [Krishnan et al. 1991] für (mathematische) Prozessmodelle gefordert wird (s. Kap. 2.3.1). Zudem beinhaltet VeDAs Metamodell ein Device–Connection–Konzept sowie Vorkehrungen zur Dekomposition von Prozeßmodellen (vgl. Kapitel 3), die bereits teilweise zur geforderten Übersichtlichkeit und Verständlichkeit beitragen.

Diese Arbeit befaßt sich somit primär mit der Spezifikation einer objektorientierten/objektbasierten Modellierungssprache für VeDa, die sowohl die vorgestellten Konzepte der Wiederverwendbarkeit als auch der Variierbarkeit und Modifizierbarkeit unterstützt. Zudem werden die zur Realisierung dieser Eigenschaften notwendigen Dienste definiert. Die Arbeit berücksichtigt dabei aus den in den Abschnitten 1.1.2 und 1.2.2 genannten Gründen auch Speichereffizienzaspekte und Verständlichkeit für nicht datenbankgeschulte Benutzer.

Folgende Fragen testen also das Objektmodell und ein darauf aufbauendes DBMS auf Anforderungserfüllung:

- Können alle gewünschten Zusammenhänge in einem Modell dargestellt werden? (Mächtigkeit der Repräsentation)
- Welcher zusätzliche Aufwand entsteht bei der Repräsentation ähnlicher Modelle? (Effizienz der Repräsentation)

- Kann eine intuitive Hierarchie von Modellbausteinen erstellt werden? (Übersichtlichkeit)
- Wie weit weicht diese 'optimale' Hierarchie von einer Hierarchie ab, die im Laufe der Modellierung entsteht? (Einordnung)
- Läßt sich durch Modifikation vorhandener Modellteile ein neues, ähnliches Modell erstellen? (Adaption)
- Werden die 'intuitiv' ähnlichsten Bausteine gefunden? Treten Unterschiede bei vorgegebener und entstandener Bausteinhierarchie auf? (Suche)

Die genaue Bedeutung dieser Fragen wird sich in den folgenden Kapiteln klären. Zuerst werden Sprache und Metamodell von VeDa beschrieben, das im Rahmen dieser Arbeit auf eine formale Grundlage gestellt wurde und die Basis für die weiteren Untersuchungen bildet. Anhand eines Beispiels aus der Prozeßtechnik erläutert Kapitel 3 dann die Anwendungsproblematik nochmals genauer und skizziert ein die obigen Anforderungen erfüllendes System. Den allgemeinen Problemen der objektorientierten Repräsentation ingenieurwissenschaftlicher Modelldaten und der dazu vorgeschlagenen Ansätze widmet sich Kapitel 4, das insbesondere auch auf die in Abschnitt 1.1.4 aufgeführten Systeme näher eingeht. Anschließend stellt Kapitel 5 zwei Konzepte vor, durch deren Anwendung in VeDa die gewünschten Eigenschaften erreicht werden. Kapitel 6 formalisiert beide Konzepte aufbauend auf der formalen Grundlage von *ConceptBase* und Kapitel 7 legt dar, warum die vorgestellte Modellierungssprache die Anforderungen letztlich nicht erfüllt und mit welchen Problemen bei entsprechenden Ansätzen grundsätzlich zu rechnen ist.

Kapitel 2

Modellierungssprache und Datenmodell VeDa

Die vorliegende Arbeit nutzt und erweitert das Verfahrenstechnische Datenmodell VeDa. VeDa wird seit 1990 von Marquardt et al. zu dem Zweck entwickelt, Wissen über die Modellierung verfahrenstechnischer Prozesse sowohl auf systematischer als auch auf Einzelfallebene zu repräsentieren. VeDa umfaßt neben dem domänenspezifischen Modell die Definition der Modellierungssprache, in der dieses Datenmodell notiert wird. Diese Arbeit verwendet somit eine andere Terminologie als etwa [Marquardt 1996; Nagel et al. 1995; Maffezzoni und Girelli 1998], die das domänenspezifische Modell (in Teilen) mit zur Modellierungssprache zählen[1]. Wie viele Ansätze zur Datenrepräsentation im Ingenieurbereich (vgl. etwa [Kemper und Moerkotte 1994]) ist auch VeDas Datenmodellierungssprache (im folgenden kurz VDDL[2] genannt.) objektorientiert, XSgenauer framebasiert. Die framebasierte Notation kann auf den Einfluß von Wissensrepräsentationssprachen wie KL-ONE [von Luck 1990] auf VeDas Entwicklung zurückgeführt werden. Im Verlauf der Überarbeitungen der letzten Jahre wurden wissensrepräsentationsnahe Konstrukte jedoch zugunsten einer strenger typisierten, mehr datenbankbezogenen Sicht zurückgedrängt [Baumeister und Marquardt 1998]. Hierbei erfolgte eine Aufteilung von VeDa in vier Abstraktionsebenen (vgl. Bild 2.1), an denen sich auch die weitere Struktur dieses Kapitels orientiert. Eine ähnliche Aufteilung empfiehlt beispielsweise der IRDS–Standard [ISO 1990].

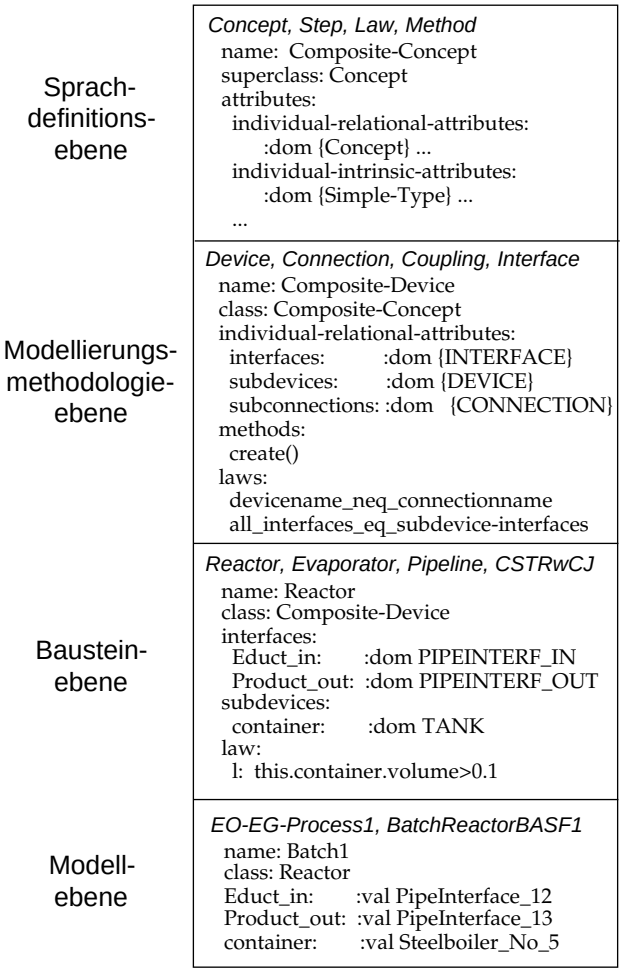


Abbildung 2.1: Die vier Ebenen von VeDa jeweils mit Beispielklasse (aus [Baumeister 1996])

Die **Sprachdefinitionsebene** enthält die Modellierungssprache, in der alle darunterliegenden Ebenen repräsentiert werden. Im Unterschied zu Sprachen wie *ConceptBase*, CLOS und Smalltalk, bei

[1] Die daraus häufig folgende Vermischung der beiden Ebenen wird in Abschnitt 2.3.5 analysiert.

[2] Die Bezeichnung "VDDL" ergibt sich analog zur Aufteilung in data definition langage (DDL) und data modification language (DML), die bei Datenbanksprachen Standard ist [Blaser et al. 1987].

denen mehr oder weniger große Teile des Sprachmodells für den Benutzer zugreifbar in der Sprache selbst notiert sind, werden Syntax und Semantik von VDDL lediglich im Sprachbeschreibungsdokument [Baumeister und Marquardt 1998] definiert.

In der **Modellierungsmethodologieebene** befinden sich die Konzepte zur Repräsentation verfahrenstechnischer Prozeßmodelle, also das domänenspezifische Metamodell. Die Konzepte dieser Ebene definieren abstrakt die wesentlichen Bestandteile eines Prozeßmodells und strukturieren so die in der nächsten Ebene liegenden Bausteine. Da sich diese Methodologie nur aufgrund langfristiger Forschungen ändert, erscheint sie für den Modellierer stabil und damit quasi als Sprachbestandteil. Die entsprechenden Teile VeDas werden u.a. in [Souza und Marquardt 1998; Krobb et al. 1998] definiert.

Die **Modellbausteine** der dritten Ebene ähneln den in Kapitel 1 im Zusammenhang mit der blockorientierten Modellierung erwähnten Blöcken. Sie abstrahieren von Parametern und Details der im Modell vorkommenden Komponenten, um die für eine Lösung wichtigen Aspekte in den Vordergrund zu stellen. Im Unterschied zur blockorientierten Modellierung ermöglichen Modellbausteine, durch Dekomposition verschiedene Verfeinerungsebenen auszudrücken, Verknüpfungen zwischen verschiedenen Entwurfsstufen zu ziehen sowie Modifikationen durch den Nutzer. Die zwischen den Bausteinen bestehende Spezialisierungshierarchie soll einem Modellierer die Auswahl eines seinen Anforderungen am besten entsprechenden Modellbausteins erleichtern. Ihre Modifikation und insbesondere Erweiterung *durch den Modellierer* ist erwünscht, da in ihnen das spezifische, komponentenbezogene Wissen der Modellierer erfaßt werden soll.

Die unterste Schicht enthält die eigentlichen **Modelle**, also die konkreten Beschreibungen der Realität, die der Modellierer für das jeweilige Problem am angemessensten hält. Diese Modelle enstehen durch Instanziierung der Modellbausteine, also dem Einsetzen von Parametern und Modellkomponenten in die Klassen der zweituntersten Ebene. VeDa selbst beschäftigt sich nur am Rande mit der Definition dieser Instanzen, da seine Zielrichtung die Gewinnung und Repräsentation von abstraktem Modellierungswissen ist. Implementierungen von VeDa müssen diese Schicht allerdings berücksichtigen.

Die Abhängigkeiten zwischen den Ebenen entsprechen der wohlbekannten Instanziierungsbeziehung. Es handelt sich also um eine vierstufige Instanzen–Klassen–Metaklassen–MetaMetaklassen–Hierarchie. In Kapitel 5.4.2 werden wir allerdings sehen, daß diese Aufteilung in Ebenen nicht nur Vorteile hat.

Die folgenden Abschnitte werden die Sprachdefinitionsebene, also VDDL, und die Modellierungsmethodologieebene genauer erläutern und in die objektorientierte Begriffswelt einordnen. Um auch Lesern, die nur objektorientierte Grundkenntnisse besitzen, das Verständnis zu erleichtern, werden komplexere Konzepte jeweils in Grundlageneinschüben erläutert. Diese können natürlich übergangen werden, falls die entsprechenden Begriffe bereits bekannt sind.

2.1 Sprachsyntax und -semantik

Die Syntax von VeDas Datendefinitionssprache VDDL geht auf die von Minsky [Minsky 1975] vorgeschlagenen und von anderen weiterentwickelten Frames zurück [Bibel et al. 1996, Kap. 2.6]. Da z.B. in [Russel und Norvig 1995, S.318f] gezeigt wird, wie sich Frame–Netze auf die Prädikatenlogik erster Stufe abbilden lassen, scheint die Frage nach einer (formalen) Definition der Semantik von VDDL zunächst beantwortet zu sein. VDDL geht aber über die Möglichkeiten von Frame–Netzen hinaus, indem es aus Programmiersprachen und objektorientierten Datenmodellen bekannte Konzepte wie strukturierte Typen, Methoden und Vererbung bietet. Insbesondere die Methoden führen hierbei schnell in den Bereich der Funktionskalküle bzw. aufgrund des objektorientierten Umfeldes zu den in [Abadi und Cardelli 1996] vorgestellten Objektkalkülen. Auch die Definition von VDDL durch Axiomatisierung einiger Grundbausteine und anschließender Notation in sich selbst, wie sie

beispielsweise bei *ConceptBase* oder CLOS genutzt wird, gelang bisher nicht vollständig, anders als durch die oberste Ebene von Bild 2.1 suggeriert. Aus diesen Gründen wird neben der nur kurz angerissenen Syntax die Semantik hier in natürlicher Sprache angegeben. Anschließend werden die hinter den Modifikationen im Vergleich zur DDL VEDAs aus [Marquardt et al. 1992] und die hinter einigen Abweichungen von den Standardkonzepten der Objektorientierung stehenden Entwurfsentscheidungen erläutert.

2.1.1 Sprachdefinition

Die Typisierung „objektorientiertes Datenmodell“ erlaubt noch immer einen weiten Interpretationsspielraum, wie verschiedene Übersichten und Standardisierungsversuche (etwa [Atkinson et al. 1989; Motschnig-Pitrik und Mylopoulos 1992; Heuer 1992]) zeigen. Als erstes soll darum eine kurze Einordnung in einige häufig auftretenden Variationsmöglichkeiten des objektorientierten Modells vorgenommen werden, bevor auf die Sprachdetails eingegangen wird:

VDDL ist klassenbasiert: Analog zu den meisten Smalltalk- oder C++–nahen Datenmodellen kommerzieller ODBMS, im Unterschied aber zu Programmiersprachen wie Self [Ungar und Smith 1987] und Datenmodellen wie BOS [n-dim Group 1995a] verteilt VDDL Wert- und Typinformation auf zwei unterschiedliche Sprachkonstrukte: Instanzen und Klassen.

Klassen definieren Typ, Implementierung und Extension: Definitionen der Objektorientierung ordnen Klassen eine Vielzahl von Aufgaben zu (vgl. S.25). Verschiedene Autoren schlagen darum die Trennung von Typ- und Klassendefinition [LaLonde und Pugh 1991] bzw. Typ- und Extensionsdefinition [Tresch 1995; Heuer 1992] vor. VDDL behält aus den in Abschnitt 2.3 dargestellten Gründen den ersten Ansatz bei.

Framenotation: Alternativ zur normalerweise verwendeten Notation von Objektmodellen als Objekte mit Attributen und Methoden (Objekt–Attribut-Notation) kann, wie in VEDA geschehen, auch die Frame–Slot–Facetten–Notation verwendet werden. Sie erlaubt eine einfachere Darstellung zusätzlicher semantischer Beziehungen zwischen Objekten und Attributen (vgl. hierzu auch den Abschnitt über Model.La in Kapitel 54).

Objekte und Werte werden getrennt: Objektorientierte Datenbanken unterscheiden aus Performancegründen normalerweise zwischen Objekten und Werten (vgl. [Beeri 1989; Kemper und Moerkotte 1994]). Werte sind Instanzen primitiver (integer, real, string) oder strukturierter Datentypen (set, list, array) ohne eigene Identität, während Objekte Instanzen von Klassen mit eigener, von den enthaltenen Werten unabhängiger Identität sind.

Multiple, kovariante Vererbung: VDDL erlaubt mehrere Oberklassen sowie die kovariante Spezialisierung von Attributen und Methodenparametern (siehe Abschnitt 2.3.3).

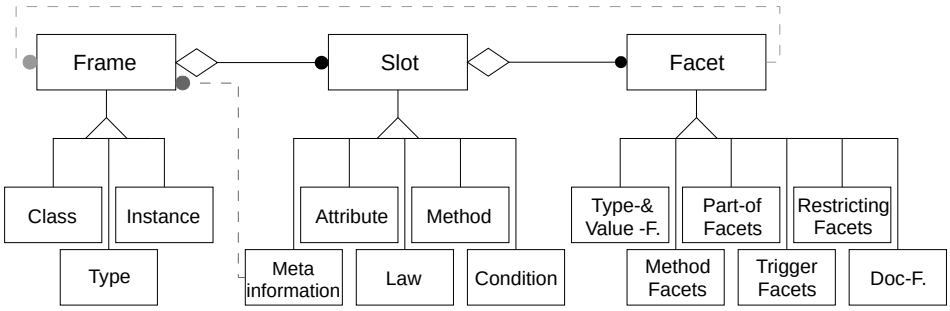


Abbildung 2.2: Frames, Slots und Facetten in VEDA

Das OMT–Diagramm in Bild 2.2 gibt einen Überblick über die Repräsentation von VeDa, die genaue Syntax findet sich in [Baumeister und Marquardt 1998]. In einer framebasierten Notation wie der von VeDa werden alle Eigenschaften eines Objektes bzw. einer Klasse in einem sogenannten Frame gesammelt. Die Eigenschaften selbst bezeichnet man als Slots, die Eigenschaften dieser Slots wiederum als Facetten. Der Kreis schließt sich, indem Facetten auf andere Frames oder Slots verweisen.

Frames

Frames bestehen aus einem Kopf mit Verwaltungsinformationen und einem Rumpf mit den Slots. Frames können Klassen, Instanzen und Typdefinitionen repräsentieren. Die Bedeutung von Klassen und Instanzen entspricht der aus der Literatur bekannten (siehe etwa [Motschnig-Pitrik und Mylopoulos 1992; Heuer 1992]). Typframes ermöglichen zusätzlich die Benennung von Typdefinitionen.

Slots

Slots speichern die in einer Klasse bzw. einer Instanz vorhandene Information. Sie bestehen aus einem Slotnamen und einer Ansammlung von Facetten, die die eigentliche Information enthalten. Der Großteil der in Bild 2.2 aufgeführten Slotarten kann lediglich in Klassenframes verwandt werden. So besitzen Typframes neben einigen Verwaltungsinformationen nur einen einzigen Slot zur Definition des Typs. Instanzen besitzen nur Attributslots, deren einzige Facette die Wertfacette `:val` ist (in der Literatur meist 'filler' oder Füllsel genannt). Aus diesem Grund werden Instanzen in vereinfachter Notation repräsentiert.

In den Klassen treten die folgenden Slotarten auf: Restriktionen (`laws:`–Slots) zur Einschränkung der erlaubten Werte von Attributen werden in VDDL als Ausdrücke der Prädikatenlogik erster Stufe in pränexer Normalform notiert[3]. Methoden (`methods:`–Slots) geben extern oder intern (in einer Abwandlung der `laws:`–Notation) implementierte Aktionen auf den Instanzen an. Durch `conditions:` und `scheduling-conditions:` wird die Sprache speziell für die Repräsentation von Ablaufschritten erweitert (vgl. [Krobb 1997]). Schließlich definieren die `attributes:`–Slots die Struktur einer Klasse in Form von Werten und Verweisen auf andere Klassen. Sie teilen sich entsprechend weiter auf in Verweis- und Internattribute (`relational/intrinsic`) sowie in Klassen-/Instanzattribute (`shared/individual`). Da diese vier Attributkategorien die für einen Attributslot erlaubten Facetten bestimmen, werden sie nicht als Facette sondern als Slotart realisiert.

Zusätzlich existieren noch Slots, die Metainformation wie etwa die Oberklassenbeziehung repräsentieren. Sie werden unten bei der Behandlung des Frame–Kopfes genauer beschrieben.

Facetten

Facetten spezifizieren allgemeine und besondere Eigenschaften der Slots. Sie bestehen aus dem Facettennamen sowie einer Spezifikation, die je nach Facette ein Typ, ein boolescher Wert oder ein Attribut- bzw. Methodenname ist. Insbesondere boolesche Facetten können auch als eine Variante zur Klassifizierung von Attributen aufgefaßt werden (vgl. [Motschnig-Pitrik und Mylopoulos 1992, S. 73f]), d.h. `:facette t` bedeutet, daß das Attribut zur Attributkategorie <facette> gehört.

Die meisten der vorhandenen **Facetten** sind nur **für Attribute** erlaubt. Deren Hauptfacetten, `:dom` bzw. `:val`, geben Typ und Wert des Attributs an und entsprechen somit den analogen Einträgen der Objekt–Attribut–Notation. Durch zusätzliche Facetten können jedoch weitere Eigenschaften und Restriktionen der Attribute angegeben werden:

[3] Die verschiedenen Begriffe aus der Logik werden beispielsweise in [Ebbinghaus et al. 1992] erläutert.

Teil–Ganzes–Facetten: `:comp` gibt an, ob das durch das Attribut referenzierte Objekt als Teil des referenzierenden Objekts zu betrachten ist. `:dep` und `:exc` entsprechen den in [Kim 1990] definierten Eigenschaften der Existenzabhängigkeit des referenzierten Objekts und der Exklusivität der part-of Referenz. Für interne ('intrinsic') Attribute gelten die `:dep` und `:exc` Facetten zwangsläufig.

einschränkende Facetten: Außer durch die Typfacette `:dom` können die möglichen Werte eines Attributs durch weitere Facetten eingeschränkt werden. Die `:inv`–Facette gibt den Namen des Attributs des referenzierten Objekts an, das auf das referenzierende Objekt zurückverweist und stellt dadurch gleichzeitig die kreuzreferentielle Integrität [Schmidt 1987, S.36f] sicher. `:req` verbietet den Wert `Nil` für ein Attribut, und `:readonly` verbietet jegliche Schreibzugriffe nach der Initialisierung.

Trigger–Facetten: Trigger–Facetten lösen Methodenaufrufe aus, wenn ein Zugriff auf ihr Attribut stattfindet. Sie können beispielsweise zur Implementierung von Konsistenzbedingungen, datenmodellseitigen Cache–Strategien oder zur Informationsbereitstellung genutzt werden. Zum Einsatz kommen sechs verschiedene Triggertypen, die sich in der Art des Zugriffs (`:do_X_read` oder `:do_X_write`) und dem Zeitpunkt der Triggerausführung (`:do_before_Y`, `:do_instead_Y` oder `:do_after_Y`) unterscheiden. Diese Einteilung bietet gegenüber den `:act` und `:get` Facetten des VEDAS in der Version aus [Marquardt et al. 1992] den Vorteil der genaueren Bestimmbarkeit des Ausführungszeitpunktes.

Dokumentationsfacette: `:doc` erlaubt die Angabe eines Strings zu Dokumentationszwecken.

Restriktionen, also `laws`, besitzen theoretisch drei Facetten: `:doc` wie bei den Attributen zur Dokumentation, `:refinable` zur Freigabe der Verfeinerung in Unterklassen sowie eine Facette für die eigentliche Restriktion. Die Restriktion wird in einer an [Urban 1989] angelehnten Schreibweise für Terme der Prädikatenlogik 1. Stufe (PL1) notiert, die es erlaubt, Aussagen aus dem Kern (also dem quantorenfreien Teil) des logischen Ausdrucks herauszuziehen und dem quantifizierenden Teil zuzuordnen. Diese rein syntaktische Erweiterung vereinfacht die Formulierung logischer Ausdrücke, da die Einschränkungen der quantifizierten Variablen und die eigentliche Aussage getrennt werden. Syntaktisch werden logische Ausdrücke mittels Schlüsselwörtern wie `:all`, `:some`, `:and` und `:eq` notiert. Da ein solcher logischer Ausdruck immer mit `:all`, `:some` oder `:holds` beginnt, wird auf die explizite Angabe einer Restriktionsfacette (etwa `:rule`) verzichtet.

Methodenslots können drei Facetten enthalten. Zum einen erfolgt die Angabe der Signatur einer Methode mittels der `:interface`–Facette in einer der mathematischen Funktionsdefinition ähnlichen Notation: <`Parametername`$_1$ > <`Parametertyp`$_1$ > `x` <`Parametername`$_2$ > <`Parametertyp`$_2$ > `x ...->` <`Ergebnistyp`> . Zum anderen wird für die Methodenimplementierung entweder mittels `:extern` auf ein externe Routine verwiesen oder hinter der `:intern`–Facette folgt eine Methodenimplementierung mittels der auch in den Restriktionen verwandten Notation für PL1–Terme.

Der Frame–Kopf

Zusätzlich zu den oben erläuterten Slots des Frame–Rumpfes existieren einige weitere Slots im Kopf jedes Frames, die Verwaltungsinformation und die Beziehungen zwischen den Frames enthalten. Bei Klassen sind diese Framebeziehungen im `superclasses:`– und `metaclasses:`–Slot enthalten. Beide werden weiter unten erläutert.
Typframes enthalten den `typedef:`-Slot, der die Typdefinition enthält, die benannt werden soll.
Die zu einer Instanz gehörige Klasse gibt der `classes:`–Slot des Instanzenframes an. Vorläufig darf jeder Instanz nur eine Klasse zugeordnet werden. Die Erweiterung auf Multiple Instanziierung erfolgt erst in Kapitel 5.3. Instanziierung in VDDL bedeutet strikte Instanziierung, d.h. die Instanz darf keine Erweiterungen gegenüber der Klasse besitzen, wie sie KSL [Ibrahim et al. 1991] durch „Traps“ oder VODAK [Klas und Schrefl 1995] durch „Instanztypen“ zu erreichen versuchen.

Ober- und Unterklassen

Klassen werden im allgemeinen mittels einer Halbordnung in einer Ober-/Unterklassentaxonomie angeordnet. In VDDL verweist hierzu der `superclasses:`–Slot auf die Oberklassen einer Klasse. Die Halbordnung wird meist als Spezialisierungsrelation oder Vererbungsbeziehung bezeichnet und trägt damit zwei ihrer Eigenschaften im Namen: Oberklassen sind Generalisierungen ihrer Unterklassen, und die Unterklassen erben Struktur und Verhalten von ihren Oberklassen. Als dritte Eigenschaft kommt die Obertyprelation hinzu [Taivalsaari 1996; LaLonde und Pugh 1991], die einer Substituierbarkeitsrelation entspricht. Wie die meisten objektorientierten Datenmodelle realisiert VDDL alle drei Funktionen gemeinsam. Vor- und Nachteile einer Aufspaltung von Spezialisierung, Obertypen und Vererbung in VDDL werden im Rahmen der Typ–Klassen–Trennung in Abschnitt 2.3 diskutiert.

Die Ober-/Unterklassenbeziehung hat in VDDL die folgende Bedeutung:

- Die Unterklasse erhält alle Attribute, Methoden und Restriktionen der Oberklassen mit all ihren jeweiligen Facetten.
- Nicht vererbt werden hingegen der im Klassenkopf aufgeführte Name, der Kommentar und die Metaklassen[4]. Letztere werden nicht vererbt, da die Vererbung einiger Metaklassen wie `abstract` sie effektiv unbenutzbar machte [Jeusfeld 1997]).
- Geerbte Slots dürfen nur nach den in der Liste unten aufgeführten Regeln modifiziert werden.
- Mehrere Oberklassen sind erlaubt (Multiple Vererbung). Wird ein Slot gleichen Namens auf mehreren Wegen geerbt (horizontale Namenskonflikte nach [Knudsen 1988]) und gehen alle Slots auf einen Slot in einer gemeinsamen Oberklasse zurück, so muß in der erbenden Klasse ein Slot angegeben werden, der alle in Konflikt stehenden Slot nach den Regeln in der Liste unten spezialisiert. Gibt es eine solche gemeinsame Spezialisierung nicht, oder gehen die Slots auf unterschiedlichen Klassen zurück, so zählt dies als Fehler.
- Die Instanzen einer Unterklasse sind auch Instanzen der Oberklasse, insbesondere müssen alle Restriktionen der Oberklassen erfüllt werden. Allerdings wird keine Substituierbarkeit garantiert, d.h. Methodenaufrufe oder Attributzuweisungen können u.U. zu Laufzeitfehlern führen. Dies wird genauer in Abschnitt 2.3.3 ausgeführt.

Wie erwähnt dürfen geerbte Slots von der erbenden Klasse modifiziert werden. Allerdings muß dabei die Liste der folgenden Regeln beachtet werden, die, grob gesagt, nur eine kovariante Spezialisierung aller Attribute, Methodenparameter und -ergebnisse zuläßt[5].

- Die Domäne eines Attribut muß gleich bleiben oder spezialisiert werden. Hat das Attribut einen strukturierten Datentyp als Domäne, so muß die Struktur des Typs gleichbleiben, und die Typen der Komponenten dürfen spezialisiert werden. Bei Mengen- und Listenkardinalitäten gelten die natürlichen Enthaltenseinbeziehungen.
- Bei allen booleschen Facetten außer `:refinable` ist true spezieller als false. Bei `:refinable` ist false spezieller als true.
- Trigger und `:doc`-Facetten dürfen beliebig geändert werden.
- `:inv` und `:alias`-Facetten dürfen nicht verändert werden.
- Ein mittels `:init` angegebener Initialisierungswert spezialisiert jeden anderen Initialisierungswert, einschließlich des nicht vorhandenen, sofern er den korrekten Typ hat.
- Spezialisierungsbeziehungen zwischen PL1-Regeln sind allgemein unentscheidbar[6], d.h. sie können nicht immer überprüft werden. `laws` dürfen darum beliebig geändert werden, wenn

[4] Wie bereits erwähnt, werden Begriffe wie "Metaklasse" in entsprechenden Grundlageneinschüben erläutert. Zur Bedeutung von "Metaklasse" siehe darum Seite 21.

[5] Diese Regeln führen dazu, daß VDDLs Vererbungsregeln restriktiver sind als die vieler anderer objektorientierter Systeme (vgl. [Stefik und Bobrow 1986] und [Kim 1990, S. 44/45]), in denen die lokale Definition eines Slots/Attributs die geerbte überschreiben kann. Die hier gewählte Variante existiert so auch in KL–ONE [von Luck 1990] und erlaubt verbesserte Konsistenztests der Datenbank.

[6] Dies folgt direkt aus der Unentscheidbarkeit der Allgemeingültigkeit von PL1-Formeln (s.z.B. [Ebbinghaus et al. 1992]).

ihre `:refinable` Facette den Wert `t` hat, da wir andernfalls keine Änderungen zulassen dürften.

- Methodenimplementierungen leiden unter dem gleichen Problem der Nichtvergleichbarkeit. Die Methodenspezialisierung unterliegt darum nur der Beschränkung, daß Parameter- und Ergebnistypen der Signatur spezieller oder gleich sein müssen.
- `shared` Attribute sind spezieller als `individual` Attribute.

VeDas Ober-/Unterklassenbeziehung erlaubt also signaturkompatibel Modifikationen [Wegner und Zdonik 1988] in der erbenden Klasse. Sie sichert aber weder Substituierbarkeit (vgl. S. 25) zu, noch erlaubt sie Vererbungsausnahmen ([Meyer 1996], 'Cancellation' in [Wegner und Zdonik 1988]). Vor- und Nachteile dieser Modifikationsregeln finden sich in Kapitel 2.3.3.

Metaklassen

Die durch den `metaclasses:`–Slot referenzierten Sprachmetaklassen erlauben es, besondere Spracheigenschaften für einzelne Klassen zu aktivieren. Klassen können durch sie als nichtinstanziierbar (`abstract`) oder als nur disjunkt spezialisierbar (`disjunct`) gekennzeichnet werden. Metaklassen können zudem die Verwendung von `:comp`–Facetten (`composite-concept` vs. `elementary-concept`) ermöglichen oder tragen die für die Slots `conditions:` und `scheduling- conditions:` notwendige strukturelle und semantische Information (`has-condition`, `has-scheduling-condition`). Auch der in VDDL durch die Slots `class:`, `type:` und `instance:` am Anfang eines Frames selektierte Frametyp entspricht eigentlich einer Metaklasse, wird aber aus Vereinfachungsgründen nicht in `metaclasses:` erwähnt.

Grundlagen 2.1: Metaklassen, Metamodellierung

Wiederholt man den zwischen Instanz und Klasse liegenden Abstraktionsschritt erneut ausgehend von einer Klasse, so erhält man eine **Metaklasse**, also ein Konzept, das Struktur und Verhalten von *Klassen* abstrakt beschreibt.

Die Anwendungsmöglichkeiten der Metamodellierung sind vielfältig (vgl. [Baumeister 1996]). In Programmiersprachen werden sie vorwiegend eingesetzt, um Introspektion und Sprachadaptierbarkeit zu erlauben. **Introspektion** gibt einem Programm die Möglichkeit, Informationen über das eigene Schema zu verarbeiten, und wird beispielsweise von Smalltalk [Goldberg und Robson 1989] und der C++–Erweiterung Metaflex [Johnson und Palaniappan 1993] angeboten. Werden Metaklassen zur **Sprachadaptierbarkeit** genutzt, so sind mehr oder weniger große Teile der Sprachdefinition mit Hilfe von Metaklassen in der Sprache selbst definiert. Änderungen an den Metaklassen rufen dann Verhaltensänderungen der Sprache hervor. Dieser Mechanismus wird in geringem Ausmaß beispielsweise von Smalltalk oder der C++–Erweiterung aus [Chiba und Masuda 1993] sowie umfangreicher durch das Metaobject–Protocol von CLOS [Kiczales et al. 1991] angeboten.

Bei der Datenmodellierung werden Metaklassen häufig zur **Modellierungsmodellierung** eingesetzt. Wie in [Brinkkemper 1990, Kap. 2] ausgeführt wird hierbei die Modellierungssystematik eines Datenmodells in eine darüberliegende Metaebene destilliert. Dies kann ggf. mehrfach wiederholt werden. Brinkkemper nutzt dies zur Modellierung von Informationsystemen. Andere Anwendungen sind die Definition von Attributklassen [Mylopoulos et al. 1990; Klas und Schrefl 1995] oder die Datenbank–Integration [Jeusfeld und Johnen 1995]. Modellierungsmodellierung bieten die meisten Datenmodelle, die Metaklassen unterstützen, z.B. *ConceptBase* oder VODAK.

Die Modellierungsmethodologieebene von VeDa ist eine Anwendung von Metaklassen zur Modellierungsmodellierung. Die Sprachdefinitionsebene wäre, wenn sie explizit repräsentiert würde, eine Anwendung zur Sprachadaptierbarkeit.

Im `metaclasses:`–Slot dürfen nur die durch die Sprache vorgegebenen Metaklassen aufgeführt werden, die nicht mit den Modellierungsmethodologie–Metaklassen verwechselt werden dürfen, die im folgenden Abschnitt definiert werden. Diese stellen Struktur- und Verhaltensvorgaben für die die Modellbausteine repräsentierenden Klassen auf, während Sprachmetaklassen in gewissem Sinne die Sprache selbst ändern. Wie die Modellierungsmethodologie–Metaklassen in VDDL ausgedrückt werden können, behandelt Abschnitt 5.4.2.

Einige Beispielframes in VDDL finden sich in Abschnitt 3.2. Wesentlich umfangreichere Beispiele enthalten etwa [Souza und Marquardt 1998] und [Krobb et al. 1998]

2.2 Metamodell

VDDL selbst enthält als Datenmodellierungssprache kein verfahrenstechnisches Wissen. Die grundlegenden Modellierungsprinzipien werden vielmehr in der im folgenden vereinfacht beschriebenen Metaebene festgelegt. Eine umfangreichere, über den notwendigen Erklärungsbedarf für diese Arbeit hinausgehende Definition findet sich in [Souza und Marquardt 1998].

Stark vereinfachend kann man die Metaklassen von VeDa mit vier abstrakten Grundschemata darstellen. Auf oberster Ebene findet eine Aufteilung der Konzepte nach den schon früher erwähnten verschiedenen **Stufen des Entwicklungsprozesses** statt, was als OMT–Diagramm in Abbildung 2.3 dargestellt wird. Jede der drei Klassen repräsentiert dabei jeweils die abstrakte Oberklasse für Konzepte aus der jeweiligen Sicht. Einige beispielhafte Unterklassen sind jeweils unter den Klassen aufgeführt.

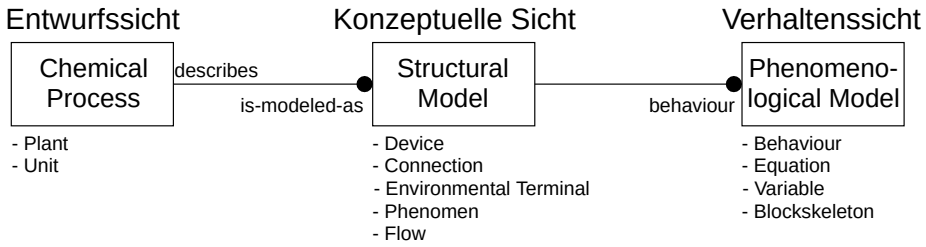


Abbildung 2.3: Grobstrukturierung des Metamodells von VeDa

“Chemical Process“ sind Konzepte der Designebene, die sich weiter z.B. in “Plant“ und “Unit“ gliedern. Sie dienen zur Verknüpfung von Designanforderungen (z.B. zu erbringender Funktionalität) und -dokumenten (etwa Flowsheets) mit den Modellen. Das “Structural Model“ (hier auch konzeptionelles Modell genannt) enthält die strukturellen, hierarchisch aggregierten Modelle des verfahrenstechnischen Prozesses, sowie Informationen wie auftretende Phänomene, Stoffe und Zwangsbedingungen, die bei der Erstellung der Verhaltensbeschreibung hilfreich sind. Diese wird dann mittels der “Phenomenological Models“ dargestellt, wobei je nach verwendetem Simulatortyp mathematische Gleichungen und Variablen oder aber Verhaltensblöcke zum Einsatz kommen.

Die Trennung dieser Bereiche erhöht Übersichtlichkeit und Verständlichkeit der Modelle[7], zieht allerdings auch teilweise parallele Spezialisierungs- und Aggregationshierarchien bzw. auf Klassenebene nur schwach gekoppelte Modelle nach sich. Im weiteren konzentrieren wir uns auf den am besten ausgearbeiteten Bereich der strukturellen Modelle, in dem sich auch die Beispiele aus Kapitel 3 wiederfinden.

Den strukturellen Modellen liegt als Grobstruktur ein bipartiter Graph zugrunde. Knoten sind hierbei **“Devices“ und “Connections“**, die über “Couplings“ als Kanten verknüpft werden. Bild 2.4 zeigt dies erst einmal in VeDa–Flowsheet–Darstellung, da selbst ein stark vereinfachtes OMT-Diagramm wie 2.5 die Zusammenhänge nicht so gut verdeutlichen kann. Die Devices X und Y sind hier über die Connection Z miteinander verbunden. Connections sind im Gegensatz zu anderen Modellierungssprachen (Omola, Model.La, vgl. Abschnitt 4.3.1) vollwertige Objekte, da sie eigenes Verhalten besitzen. Sie bestimmen aus den von den Devices gelieferten Prozeßzustandsgrößen die treibenden Kräfte der sie durchquerenden Flüsse und aus diesen wiederum die Flüsse selbst. Devices berechnen analog ihre Zustandsgrößen aus den Flußgrößen, die ihnen die sie umgebenden

[7]Schön läßt sich dies in pdXi [ISO 1998] sehen, mit in ’performance parameter’, ’design criteria’, und ’assembly’ aufgespaltenem, verständlichem ’shell and tube heat exchanger unit’–Modell und nicht aufgeteiltem, unübersichtlichem ’mass transfer equipment’–Modell.

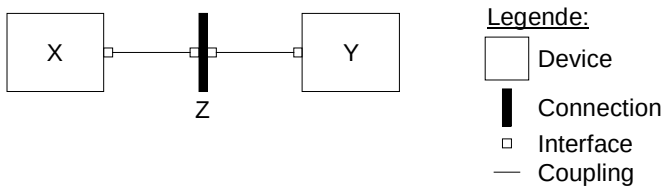


Abbildung 2.4: Devices und Connections als Fließbild

Connections mitteilen. Mit diesen wenigen Grundbausteinen lassen sich quasi alle verfahrenstechni-

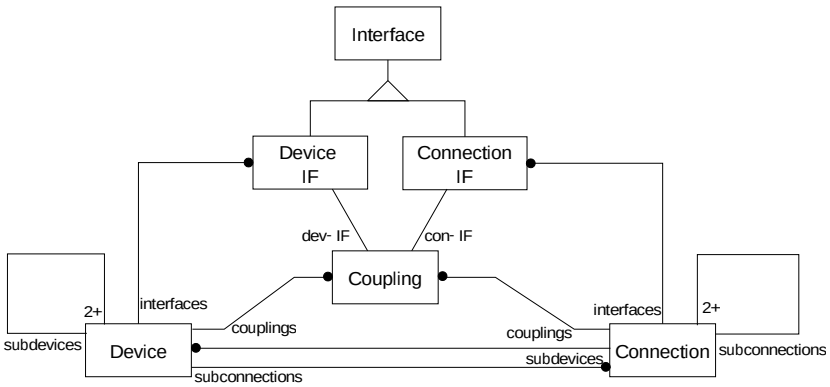


Abbildung 2.5: Devices und Connections als OMT–Diagramm

schen Prozesse strukturell modellieren. Abbildung 2.5 zeigt die selben Konzepte stark vereinfacht gegenüber [Souza und Marquardt 1998] in Form eines OMT–Diagramms. Es fällt auf, daß nicht nur Devices aus Subdevices und Subconnections aggregiert werden können sondern auch Connections, etwa um die Abläufe an einer Phasengrenze genauer modellieren zu können. Der notwendige 'Verwaltungsoverhead' in Form zusätzlicher Metaklassen wird im folgenden erläutert.

Der 'Overhead' äußert sich beispielsweise in der Tatsache, daß Abbildung 2.5 nur den zusammengesetzten Fall modelliert, aber auch elementare Bausteine notwendig sind. Zudem trennt VeDa die in obiger Abbildung noch weitgehend monolithisch aussehenden Devices und Connections (und mit etwas anderer Benennung auch Variablen) jeweils in **Schnittstelle**, Hauptmodell **und Implementierung** auf. Dies ähnelt der Unterteilung von Funktionen in Programmiersprachen in Deklaration (Signatur) und Definition (Implementierung). Abbildung 2.6 zeigt die dazu notwendigen Konzepte. Interfaces repräsentieren Teile der Spezifikation eines Bausteins, nämlich Zahl und Art der ein-

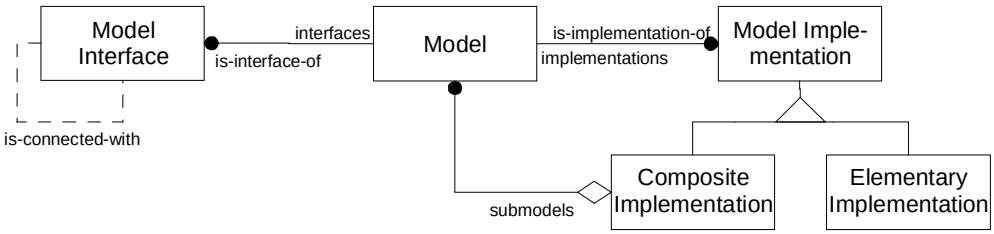


Abbildung 2.6: Trennung von Modell, Schnittstelle und Implementierung

und austretenden Phasen bzw. Flußgrößen. Die Implementierung hingegen enthält entweder die Dekomposition des Modells in weitere Bausteine ('composite implementation') oder Angaben zur Verhaltensimplementierung wie Phänomene, Geometrie und auftretende Prozeßgrößen ('elementary implementation'). Zusammen mit der Aufteilung in Devices und Connections verringert diese Trennung von Interfaces und Implementierung die informatische Kopplung zwischen Modellteilen, da die Verschaltungsmöglichkeiten sowie die ausgetauschten Informationen nur durch die Interfaces bestimmt werden.

Schließlich enthält VeDa noch einfache Mechanismen zur **Alternativenverwaltung** im Metamodell (siehe Abbildung 2.7). Alternative Modelle sind über die 'active'- und 'possible'-slots untereinander verbunden. Auf das jeweils aktive Modell verweisen alle alternativen Modelle per 'active'-Slot. Die Auflösung dieses Verweises muß in allen Methoden/Regeln, die sich nur auf die aktive Alternative beziehen sollen, durch den Werkzeugprogrammierer erfolgen.

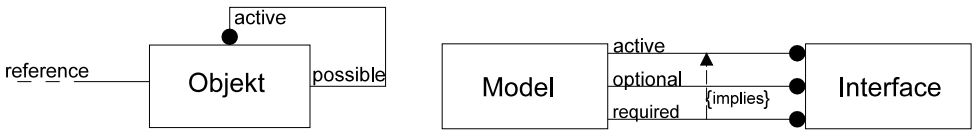


Abbildung 2.7: Alternativenverwaltung

Interfaces können drei Zustände annehmen: 'active', 'required', was gleichzeitig 'active' erzwingt, und 'optional', was das Zu- und Abschalten des Interfaces zu einem Modell erlaubt. Die Zustände werden in VeDa über die Zugehörigkeit zu den drei entsprechenden Slots ausgedrückt.

Wie wir gesehen haben, kann mit Hilfe der Modelle der Metaebene recht einfach die Grundphilosophie von VeDa beschrieben werden, wenn auch die vollständige Metaebene etwas komplizierter ist als hier dargestellt.

2.3 Diskussion

Die Beschreibung von VeDa in den vorhergehenden Abschnitten überging einige, nur kurz erwähnte Entwurfsentscheidungen, um ein zusammenhängenderes Bild von VeDa geben zu können. Darum sollen hier nun die Fragen des fehlenden Information–Hiding, der Trennung von Typen und Klassen und, damit zusammenhängend, der Sinnhaftigkeit der gewählten Spezialisierungsregeln behandelt werden.

2.3.1 Objektorientiert ohne Kapselung?

Verkapselung ('Encapsulation' oder auch 'information hiding') bezeichnet in objektorientierten Datenmodellen das Verbergen der Implementierung einer Klasse zugunsten ihrer Schnittstelle. Bei vollständiger Verkapselung sind von außen nur noch die Signaturen bestimmter Methoden sichtbar. Die Klassen werden also zu abstrakten Datentypen. Ohne Verkapselung sind alle Attribute und alle Methodenimplementierungen von außen sichtbar, so daß sich die Klassen um Methoden erweiterten Records annähern. Verkapselung vereinfacht das Verständnis einer Klasse, da im besten Fall nur die Schnittstelle verstanden werden muß. Insbesondere aber erlaubt die Verkapselung den Aufwand für Implementierungsänderungen auf die jeweilige Klasse zu begrenzen. Verkapselung wird darum von vielen Autoren als wesentliches [Atkinson et al. 1989; Kemper und Moerkotte 1994] oder sogar zwingendes Merkmal [Booch 1994] objektorientierter Sprachen und Modelle gesehen.

Analog zu [Heuer 1992, S.343f] und auch [Atkinson et al. 1989] sehen wir allerdings das Problem des Attributzugriffs durch eine deklarative Anfragesprache. In einem vollständig gekapselten Datenmodell kann auf Attribute nur noch durch Aufruf von Zugriffsfunktionen zugegriffen werden. Dies erschwert die Nutzung von Indizes, Restriktions- und Anfrageoptimierern.
Der Schutz vor weitreichenden Implementierungsänderungen aufgrund kleiner Schemaänderungen muß zudem in Systemen wie VeDa, bei denen aufgrund von Benutzereingriffen Schemaänderungen häufig auftreten, teilweise durch sich an das Schema anpassende Programme erfolgen. Dies um so mehr, da eine VeDa-Datenbank nicht, wie häufig vorkommend, die Persistenz der Objekte eines Programms sicherstellen soll, sondern Daten für eine Vielzahl selbständiger Werkzeuge mit eigenen Funktionalitäten bereithält. Dadurch kann den Klassen lediglich Grundfunktionalität zugeordnet

werden. Drei häufigere Vertreter — Umbenennungen, berechnete Attribute und Konsistenzbedingungen — lassen sich durch `:alias`–Facetten, `:do-instead`–Trigger, und `laws` auch in VDDL für den Benutzer transparent ausdrücken. Trotz der verbleibenden Probleme sind Attributdefinitonen in VeDa darum nicht verkapselt.

2.3.2 Typ–Klassen–Trennung

Grundlagen 2.2: Funktionalitäten von Klassen

Ähnlich wie prozedurale Programmiersprachen zwischen Deklaration und Implementierung von Prozeduren unterscheiden, kann man auch in Objektmodellen verschiedene Teilfunktionalitäten feststellen, aus denen sich das Gesamtkonzept Klasse zusammensetzt (siehe z.B. [Black et al. 1986; Atkinson et al. 1989]). Der Prozedur*deklaration* entspricht hierbei der **Klassentyp**. Er umfaßt die Typen der Attribute, die Signatur der Methoden sowie Restriktionen, Vor-/Nachbedingungen und Invarianten der Klasse bzw. ihrer Methoden. Die **Klassenimplementierung** (häufig auch selbst als "Klasse" bezeichnet) umfaßt die zur Speicherung der Attribute notwendigen Strukturen sowie die Implementierungen der Methoden. Als dritten Teil der Klassenfunktion enthält die **Klassenextension** alle Instanzen einer Klasse, so daß über diesen Funktionen aufgerufen werden können. Schließlich dient eine Klasse noch als **Instanzfabrik**, d.h. sie stellt eine Klassenmethode, häufig 'new' genannt, zur Verfügung, durch die sich eine neue Instanz von ihr erzeugen läßt.

Will man Klassenfunktionalitäten aufteilen, so handelt es sich hierbei im Bereich der objektorientierten Programmiersprachen (OOPL) meist um Typ und Implementierung, während im Datenbankbereich gerne die Extension abgespalten wird. Wir beschäftigen uns hier zunächst mit ersterem. Die Vorteile und auch die Nachteile der Typ–Klassen–Trennung ergeben sich dadurch, daß die entstehende Typ– und die entstehende Klassenhierarchie mehr oder weniger stark voneinander abhängig sein können. Die diese Hierarchien aufspannenden Beziehungen haben viele Namen (vgl. [Taivalsaari 1996, S.445]), wir wählen Obertypbeziehung für die die Typhierarchie aufspannende Beziehung und Vererbungsbeziehung für die die Klassenhierarchie aufspannende Beziehung.

Grundlagen 2.3: Obertyp, Vererbung, Spezialisierung, . . . — Varianten derselben Relation (Teil 1)

Gemäß der in Grundlagen 2.2 aufgeführten Aufteilung einer Klasse in Typ, Implementierung und Extension können auch die Beziehungen zwischen Klassen aufgespalten werden in Obertyp-, Vererbungs- und Teilmengenbeziehung.

Die Anordnung der Klassentypen gemäß der **Obertyp**relation zeigt an, welche Klassenschnittstelle in welcher anderen enthalten ist, also Objekte welcher Klassen für Objekte einer bestimmten Klasse substituiert werden können, ohne daß Typkonflikte auftreten. Diese unter dem Namen **Substituierbarkeit** bekannte Eigenschaft wird in [Liskov und Wing 1993] verschärft, indem auch die in Methodenschnittstellen auftretenden Vor- und Nachbedingungen berücksichtigt werden, um auch die Kompatibilität der Semantik der Klassen vergleichen zu können.

Obwohl es auch in der **Vererbung**sbeziehung um die Semantik von Klassen geht, ist die Ersetzbarkeit von Instanzen hier nicht das Ordnungskriterium. Vielmehr werden die Klassen nach der gegenseitigen Verwendung von Implementierungsteilen angeordnet. Eine Klasse erbt also von einer anderen, wenn sie deren Attributstrukturen und Methoden wiederverwendet. Diese geerbten Komponenten müssen nicht in der Schnittstelle der Klasse vorkommen. Der Unterschied zwischen dem Erben von Strukturen und ihrer Verwendung durch Referenzieren der entsprechenden Klasse ist somit gering: Bei der Vererbung ist keine Dereferenzierung notwendig und es müssen keine Teilobjekte vernichtet werden, um das Hauptobjekt zu löschen.

Bedingen sich Obertyp- und Vererbungsbeziehung gegenseitig („subclassing is subtyping" in [Abadi und Cardelli 1996]), so bestehen quasi nur syntaktische Unterschiede zwischen einer Sprache mit getrennten Typen und Klassen und einer Sprache, die beides mit einem einzigen Konzept erfaßt.

Verzichtet man auf einen Zusammenhang zwischen Obertyp- und Vererbungshierarchie („subclassing is not subtyping"), so steigt die Möglichkeit zur Wiederverwendung von Klassenimplementierungen

unter verschiedenen Typen stark an, da nun Vererbungbeziehungen nach Nützlichkeitsgesichtspunkten statt durch Substituierbarkeit begrenzt erzeugt werden können. [Porter 1992] gibt entsprechende Beispiele aus dem Bereich der Kollektionstypen. In der Verfahrenstechnik könnte beispielsweise eine Reaktivdestillationskolonne von einer anderen Klasse heuristische Methoden zur groben Abschätzung von Stoffumsatz, Wärmeabgabe, oder ähnlichem erben, obwohl die Klassen in ihren Interfaces nicht übereinstimmen. Als nachteilig erweist sich jedoch, daß nicht mehr aufgrund der Struktur eines Objekts (also seiner Implementierung) auf seinen Typ geschlossen werden kann ([Abadi und Cardelli 1996, Kap. 3.4] gibt ein nicht offensichtliches Beispiel). Die automatische Einordnung von Objekten oder Klassen, von denen nur die Struktur nicht aber der Typ bekannt ist, in die Typhierarchie wird dadurch unmöglich.

Die in [Abadi und Cardelli 1996] vorgestellte Alternative „subclassing implies subtyping“ ermöglicht zwar wieder den Typ eines Objekt/einer Klasse zu schlußfolgern, verliert aber viele Wiederverwendungsmöglichkeiten, da sich diese gerade ergeben, wenn Implementierungen geerbt werden können, ohne den Typ einer Klasse zu beeinflussen.

Grundlagen 2.4: Obertypen, Vererbung, Spezialisierung, ... — (Teil 2)

Verschiedene Autoren [LaLonde und Pugh 1991; Taivalsaari 1996] weisen darauf hin, daß neben der Obertyp- und der Vererbungsbeziehung noch die **Spezialisierung**sbeziehung (die auch als 'IsA' bezeichnet wird) unterschieden werden muß. Sie beschreibt, dem intuitiven Verständnis von Spezialisierung entsprechend, die Beziehung von Spezialfällen zu ihren allgemeinen Oberbegriffen. D.h. die Klassen werden nach den Restriktionen, die sie ihren Instanzen auferlegen, angeordnet. Formal kann man dies folgendermaßen schreiben (M_θ entspricht dabei dem Klassenbegriff mit $\theta(x) \in$ PL1 als Restriktion über alle Instanzen):

$$\begin{aligned} M_\theta &= \{x \in \text{World} \mid \theta(x)\} \\ M_{\theta_1} \text{ IsA } M_{\theta_2} &\overset{\text{def}}{\Leftrightarrow} \forall x \in \text{World} : \theta_1(x) \Rightarrow \theta_2(x) \\ &\Leftrightarrow \forall x \in M_{\theta_1} \Rightarrow x \in M_{\theta_2} \end{aligned}$$

Der Unterschied zwischen Spezialisierungs- und der Obertypbeziehung ist für die häufig diskutierten Paradoxien der Art “Ist Quadrat eine Unterklasse von Rechteck?“ verantwortlich. Während aus mathematischen Gesichtspunkten jedes Quadrat auch ein Rechteck ist (d.h. die Klasse Quadrat ist eine Spezialisierung von Rechteck), kann ein Quadrat einige Operationen die ein Rechteck zur Verfügung stellt nicht ausführen, z.B. die unabhängige Skalierung von Höhe und Breite. Quadrate können somit nicht für Rechtecke substituiert werden, Quadrat ist also kein Untertyp von Rechteck, wohl aber eine Spezialisierung, was in Sprachen, die beide Beziehungen mittels nur einer Beziehung ausdrücken, zum obigen Paradoxon führt.
Die letzte Komponente der Ober-/Unterklassenrelation ist die **Teilmengenbeziehung zwischen den Klassenextensionen**, formal etwa:

Definiere $E_{Basis} = \{x \in \text{Objekt} \mid \text{insert}(x, E_{Basis})\}$ die Extension einer Basisklasse
wobei $insert(x, E)$ das Prädikat ist, das alle Objekte x mit den Basisextensionen E, in die sie eingefügt wurden, verbindet
und $E_{E',\theta} = \{x \in E' \mid \theta(x)\}$ eine abgeleitete Extension
E_1 Teilextension von $E_2 \overset{\text{def}}{\Leftrightarrow} E_1 \subset E_2$

In [Heuer 1992] wird diese Beziehung mit der Spezialisierungsbeziehung zusammengefaßt. Dies ist einerseits korrekt, da auch bei der Spezialisierungsbeziehung obige Teilmengenrelation gilt. Bei der Spezialisierung kann dies jedoch immer auf verschärfte Restriktionen für die Instanzen der entsprechenden Klasse zurückgeführt werden, während beispielsweise die Klassenextensionen in GOM [Kemper und Moerkotte 1994] oder die 'some'–Klassen (=Extensionen) in COCOON [Tresch 1995] prozedural definiert sind, d.h. genau das enthalten, was in sie 'eingefügt' wurde, ohne daß eine entsprechende Restriktion existieren müßte.

Für VeDa ist eine Unterscheidung zwischen Typen und Klassen allerdings weniger attraktiv und wird darum auch nicht vorgenommen. Nicht nur, daß auch hier Meyers Kritik [Meyer 1996] greift, daß die Aufteilung der Vererbungsbeziehung die Sprache verkompliziere und es besser sei, die verschiedenen Funktionen der Vererbungsbeziehung (er identifiziert zwölf) nur durch eine Beziehung auszudrücken. Hinzu kommt, daß VeDas Datenmodell strukturlastig ist, d.h. die Klassen enthalten zahl-

reiche Attribute mit zugehörigen Restriktionen, aber nur wenige Methoden. Bei einer Typ-Klassen-Trennung würden aber nur die Methodenimplementierungen den Klassen zugeordnet, Attribut- und Restriktionsdefinitionen jedoch den Typen, so daß die Verwendung der gleichen Klassenimplementierung in mehreren Typen an Attraktivität verliert. Übernommen werden könnte allerdings der in [Abadi und Cardelli 1996] und anderen vorgeschlagene Einsatz der „self type specialization", der die Spezialisierung von Attributen und Methoden gemäß der Spezialisierung der sie enthaltenden Klasse teilweise zu automatisieren hilft.

2.3.3 Spezialisierung vs. Kontravarianz

Interessanter als die Trennung von Obertypen- und Vererbungsrelation ist in VEDA hingegen der Unterschied zwischen der Spezialisierungs– und der Obertyprelation. Stark getypte Programmiersprachen geben der Obertypbeziehung den Vorzug vor der Spezialisierungsbeziehung, um Typtests zur Compile-Zeit zu ermöglichen.

Einige andere Sprachen geben allerdings der Spezialisierung den Vorzug vor der Substituierbarkeit bzw. ermöglichen zumindest entsprechende Oberklassenbeziehungen. Hierunter fallen sowohl zur konzeptionellen Datenmodellierung bestimmte Sprachen wie VODAK [Klas und Schrefl 1995], O–TELOS [Jeusfeld 1992], O_2 [Bancilhon et al. 1992] und eben VEDA als auch einige Programmiersprachen wie etwa SMALLTALK [Goldberg und Robson 1989] und EIFFEL [Meyer 1997]. Smalltalk läßt beliebige Änderungen zwischen Ober– und Unterklassen zu, da die gesamte, schwache Typprüfung zur Laufzeit stattfindet. Eiffel erlaubt es, geerbte Methoden zu verbergen, wodurch die Ober-/Unterklassenrelation verschiedenen Zwecken dienen kann[8]. In VODAK können Methodensignaturen beliebig überschrieben, allerdings nicht versteckt werden, was zur Modellierung des obigen Beispiels ausreicht. (Die Methode Skalierung erhält dann beispielsweise bei Quadrat einen statt zwei Parameter). Die Fähigkeit, die Spezialisierungsbeziehung repräsentieren zu können, wird hier also entweder mit einem sehr schwachen Typkonzept oder der Notwendigkeit von Ausnahmen erkauft.

O–Telos, welches allerdings keine Methoden kennt, O_2 und VDDL stellen von den aufgeführten Sprachen die strengsten Anforderungen an die Typänderungen im Rahmen der Ober-/Unterklassenbeziehung: Statt der Kontravarianz von Methodenparametern, die stark getypte Programmiersprachen fordern, dürfen Parameter nur kovariant spezialisiert werden. Hierdurch ergibt sich automatisch

Grundlagen 2.5: Ko- und Kontravarianz

Aus dem Wunsch Typtests zur Compile-Zeit mit der Verfeinerung von Methodensignaturen zu verbinden entwickelte Cardelli die Ko-/Kontravarianz [Cardelli 1988]. Aus der Substituierbarkeit von Instanzen eines Untertyps für den Obertyp folgt, daß die Typen der Eingabeparameter im Untertyp verallgemeinert werden oder gleich bleiben müssen (Kontravarianz), während die Typen der Rückgabeparameter spezialisiert werden oder gleich bleiben müssen (Kovarianz). Andernfalls könnte es zu Typfehlern zur Laufzeit kommen, wenn eine Methode des Untertyps unter der Signatur des Obertyps aufgerufen wird.
Für jedes Attribut einer Klasse existiert implizit eine Zugriffsmethode (mit dem Attribut als Rückgabeparameter) und eine Änderungsmethode (mit dem Attribut als Eingabeparameter). Aus den obigen Restriktionen folgt darum, daß sich der Typ eines Attributs in der Unterklasse nicht ändern darf (Invarianz), wenn die Unterklasse die Oberklasse substituieren können soll.

statt der in streng getypten Sprachen geltenden Invarianz von Attributklassen die Möglichkeit deren kovarianter Spezialisierung. Das Quadrat–Beispiel läßt sich so zwar nicht modellieren, jedoch eine Vielzahl anderer Spezialisierungen (etwa das Vegetarierbeispiel aus [Abadi und Cardelli 1996]), da kovariante Attributspezialisierung mit der Definition der Klassenspezialisierung weitgehend übereinstimmt.

[8] Auch in C++ können geerbte Methoden versteckt werden, allerdings muß hierzu die gesamte Ober-/Unterklassenbeziehung verborgen werden, so daß in C++ für alle 'sichtbaren' Beziehungen die Obertypbedingung gilt.

Die auf Seite 20 f aufgeführten Spezialisierungsregeln sind somit im Vergleich zu den meisten objektorientierten Sprachen zwar ungewöhnlich, machen aber im Anwendungsgebiet durchaus Sinn, wobei das möglichst frühzeitige Erkennen von Programmierfehlern hinter der möglichst natürlichen Modellierung des Problemfeldes zurücktreten muß (vgl. die Abwägung in [Taivalsaari 1996] zwischen Ausdruckstärke und struktureller Klarheit). An einigen Stellen, an denen die Verwendung dieser Spezialisierungsbeziehung unpassend erschien, insbesondere bei den strukturierten und primitiven Datentypen, wurde allerdings durch nicht–objektorientierte Definitionstechniken von dieser Semantik abgewichen.

2.3.4 Verbindungen zu TWR–Sprachen

Am Anfang des Kapitels haben wir gehört, daß einige Kernideen von VeDas Repräsentationsformalismus auf KL–ONE bzw. allgemein auf Sprachen zur terminologischen Wissensrepräsentation zurückgehen. Eines der Ziele der Überarbeitung VDDLs war es, ein in konventionellen Datenbanken repräsentierbares Datenmodell zu erhalten, so daß sich die Frage nach verbliebenen Ähnlichkeiten zu KL–ONE stellt.

Die entscheidenden Fähigkeiten von TWR–Systemen, nämlich die automatische **Subsumption und Klassifikation** von Konzepten und Objekten, waren bereits in VeDa in der Version aus [Marquardt et al. 1992] aufgrund der eingeführten Restriktionen, Methoden und einiger Facetten nicht mehr möglich.

Grundlagen 2.6: Terminologische Wissensrepräsentation (TWR) und KL–ONE

Die zu einem nicht unbedeutenden Teil auf R. Brachman und die von ihm entworfene Sprache KL–ONE [von Luck 1990] zurückgehenden **Terminologischen Wissensrepräsentationssprachen** sind, vereinfacht gesagt, Varianten der Prädikatenlogik erster Stufe mit dem Ziel, Zeit- und Speicherkomplexität der Berechnung verschiedener Folgerungen zu verringern [Russel und Norvig 1995, S. 331]. KL–ONE repräsentiert die Welt mittels durch terminologische Axiome beschriebenen Konzepten, wobei zwischen **definierten** und **primitiven** Konzepten unterschieden wird. Bei ersteren sind notwendige und hinreichende Zugehörigkeitsbedingungen bekannt, bei letzteren nur notwendige. Somit lassen sich über primitiven Konzepten einfacher Schlußfolgerungen treffen, während definierte Konzepte die interessanteren sind. Eine der wesentlichen Folgerungen, die TWR–Systeme über diesen Konzepten ableiten, stellt die **Subsumption** dar. Sie behandelt die Fragestellung, ob ein Konzept vollständig in einem anderen enthalten ist, spannt also genauso wie die auf S. 26 definierte Spezialisierung eine Teilmengenhierarchie auf.

Die in KL–ONE vorhandene Unterscheidung zwischen **primitiven und definierten Konzepten**, die im alten VeDa begrenzt durch den Klassenslot `closed` modelliert wurde, entfällt aus zwei Gründen. Zum einen lehrt die Erfahrung, daß sich in komplizierteren Modellen ursprünglich als hinreichend angesehene Definitionen bei Schemaänderungen als lediglich notwendig herausstellen können. Zum anderen ist aufgrund der fehlenden Subsumption und Klassifikation die Frage der Vollständigkeit der Beschreibung eines Konzepts weniger interessant (vgl. auch [Bibel et al. 1996, Kap.2.8.1]).

Im Zuge der Spezialisierung von Konzepten kennt KL–ONE Attributrestriktionen (im Sprachgebrauch von KL–ONE Rollenrestriktionen genannt) und Attributdifferenzierung. **Attributrestriktionen** sind nur eine andere Umschreibung für die in VDDL erlaubten Spezialisierungen von Attributtyp und -kardinalität, so daß sich aus der Abstammung aus KL–ONE eine zusätzliche Begründung für die in Abschnitt 2.3.3 erläuterten Spezialisierungsregeln ergibt.
Attributdifferenzierung erlaubt es, in einer Unterklasse ein Attribut in mehrere "Teilattribute" zu zerlegen (beispielsweise 'Kinder' in 'Söhne' und 'Töchter'). Das alte VeDa enthielt die Facette `:as`, die u.a. Attributdifferenzierungen erlaubte.

Beispiel 2.1 *Die folgende Slotdefinition in der Klasse* `Device` *erlaubte drei verschiedene Arten von Interfaces anzugeben, wobei in Unterklassen die einzelnen Elemente der Interfacemengen noch bezeichnet werden konnten.*

```
interface:   :dom (:as signal-io (:as <word> SIGNAL-INTERFACE )) x
               (:as pipe-io (:as <word> PIPE-INTERFACE)) x
               (:as phase-io (:as <word> PHASE-INTERFACE))
```

Attributdifferenzierungen werden hier also bereits in der Oberklasse angegeben. Aufgrund der Unverständlichkeit der so schnell größer werdenden Typausdrücke wurde die `:as`–Facette im Rahmen der Überarbeitung der strukturierten Typen gestrichen. Die Verwendung von Metaklassen ermöglicht einen zur Attributdifferenzierung ähnlichen Mechanismus. In Kapitel 5.4.2 wird gezeigt werden, wie Metaklassen in VEDA per Attributgruppierung emuliert werden können, so daß auch in VDDL Attributdifferenzierung in den Unterklassen möglich ist.

KL-ONEs **Role–Value–Maps**, also die Gleichsetzung von Attributen, konnte im alten VEDA durch die Verwendung eines Attributpfades als Typbezeichner erreicht werden. Wie das folgende Beispiel zeigt, kann der Benutzer den Typ eines Attributs nur noch sehr schwer ersehen.

Beispiel 2.2

```
class:          Structural Coupling
relation:     :dom #>= 2{this.in-description-of.devices.@<word>.
                active-interfaces@signal-io@<word> x
                this.in-description-of.devices.@<word>...}
```

(Beim Attribut `relation` *handelt es sich also um eine Menge mit mindestens zwei 2-Tupeln, deren Komponenten jeweils auf Interfaces verweisen).*

Aus diesem Grund werden Attributidentitäten nun durch die `:alias`–Facette in einem der beiden Attribute dargestellt, während die `:dom`–Facette eine normale Typdefinition enthält. `:alias` erhielt den Vorzug vor einer `:identical`–Facette, da die gewünschte automatische Identitätserhaltung durch `:alias` deutlicher wird, während `:identical` auch eine Restriktion sein könnte. Die in KL–ONE ebenfalls möglichen Teilmengenbeziehungen zwischen Attributen können nur als Restriktion oder als Trigger formuliert werden.

Insgesamt enthält VDDL also noch einige von KL–ONE abgeleitete Konzepte. Zielrichtung und Notation sind jedoch eine andere. In den Arbeiten von U. Sattler [Sattler 1998] werden allerdings Erweiterungen der TWR–Sprache KRIS vorgestellt, die es erlauben, ein eingeschränktes VEDA zu repräsentieren, wodurch auch Subsumption und Klassifikation wieder möglich werden.

2.3.5 Vorteile der Metamodellierung

Wie Kapitel 4.3 zeigen wird, finden sich die vier Ebenen aus Bild 2.1 in verfahrenstechnischen Modellierungssprachen nicht explizit wieder, obwohl die in ihnen enthaltene Information meist vorhanden ist. Kann also auf eine oder mehrere der Ebenen verzichtet werden oder müssen sie explizit modelliert werden?

Selbstverständlich müssen sowohl die Sprachdefinitionsebene als auch die Modellebene existieren, da ansonsten das eigentliche Ziel — die Repräsentation von Modellen — und das Mittel zur Erreichung dieses Ziels — die Sprache — nicht vorhanden wären. Allerdings kann die Sprachdefinition wie in VEDA implizit erfolgen, wenn man auf Introspektion und Adaptibilität zu verzichten bereit ist.
Der **Verzicht auf die Metaklassenebene** führt zum Verlust der Information über die grundlegende Modellierungsmethodik. In Bild 2.1 ließe sich etwa nicht mehr feststellen, welche Semantik der `container`:-Link des Reaktors trägt, d.h. ob es sich zwangsweise um einen Verweis auf ein Subdevice handelt oder ob er nur rein zufällig auf ein anderes Device zeigt. Da diese Informationen jedoch zur Interpretation des Modells durch Applikationen (etwa des Gleichungsaggregierers) unerläßlich ist, existieren die Informationen dieser Metaebene in fast allen verfahrenstechnischen

Modellierungssprachen auf die eine oder andere Weise. Dort wo diese Ebene nicht explizit vorhanden ist, wird versucht, die entsprechende Information durch Sprachkonstruktoren oder Kommentare zu erfassen. Repräsentiert man diese Ebene implizit, indem man ihre Information etwa in die Modellierungssprache verlagert, so werden Änderungen der Modellierungsmethodologie erschwert und die Sprache verliert an Allgemeinheit und wird unübersichtlicher. Die in [Nagel et al. 1995] vorgestellten Sprachen LCR (zur Modellierung chemischer Reaktionen) und Model.La (zur Modellierung verfahrenstechnischer Prozesse) ließen sich beispielsweise auch durch nur eine Sprache mit verschiedenen Metamodellen repräsentieren, wenn Sprache und Methodologiemodell getrennt wären.
Modellierung **ohne die Modellbausteinebene** (d.h. die Modelle sind direkte Instanzen der Methodologieebene) verhindert die Einordnung von Bausteinen in eine Hierarchie sowie die Benennung unterschiedlicher Attribute der gleichen Semantik. Es leidet also die Darstellung der Information in dem verfahrenstechnischen Modellierer vertrauten Konzepten. Implizit kann ein Großteil dieser Information in der Modellebene repräsentiert werden. Entweder durch Verwendung eines klassenlosen Objektmodells (siehe Kapitel 4.1.1) oder durch zusätzliche Referenzen zwischen Objekten und Mengen aus ihnen. Hierbei besteht allerdings die Wahl zwischen einer aufwendigen Nachbildung der Mechanismen der Bausteinebene oder dem Verlust eines Teils der Information (etwa Typisierungsinformation).

Es wird also die Information sämtlicher in Bild 2.1 dargestellten Ebenen benötigt. Wird die zweite oder dritte Ebene implizit repräsentiert, geht im Normalfall Information verloren, so daß die Aufteilung in die entsprechenden Ebenen vorteilhaft erscheint.

2.3.6 Zusammenfassung

Dieses Kapitel hat die Aufteilung des Datenmodells VEDA in vier Ebenen sowie die in Sprachdefinitions- und Modellierungsmethodologieebene auftretenden Konstrukte vorgestellt. Die benutzte objektorientierte Datenmodellierungssprache VDDL weicht in Punkten wie Verkapselung und Klassentaxonomie von den Standardisierungsvorschlägen der Manifeste ab, um dem Anwendungsgebiet gerechter zu werden. Die explizit als Metaklassen repräsentierte Modellierungssystematik erlaubt es dem Benutzer, die seiner Arbeit zugrunde liegenden Konzepte zu inspizieren und ggf. mit vergleichsweise geringem Aufwand seinen Bedürfnissen anzupassen. Im Rahmen dieser Arbeit wurde die Aufteilung VEDAs in vier Ebenen explizit gemacht, sowie verschiedene Anpassungen der ursprünglichen Modellierungssprache in Richtung objektorientierte Datenmodelle vorgenommen und zusammen mit den weiter bestehenden Unterschieden begründet.

Selbstverständlich konnte auf die Modellierungssprache nicht erschöpfend eingegangen werden, da dies den Rahmen der Arbeit sprengen würde. Weitergehende Informationen zu Fragen wie Namenskonventionen, logischen Operatoren, Semantik der strukturierten Datentypen, Kompatibilitätsmatrix der Facetten sowie die integrierte Behandlung der in Kapitel 5 vorgestellten Erweiterungen finden sich in [Baumeister und Marquardt 1998].

Kapitel 3

Ein Beispiel zur Wiederverwendung von Modellen

Kapitel 1 wies auf die Anforderungen bei der Repräsentation und der Wiederverwendung verfahrenstechnischer Prozeßmodelle hin. Anhand eines Beispiels aus der Verfahrenstechnik werden diese Probleme nun eingehender untersucht, indem die zur Wiederverwendung notwendigen Operationen herausgestellt und auf ihre Realisierbarkeit untersucht werden.

3.1 Das Beispiel

Das Grundszenario des Beispiels besteht in der Modifikation eines vorhandenen Modells, um ein Modell für einen noch nicht vorhandenen Prozeßteil zu gewinnen. Zudem wird die Erweiterung eines Modells mittels vorhandener Modellbausteine, sowie die Einordnung erzeugter Modelle in die Bausteinhierarchie demonstriert. Das Beispiel geht auf die in den Arbeiten von Deutsch und Hahn vorgestellten Modelle für Schlaufenreaktoren und Destillationskolonnen [Deutsch 1994; Hahn 1997], auf die entsprechenden Ausführungen in [Sattler 1988] sowie auf zahlreiche Anregungen von Bernd Lohmann, Volker Gehrke und Dirk Müller zurück.

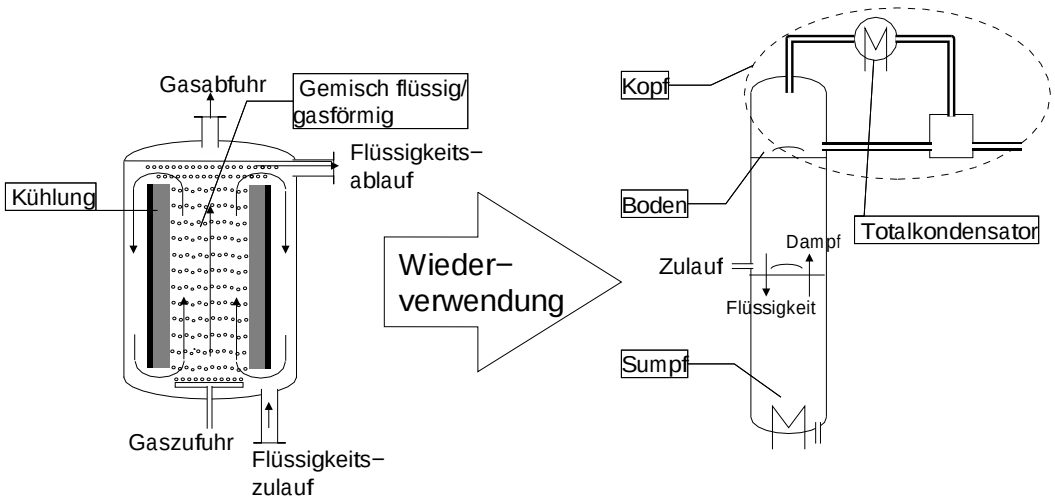


Abbildung 3.1: Schlaufenreaktor und Glockenbodendestillationskolonne

Als Ausgangspunkt des Szenarios befinde sich in dem zu einem Modellierungswerkzeug gehörigen DBMS ein fertiges Modell eines Schlaufenreaktors (vgl. [Hahn 1997]) sowie andere Modellbausteine

aus früheren Modellierungsaufgaben. Insbesondere sind zwar Bausteine für einige Destillationskolonnentypen zu finden, aber keiner für eine Reaktivdestillationskolonne[1]*.*
Der uns im folgenden begleitende Modellierer, Herr M., hat die Erstellung eines Gleichgewichtsmodells einer fünfbödigen Reaktivdestillationskolonne als Aufgabe erhalten. Ein kurzer Überblick über die vorhandenen Modellbausteine überzeugt auch ihn von der bereits vom Erzähler verratenen Tatsache, daß ein passender Modellbaustein nicht existiert. Da Herr M. vermeiden möchte, die Stöchiometrie und andere mit der Reaktion zusammenhängende Daten in das Modell eintragen zu müssen, beginnt er, nach Modellen zu suchen, in denen eine möglichst ähnliche Reaktion vorkommt und deren Modell dem einer Reaktivdestillationskolonne ähnlich sieht. Diese ***Suche*** *führt Herrn M. schließlich zu dem Modell des Schlaufenreaktors, das sich durch die Modellierung der Ortsabhängigkeiten mittels Diskretisierung in fünf übereinanderliegende Bilanzräume gut als Ausgangspunkt für die Kolonnenmodellierung eignet.*

Bild 3.1 zeigt Prinzipienskizzen des Schlaufenreaktors und der Kolonne. Im Verlauf der Modellierung soll aus dem Modell des links skizzierten Blasensäulenreaktors eines für die rechts dargestellte Glockenbodendestillationskolonne erzeugt werden. Betrachtet man die entsprechenden Komponenten–Verknüpfungs–Diagramme gemäß VeDa–Systematik in Abbildung 3.2, so fällt auf, daß Kühlung und Rückfluß des Reaktors in der Kolonne keine Entsprechung haben, während sich die gleichmäßige Struktur der Diskretisierungsebenen des Reaktors in der Kolonne in den Böden wiederfindet, wobei allerdings leichte Variationen durch Kopf, Sumpf und den Zulaufboden entstehen.

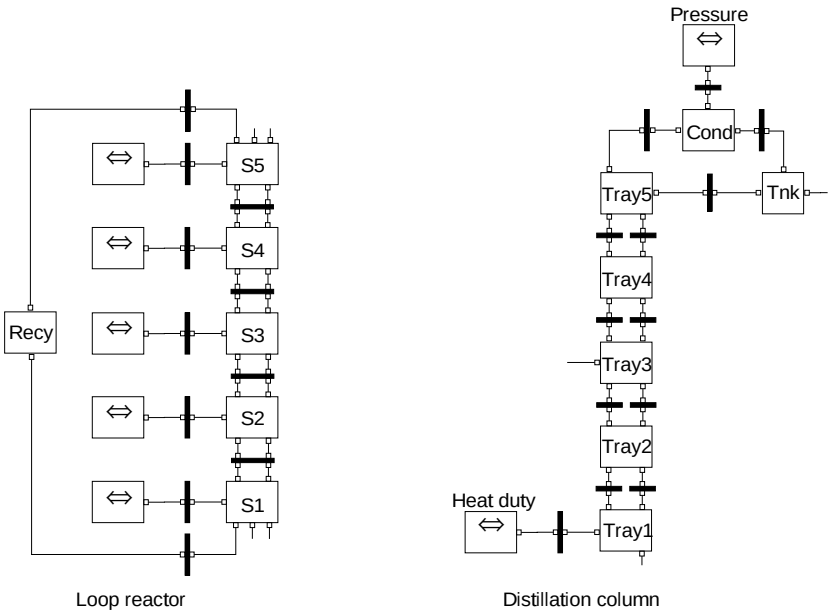


Abbildung 3.2: Komponenten–Verknüpfungs–Diagramme der beiden Modelle des Beispiels

Herr M. macht sich wegen der überflüssigen Teile wenig Sorgen, da ***Löschoperationen*** *schnell ausgeführt sind. Nachdem er also das Reaktormodell kopiert hat, um das bestehende Modell nicht zu zerstören, und die überflüssigen Teile entfernt hat, beginnt der kompliziertere Teil der Arbeit: Einfügen von Kopf und Sumpf sowie die Anpassung des Modells für die Böden.*

Abbildung 3.3 zeigt, wie die Operation "Löschen" in einem entsprechenden Werkzeug aussehen könnte.

Der Kolonnenkopf sollte nach Herrn M.s Vorstellungen ein Totalkondensator sein. Ein entsprechender Modellbaustein findet sich auch in der Datenbank. Allerdings verlangt die Modellierungsumgebung aufgrund der in der Datenbank vorhandenen Klassen und ihrer Beziehungen eine ***Auswahl*** *der*

[1] Die in den Modellen auftretenden Stoffe und die mathematischen Gleichungen werden im weiteren nicht betrachtet, da ansonsten der Umfang Beispiels gesprengt würde

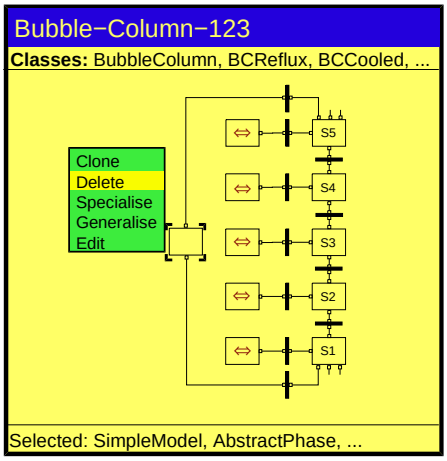


Abbildung 3.3: Löschen der überflüssigen Teile

Rückfluß- und Druckregelung des Kopfes und bietet den Einbau eines Kondensatsammelbehälters und die Modellierung einer Flüssig/Flüssig–Phasentrennung an (vgl. Abbildung 3.4). Durch Kombination der zu diesen Eigenschaften angebotenen Alternativen erhält Herr M. schließlich einen zur Umgebung offenen Totalkondensator mit Kondensatsammelbehälter und durch Pumpen kontrolliertem Rück- und Abfluß.

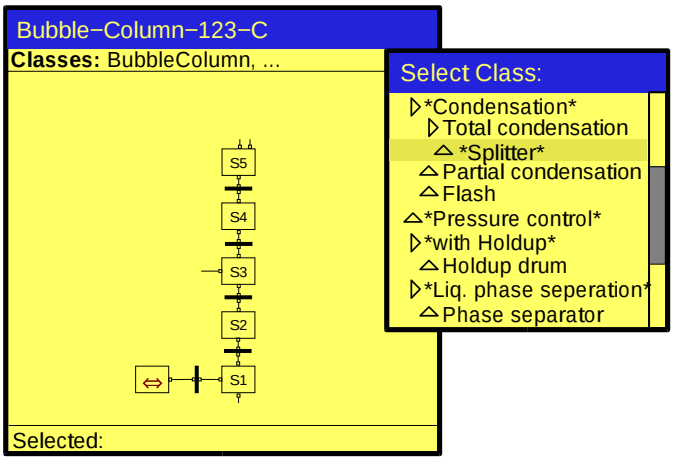


Abbildung 3.4: Auswahl verschiedener Aspekte des Kolonnenkopfes

Bei der Auswahl eines Bausteins müssen diverse mehr oder weniger unabhängige Eigenschaften berücksichtigt werden, im Beispiel etwa die Art der Kondensation im Kopf, die Regelung der Rückflußmenge oder das Auftreten eines Phasenzerfalls. Eine Auswahl der entsprechenden Eigenschaften des Kolonnenkopfes zeigt Bild 3.4, sich ergebende verschiedene Realisierungsmöglichkeiten zeigt Bild 3.7. Zur Organisation dieser Varianten werden ggf. mehrere Hierarchien benötigt. Genaueres findet sich im Unterabschnitt „Select from Hierarchy“ auf Seite 37.

Während obiger Erklärungen hat Herr M. auch den Sumpf modelliert, dessen Realisierung hier nicht weiter interessieren soll. Das Modellierungssystem ist bei seinen ständigen Vergleichen des Modells von Herrn M. mit den in der Datenbank vorhandenen Modellbausteinen inzwischen zu dem Schluß gekommen, daß Herr M. eine Destillationskolonne modelliert. Da Benennung und Typ der die Böden repräsentierenden Slots nicht ganz dem Destillationskolonnenbaustein entsprechen, ***klassifiziert*** *es Herrn M.s Modell vorläufig als noch unbekannte Kolonnenvariante und bietet Herrn M. an, ausgehend vom Destillationskolonnenbaustein ein neues Modell zu erzeugen oder zu versuchen, das erzeugte Modell an den Baustein anzupassen.*

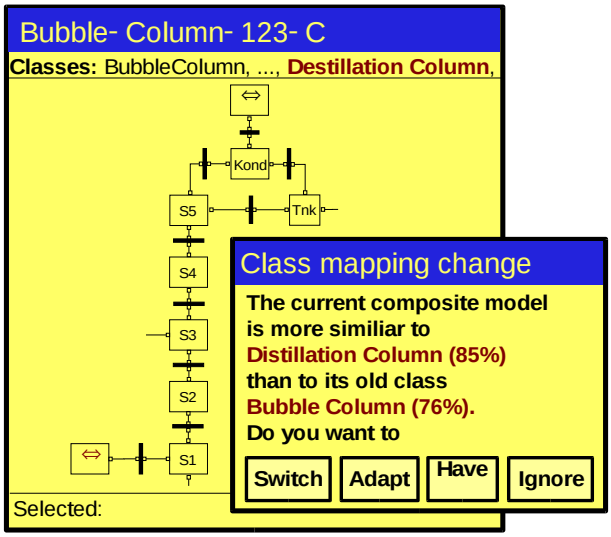


Abbildung 3.5: Einordnung in Klassen

Um dem Modellierer die Feststellung zu erleichtern, ob er sich auf dem richtigen Weg befindet, wird ständig angezeigt, in der 'Nähe' welcher Konzepte sich der bearbeitete Modellteil derzeit befindet (s. auch die Kopfzeile in Bild 3.5). Dies ermöglicht nicht nur ein ständiges Feedback an den Modellierer, sondern kann auch die Mehrfachentwicklung schon vorhandener aber nicht gefundener Modelle vor ihrer Fertigstellung aufdecken. Abschnitt 3.3.4 enthält weitere Erläuterungen zu dieser Funktionalität.

Herr M. freut sich über die Bestätigung, daß er auf dem richtigen Weg ist, beschließt aber die Hilfsangebote zu ignorieren, da er ja bereits weiß, daß die Datenbank sein gewünschtes Modell nicht enthält.
Das Modell für die Kolonnenböden möchte Herr M. aus dem im Schlaufenreaktormodell vorhandenen Modell der Kaskadenstufen ableiten, da es bereits eine ähnliche Reaktion beinhaltet. Das Kaskadenstufenmodell besitzt natürlich außer der Reaktion noch weitere Festlegungen, so z.B. die räumliche Auflösung, Aggregatzustände und andere physikalische Phänomene. Um versehentliche Änderungen zu vermeiden, friert Herr M. vorerst alle mit der Reaktion zusammenhängenden Attribute ein, während er die restlichen Slots ***nach Belieben ändern*** *können möchte. Unter anderem*

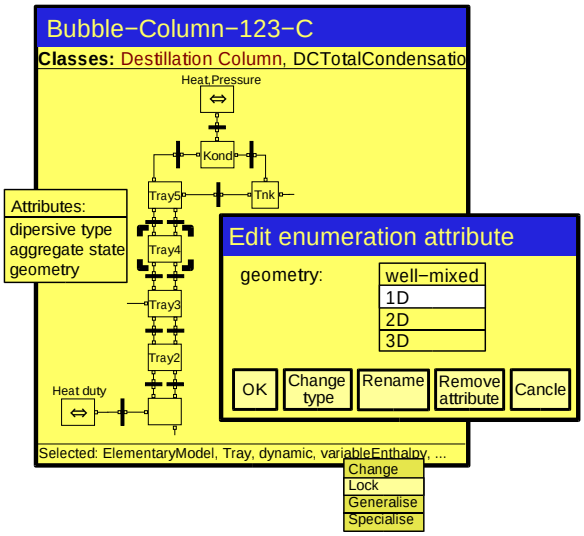


Abbildung 3.6: Änderungen an einem Kolonnenbodens

fügt er die für die Bestimmung des auf dem Boden befindlichen Volumens benötigten Prozeßgrößen hinzu und ändert die geometrische Repräsentation auf eine eindimensionale (s. Bild 3.6).

Änderungen können also Attribute entfernen (Löschen der überflüssigen Bausteine) oder hinzufügen (neue Prozeßgrößen) und ihren Wert oder ihren Typ ändern (eindimensionale Geometrie). Alle Änderungen müssen direkt durchführbar sein, ohne daß sich der Modellierer Gedanken über notwendige Anpassungen von Verwaltungsstrukturen (also etwa Klassen) machen muß.

Verlassen wir nun Herrn M. und wenden uns den in diesem Beispiel auftretenden Daten und Operationen zu.

3.2 Die VeDa–Klassen

Aufgrund des begrenzten Platzes sollen hier nur zwei mögliche Bausteinklassen (also zwei Klassen aus VeDas Bausteinebene (s. Bild 2.1)) für den Reaktor und die Destillationskolonne angegeben werden[2]. Sie enthalten den Aufbau der jeweiligen Einheit aus Untereinheiten und Verbindungen, wobei nicht alle Attribute angegeben wurden. Der Schlaufenreaktor besteht somit aus einer Anzahl von Reaktions- und Kühlzellen mit einem Stoffffluß zwischen den Reaktionszellen und einem Wärmeübergang zwischen Reaktions- und zugehöriger Kühlzelle. Ein Objekt dieser Klasse wird im Beispiel transformiert in ein Objekt der Destillationskolonnenklasse. Hier sind mehrere Böden über zwei Stoffströme verbunden sowie oben durch einen Kondensator und unten durch eine Beheizung begrenzt.

```
class:                          COMPOSITE-LOOPREACTOR-1DD-IMPLEMENTATION
documentation:                  "one-dimension distributed model of the loop reactor
                                   discretized using well-mixed subcells"
metaclasses:                    COMPOSITE-CONCEPT-FRAME
superclasses:                   LOOPREACTOR-IMPLEMENTATION
individual-intrinsic-attributes:
   cellNumber:                  :dom INTEGER
individual-relational-attributes:
   reactionCell:                :dom #>=2{ LOOPREACTORSEGMENT }
                                   :to_group subdevices :req t
   coolingCell:                 :dom #>=2{ HEAT-SINK }
                                   :to_group subdevices
   recycle:                     :dom STIRRED-TANK
                                   :to_group subdevices :req t
   massConnection:              :dom #>=1{ DIRECT-CONNECTION }
                                   :to_group subconnections :req t
   heatConnection:              :dom #>=2{ HEAT-TRANSFER-THROUGH-WALL }
                                   :to_group subconnections

laws:
   cellCount:                   :holds :count(reactionCell) :eq cellNumber :and
                                   :count(coolingCell) :eq cellNumber :and
                                   :count(heatConnection) :eq cellNumber :and
                                   :count(massConnection) :eq cellNumber - 1
END

class:                          COMPOSITE-REACTIVE-DISTILLATIONCOLUMN-IMPLEMENTATION
metaclasses:                    COMPOSITE-CONCEPT-FRAME
superclasses:                   REACTIVE-DISTILLATIONCOLUMN-IMPLEMENTATION
individual-intrinsic-attributes:
   trayNumber:                  :dom INTEGER
```

[2] Die in den Klassen auftauchende `:to_group`–Facette wird in Kapitel 5.4 genauer erläutert.

```
individual-relational-attributes:
   trays:                  :dom #>=2{ TRAY-WITH-REACTION }
                              :to_group subdevices :req t
   vaporStream:            :dom #>=1{ BUBBLECAP-FLOW }
                              :to_group subconnections :req t
   liquidStream:           :dom #>=1{ WEIR-FLOW }
                              :to_group subconnections :req t
   condensor:              :dom TOTAL-CONDENSOR :req t
                              :to_group subdevices
   heating:                :dom HEAT-SOURCE
                              :to_group subdevices

laws:
   trayCount:              :holds :count(trays) :eq trayNumber :and
                              :count(vaporStream) :eq trayNumber - 1 :and
                              :count(liquidStream) :eq trayNumber - 1
END
```

Wie man sieht besitzen beide Klassen die gleiche, regelmäßige Struktur, allerdings wurden einige Attribute gelöscht oder eingefügt; alle Attribute wurden umbenannt. Das Ziel, die Informationen über die Reaktion wiederzuverwenden, läßt sich bei einer solchen Klassenstruktur nur erreichen, falls `LOOPREACTORSEGMENT` und `TRAY-WITH-REACTION` eine gemeinsame Oberklasse besitzen, die genau diese Informationen enthält, da andernfalls die entsprechenden Objekte nicht von der einen Klasse zur anderen transferiert werden können. In einer Vererbungshierarchie mit einfacher Vererbung erscheint dies eher unwahrscheinlich, da die Haupteinordnungskriterien "Boden" bzw. "Reaktorsegment" sind. Bei multipler Vererbung, die von VeDa unterstützt wird, wird dies zwar möglich, es ergeben sich aber die auf Seite 38 und ausführlicher ab Seite 53 dargelegten Probleme.

3.3 Die Operationen

In obigem Szenario treten mehr oder weniger deutlich verschiedene, im Verlauf des Modellierungsprozesses notwendige Operationen auf. Sie werden im folgenden identifiziert, beschrieben und bezüglich ihrer Realisierbarkeit untersucht. Tabelle 3.1 auf Seite 39 faßt die Ergebnisse zusammen.

3.3.1 Browse Database

Ein kurzer Überblick über die vorhandenen Modellbausteine überzeugt Herrn M., daß ein für die Reaktivdestillation passender Modellbaustein nicht existiert.

Um einen Überblick über vorhandene Modelle oder Bausteine zu gewinnen, kann ein Benutzer es vorziehen, den Inhalt der Datenbank nach eigenen Kriterien zu durchstöbern[3], statt in einer Anfrage die gesuchten Elemente zu spezifizieren.

Entsprechende Operationen sind in allen objektorientierten Datenmodellen einfach möglich, da Objekte die Daten zu intuitiven Einheiten zusammenfassen und durch Verweise verknüpfen. Mit zunehmender Größe der Datenbank sind jedoch Strukturierungsmechanismen erforderlich, um die Zahl der vom Benutzer zu betrachtenden Daten zu begrenzen. Dies kann durch eine Hierarchie auf den Daten erfolgen, durch die Einschränkung der angezeigten Objekte mittels einer Anfrage (also der Generierung einer eingeschränkten Sicht) oder durch die Selektion eines Startpunkts per Anfrage oder Verweissammlung, von dem aus weiter gestöbert werden kann. Betrachten wir den Fall der Hierarchie genauer.

[3] deutsche Übersetzung von "Browsen"

Select from Hierarchy

Der Kolonnenkopf sollte nach Herrn M.s Vorstellungen ein Totalkondensator sein. Ein entsprechender Modellbaustein findet sich auch in der Datenbank. Allerdings verlangt die Modellierungsumgebung aufgrund der in der Datenbank vorhandenen Klassen und ihrer Beziehungen noch Entscheidungen über die Druck- und Rückflußregelung des Kopfes und bietet den Einbau eines Kondensatsammelbehälters und die Modellierung einer Flüssig/Flüssig–Phasentrennung an.

Diesmal wurde also durch das Stöbern ein Baustein gefunden. Entscheidend an diesem Fall ist jedoch, daß ein gewünschter Baustein entlang der Spezialisierungshierarchie anhand mehrerer Kriterien (hier zusätzlich zur Art des Kondensators etwa noch Regelungen und Vorhandensein eines Kondensatsammelbehälters) ausgewählt wurde.

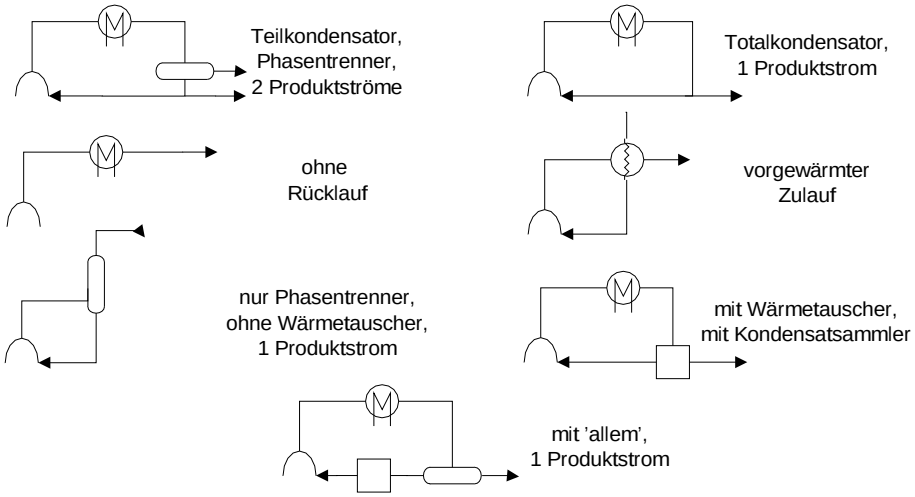


Abbildung 3.7: Verschiedene Realisierungsmöglichkeiten des Kolonnenkopfes

Eine Hierarchie über den Modellbausteinen erlaubt es, sich sukzessive dem gewünschten Baustein anzunähern, indem auf jeder Hierarchieebene Alternativen ausgeschlossen werden. Bei der Anordnung von Klassen in einer einfachen Spezialisierungshierarchie ohne Mehrfach–Spezialisierung (also in einem Baum) kann allerdings in jeder Ebene nur nach einem Kriterium spezialisiert werden (links in Bild 3.8). Selbst die bekannteste aller Taxonomien, die biologische Klassifizierung der Lebewesen, kann erst vollständig und korrekt so angegeben werden, seitdem als einziges Kriterium die mit Hilfe der molekularen Biologie bestimmbare Abstammung der Arten gewählt wird [Dawkins 1987, S. 269ff][4]. In vielen Fällen wird jedoch nach mehreren Kriterien klassifiziert (etwa in Konstrukti-

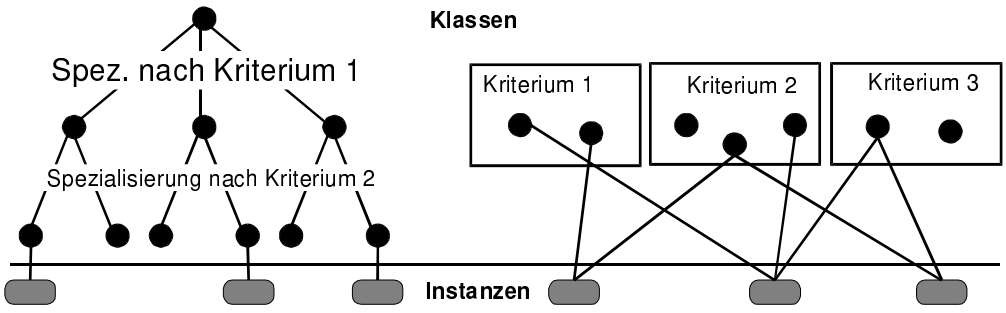


Abbildung 3.8: Kriterienzuordnung durch Hierarchie und durch direkte Selektion

[4]Die Klassifizierung nach anderen extern meßbaren Kriterien wie Größe, Aussehen, Verhalten oder Metabolismus führt nicht nur zu Fehlklassifizierungen aufgrund paralleler, unabhängiger Entwicklungen [Dawkins 1987, S. 269], sondern ergibt aufgrund des Zieles, einen Klassifikations*baum* zu erhalten, die Probleme der nichtfliegenden Vögel (Strauß, Pinguin) oder der säugenden 'Fische' (Wal, Delphin).

onskatalogen für Plastikverbinder [Neumann 1995] nach Belastbarkeit, Lösbarkeit und Durchlässigkeit), oder Objekte werden mehreren Klassen zugeordnet (etwa bei Literaturklassifikationsschemata [Prieto-Diaz 1991]). Werden solche Fälle wie links in Bild 3.8 modelliert, so müssen die Unterscheidungen nach den verschiedenen Kriterien nacheinander in der Hierarchie auftauchen, d.h. spätere Unterscheidungen tauchen mehrfach an verschiedenen Stellen der Hierarchie auf. Eine vorteilhaftere Alternative ergibt sich, wenn man wie rechts in Bild 3.8 gleichzeitig Alternativen nach mehreren Kriterien selektieren kann.

Die Spezialisierung nach mehreren unabhängigen Kriterien kann in objektorientierten, klassenbasierten Systemen durch multiple Spezialisierung[5] ausgedrückt werden. Im Beispiel hätte Herr M. dann eine Klasse namens `Open-PumpRegulated-TotalCondenser-Head` ausgewählt. Allerdings führt dies, wie schon an der Länge des Klassennamens erkennbar, zu einer exponentiellen Klassenexplosion: Neben den Möglichkeiten, die Bild 3.7 zeigt, muß etwa auch noch entschieden werden, wie die Druck und Splitfaktor geregelt werden. Eine Methodik, die ähnlich wie Abbildung 3.8 rechts ohne solche Klassen auskommt, wäre daher zu bevorzugen.

3.3.2 Search Database

Die Suche in der Datenbank nach Modellen, in denen eine möglichst ähnliche Reaktion vorkommt, führt Herrn M. schließlich zu dem Modell des Schlaufenreaktors [. . .]

Sind dem Benutzer hinreichend genaue Kriterien bekannt, die die gesuchten Modelle/Bausteine beschreiben, so lassen sie sich durch eine Anfrage an das DBMS schneller finden, als durch das Stöbern in der Datenbank.

Für die Formulierung der Anfrage bestehen verschiedene Möglichkeiten. Für den Benutzer am einfachsten ist die Vorgabe eines Beispiels, zu dem ähnliche Modelle gesucht werden sollen. Die Benutzung einer deklarativen Anfragesprache, wie sie in den meisten kommerziellen ODBMS als SQL–Abart vorhanden ist, erlaubt die Spezifikation komplexer Beziehungen zwischen den Daten um den Preis des notwendigen Lernaufwandes des Benutzers. Eine Mischform stellen parametrisierbare Anfrageskelette dar, die dem Benutzer erlauben, vorgegebene, komplexe Anfragen ohne Kenntnisse einer Anfragesprache zu stellen. Beispiele für alle diese Verfahren finden sich etwa in [Date 1995]. Im folgenden soll der Fokus auf die beispielgetriebenen Anfragen gelegt werden.

Search by Example

[. . .] beginnt er, nach Modellen zu suchen, in denen eine möglichst ähnliche Reaktion vorkommt und deren Modell dem einer Reaktivdestillationskolonne ähnlich sieht.

Deklarative Anfragen bereiten im objektorientierten Paradigma Probleme, die in relationalen Modellen nicht bestehen; in [Heuer 1992] werden z.B. folgende erwähnt: die fehlende Objektidentität von Anfrageergebnissen, die Datenkapselung und die mangelnde Optimierbarkeit von Methodenaufrufen. Für einfache beispielgetriebene Anfragen eignet sich ein objektorientiertes Modell allerdings besser als ein relationales. So ist das in [Date 1995] vorgestellte „Query by Example“ für IBMs relationale Datenbank zwar fast genauso mächtig wie die ebenfalls vorhandene Anfragesprache SQL, erfordert dafür aber genau wie diese das Verständnis der zugrundeliegenden Relationen sowie der Verwendungsweise von Joins. Beschränkt man sich auf Gleichheit oder Ähnlichkeit, schließt also Bereichsanfragen, Platzhalter und Negation aus, so können beispielgetriebene Anfragen in objektorientierten Modellen aufgrund der 'Clusterung' von Attributen zu Objekten einfach durch Vorgabe einer entsprechend mit Werten gefüllten Instanz angegeben werden. Dies macht sich beispielsweise der Ansatz des „Case-based Reasoning“ [Kolodner 1993] zunutze, wenn zu einem vorhandenen Problem/Ansatz ähnliche gesucht werden sollen. Auch obige Suche nach Bausteinen, in denen eine

[5] Multiple Spezialisierung bedeutet, daß eine Klasse mehrere Oberklassen haben kann.

Operationen	Parameter	Ergebnis	Semantik	benötigt
Suchoperationen:				
Browse	Startpunkt	(Klasse / Instanz)	keine; Benutzer verfolgt Links nach eigenem Gutdünken	vorteilhaft sind: • Hierarchien • Suchraumeinschränkungen
Select from Hierarchy	Startpunkt	Klasse / Instanz	nur Verfolgung hierarchischer Links	
Query	Anfrage	(Menge von) Instanzen	abhängig von Anfrageart	• Anfragesprache oder Skelette • Durchbrechen der Verkapselung
Query by example	Instanz	(Menge von) Instanzen	liefert alle dem Beispiel ähnlichen oder vom Beispiel subsumierten Instanzen	• Vergleichsfunktion • intuitive Beispielnotation
Compare	2+ Instanzen	Zahlenwert(e)		• Ähnlichkeitsmaß • domänenabhängig oder domänenunabhängig
Änderungsoperationen:				
Clone	Instanz	Instanz	kopiere Objekt sowie ggf. referenzierte Objekte	• rekursives Kopieren
Modify				
Modify values	Instanz	Instanz	ändert vorhandene Attributewerte	• Objektversionierung
Modify type	Instanz & Klasse	Instanz & Klasse	modifiziert Typ des Objektes, adaptiert Instanz an neuen Typ	• ggf. Schemaevolution
Classify	Instanz & Klasse	Platz in Klassenhierarchie	ordnet ein modifiziertes Objekt in Klassenhierarchie ein	• Klassifizierungsalgorithmus

Tabelle 3.1: Operationen im Szenario

ähnliche Reaktion vorkommt, kann beispielgetrieben formuliert werden, indem etwa die folgenden VeDa–Frames als Suchvorgabe gegeben werden.

```
instance:                Searchexample
classes:                 ELEMENTARY-DEVICE-IMPLEMENTATION
   reaction:             Xy-z-reaction :to_group phenomena
END

instance:                Xy-z-reaction
classes:                 REACTION
   stochiometrie:        Xy-z-stoch
...
END
```

Mit dem entsprechenden Werkzeug wird der Benutzer natürlich nicht obige textuelle Repräsentation spezifizieren, sondern sich einige Objekte per Point&Click generieren, die dann als Beispiel für die Anfrage dienen.

3.3.3 Copy & Modify

Nachdem er also das Reaktormodell kopiert hat, um das bestehendes Modell nicht zerstören, und die überflüssigen Teile entfernt hat, beginnt der kompliziertere Teil der Arbeit: Einfügen von Kopf und Sumpf sowie die Anpassung des Modells für die Böden

Wie bereits festgestellt, erfordert die Wiederverwendung von Modellbausteinen in der verfahrenstechnischen Prozeßmodellierung fast immer deren Adaption an die Problemstellung. So müssen im Beispiel die Reaktorsegmente und die Reaktorstruktur in die Kolonnenböden und die Kolonnenstruktur umgewandelt werden. Solche Änderungen, aber auch einfachere wie das Hinzufügen einzelner Untereinheiten oder die Änderung von Attributwerten über die Typgrenzen hinaus, modifizieren den Typ des entsprechenden Objekts. Es wird allgemein angenommen, daß es für Menschen intuitiver ist, entsprechende Änderungen direkt auf der Instanzebene durchzuführen, statt nach Erzeugung bzw. Modifikation der in den Klassen vorhandenen Abstraktion diese zu instanziieren [Lieberman 1986].

Die Modifikation auf der Instanzebene nennt sich auch “Copy & Modify“, entsprechend kann man das klassenbasierte Vorgehen als “Abstract & Instantiate“ bezeichnen. Wie der Name bereits sagt, läßt sich “Copy & Modify“ in eine Kopier- und eine Änderungsoperation unterteilen.

Copy

Die Kopieroperation an sich stellt keine Besonderheit dar. Da die Objekte nach dem Kopieren modifiziert werden sollen, müssen sie vollständig, d.h. einschließlich aller referenzierten Objekte kopiert werden (sogenanntes “Deep–Copy“ oder auch “Klonen“). Um den dadurch entstehenden Platzbedarf zu begrenzen, sind Schemata vorstellbar, die die Kopieroperation tatsächlich erst bei einem Änderungszugriff durchführen oder aber nur die Änderungen speichern. Legt man vor jeder (größeren) Änderung eines Objekts eine Kopie an, so erhält man inhärent eine Versionierung des Objekts.

Modify

Allgemein kann man sagen (nach [Krueger 1992, S.143], Erweiterung durch Autor kursiv): Um ein Modellelement effektiv wiederzuverwenden, muß man es schneller finden *und anpassen* können als es dauert, das benötigte Element direkt zu erzeugen.

Um diese einfache und schnelle Modifizierbarkeit zu erreichen, sollten beliebige Änderungen direkt am Objekt zulässig sein. Dies umfaßt zum einen das Abändern beliebiger Attributwerte gemäß ihrem vorgegebenem Typ (Wertmodifikation). Außerdem zählen hierzu natürlich auch das Hinzufügen, Entfernen und Umbenennen von Attributen, die direkte Änderung eines Attributtyps sowie die indirekte Typänderung, indem ein Wert für das Attribut spezifiziert wird, der nicht mit dessen Typ übereinstimmt (Typmodifikation, da sie den Typ des Objekts ändern). Eine letzte Modifikationsmöglichkeit stellt das Bewegen des Objektes in der Klassenhierarchie dar, etwa wenn anfangs nicht alle möglichen Festlegungen getroffen wurden (vgl. Abschnitt 18) und dies erst in späteren Iterationszyklen geschieht. Auf Objekt oder Klassenhierarchie hat diese "Objektmigration" genannte Änderung keine weiteren Auswirkungen.

Es sei darauf hingewiesen, daß Typmodifikationen zu Schemaevolution zur Laufzeit führen können, d.h. zu einer Änderungen an einer oder mehreren Klassen. Darum müssen, falls eine Versionierung gewünscht wird, auch Schemaversionierungsmechanismen existieren.

3.3.4 Modify & Classify

Das Modellierungssystem ist bei seinen ständigen Vergleichen des Modells von Herrn M. mit den in der Datenbank vorhandenen Modellbausteinen inzwischen zu dem Schluß gekommen, daß Herr M. eine Destillationskolonne modelliert. Da Benennung und Typ der die Böden repräsentierenden Slots nicht ganz dem Destillationskolonnenbaustein entsprechen, klassifiziert es Herrn M.s Modell vorläufig als noch unbekannte Kolonnenvariante [...]

Bei der Beschreibung der Suchoperationen wurde erwähnt, daß eine Ordnung der Datenbasis durch Hierarchien zumindest wünschenswert erscheint. Werden Objekte im Rahmen der Wiederverwendung angepaßt, also modifiziert, sollten sie darum in bestehende Hierarchien eingeordnet werden. In einem klassenbasierten System wie VeDa ist dies insbesondere die Spezialisierungshierarchie zwischen den Klassen. Nach einer Typänderung eines Objekts muß also eine passende Klasse für dieses Objekt gefunden bzw. erzeugt werden; das Objekt muß somit in die Klassenhierarchie eingeordnet — kurz: klassifiziert — werden.

Classify

Prinzipiell existieren zwei Möglichkeiten ein Objekt zu klassifizieren. Zum einen kann der Benutzer aufgrund seines Metawissens eine passende Klasse für ein Objekt selektieren. Zum anderen kann das System versuchen, dieses Metawissen aufgrund der Struktur des Objekts (also dessen Attributen) und ggf. vorhandener Klassen zu erschließen und das Objekt ohne Befragung des Benutzers automatisch zu klassifizieren. Ggf. können hierbei auch in den Klassen vorhandene Restriktionen zum Ausschluß von Objekten genutzt werden.

Um dem Benutzer sofortigen Feedback geben zu können, muß im Unterschied zur Ausgangslage bei den meisten in der Literatur angegebenen Verfahren zur Klassifikation (vgl. Kapitel 4.2.2) in diesem Anwendungsfall ständig und inkrementell klassifiziert werden, d.h. nach (quasi) jeder Änderung muß überprüft werden, ob sich das Objekt in einer neuen Klasse befindet. Auch muß davon ausgegangen werden, daß sich sehr häufig keine exakt passende Klasse finden läßt. Zum einen, da für die Zwischenzustände eines gerade modifizierten Objekts keine zugehörige Klasse existiert, zum anderen, da sich das Anwendungsgebiet, wie erwähnt, durch viele, leicht voneinander abweichende Objekte auszeichnet.

3.4 Zusammenfassung

Dieses Kapitel zeigte anhand eines idealisierten Beispiels, welche Operationen notwendig sind, um ein Modell auf möglichst einfachem Weg wiederverwenden zu können. Man beachte, daß keine der durch

den Modellierer durchgeführten Handlungen die Kenntnis einer Anfragesprache oder ähnlicher Formalismen benötigt. Selbstverständlich spiegelt sich in diesen Operationen der Wiederverwendungszyklus von Seite 10 wieder: Suchen, Anpassen, Einordnen, wobei in Tabelle 3.1 die Suchoperationen aufgrund ihrer Vielfalt den meisten Platz einnehmen. Diese Arbeit wird sich im folgenden allerdings auf Probleme und Lösungen konzentrieren, die im Bereich der Änderungsoperationen entstehen, wenn die für die Suche vorteilhafte Ordnungshierarchie erhalten bleiben soll. Die Suchoperationen selbst bleiben weitgehend ausgeklammert, da sie, abgesehen von der Hierarchie, wenig direkte Auswirkungen auf die Repräsentation der Daten haben[6].

Abschnitt 3.2 deutete bereits eine der Schwierigkeit an, die sich aus dem Wunsch nach gleichzeitiger beliebiger Modifizierbarkeit und Erhalt einer Ordnungshierarchie ergeben. Das nächste Kapitel wird diese Schwierigkeiten genauer ausarbeiten und einen Überblick über vorhandene Lösungsansätze geben.

[6] Bei Auswirkungen wie Indizes und Clusterung von Daten wird davon ausgegangen, daß sie ggf. durch die verwendete Datenbank zur Verfügung gestellt werden.

Kapitel 4

Wiederverwendungsmechanismen in objektorientierten Datenmodellen

Wie die vorhergehenden Kapitel gezeigt haben, zeigen gängige klassenbasierte Objektmodelle wie VeDa Schwächen bei Wiederverwendung und Erweiterung vorhandener Konzepte und bieten die zur Durchführung des Szenarios notwendigen Operationen nur teilweise an. Die mangelnde Flexibilität und Manipulierbarkeit wurde in [Lieberman 1986; Borning 1986] kritisiert und führte in der Folgezeit zu einigen auf flexibleren Mechanismen beruhenden objektorientierten Sprachen (etwa den prototypbasierten Delegation [Lieberman 1986], Self [Ungar und Smith 1987] sowie Sciores Objekthierarchien [Sciore 1989]) und zu Abwandlungen vorhandener Prinzipien in klassenbasierten Sprachen (Mixins [Bracha und Cook 1990], Rollen [Wieringa et al. 1995], Multiple Instanziierung). Dieses Kapitel gibt eine Übersicht über Vor- und Nachteile der verschiedenen Varianten, zeigt Möglichkeiten zur Realisierung der Modifikations- und Klassifikationsoperationen und ordnet verschiedene in Verfahrenstechnik und Ingenieurwesen eingesetzte Modellierungssprachen und -umgebungen entsprechend ein.

4.1 Objektmodelle

Die Objektmodelle werden anhand der im vorigen Kapitel aufgezeigten Probleme sowie in der Literatur vorgeschlagener Kriterien verglichen. Der hier richtungsweisende Artikel „The Treaty of Orlando“ [Stein et al. 1989] stellt etwa die Eigenschaften *Empathy* und *Templates* vor, um die Vererbungs- und Objektgenerierungsmechanismen verschiedener klassen- und prototypbasierter Sprachen zu vergleichen.

Empathy umschreibt hierbei den Mechanismus, durch den ein Objekt (der Empathy–Nehmer) nach außen den Anschein erwecken kann, daß in Wirklichkeit in anderen Objekten (den Empathy–Gebern) enthaltene Information von ihm verwaltet wird, beispielsweise indem es sie von ihnen erbt bzw. auf sie delegiert. Empathy–Beziehungen können nach Dynamik, Automatik und betroffenen Objekten klassifiziert werden. Die Dynamik einer Empathy–Beziehung bezeichnet den Zeitpunkt, zu dem sie ausgewertet wird: bei jeder Ausführung einer Operation (*dynamisch*) oder nur bei der Objektkreierung (*statisch*). *Implizite* Empathy bezeichnet einen Sprachautomatismus, der bei Bedarf automatisch den Empathy–Beziehungen folgend die benötigte Information holt. *Explizite* Empathy hingegen erfordert explizite Aufrufe der übernommenen Methoden des Empathy–Gebers. *Objekt–Empathy*–Beziehungen erlauben unterschiedliche Empathy–Geber von Objekt zu Objekt, während bei *Gruppen–Empathy* sich eine Menge von Objekten die gleiche Beziehung teilt. Das typische klassenbasierte Objektmodell besitzt also statische, implizite Gruppen–Empathy zwischen den Klassen.

Als **Templates** werden diejenigen Modellbestandteile bezeichnet, die bei der Erzeugung eines neuen Objekts als Vorlage dienen. Man unterscheidet sie anhand ihrer Gültigkeitsdauer und erneut nach den betroffenen Informationen. Aus einem *strikten* Template erzeugte Objekte können nicht von den

in der Vorlage enthaltenen Informationen abweichen, während *lose* Templates beliebige nachträgliche Änderungen zulassen. *Klassen–Templates* prägen dem generierten Objekt Attributstruktur und ggf. Methoden auf, während *Objekt–Templates* zusätzlich noch die Attributwerte belegen. Die klassenbasierte Objektorientierung benutzt ihre Klassen also als strikte Klassen–Templates.

Diese Klassifizierungsalternativen aus [Stein et al. 1989] müssen allerdings erweitert werden, um alle Unterschiede der hier vorgestellten Sprachen erfassen zu können: Bei Empathy wird zusätzlich nach den Einschränkungen der möglichen Empathy–Beziehungen unterschieden, d.h. ob bestimmte Beziehungsarten verboten sind. Bei Templates wird auch die Dynamik beachtet, d.h. inwiefern ein Objekt seine Templates nach seiner Generierung ändern kann.

Weitere Kriterien sind die Typisierungsmöglichkeiten und die zur Erzeugung einer Suchhierarchie verwendbaren Ordnungsbeziehungen. Einige Objektmodelle schränken die Möglichkeit der **Typisierung** von Attributen oder Objekten ein. Sei es, daß Typüberprüfungen grundsätzlich nur zur Laufzeit möglich sind oder sei es, daß die möglichen Typen eingeschränkt sind. Späte oder eingeschränkte Typtests können Auswirkungen auf die Qualität der Entwurfsergebnisse haben. Vorhandensein und Aussehen der **Ordnungshierarchien** im jeweiligen Objektmodell haben einen entscheidenden Einfluß auf die Einfachheit von Such- und Stöber–Operationen.

$\#I$	Zahl der Instanzen/Objekte
$\#K$	Zahl der Klassen
$\#A$	Anzahl der sich in Typ oder Namen unterscheidenden Attribute
$\#ApI$	durchschnittliche Attributzahl pro Instanz
PV	Platzbedarf eines Verweises
PN	Platzbedarf eines durchschnittlichen Attributnamens
PT	Platzbedarf einer durchschnittlichen Attributtypinformation
PW	Platzbedarf für den durschnittlichen Wert

Tabelle 4.1: Zur Berechnung des Speicherbedarfs verwendete Variablen

Aus Implementierungssicht ergibt sich das Kriterium des mit dem Objektmodell verbundenen Speicher–Overheads, der durch die benötigten Verwaltungsstrukturen und redundante Speicherung entsteht. Der **zusätzliche Speicherverbrauch** S_{ov} läßt sich natürlich nur abstrakt berechnen, da er von den Einzelheiten der jeweiligen Implementierung abhängt. Entsprechend werden für viele der in Tabelle 4.1 aufgeführten Parameter vereinfachende Annahmen getroffen, bzw. sie werden mit Durchschnittswerten besetzt: $PV \simeq 4$ Bytes, $PN \simeq 10$ Bytes, $PT \simeq 6$ Bytes (für Domäne, Kardinalität und Facetten), $PW \simeq 4$ Bytes (Wert oder Verweis). Der für Methodeninterface und -implementierung benötigte Platz wird vernachlässigt. Ebenso werden die Namen, die ggf. Klassen und Instanzen gegeben werden, nicht berücksichtigt, da es sich hierbei um zusätzliche Information handelt, die je nach verwendetem Objektmodell vorhanden ist oder nicht. Der Overhead ergibt sich dann als Quotient aus dem tatsächlich verbrauchten Speicher und dem 'optimalen' zur Speicherung aller Informationen benötigtem Speicher S_{opt}:

$$S_{opt} = \#I * \#ApI * PW + \#A * (PT + PN)$$

Als 'optimal' wird hierbei der Platzbedarf der in den Objekten enthaltenen Information ohne Verwaltungsinformation angenommen. D.h. es wird für jede Instanz sämtliche Wertinformation abgespeichert, die Typ- und Namensinformation für jedes Attribut nur einmal verzeichnet und keine Verknüpfungen zwischen Objekt- und Typinformation benötigt. S_{opt} kann in der Realität nie erreicht werden, da ein entsprechendes System aufgrund der fehlenden Verwaltungsinformation keinen Zugriff mehr auf die gespeicherten Informationen hätte. Dennoch gibt die Formel nicht den minimalen Speicherbedarf an, da zur Rechenvereinfachung Kompressionsmöglichkeiten in der Instanzdarstellung ignoriert werden (etwa wenn mehrere Instanzen gleiche Werte haben). Um eine einfachere Vergleichsmöglichkeit zu haben, werden für die in Tabelle 4.2 auf Seite 48 beschrieben vier Szenarien jeweils Zahlenwerte ausgerechnet und in Tabelle 4.3 aufgeführt.

Eher subjektiv ist das Kriterium der **Verständlichkeit** des Objektmodells. Es besitzt aber hier eine hohe Relevanz, da sich Nicht-Informatiker als Modellierer und Reorganisierer betätigen sollen, so daß ein schwer zu erlernendes und zu benutzendes Objektmodell vermieden werden sollte. In [Krueger 1992, S.136] wird bei der Definition der kognitiven Distanz unterschieden zwischen dem Aufwand, vom anfänglichen Design zur Darstellung desselben in der Sprache der Wiederverwendungstechnik zu gelangen, und dem Aufwand, von dieser Darstellung aus ein ausführbares Programm zu erhalten. Da letzteres im hier behandelten Anwendungsgebiet dem Schritt vom konzeptionellen zum mathematischen Modell entspricht, dessen Aufwand durch Wiederverwendung im konzeptionellen Modell nur gering beeinflußt wird, wird im folgenden zwischen dem jeweils notwendigen kognitiven Aufwand unterschieden,

a) ein Objekt zu suchen,
b) ein Objekt entsprechend den Anforderungen zu modifizieren,
c) ein modifiziertes Objekt einzuordnen.

Es sei bereits jetzt darauf hingewiesen, daß es kein gemeinsames Minimum dieser drei Aufwände geben kann, da sich einfache Suche/Modifizierbarkeit und einfache Einordbarkeit nur schlecht vereinbaren lassen. Dies wird auch durch Taivalsaaris Aussage nahe gelegt, daß Wiederverwendbarkeit und konzeptionelle Klarheit gegensätzliche Ziele sind [Taivalsaari 1996, S.470].

Die vorgestellten Kriterien (Empathy, Templates, Typisierung, Ordnungshierarchie, Speicherverbrauch und Verständlichkeit) umfassen alle in Kapitel 1 genannten Anforderungen an das Datenmodell. So erlauben dynamische Templates und dynamische Empathy eine einfache Modifizierbarkeit, während lose Templates sich auf die Variierbarkeit positiv auswirken. Die Vorteile einer Ordnungshierarchie für die Suche und damit die Übersichtlichkeit legte bereits Kapitel 3 dar. Ohne ausreichende Typisierung von Attributen vereinfacht sich zwar die Erstellung dieser Hierarchie, ihr Nutzen sinkt aber gleichzeitig. Der Speicherverbrauch eines Objektmodells kann Auswirkungen auf seine Skalierbarkeit bei großen Objektmengen haben und die Notwendigkeit eines verständlichen Modells wurde bereits in Kapitel 1 erörtert.

4.1.1 Prototypbasierte Modelle

Lieberman kritisierte 1986 die klassenbasierte Objektorientierung als nicht dem natürlichen Problemlösungsverhalten von Menschen entsprechend, da es schwieriger sei, zuerst eine Abstraktion eines Konzepts zu gewinnen und dann Beispiele zu erzeugen als umgekehrt[1]. Zumal die bei der Abstraktion gewünschten essentiellen Eigenschaften eines Konzepts sich nur schwer identifizieren ließen, wenn Beispiele fehlen, die die nichtessentiellen Eigenschaften nicht besitzen.
Aus dieser Kritik heraus schlugen Lieberman [Lieberman 1986] und Borning [Borning 1986] ein auf Beispielen beruhendes Objektmodell vor. Die Objekte selbst dienen dabei als Vorlagen für neue, per sogenanntem "Cloning" erzeugte Objekte. Statt auf Klassen und Superklassen verweisen Objekte aufeinander als beispielhafte Implementierungen von Verhalten oder Datenwerten. Letzteres wird als "Delegation" bezeichnet.

Prototypbasierte Sprachen speichern also sowohl Werte als auch Typinformation in den einzelnen Objekten[3]. Dadurch wird bei der Delegation nicht nur der Typ sondern auch der Wert eines Attributs übernommen. In Abbildung 4.1 rechts würde also die Nachricht `Tray123.geometry` vom Objekt `Tray123` an `HetPh321` weitergeleitet und von dort mit dem Wert `GEO213` beantwortet. Für den Benutzer erscheint also `Tray123` ein Attribut `geometry` mit Wert `GEO213` zu haben. Neue Objekte werden aus vorhandenen erzeugt, indem entweder ein neues, leeres Objekt generiert wird, das auf das

[1] „People seem to be a lot better at dealing with specific examples first, then generalizing from them than they are at absorbing general abstract principles first, and later applying them in particular cases." [Lieberman 1986, S.215]

[2] Wie in Kapitel 2 erwähnt, kann ein Attribut in diesem Zusammenhang durch eine Lese- und eine Schreibmethode repräsentiert werden.

[3] Klassenbasierte Sprachen hingegen speichern nur die Wertinformation in den Objekten die Typinformation hingegen in Klassen (vgl. Seite 51).

Grundlagen 4.1: Vererbung, Delegation und die Bedeutung von "self"

Vererbung und Delegation sind verschiedene Mechanismen, um Methoden und Attribute durch mehrere Objekte gemeinsam zu nutzen. Klassen **vererben** einander Methoden und Attributstruktur, die dann von den Instanzen genutzt werden kann, während Objekte per **Delegation** auch die Attribut*werte* anderer Objekte als eigene ausgeben.
Zum Verständnis beider Mechanismen ist zu beachten, daß das Delegieren einer Methode bzw. das Nutzen eine geerbten Methode[2] keineswegs einem einfachen Aufruf der neuen Methode durch die delegierende entspricht. Vielmehr wird ein meist "self" oder "this" genannter, zusätzlicher und für den Benutzer unsichtbarer Parameter übergeben, der auf das delegierende Objekt/die erbende Klasse verweist und genutzt wird, um in der delegierten/geerbten Methode vorhandene Methodenaufrufe wieder auf das ursprüngliche Objekt/die ursprüngliche Klasse zu beziehen.

ursprüngliche Objekt delegiert (`Tray125` in Bild 4.1), oder indem das ursprüngliche Objekt komplett — also einschließlich etwaiger Delegationsverweise — kopiert wird (*Cloning*; `Tray124` in Abbildung 4.1). Anschließend können Werte, Attribute und Methoden des Objekts modifiziert werden. Man

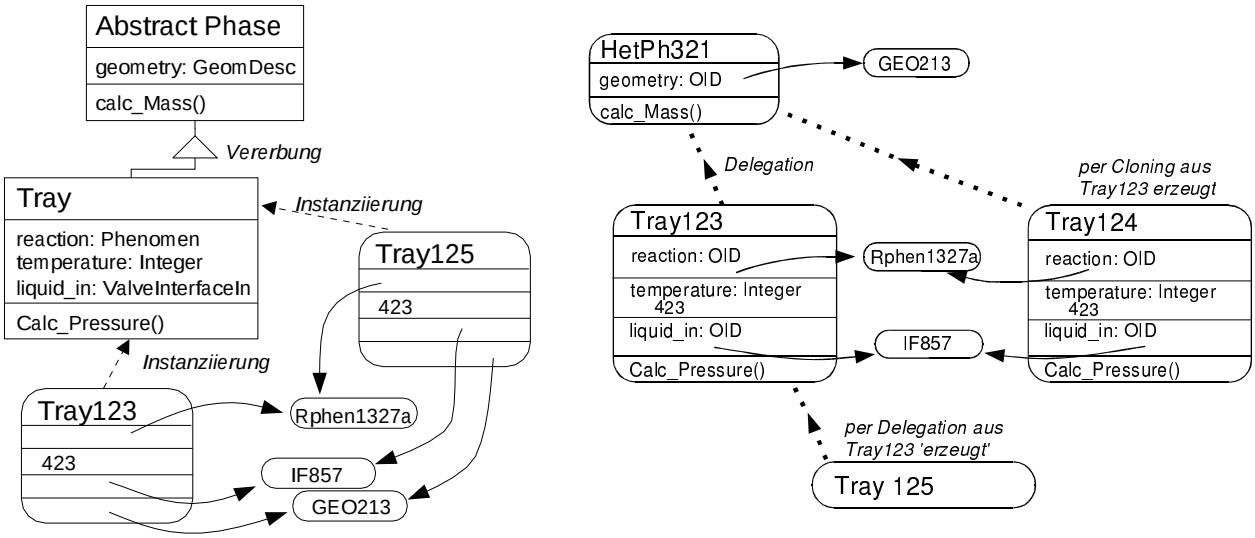


Abbildung 4.1: Klassenbasierte und prototypbasierte Repräsentation eines Kolonnenbodens (vereinfachtes Beispiel)

beachte, daß das Löschen eines durch Delegation erlangten Attributs nur möglich ist, indem zu einem anderen Objekt delegiert wird, das dieses Attribut nicht besitzt. Delegation ist also eine "name-compatible modification" (s. S. 53), die nur das Überschreiben von Wert oder Typ eines ererbten Attributs zuläßt, nicht aber das Löschen.

Nur lose Templates, keine Delegation

Verwendet man wie in [Borning 1986] oder in [Gross-Hardt und Vossen 1993] Objekte nur als gegenseitige Kopiervorlagen, ohne Verhalten oder Struktur gegenseitig zu erben/delegieren, so erhält man die bekannte Vorgehensweise „Copy & Modify" in ihrer Reinform. Zwar bietet dies ein Maximum an Flexibilität bei der Erstellung neuer Objekte, da die oben erwähnte Löschbeschränkung der Delegation nicht besteht, aber die entstehenden Modelle verbrauchen in den meisten Fällen unökonomisch viel Speicher: $S_{templ} = \#I * \#ApI * (PW + PN + PT')$[4], wie auch die Werte der zweiten Zeile von Tabelle 4.3 zeigen. Zudem sind Korrekturen fehlerhafter Objektstrukturen/-werte aufwendig, da sie nicht automatisch zu allen betroffenen Objekten propagiert werden [Taivalsaari 1996, S.451]. In allen

[4] Wie im nächsten Abschnitt ausführlich beschrieben, ist auch hier die vorhandene Typinformation stark eingeschränkt. Darum reichen $PT' \simeq 2$ Bytes zu ihrer Darstellung.

im folgenden vorgestellten Objektmodellen verwenden darum Objekte auf verschiedene Arten Teile ihrer Definitionen gemeinsam (*Sharing*).

Lose Templates und Delegation

Die in [Lieberman 1986] vorgestellte Sprache DELEGATION besitzt keine Templates. Da sich dadurch die Generierung typgleicher, aber in ihren Werten unterschiedlicher Objekte deutlich verkompliziert, führten spätere prototypbasierte Sprachen zusätzlich Templates ein. Aus diesem Grund soll das templatelose Modell hier nicht weiter betrachtet werden sondern nur die Kombination von Delegation und Templates.

Durch die Verwendung von Objekten als lose Templates, wie sie etwa bei prototypbasierten Objektmodelle existieren, lassen sich Objekttypen schnell modifizieren. Die Empathy–Beziehung kann entweder zum Generierungszeitpunkt eines Objekts als auf das Template–Objekt zeigend festgelegt werden, oder ein dynamisch änderbares "parent"–Attribute repräsentiert sie (etwa in SELF). Da die Empathy eine 'Vererbung' von Werten auf Objektebene erlaubt, können Speicherplatz gespart und Gleichheitsbedingungen zwischen einzelnen Attributen einfach sichergestellt werden.

Mit dieser Wertvererbung allerdings zeigt sich bereits das erste der **Probleme** dieses Ansatzes. Wertänderungen in einem Objekt können andere Objekte beeinflussen. Wird dies nicht gewünscht, so müssen die entsprechenden Attribute einschließlich ihrer Typinformation in jedem Objekt gespeichert werden oder dürfen nur als überschreibbare Default–Werte betrachtet werden [Bardou und Dony 1996, S.129f]. Zwar läßt sich dies automatisieren, der Speicherplatzbedarf nähert sich dadurch aber dem an, der sich bei völligem Verzicht auf Delegation ergibt. Ein weiteres Problem ergibt sich daraus, daß die Empathy–Geber (also die Objekte, auf die delegiert wird) beliebige Objekte sind. Es kann darum nicht davon ausgegangen werden, daß ein Empathy–Geber ein typisches Beispiel für das von ihm zu repräsentierende Konzept darstellt [Sciore 1989, S.109]; möglicherweise ist er einfach das erste erzeugte bzw. das als erstes gefundene Beispiel für das Konzept. Delegation entspricht damit am ehesten der Implementierungsvererbung zwischen Klassen, die in [Taivalsaari 1996] ebenfalls als "inheritance for convenience" bezeichnet wird. Delegation spannt aber im Gegensatz zur Spezialisierungs- oder die Subtyprelation in klassenbasierten Systemen keine Ordnungshierarchie auf. Eine Subtyprelation kann in diesem Ansatz auch darum nicht entstehen, da weder durch den losen Templatemechanismus noch durch die Delegation Typzusicherungen gemacht werden. Wegen der Nichtexistenz von Klassen können Verweisattribute nämlich nicht getypt werden. Auch der Typ eines Objekts (also Zahl und Namen seiner Attribute und Methoden) kann nur durch Betrachtung des Objekts selbst festgestellt werden, nicht aber aufgrund des Templates, aus dem es erzeugt wurde, oder aufgrund der Objekte, zu denen es delegiert. Der Typ eines Objekts besteht somit nur aus Zahl und Namen seiner Attribute und Methoden sowie aus der Zahl der Parameter der einzelnen Methoden. Lediglich Attribute, die primitive Datentypen beinhalten, besitzen somit eine Typspezifikation, Verweisattribute sind typlos. Es ist klar, daß hierdurch fast jegliche Typprüfung benutzerdefinierter Typen vor der Laufzeit unmöglich wird.

Bei der Berechnung des **Speicherbedarfs** bringt diese schwache Typisierung allerdings Vorteile. Da bei Verweisattributen keine Domäne bekannt ist und primitive Typen nicht eingeschränkt werden können, wird der für Typinformation benötigte Platz in diesem Modell mit lediglich 2 Bytes veranschlagt (und darum mit PT' bezeichnet). Als Gesamtspeicherbedarf ergibt sich dann $S_{prot} = \#I * (PV + \#nApI * (PW + PN + PT'))$ wobei $\#nApI$ die Zahl der durchschnittlich aufgrund von Wertänderungen neuzudefinierenden Attribute pro Instanz angibt. In den vier Szenarien ergeben sich damit die Zahlen der dritten Zeile von Tabelle 4.3. Im Szenario S1 muß beachtet werden, daß die Wertvererbung nur in begrenztem Umfang in Anspruch genommen werden kann, da überflüssige Attribute beim Delegieren nicht entfernt werden können, so daß nur bei Namenskompatibilität zwischen den Objekten delegiert werden kann. Es wird darum angenommen, daß durchschnittlich lediglich zwei Attribute delegiert werden können. Um bei Szenario S3 ein realitätsnahes Beispiel zu erhalten, geht die Berechnung davon aus, daß von einer Hierarchiestufe zur nächsten nur jeweils 1/4

der Attributwerte nicht geändert wird.
Wie zu erwarten, schneiden prototypbasierte Verfahren beim Speicherverbrauch dann besonders gut ab, wenn viele einzigartige Objekte repräsentiert werden müssen, und schlecht, wenn zwischen den Objekten strukturelle Ähnlichkeiten, aber keine Wertgleichheiten bestehen.

Objektszenario (S1): 1000 Objekte, die sich jeweils paarweise in (mindestens) vier Attributen wert- und typmäßig unterscheiden. Jedes Objekt besitzt 10 Attribute. Insgesamt gibt es also 4006 einzigartige Attribute mit unterschiedlichen Werten.

Klassenszenario (S2): 1000 Objekte, alle mit exakt den gleichen 10 Attributen, aber unterschiedlichen Werten.

Vererbungsszenario (S3): 1000 Objekte mit je 10 Attributen, die sich in 200 Gruppen mit Objekten jeweils gleicher Attribute befinden. Die Gruppen sind wiederum hierarchisch strukturiert, wobei die Attribute sich aus einer Hinzufügung jeweils zweier Attribute pro Stufe ergeben (insgesamt also 5 Hierarchiestufen und 570 verschiedene Attribute).

Mehrfachvererbungsszenario (S4): 1000 Objekte mit je 10 Attributen, die sich in 64 Gruppen mit Objekten jeweils gleicher Attribute befinden. Die Gruppen ergeben sich aus der Hierarchie von Bild 4.2, d.h. jede Gruppe bezieht ihre Attribute aus je einer der vier Untergruppen der Gruppen 1, 2 und 3. Der Speicherbedarf in diesem Szenario wird in Tabelle 4.3 nicht immer mit Hilfe der angegebenen Formel berechnet, da die Grundannahme der gleichmäßigen Verteilung der Attribute auf die Klassen nicht mehr korrekt ist. $S4_K$ gibt zusätzlich die Zahl der benötigten Klassen an.

Tabelle 4.2: Szenarien zur Berechnung des Speicherverbrauchs

Beispiele für solche 'reinen' prototypbasierten Sprachen sind SELF [Ungar und Smith 1987], welches aber durch die verstärkte, methodologisch geforderte Verwendung von reinen Methodenobjekten ("traits") als Empathy–Gebern die Sprache schon leicht in Richtung Klassenbasierung erweitert, sowie im ingenieurwissenschaftlichen Bereich CLOOD [Gross-Hardt und Vossen 1993], das zusätzlich Teil–Ganzes–Hierarchien auf Objektebene per Und/Oder–Graphen versioniert.

Getypte prototypbasierte Objektmodelle

Die Typarmut prototypbasierter Objektmodelle erhöht zwar ähnlich wie in SMALLTALK die Flexibilität der Sprache, jedoch können selbst einfache, häufig auf Tippfehlern beruhende Typfehler erst zur Laufzeit erkannt werden. Um dem abzuhelfen, entstanden verschiedene Sprachen, die versuchen, die in prototypbasierten Modellen durch Objekte repräsentierten Typen zu Typtests zu nutzen.

Omega von [Blaschek 1991] geht hierbei soweit, daß es trotz anderslautender Bezeichnung eher als klassenbasierte Sprache zu bezeichnen ist denn als prototypbasierte: Der von einem prototypischen Objekt definierte Typ legt die Struktur aller Objekte dieses Typs fest; Typen erben voneinander gemäß den Regeln der strengen Typisierung [Kemper und Moerkotte 1994]; der Wert eines Attributs wird nur bei sogenannten 'shared' Attributen (ähnlich den Klassenattributen in SMALLTALK) vererbt, und jedes neue Attribut erfordert bei der Definition die Angabe eines Typs. Die Unterschiede zu klassenbasierten Systemen beschränken sich also auf die stets vorhandenen Defaultwerte bei der Objekterzeugung und die für den Benutzer als Einheit erscheinenden Prototypobjekte und Typen.

Tabelle 4.3: Speicherbedarf und -overhead verschiedener Objektmodelle

Nr.	Objektmodell	Formel für den Speicherbedarf	S1	$S1_{ov}$	S2	$S2_{ov}$	S3	$S3_{ov}$	S4	$S4_{ov}$	$S4_K$
1	'optimal'	$\#I * \#ApI * PW +$ $\#A * (PT + PN)$	104096	1,00	40160	1,00	49120	1,00	40488	1,0	–
2	nur lose Templates (ohne Delegation)	$\#I * (\#ApI *$ $(PW + PN + PT'))$	160000	1,54	160000	3,98	160000	3,25	160000	3,95	–
3	prototypbasiert (mit Delegation)	$\#I * (PV + \#nApI *$ $(PW + PN + PT'))$	132000	1,27	164000	4,08	158932	3,23	136800	3,38	–
4	prototypbas. mit Attr.typisierung	$\#I * (PV + \#nApI *$ $(PW + PN + PT))$	164000	1,58	204000	5,08	195480	3,98	203120	5,02	–
5	klassenbasiert	$\#I * (PV + \#ApI * PW) + \#K *$ $(PV + \#nApK * (PT + PN))$	192000	1,84	44164	1,10	58820	1,20	50564	1,25	69
6	Multiple Spezialisierung	$\#I * (PV + \#ApI * PW) + \#K *$ $(PV * (\#SKpK + 1) + \#nApK * (PT + PN))$	196000	1,88	44168	1,10	59960	1,22	45600	1,13	80
7	Mixins	$\#I * (PV + \#ApI * PW) + \#K *$ $(PV * (\#SKpK + 1) + \#nApK * (PT + PN))$	196000	1,88	44168	1,10	59960	1,22	45568	1,13	80
8	Rollen	$\#I * (PV * (\#RpI + 1) + \#ApI * PW) +$ $\#K * (PV + \#nApK * (PT + PN))$	192000	1,84	44164	1,10	58820	1,20	52544	1,30	16
9	Multiple Inst.	$\#I * (PV * (\#KpI + 1) + \#ApI * PW) +$ $\#K * (PV + \#nApK * (PT + PN))$	196000	1,88	48164	1,20	62820	1,28	56512	1,40	16
10	Hybrid	unklare Implementierung	–	–	–	–	–	–	–	–	–
11	Objektspez.	$(\#I + \#Templ) * (2 * PV * \#TOpI + \#ApI$ $*PW) + \#K * (PV + \#ApK * (PT + PN))$	172000 212000	1.65 2.04	48164 48212	1.20 1.20	90260 113060	1.84 2.30	12208 13788	0.30 0.34	80 80

$\#SKpK$ = durchschnittliche Superklassenzahl (einschließlich Mixins) pro Klasse: 1 in S1 bis S3, unterschiedlich in S4
$\#RpI$ = durchschnittliche Rollenzahl pro Instanz: 0 in S1 bis S3, 2 in S4
$\#KpI$ = durchschnittliche Klassenzahl pro Instanz: 1 in S1 bis S3, 3 in S4
$\#Templ$ = Anzahl der vorhandenen Templates; in der oberen Wertereihe keine Templates, in der unteren: S1:1000, S2:1, S3:285
$\#TOpI$ = durchschnittliche Teilobjektzahl pro Instanz

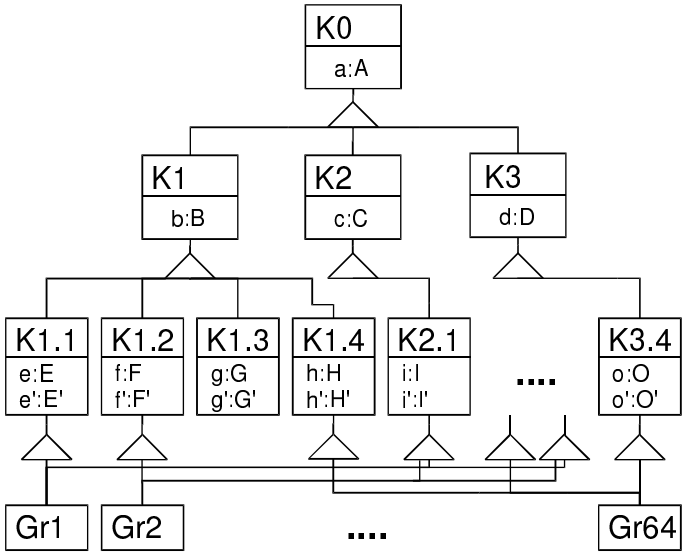


Abbildung 4.2: Hierarchie in S4

BOS [Dutoit et al. 1996; n-dim Group 1995a] erweitert quasi das Objektmodell von SELF um eine Typfacette in jedem Attribut eines Objekts. Diese verweist auf ein anderes Objekt P und schränkt die erlaubten Werte des Attributs auf diejenigen Objekte ein, die direkt oder indirekt nach P delegieren. Allerdings kann auch durch diese Typisierung von Attributen keine wirkliche Typsicherheit garantiert werden, da ein Delegationsverweis von O nach P wirkungslos wird, wenn in O alle in P definierten Attribute und Methoden neu definiert werden[5].
BOS umgeht das dadurch entstehende Problem teilweise durch die Benutzung eines Multi-Dispatch-Verfahrens[6]. Bei Attributzugriffen/-zuweisungen wird laut [Cunningham 1998] aber auf jeden Fall ein in O definiertes Attribut genutzt statt des Attributs in P. Eine Typsicherheit bei Attributzugriffen ist also nicht gegeben.

Bewertung prototypbasierter Modelle

Die zusätzliche Flexibilität und das einfachere Objektmodell prototypbasierter Sprachen werden also erkauft durch den Verzicht auf aussagekräftige Typsysteme und Ordnungstaxonomien sowie auf eine effiziente Speichernutzung. Allerdings ist die Unterscheidung zwischen prototypbasierten und klassenbasierten Sprachen nicht so strikt, wie es scheint. Nicht nur die fließenden Übergänge ins klassenbasierte Modell im oben erwähnten OMEGA bzw. ins prototypbasierte in Wiebes klassenbasierter 'Partial Instantiation' [Wiebe 1988] lassen diesen Schluß zu. Bereits 1987 zeigte [Stein 1987], daß die in prototypbasierten Sprachen verwendete Delegation die gleiche Mächtigkeit wie die Vererbung zwischen Klassen besitzt und sich beide Mechanismen aufeinander abbilden lassen. Der zugehörige Beweis läßt sich in der Praxis konstruktiv nur schlecht verwenden, da er Instanzattribute auf Klassenattribute abbildet und sich damit in die Abhängigkeit der auf Klassenebene meist strengeren Typrestriktionen begibt[7]. Er führte aber zu Versuchen, klassenbasierte Konzepte und prototypba-

[5] D.h. wenn Qs Attribut a die Typfacette P hat, kann a selbst dann auf O verweisen, wenn O alle Methoden und Attribute selbst definiert (und somit niemals delegiert), solange es nur einen Delegationslink nach P besitzt.

[6] D.h. die Entscheidung über die auszuführende Methode wird gleichberechtigt aufgrund der Typen aller Parameter getroffen.

[7] Beim Übergang von Delegation zu Vererbung werden in Steins Beweis sämtliche prototypbasierten Objekte in Klassen und ihre Attribute in Klassenattribute umgewandelt. Würde dies zur Abbildung einer prototypbasierten Sprache in eine klassenbasierte genutzt, so ergäbe sich nicht nur eine ungewöhnliche Objektrepräsentation, es entstünde auch das Problem, daß Klassenattribute in vielen Sprachen (z.B. SMALLTALK und *ConceptBase*) nicht verfeinert/überschrieben werden dürfen, da dies Subtyp- bzw. Spezialisierungsbedingungen verletzt. Überschreiben von Klassenattributen ist aber notwendig, um das bei Delegation mögliche beliebige Modifizieren von Attributen abbilden zu können.

sierte Konzepte in einem Objektmodell zu vereinen. Diese Vereinigungen werden in Kapitel 4.1.3 vorgestellt.

Als wesentliches Unterscheidungsmerkmal zwischen prototyp- und klassenbasierten Systemen verbleibt nach Steins weitgehender Gleichsetzung von Vererbung und Delegation der Unterschied zwischen losen und strikten Templates. Dies auch deshalb, weil lose Templates in prototypbasierten Modellen einerseits die flexible Handhabung von Objekten ermöglichen, andererseits aber auch für die Ineffizienzen in Speicherplatz und Datenbankindizierung [Zdonik 1990] sowie für das Fehlen von Typisierung und Ordnungshierarchien verantwortlich sind. Die gleichen negativen Effekte treten bei den Ansätzen zur Lockerung der Templatebeziehung in klassenbasierten Modellen (s. S. 52) ebenfalls auf.

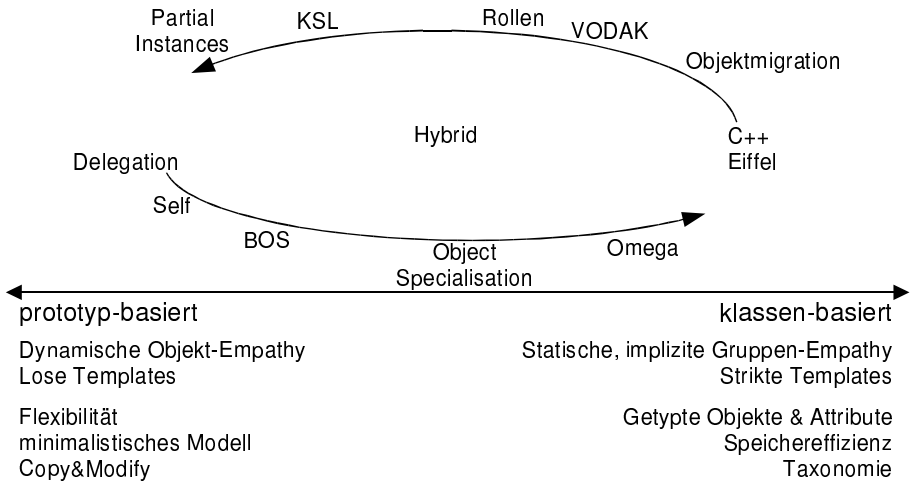


Abbildung 4.3: Einordnung verschiedener Objektmodelle

4.1.2 Klassenbasierte Modelle

Die Kenntnis der klassischen, klassenbasierten Objektorientierung wird zwar in dieser Arbeit vorausgesetzt, dennoch soll kurz auf ihre hier relevanten Eigenschaften eingegangen werden, ehe ihre Varianten erläutert werden. Klassenbasierte Objektorientierung trennt die Typ- und Wertinformation eines Attributs und speichert sie in verschiedenen Konzepten (vgl. Abbildung 4.1 links): Attributtypen und -namen finden sich zusammen mit den Methoden in der **Klasse** wieder und können dort vererbt werden. Die Attributwerte werden in den **Instanzen** der Klasse gespeichert, und es kann auf sie von anderen Instanzen nur per Verweis, nicht aber per Empathy zugegriffen werden. Da Instanzen fest mit ihren Klassen verbunden sind und nicht in ihrem Typ variieren, kann mit Hilfe der zwischen den Klassen bestehenden Subtyp- bzw. Spezialisierungshierarchie eine Hierarchie über den Objekte aufgebaut werden.

Neue Instanzen werden ausgehend von einer Klasse aufgrund der dort vorgegebenen Struktur erzeugt. Das neu entstandene Objekt muß dann normalerweise mit Werten initialisiert werden. **Änderungen** an Objektstruktur und -typ können nur durch Erzeugen einer neuen Klasse mit den entsprechenden Modifikationen, dem Kreieren einer Instanz dieser Klasse und dem Kopieren der gleichgebliebenen Attribute aus der alten in die neue Instanz vorgenommen werden.
Wie am Anfang dieses Kapitels bereits erwähnt, stellen Klassen im Sinne der Aufteilung von [Stein et al. 1989] strikte, statische Klassentemplates dar, und die Kombination aus Vererbung und Instanziierung entspricht einer statischen, impliziten Gruppen–Empathy.

Strikte Templates sind der Hauptunterschied zwischen klassen- und prototypbasierter Objektorientierung. Möchte man also mit einer klassenbasierten Sprache eine ähnliche **Flexibilität** wie mit

einer prototypbasierten erreichen, so kann man zum einen versuchen, die Striktheit der Templates aufzulösen. Entsprechende Ansätze werden im folgenden Unterabschnitt „Instanzenmodifikationen“ behandelt. Alternativ kann man aber auch den Wechsel des oder der strikten Templates eines Objekts zur Laufzeit ermöglichen, indem man den Objekten erlaubt, dynamisch ihre Instanziierungsbeziehungen zu ändern. Der zugehörige Mechanismus, die Objektmigration, wird in Abschnitt 4.1.4 beschrieben. Können die Klassen dadurch gewechselt werden, müssen noch Mechanismen zur Verfügung gestellt werden, um passende Klassen schnell erzeugen, modifizieren oder kombinieren zu können. Entsprechende Verfahren sind Thema der letzten Unterabschnitte diese Abschnitts, wobei die ersten drei Verfahren keine über die im normalen klassenbasierten Modell vorhandenen Konzepte hinausgehenden Mechanismen benötigen, die letzten beiden jedoch Mixin–Klassen zur Klassenmodifikation und Rollen zur klassenbasierten Instanzmodifikation verwenden.

Instanzmodifikationen

Instanzspezifische Modifikatoren erlauben es, auch in klassenbasierten Sprachen mit eigentlich strikten Templates eine Instanz unabhängig von ihrer Klasse zu erweitern bzw. zu ändern. Je nachdem, wie weit in der Klasse vorhandene Attribute revidiert oder gar entfernt werden können, nähern sich sowohl die Flexibilität als auch die mangelnde Typsicherheit der Instanzen derjenigen von prototypbasierten Systemen an.

KSLs “**Trap**“–Mechanismus [Ibrahim et al. 1991] erlaubt es beispielsweise einer Instanz, beliebige Nachrichten[8] an sie abzufangen, bevor diese zur zugehörigen Klasse gelangen und somit zusätzliches oder geändertes Verhalten anzubieten. Bezüglich Flexibilität und Typsicherheit entspricht dies weitgehend einer prototypbasierten Sprache.

Durch die in VODAK vorhandenen “**Instance–Types**“ [Klas und Schrefl 1995, S.32ff] lassen sich dagegen nur Erweiterungen der Objektstruktur vornehmen bzw. vorhandene Methodenimplementationen ändern [Klas und Schrefl 1995, S.39f]. Als Ausgleich für die verringerten Modifikationsmöglichkeiten erhält man aber eine Signaturkompatibilität zwischen Klasse und Objekt, wie sie sich äquivalent auch durch eine Unterklasse mit Subtypcharakter (vgl. S. 25) ergeben würde.

Als vorteilhaft erweist sich das Konzept der Instanzmodifikation für die Modellierung von Einzelinstanzen, sogenannter Singletons, also von Instanzen, die jeweils einzige Instanz ihrer Klasse bleiben müssen, sowie für die Modellierung von seltenen Abweichungen eines Objekts von der Klassenstruktur in Form zusätzlicher oder fehlender Attribute/Methoden. Die Informatikliteratur verwendet als Beispiel für letzteres gerne dreibeinige oder fliegene Elefanten [Stein et al. 1989]. In der Prozeßtechnik kommen als Beispiele etwa aus dem Bereich der Unit Operations ein Reaktor, in dem (wegen fehlendem Katalysator) keine Reaktion stattfindet, oder eine durchgeschlagene Pumpe, die in falscher Richtung durchströmt werden kann, in Frage. Negativ schlägt bei den Instanzmodifikationen jedoch zu Buche, daß insbesondere Mechanismen wie die Traps von KSL es einer Instanz erlauben, sich beliebig weit von der Typvorgabe der Klasse zu entfernen, und somit Typtests und Suchoperationen erschweren.

Vererbung als inkrementelle Modifikation

Will man — etwa zwecks Migration eines Objekts — eine Klasse modifizieren, so besteht die einfachste Möglichkeit darin , eine neue Klasse als Subklasse der Klasse zu erzeugen und dabei die gewünschten Modifikationen vorzunehmen. Dies entspricht der Betrachtungsweise von Vererbung als inkrementellem Modifikationsmechanismus [Wegner und Zdonik 1988]. In einer an [Wegner und Zdonik 1988] angelehnten Notation schreiben wir hierfür SK = K $\oplus\triangle$M(K), wobei SK für die Unterklasse steht, K für die Klasse, $\triangle$M für den jeweiligen Modifikator (vgl. Bild 4.4) und $\oplus$ für die Operation, mittels der der Modifikator Attribut- und Klassendefinitionen in K überschreibt. In

[8] Also Attributlese- und -schreiboperationen sowie Methodenaufrufe.

Abbildung 4.4 ergänzt △M(K) die Klasse K um das Attribut c, ändert den Typ von b sowie die Implementierung der Methode p(). Man beachte, daß △M(K) auf Ks Eigenschaften zurückgreifen kann, im Beispiel indem es die Methode K.p() zur Definition von p() benutzt. Die in △M zulässigen Modifikationen richten sich nach den mit der Vererbungsrelation verknüpften Typbeschränkungen, meist sind nur signaturkompatible Modifikationen erlaubt[9]. In [Wegner und Zdonik 1988] werden insgesamt vier **Modifikationsstufen** definiert: *Cancelation*, *name-*, *signature-* und *behaviour–compatible modification*. Die erste macht keine Zusicherungen fuer SK, die zweite garantiert weiter vorhandene Attribut- und Methodennamen, die dritte macht Aussagen über die von K zu SK erlaubten Typänderungen von Attributen und Methoden und die vierte sichert gleiches Verhalten aller in K vorhandenen Methoden zu.

Cancelation erhält man beispielsweise bei rein templatebasierten Objektmodellen, Namenskompatibilität wird von Delegation unterstützt, die meisten klassenbasierten Objektmodelle sichern Signaturkompatibilität zu und Verhaltenskompatibilität tritt in der Praxis kaum auf, da sie sich nicht überprüfen läßt. Gegenüber prototypbasierten Systemen existieren in klassenbasierten also nur eingeschränkte Modifikationsmöglichkeiten.

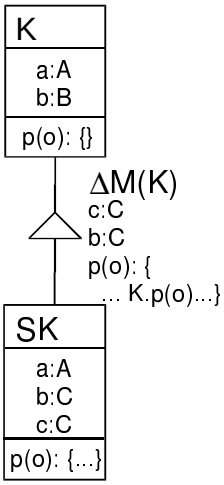


Abbildung 4.4: SK = K ⊕△M(K)

Häufig besteht der Wunsch, Modifikatoren △M, die zur Erzeugung einer Unterklasse genutzt wurden, auch für andere Klassen zu benutzen [Bracha und Cook 1990], insbesondere wenn sich Klassen und ihre möglichen Modifikationen jeweils ähneln, wie im Beispiel aus Kapitel 3. Verzichtet man auf den Rückbezug des Modifikators (also auf das K in △M(K)), was in vielen Fällen ausreicht, so läßt sich diese **Modifikator'wiederverwendung'** durch Multiple Vererbung erreichen, indem die Oberklassen einer Klasse als gegenseitige Modifikatoren aufgefaßt werden. Legt man Wert auf den Rückbezug des Modifikators, so gelangt man zu Konzepten wie Mixin–Klassen und Rollen, die ab Seite 56 besprochen werden.

Multiple Vererbung

Multiple Vererbung, also das Erben von Attributen und Methoden von mehr als einer Klasse, sieht in der oben benutzten Notation von Wegner und Zdonik folgendermassen aus: SK = (K_1 ⊕ K_2) ⊕△M(K_1 ⊕ K_2) (ggf. mit weiteren direkten Oberklassen). Verzichtet man auf den Modifikator, d.h. die Inhalte der beiden Klassen können in der Subklasse nicht mehr spezialisiert oder erweitert werden, so ergibt sich SK = K_1 ⊕ K_2. Nun modifiziert ausschließlich K_2 K_1 (oder umgekehrt) und die Unterklasse wird häufig Vereinigungsklasse genannt, da sie die Vereinigung der Definitionen von K_1 und K_2 darstellt[10].

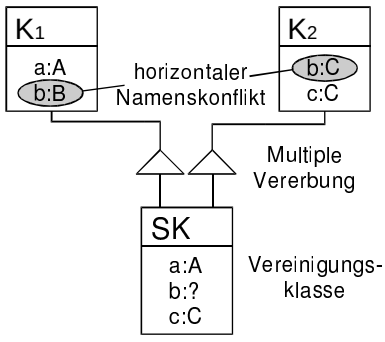


Abbildung 4.5: SK = K_1 ⊕ K_2

Diese gegenseitigen Modifikationsmöglichkeiten sind allerdings weitgehend auf die Erweiterung um zusätzliche Attribute/Methoden begrenzt, zudem können Methoden aus K_2 nicht auf Methoden und Attribute von K_1 zugreifen, da kein Rückbezug der Art △M(K_1) genutzt werden kann. Die Behandlung in K_1 *und* K_2 vorhandener Attribute und Methoden fällt unter den Begriff „Auflösung von Namenskonflikten" (siehe etwa [Mezini 1997; Knudsen 1988]), welche nur unter bestimmten Voraussetzungen die Modifikation der vorhandenen Attribute zuläßt. Betrachtet man etwa das in [Stein et al. 1989] gestellte

[9] Wobei sich aus den unterschiedlichen Definitionen der Signaturkompatibilität die auf Seite 25 vorgestellten Unterschiede zwischen Subtyp- und Spezialisierungsrelation ergeben.

[10] Im Englischen hingegen "intersection class" da sie die Schnittmenge der Instanzen von K_1 und K_2 enthält.

"Coloured Shape" Problem der nachträglichen **Abänderung ganzer Klassenhierarchien**[11], so stellt man fest, daß sich dies aufgrund der Namenskonfliktprobleme mit multipler Vererbung nur teilweise lösen läßt. In Abbildung 4.6 kann `FarbigerKreis` zwar das Farbattribut von `FarbForm` erben, seine Darstellungsmethode (also `Zeichnen()`) kann aber von `FarbForm` nicht überschrieben werden. Dort kann lediglich eine zusätzliche Methode `FarbZeichnen()` definiert werden. Nur wenn man statt der multiplen Vererbung von `FarbForm` jeweils ein $\triangle$M(K) einsetzt, indem man in allen Vereinigungsklassen die Definition von `Zeichnen()` entsprechend überschreibt, werden farbige Formen automatisch richtig gezeichnet. Für die Generierung neuer Klassen entsprechend den Migrationswünschen eines Objekts reichen die Möglichkeiten der multiplen Vererbung trotzdem häufig aus. Allerdings entsteht ein anderes Problem: die **Explosion der Klassenanzahl**.

Daß das klassenbasierte Objektmodell äußerst ineffizient wird, wenn sich die Klassenzahl der Objektzahl annähert, zeigt schon Szenario S1 in Tabelle 4.3. Auch die Übersichtlichkeit der Klassenhierarchie leidet darunter. Da bei der Verwendung von Multipler Vererbung für jede gewünschte Kombination von Oberklassen eine Vereinigungsklasse erzeugt werden muß, im ungünstigsten Fall also $2^{\#OK} - 1$ [Odell 1992; Bardou und Dony 1996; Sciore 1989], nimmt deren Zahl stark zu, was das Objekt/Klassen–Verhältnis entsprechend verschlechtert. Die entsprechenden Probleme treten noch weitaus stärker auf, wenn in einem Modell mit statischer Klassenhierarchie alle durch Multiple Vererbung erwünschten Klassen bereits vor der Generierung von Objekten erzeugt werden müssen oder zur Verbesserung der Wiederverwendbarkeit Klassen in feingranulare Einheiten aufgeteilt werden ('fine–grain inheritance' in [Johnson und Rees 1992]).

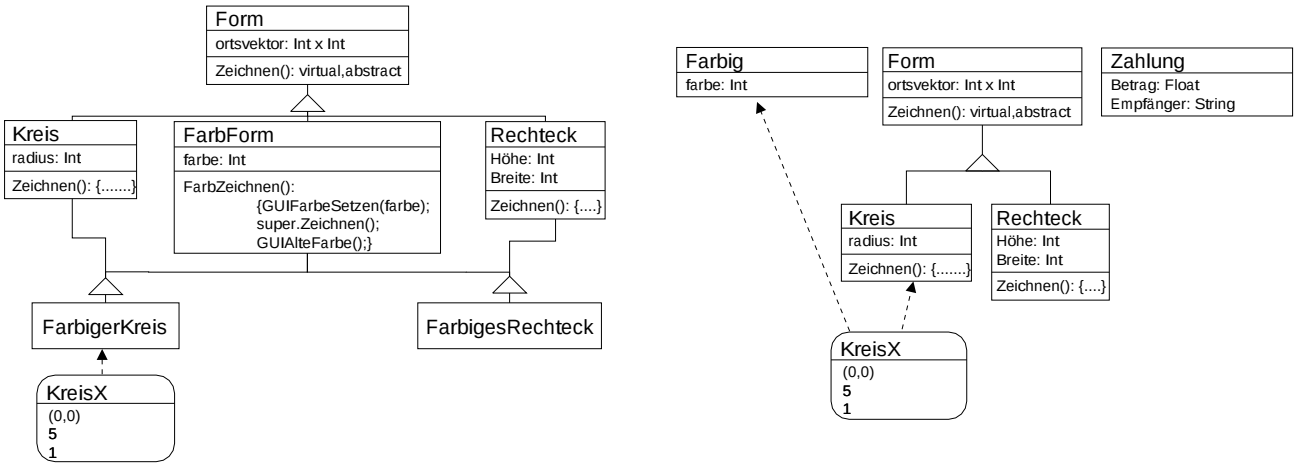


Abbildung 4.6: Unvollständige Modellierung des "Coloured–Shape"–Problems [Stein et al. 1989] mit multipler Vererbung und multipler Instanziierung

Multiple Instanziierung

Der Einsatz generischer Klassen [Meyer 1986; Eisenecker 1997] oder des 'abstract factory' Patterns aus [Gamma et al. 1996] führen zwar zu einer gewissen Erleichterung bei der Klassen- bzw. Instanzgenerierung, wirkliche Abhilfe von den Problemen der Multiplen Vererbung bringen sie aber nicht, da Vereinigungsklassen weiterhin benötigt werden. Erst durch **Verlagerung** der Kombination von der Klassen- **auf die Instanzebene**, d.h. bei der Verwendung multipler Instanziierung, treten Vereinigungsklassen und die mit Ihnen einhergehende Klassenexplosion nicht mehr auf, da die Kombination der Klasseneigenschaften 'just–in–time' in der Instanz stattfindet. In der Instanz kann

[11] In einer Klassenhierarchie von geometrischen Formen sollen nachträglich durch ein entsprechendes Attribut und Zeichenmethoden farbige Körper ermöglicht werden. Die Modifikation der Oberklasse "Form" wird explizit ausgeschlossen, um auch weiterhin schwarzweiße Körper darstellen zu können.

allerdings keine Änderung dieser Eigenschaften mehr herbeigeführt werden. Dies entspricht jedoch genau der Mächtigkeit, die sich bei der Multiplen Vererbung unter Verzicht auf den Modifikator M auch ergab. Dementsprechend läßt sich das "Coloured Shape"–Problem auch mit multipler Instanziierung allein nicht lösen. Abbildung 4.6 rechts zeigt, daß das Objekt `KreisX` die Klassen `Kreis` *und* `Farbig` instanziiert. Allerdings kann auch hier die `Zeichnen()`–Methode von `Kreis` nicht durch `Farbig` angepaßt werden. Da `Farbig` keine Verbindung zu `Kreis` bzw. `Form`, kann es die dort definierte Methode nicht benutzen, weil bei Multipler Instanziierung — wie bei Multiple Vererbung — der Modifikator $\triangle$M keine Rückbezug zu K hat. Würde `Farbig` eine eigene `Zeichnen()`–Methode definieren, so wäre bei einer Instanz von `Kreis` und `Farbig` unklar, welche der beiden ausgeführt werden soll.

Es stellt sich bei der die multiple Instanziierung repräsentierenden Gleichung $I = Inst\_of(K_1 \oplus K_2)$ also die Frage nach der Bedeutung von '$\oplus$' bei **Namenskonflikten**. Entsprechende Vorschläge in der Literatur, die sich mit multipler Instanziierung meist nur in Zusammenhang mit der Darstellung verschiedener Sichten auf Objekte beschäftigen, verbieten meist entsprechende Konflikte. In [Nguyen et al. 1992] werden beispielsweise unabhängige Klassen vorausgesetzt und [Stein 1991] sieht eine Instanz bei jedem Zugriff als zu nur einer Klasse gehörig an. Lediglich in [Marino et al. 1990] wird die Überlappung von Attributen verschiedener Perspektiven eines Konzepts[12] angedeutet, andererseits fordern sie aber wie in [Stein 1991] die unabhängige Bearbeitbarkeit jeder Perspektive eines Objekts, was Namenskonflikte an und für sich ausschließt.

Neben der Behebung von Namenskonflikten ergibt sich bei der Verwendung von Multipler Instanziierung das Problem der **Instanziierungsbeschränkung**. Bei unbeschränkter Verwendung, wie etwa in [Stein 1991] und [Nguyen et al. 1992] vorgeschlagen, kann ein Objekt Instanz beliebiger Klassen sein, selbst wenn diese konzeptionell ohne Gemeinsamkeiten sind oder sich in der Realität wechselseitig ausschließen. In Abbildung 4.6 rechts ergäbe es z.B. wenig Sinn, wenn `KreisX` noch zusätzlich die Klasse `Zahlung` instanziieren würde. Bei Multipler Vererbung werden die möglichen Instanziierungen implizit beschränkt, indem nur für die gewünschten Kombinationen Vereinigungsklassen in das Schema aufgenommen werden. Die Multiple Instanziierung verzichtet hingegen auf Vereinigungsklassen, so daß diese Möglichkeit hier nicht besteht und zusätzliche Mechanismen nötig sind. Zum einen können neue Konzepte eingeführt werden, um die Restriktionen auszudrücken, etwa die Perspektiven in [Marino et al. 1990], die eine Verschärfung der im übernächsten Abschnitt beschriebenen Rollen sind. Zum anderen kann die vorhandene Vererbungsrelation zwischen den Klassen genutzt werden, indem ihre Aufgaben erweitert werden.

So schlagen [Meyer 1997, Kap. 25.10] und [Fowler und Scott 1997, S.77ff] „View Inheritance" bzw. „Multiple Classification"[13] vor. Beide Verfahren erlauben es einer Klasse, ihre Objekte nach verschiedenen Kriterien zu partitionieren und über jeder Partition eine Subklassenhierarchie zu erstellen. Ursprünglich für die objektorientierte Analyse und Design von nach mehreren Kriterien spezialisierten Klassen konzipiert, lassen sich diese Verfahren auch zur Einschränkung der Multiplen Instanziierung verwenden, indem unabhängige Modifikatoren einer Klasse in verschiedene Unterklassenhierarchien plaziert werden, sich wechselseitig ausschließende Modifikatoren jedoch in der selben. Analog kann die in [Prieto-Diaz 1991] vorgestellte und ursprünglich aus der Bibliotheksverwaltung stammende „FacettedClassification" angewendet werden, die die Unterklassengraphen allerdings auf jeweils eine Ebene beschränkt (also eigentlich nur mehrere Listen sich gegenseitig ausschließender Klassen bereitstellt).

Alternativ können auch die in Abschnitt 4.1.4 vorgestellten Transitionsrestriktionen zum Einsatz kommen. Da jede Instanz zu mindestens einer Klasse gehören muß, können die zusätzlich instanzi-

[12] Jede Perspektive einer Klasse wird in diesem Modell durch einen eigenen Spezialisierungsbaum repräsentiert. Eine Instanz muß aus jedem dieser Bäume eine Klasse instanziieren. Besonders gekennzeichnete Klassen in den verschiedenen Spezialisierungsbäumen eines Konzepts enthalten in jeder Perspektive die selben Objekte und Attribute.

[13] Es sei darauf hingewiesen, daß „Multiple Klassifizierung" in der Literatur sowohl für das hier beschriebene Verfahren als auch generell für alle Verfahren der Multiplen Instanziierung verwendet wird. Ältere Texte benutzen es teilweise auch äquivalent zu Multipler Vererbung.

ierbaren Klassen aufgrund der von dieser Klasse ausgehenden Übergänge eingegrenzt werden.

Mixins

Mixin-Klassen [Booch 1994] und Mixin–basierte Vererbung [Bracha und Cook 1990], hier unter dem Begriff Mixins zusammengefaßt, treiben als Fortentwicklung der Multiplen Vererbung das Konzept der „Vererbung als inkrementeller Modifikation“ auf die Spitze, indem sie einige (bei Mixin-Klassen) oder alle (bei Mixin–basierter Vererbung) der bei der Vererbung verwendeten Modifikatoren explizit machen. Diese Modifikatoren, die Mixins, lassen sich an beliebiger Stelle der Klassenhierarchie zur Generierung einer neuen Klasse einsetzen, da der Rückbezug automatisch zum Anwendungszeitpunkt hergestellt wird (das K in $\triangle$M(K) wird also dynamisch und u.U. für jeden Rückbezug getrennt festgelegt). Mixins können das “Coloured Shape“ Problem (vgl. S. 54) ohne Code–Duplizierung lösen. Die linke Hälfte von Abbildung 4.7 zeigt, wie die Klassen `Kreis` und `Rechteck` durch den expliziten Modifikator `Farb-Mixin` zu ihren farbigen Versionen erweitert werden. Man beachte die Verwendung von `super.Zeichnen()` in `Farb-Mixin` ohne daß `Farb-Mixin` eine Oberklasse besäße. Es handelt sich bei `super` um den oben erwähnten dynamischen Rückbezug, der erst bei der Definition von `FarbigerKreis` bzw. `FarbigesRechteck` ausgewertet wird und dann auf `Kreis` respektive `Rechteck` verweist. Wie auch die Abbildung zeigt, reduzieren Mixins nur die Zahl der benötigten Modifikatoren. Da aber auch weiterhin Vereinigungsklassen benötigt werden, verringert sich die Klassenexplosion nur unwesentlich im Vergleich zur Multiplen Vererbung.

[Steyaert und Meuter 1995] definieren als Abwandlung der Mixin–Klassen Mixin–Methoden zum Einsatz in einem klassenlosen Modell. Objekte enthalten für jeden Mixin, der ihnen zugeordnet ist, eine Methode, durch deren Aufruf das Objekt entsprechend modifiziert wird. Allerdings erfordert dieser Ansatz, der gleichzeitig die Kapselung der Objekte erhalten soll, die Definition aller auf ein Objekt anwendbaren Mixin–Methoden in diesem selbst, so daß unvorhergesehene, nachträgliche Erweiterungen unmöglich werden.

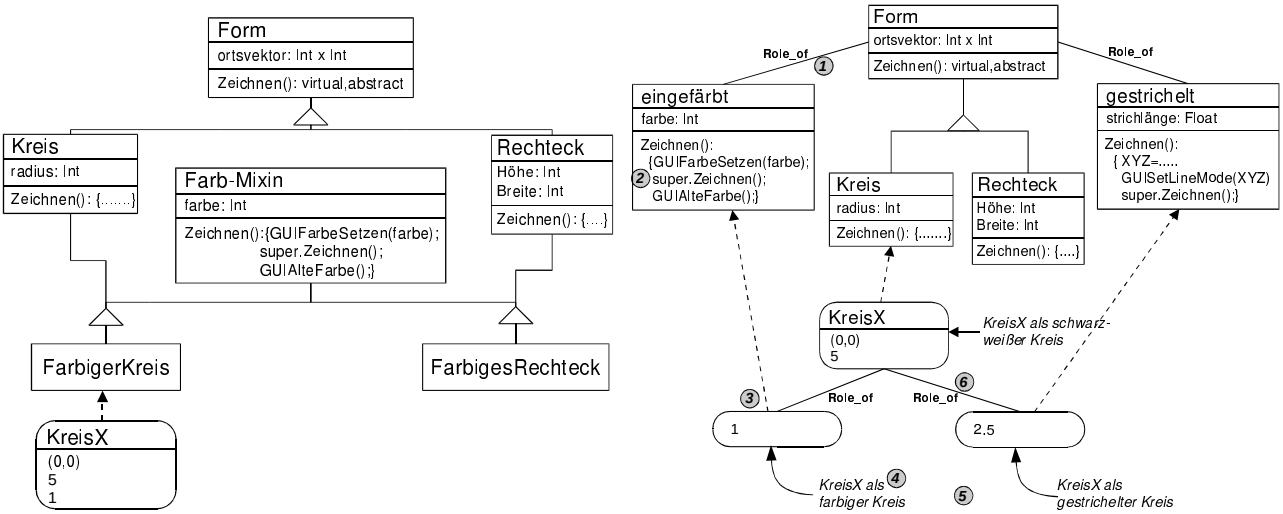


Abbildung 4.7: Modellierung des ’Coloured–Shape’–Problems mit Mixin–Klassen und mit Rollen

Rollen

Das Konzept der Rollen in der objektorientierten Modellierung kann man wie in [Wieringa et al. 1995; Kristensen 1995; Richardson und Schwarz 1991] aus der Erkenntnis motivieren, daß ein (persistentes) Objekt als Abbild einer Entität der realen Welt verschiedene, sich im Laufe seines Lebenszykluses ändernde und überlappende Rollen übernimmt („spielt“), die jeweils bestimmte Eigenschaften und

Fähigkeiten von ihm erfordern. So definiert besitzen Rollen zwar noch keine Semantik, aber eine Methodik für ihren Einsatz. Man kann Rollen aber auch aufgrund der Gemeinsamkeiten der zu ihrer Realisierung verwendeten Mechanismen (vgl. [Gottlob et al. 1996; Kristensen 1995]) sehen als einen mit Klassen verknüpften (1) klassenähnlichen Formalismus, der den Rückbezug (2) auf diese Klassen ermöglicht, und es erlaubt, bestehende Instanzen durch "Annehmen" der Rolle (3) dynamisch zu modifizieren. Dadurch können sich verschiedene erweiterte oder geänderte Ansichten (4) des Basisobjektes ("Rollenspieler" genannt) ergeben, die voneinander unabhängig und isoliert sind (5), aber von der Existenz des Basisobjektes abhängig sind (6). Bild 4.7 rechts zeigt diese Eigenschaften als grau hinterlegte Nummern. Die Verwendung von Rollen zur Lösung des 'Coloured–Shape'–Problems ergibt, wie man sieht, eine einfachere Klassenhierarchie als Mixins. Rollen werden, ganz im Sinne der Multiplen Instanziierung, dynamisch durch zusätzliche Klassenverweise bzw. Objektfragmente zu einem Objekt hinzugebunden. Im Gegensatz zur Multiplen Instanziierung kann ein Objekt jedoch immer nur unter einer von mehreren Rollen betrachtet werden. `KreisX` ist also ein farbiger und gestrichelter Kreis, wird aber, abhängig vom benutzen Verweis, nur als eins von beiden gezeichnet (Ausnahme: die im nächsten Abschnitt erwähnte Methodenaufrufkonvention in [Albano et al. 1995]).

In einer dritten Betrachtungsweise kann man Rollen also als ein Verfahren zur Instanziierungsbeschränkung der Multiplen Instanziierung (vgl. S. 55) bezeichnen, bei dem lediglich an einer bereits instanziierten Klasse anhängende Klassen zusätzlich instanziiert werden dürfen und das einen ungewöhnlichen Sichtenmechanismus auf die Instanzen bietet.

Ausprägungen des Rollenkonzepte in der Literatur Während obige Punkte die Gemeinsamkeiten der verschiedenen in der Literatur beschriebenen Ansätze darstellen, existieren auch diverse Unterschiede. Eine eigene **Objektidentität** der Rollenobjekte ist nur in [Wieringa et al. 1995] und [Gottlob et al. 1996] vorgesehen, bei ihnen findet der Rückbezug von der Rolle zum Objekt der Klasse (also die 'Vererbung' der Eigenschaften der Klasse) per Delegation statt. Eine eigene Objektidentität vereinfacht einerseits das Objektmodell, da Rollen mit den gleichen Mechanismen behandelt werden können wie Klassen. Andererseits müssen bei einigen Operationen wie dem Löschen und Vergleichen von Objekten Vorkehrungen getroffen werden, daß keine Inkonsistenzen entstehen. Alle anderen Ansätze unterscheiden darum zwischen Objektidentität, die bei allen Rolleninstanzen eines Objekts gleich ist, und Rollenverweisen, mit denen sich verschiedene Rollen unterscheiden lassen. Die Unterscheidung von Rollen ist notwendig, da außer in [Albano et al. 1995] Objekte **mehrmals die gleiche Rolle** annehmen können, z.B. könnte eine Person mehrmals die Rolle "Aufsichtsratmitglied" spielen. [Wieringa et al. 1995] erlaubt zusätzlich, analog zur Multiplen Klassifizierung (S. 55), mehrere, nach Thematik getrennte **Rollenpartitionierungen** einer Klasse (d.h. etwa daß eine Person aus der Partition Beruf nur eine Rolle annehmen darf), sowie die Angabe von Kardinalitätsrestriktionen für Rollen (etwa die Beschränkung der erlaubten Aufsichtsratsrollen auf zehn). In [Kristensen 1995] und [Albano et al. 1995] können Rollen in der Klasse vorhandene **Attribute überschreiben**, in [Richardson und Schwarz 1991] können sogar Teile der Klasse verborgen werden. Letzteres scheint nur auf den ersten Blick "Cancelation" zu ermöglichen. Da die verborgenen Klassenteile im Objekt weiterhin vorhanden und durch den Wechsel der Rolle wieder sichtbar werden, handelt es sich eher um einen Sichtenmechanismus. **Rückwirkungen** der Rollen **auf die Klasse** läßt als einziges [Albano et al. 1995] zu, indem der Methoden–Lookup des Basisobjektes in seinen Rollen beginnt (d.h. explizit an das Basisobjekt gerichtete Nachrichten werden möglicherweise von Methoden in einer Rolle bearbeitet). Hierdurch kann auch bestehenden Objekten Funktionalität hinzugefügt werden, auf die nicht nur per speziellem Rollenverweis zugegriffen werden kann. Allerdings wird dabei von mehrfach auftretende Methoden nur diejenige der zuletzt hinzugefügten Rolle ausgewertet, so daß die Funktionalität eines Objekts von der Reihenfolge der Rollenhinzunahmen abhängen kann. Bis auf [Richardson und Schwarz 1991] und [Stein und Zdonik 1989], die in diesem Punkt etwas unklar sind, ermöglichen alle Ansätze die Definition von **Rollen von Rollen**. Dabei wird eine Rolle nicht einer Klasse sondern einer anderen Rolle zugeordnet, und als Basisobjekt dient eine Rollenansicht. Dies kann notwendig sein, wenn etwa Personen der Rolle Angestellter in dieser Rolle weitere Rollen

besitzen können etwa Ersthelfer oder Betriebsratsmitglied. Eine besonders von Anwendungen in der realen Welt stammende Erweiterung nehmen [Wong et al. 1997] vor. Sie führen zusätzlich zu dem Rollen*spieler* den **Rollen*besitzer*** ein. Der Rollenbesitzer ist ein Objekt, ohne den eine Rolle eines anderen Objekts nicht existieren kann. Rollenbesitzer sind etwa Firmen (für Angestellte), Fachbereiche (für Dekane) oder Autos (für Fahrer). Rollenspieler für alle diese Rollen wäre aber eine Person. Rollenbesitzer können die ihnen 'gehörenden' Rollen initialisieren und sorgen dafür, daß diese Rollen die Lebenszeit ihres derzeitigen Spielers überdauern können. Zudem kann in Wongs Modell der Typ des Rollenspieler von Subrollen eingeschränkt werden, d.h. Subrollen können fordern, daß ihr Rollenspieler eine Subklasse des Rollenspielers der Oberrolle ist.

Schließlich sind die **Split Objects** in [Bardou und Dony 1996] eine Übertragung des Sichtenkonzepts der Rollen in eine prototypbasierte Sprache. Bardou und Dony gehen dabei das auf Seite 47 erwähnte Problem der Wertänderungen delegierter Attribute an, indem nur zwischen den verschiedenen Teilen eines aufgespaltenen Objekts echte Delegation stattfindet, während zwischen zwei Gesamtobjekte nur Default-Werte per Delegation übernommen werden können. Bei einer prototypbasiert als "Split Object" repräsentierten `PersonX` könnte also etwa das Teilobjekt `Employee-PersonX` vollständig nach `PersonX` delegieren, das Gesamtobjekt `PersonX` könnte von `PersonY` nur die Defaultwerte für seine Attribute per Delegation erhalten, würde also durch zukünftige Änderungen der Attributwerte in `PersonY` nicht beeinflußt.

Eine Übersicht über die Historie des Rollenkonzeptes und weitere den Rollen ähnliche Ansätze findet man in [Wieringa et al. 1995, S.70–72].

Rollen als Klassenmodifikatoren Alle Rollenkonzepte enthalten Ansätze zur expliziten Objektmigration (s.S. 61), da die zeitliche Änderung der Rollen eines Objekts wie erwähnt eine Motivation für das Rollenkonzept ist. Die Implementierung der Instanzmodifikationen erfolgt hierbei, soweit sie in den entsprechenden Artikeln beschrieben wird, meist durch aufeinander verweisende Teilobjekte (vgl. Abbildung 4.10). Migrationsrestriktionen ergeben sich zum einen aus der Anordnung der Rollen in der Klassenhierarchie, zum anderen sehen einige Ansätze zusätzlich erweiterte reguläre Ausdrücke über den Rollenamen [Kristensen 1995] respektive endliche Automaten mit Rollen als Zustände und Methoden als Transitionennamen vor [Wieringa et al. 1995].

Die einzelnen Rollen eines Objekts sind, wie oben erwähnt, wechselseitig isoliert. Dies bedeutet, daß Methodenaufrufe explizit an die jeweilige Rolle eines Objekts gerichtet werden müssen, in deren Kontext sie ausgeführt werden sollen, und keinen Zugriff auf Attribute oder Methoden benachbarter Rollen haben. Dies erleichtert zwar die Implementierung als Aufsatz auf klassische Objektmodelle, allerdings führt die **Isolation** zu einem Modellierungsproblem, das sich selbst an dem in der Literatur häufig zur Motivation verwendeten Person–Student–Angestellter–Beispiel zeigen läßt (vgl. Bild 4.8). In der Realität sind die Rollen Student und Angestellter nämlich keineswegs so unabhängig, wie es für

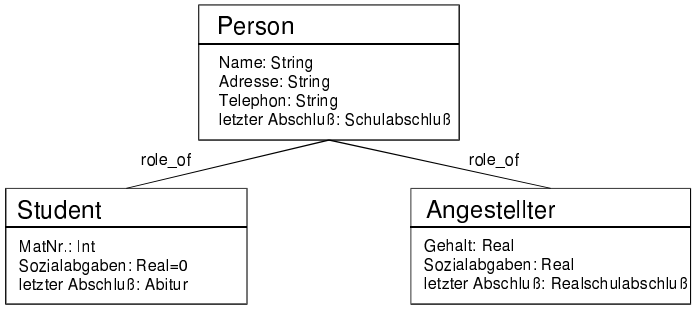


Abbildung 4.8: Student und Angestellter als Rollen von Person

ihre Modellierung als Rollen notwendig ist. So müssen für einen studentischer Angestellten keine Sozialabgaben gezahlt werden (zumindest bis Anfang 1997) und er muß mindestens das Abitur besitzen.

Wird nun ein als HiWi arbeitender Student als Person mit Rolle Student und mit Rolle Angestellter modelliert, so sind diese Einschränkungen bei der Betrachtung der Angestellten–Rolle nicht sichtbar und können somit bei Änderungen verletzt werden. Dies schränkt die Verwendungsmöglichkeit von Rollen zur Aufweichung der strikten Templatebeziehung in klassenbasierter Objektorientierung auf solche Instanzmodifikationen ein, die voneinander unabhängig sind. Für diesen eingeschränkten Fall kann dann die **Implementierung** mittels konventioneller objektorientierter Programmiersprachen wie C++ [Odberg 1995] oder SMALLTALK [Gottlob et al. 1996] einfach erreicht werden: In diesen Sprachen kann aufgrund ihrer Typ- und Methodenaufrufkonzepte die für Subklassen vorhandenen Mechanismen auch für die Definition von Rollen benutzt werden, solange keine wechselseitige Beeinflussung stattfindet.

[Kathuria 1997] kritisiert aus Sicht der objektorientierten Analyse (OOA) die Beschränkung der durch das Rollenkonzept induzierten Modellierungsmethode auf die Abbildung von "Rollen"beziehungen. Mit Hilfe der von ihm vorgestellten "**Assimilation**" und des damit möglichen "Property Modelling" können Klassen nach beliebigen Eigenschaften zerlegt werden, statt nur nach Rollen [Kathuria 1997, S.22]. Assimilation bezeichnet dabei die Vorgabe von Delegationsverweisen zwischen Klassen, wobei die Objekte, auf die ein Objekt delegiert, zusammen mit diesem neu erzeugt werden. Im "Colored Shape"–Beispiel würde die Klasse `Form` beispielsweise eine optionale Assimilationsbeziehung zu `Farbig` erhalten. Diejenigen Objekte die farbige Formen darstellen wurden diese Assimilationsbeziehung dann eingehen und ihre Schnittstelle würde dadurch automatisch um z.B. die Methode `FarbigZeichnen()` erweitert. Im Gegensatz zu Rollen existieren nur Verweise auf die Objekte, keine speziellen Assimilationsverweise. Mit dieser Semantik ähnelt die Assimilation der Komposition einer Klasse durch für den Benutzer unsichtbare Aggregationsverweise. Sie teilt auch die dort auftretenden Probleme bezüglich Methodensuche und Namenskonflikten, auf die Kathuria aber nicht weiter eingeht.

Bewertung klassenbasierter Objektmodelle

Ähnlich wie prototypbasierte Objektmodelle zeigen also auch klassenbasierte Modelle Schwächen, wenn auch in anderen Bereichen. Dem konventionellen Modell mangelt es an einfacher Modifizierbarkeit der Instanzen (der Stärke prototypbasierter Modelle) und es hat seine Stärken im Bereich der Ordnungshierarchie (der Schwäche prototypbasierter Modelle). Erweiterungen, die die Modifizierbarkeit zu verbessern suchen, führen zu einer mehr oder weniger stark eingeschränkten Verwendbarkeit der Ordnungshierarchie (Instanzmodifikationen), erhöhen die Zahl der benötigten Klassen erheblich (Multiple Vererbung, Mixin–Klassen), lassen Teilprobleme wie Namenskonflikte oder Instanziierungsbeschränkungen offen (Multiple Instanziierung, Assimiliation) oder definieren sie weg, was aber zu einer Einschränkung der Modellierungsmächtigkeit führt (Rollen). Es stellt sich somit die Frage, ob solche Probleme von Objektmodellen, die klassen- und prototypbasierte Konzepte mischen, umgangen werden können.

4.1.3 Mischung klassenbasierter und prototypbasierter Konzepte

Wirklich neue, aus der Kombination des klassen- und prototypbasierten Paradigma hervorgehende Objektmodelle sind selten (Kapitel 5.1 wird hierauf näher eingehen). Die meisten Sprachen gehen von einer der beiden Welten einen mehr oder weniger großen Schritt weit in Richtung der anderen. Die eine ausgewogene Mischung ergebende Schrittgröße scheint allerdings schwer zu treffen zu sein (vgl. Abbildung 4.3, die eine subjektive Einordnung der verschiedenen Sprachen gibt). Als Beispiele für Mischformen kann man zum einen die bereits in früheren Kapiteln beschriebenen Mixin–Methoden in [Steyaert und Meuter 1995] und die "Split Objects" in [Bardou und Dony 1996] erwähnen. Beide übertragen jeweils eine klassenbasierte Annäherung an die Flexibilität der prototypbasierten Welt zurück in eine prototypbasierte Sprache. Hiervon heben sich die Objektspezialisierung aus [Sciore

1989] und HYBRID aus [Stein 1987] allerdings nochmals ab, indem sie auf völlig neuen Konzepten beruhen, bzw. klassenbasierte und prototypbasierte Konzepte zu vereinen suchen.

Hybrid [Stein 1987, S.144ff][14] bietet die Vereinigungsmenge der im klassenbasierten und im ungetypten, prototyporientierten Paradigma vorkommenden Konzepte an. Objekte werden entweder aus Klassen instanziiert oder delegieren auf andere Objekte. Da die Klassen minimalen Templates entsprechen, können auch die durch Instanziierung gewonnene Objekte gegenüber der Klasse erweitert werden. Delegiert eine größere Zahl von Objekten auf ein Objekt, so kann dieses in eine Klasse umgewandelt werden. Als Ersatz wird ein neues Objekt erzeugt, das alle Attributwerte aufnimmt, die auch in die entsprechenden delegierenden Objekte kopiert werden. Nur die nicht von Oberklassen geerbten Attribut- und Methodendefinitionen bleiben in der Klasse zurück. Erweiterungen der vorhandenen Objekte und Klassen sind somit sehr einfach möglich. Das von Stein zur formalen Definition verwendete Modell läßt allerdings die Frage offen, wie sich die auf ein Objekt delegierenden Objekte effizient in Objekte umwandeln lassen, die die neu entstandene Klasse instanziieren. Zudem fehlen multiple Vererbung/Delegation, so daß auch Multiple Instanziierung niemals auftreten kann.

Sciores **Object Specialization** [Sciore 1989] verknüpft Delegation zwischen Objekten mit Restriktionen und Templates durch Klassen. Hierzu werden drei Hierarchien eingesetzt (vgl. Abbildung 4.9) In der Objekthierarchie delegieren Teilobjekte eines realen Objekts hierarchisch aufeinander.

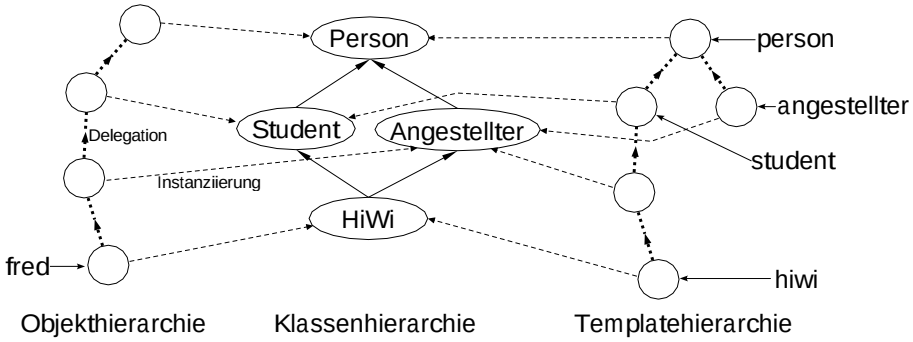


Abbildung 4.9: Die drei Hierarchien der “Object Specialization“ (nach [Sciore 1989])

Diese Art von Hierarchie findet sich auch bei den oben erwähnten “Split Objects“. Bei Sciore tragen allerdings die Objektteile lediglich Wertinformation. Attribute und Methoden werden in Klassen definiert, wobei jedes Objektteil einer Klasse zugeordnet wird. Jedes Objektteil verweist also zum einen auf dasjenige andere Objektteil, auf das es delegiert, zum anderen auf die Klassen, die seine Typinformation enthalten. Die Klassen werden wiederum in einer Klassenhierarchie angeordnet, die quasi einer namenskompatiblen Vererbungsbeziehung entspricht, da eine Subklassenbeziehung eine (indirekte) Delegationsbeziehung zwischen zugeordneten Objektteilen fordert. Die Templatehierarchie schließlich soll die Erzeugung neuer Objekte bzw. die Erweiterung vorhandener Objekte vereinfachen, indem sie sinnvolle Default–Objekte für die Klassen und ihre Kombinationen bereitstellt.

Objektspezialisierung macht also die Objekthierarchie — als Implementierungsmethode für Rollen und Multiple Instanziierung (vgl. Seite 63) — sowie die Klassenfunktionen Restriktion und Instanzgenerierung jeweils in einer eigenen Hierarchie explizit. Durch unterschiedliche Kombinationen dieser drei Hierarchien kann man viele der oben aufgeführten Objektrepräsentationen emulieren. Bei Verzicht auf Klassen- und Templatehierarchie ergibt sich eine getypte, prototypbasierte Sprache. Ohne Klassenhierarchie, aber mit einer Templatehierarchie, die für jede Klasse einen Eintrag enthält, kommt man der unbeschränkten Multiplen Instanziierung sehr nahe. Ist neben der Templatehierarchie auch die Klassenhierarchie vollständig und entsprechend gestaltet, so erhält man eine den Rollen entsprechende Semantik, wenn Objekterweiterungen nur an den durch die Klassenhierarchie

[14] Eine genauere Beschreibung befindet sich wohl in der von [Stein et al. 1989] zitierten aber nicht beschaffbaren Diplomarbeit [Mercado Jr. 1988].

vorgegebenen Stellen vorgenommen werden dürfen (d.h. die Subklassenbeziehung muß eine *direkte* Delegationsbeziehung implizieren). Sciore zeigt, wie sich Multiple Vererbung durch Linearisierung auf die singuläre Delegationshierarchie abbilden läßt [Sciore 1989, S.113f].

Ein Nachteil dieser Flexibilität ist natürlich die Notwendigkeit, das Zusammenspiel der drei Hierarchien verstehen zu müssen. Zudem muß für neuartige Objekte eine passende Klasse erzeugt werden, deren Generierung und Einordnung in die Klassenhierarchie von Sciores Ansatz nicht unterstützt wird. Auch das Problem der Namenskonflikte wird ähnlich wie beim rollenbasierten Ansatz aus [Albano et al. 1995] dadurch gelöst, daß sich eine der verschiedenen Definitionen durchsetzt, die anderen aber unsichtbar werden. Schließlich führt die hierarchische Anordnung der Objektteile zu langen Zugriffszeiten, wenn weit oben in der Hierarchie definierte Attribute über ein weit unten in der Hierarchie liegendes Objektteil angesprochen werden sollen.

Gemischte Objektmodelle verbinden also die Eigenschaften von prototyp- und klassenbasierten Systemen nur insoweit, als sie die Benutzung beider Paradigmen in einem Modell zulassen und, wie bei Hybrid, den Wechsel von Objekten zwischen ihnen erlauben. Jedes Objekt ist aber jeweils einem Paradigma, mit den entsprechenden Vor- und Nachteilen zugeordnet. Interessant sind diese Ansätze dann, wenn anfänglich unklare Konzepte im Laufe der Zeit modifiziert und schließlich, bei ausreichender Beispielzahl, fixiert werden sollen, um dann wieder Ausgangspunkt neuer, noch variabler Konzepte zu werden. Größere Änderungen an diesen prototypbasierten Objekten oder die Suche über ihnen werfen dann aber die altbekannten Probleme auf. Ein anderer Ansatz zur Erlangung von Modifizierbarkeit in klassenbasierten Systemen wird darum im folgenden Abschnitt beschrieben.

4.1.4 Objektmigration

Objektmigration ist der Übergang eines Objektes von einer Klasse in eine andere bzw. das Hinzufügen oder Entfernen von Instanziierungsbeziehungen zu/von einem Objekt[15]. Dynamische, also zur Laufzeit stattfindende, Objektmigration ist eine Voraussetzung dafür, daß in einem klassenbasierten System, also einem System mit strikten Templates, Objekte Struktur und Verhalten wie in prototypbasierten Systemen ändern können. Allerdings sind einige der hier besprochenen Punkte, insbesondere die Implementierung sich ändernder Objekte, auch für prototyp–basierte Systeme relevant.

Wir werden uns im folgenden mit vier Teilbereichen der Objektmigration beschäftigen: Wie wird die Migration angestoßen? Welche Beschränkungen gibt es? Wie werden Objekte konvertiert? Und auf welche Art kann man migrierende Objekte am besten repräsentieren?

Migrationsauslöser: Prinzipiell kann der Anstoß zur Migration eines Objekts auf zwei Arten erfolgen: Explizit vom Benutzer durch den Aufruf entsprechender Methoden von Objekt oder Klasse oder implizit durch das System, falls ein Objekt in der Klasse vorgegebene Bedingungen erfüllt. Ersteres besteht als alleinige Möglichkeit bei vielen Ansätzen, etwa dem Migrationsrahmenwerk aus [Li und Dong 1994], sowie diversen Rollenmechanismen (vgl. S. 56 f).

Die zweite Möglichkeit, also implizite Migration, erfordert eine durch das System auswertbare Angabe des Migrationszeitpunktes und des entsprechenden Ziels. Dies geschieht etwa durch mit Ausgangs- und Zielpunkt versehene Prädikate, bei deren Erfüllung die Migration stattfindet. Bei den "Predicate Classes"[16] aus [Chambers 1993] und den Transitionen der "Modes"–Klassen aus [Taivalsaari 1993] sind die Oberklassen einer Klasse der Ausgangspunkt und die Prädikatklasse bzw. der Modus der Zielpunkt. Die Klassen/Modi enthalten jeweils ein logisches

[15] Von Objektmigration ist in der Literatur auch die Rede, wenn Objekte in einem verteilten System von einem Rechenknoten zu einem anderen verlagert werden. Diese Arbeit verwendet Objektmigration aber nicht in dieser Bedeutung.

[16] Überaschenderweise definiert Chambers seine Prädikat*klassen* in Cecil, einer zwischen BOS und Hybrid anzusiedelnden, getypten prototypbasierten Sprache. Seine Definitionen werden in diesen Ausführungen in ein klassenbasiertes System zurückübertragen.

Prädikat über den Attributen und Methoden ihrer Oberklasse. Eine Instanz der Oberklassen wird automatisch auch Instanz der Prädikatsklasse bzw. des Modus, wenn sie das zugehörige Prädikat erfüllt. Gleiches gilt für KEA [Mugridge et al. 1991], das allerdings nur Prädikatsausdrücke über einem einzelnen Attribut zuläßt. Odbergs "Category Classes" [Odberg 1994] ermöglichen neben expliziten auch implizite, durch ein Prädikat ausgelöste Übergänge von beliebigen Klassen, indem die möglichen, mit Prädikat versehenen Übergänge in der Zielklasse spezifiziert werden (vgl. S. 93 und Abb. 5.4).

Migrationsbeschränkungen: Veränderungen an Objekten der realen Welt unterliegen meist biologischen, chemischen, physikalischen und anderen Beschränkungen. Für die Modellrepräsentationen dieser Objekte ergibt sich darum der Wunsch nach entsprechenden Migrationbeschränkungen. Hierfür existieren zwei Möglichkeiten. Zum einen lassen sich vorhandene Restriktionsmöglichkeiten von Klassen bzw. Attributen nutzen, um die möglichen Instanzen von Zielklassen zu beschränken (**Zustandsrestriktionen**). TROLLs Einschränkungen der Attributwertebereiche verhindern etwa die Aufnahme von gegen sie verstoßenden Objekten in eine Klasse [Hartmann et al. 1994, S.89]. Zustandsrestriktionen wirken hier also wie normale Constraints, mit der Ausnahme, daß ihre Verletzung nicht eine Inkonsistenz anzeigt sondern eine Migration auslöst (aus der Klasse hinaus) oder verhindert (in die Klasse hinein). Auch die "essential" und "exclusionary types" in [Zdonik 1990] stellen in gewissem Sinn Zustandsbeschränkungen dar, da ein Objekt, das einen Zustand (essential type) angenommen hat, diesen nicht mehr verlassen kann, bzw. ein solches, das einen Zustand (exclusionary type) verlassen hat, diesen nicht mehr annehmen kann.

Meistens werden jedoch die *Übergänge* zwischen den einzelnen Klassen eingeschränkt (**Transitionsrestriktionen**). Am deutlichsten wird dies in [Li und Dong 1994], bei denen die möglichen Übergänge explizit in einer eigenen Relation modelliert werden. Einschränkungen beziehen sich hier aber nur auf die Art der erlaubten Migrationsoperation (Klassenwechsel, Klassenhinzugewinnung) zwischen jeweils zwei Klassen, ohne Attributwerte oder andere Klassenzugehörigkeiten zu berücksichtigen. Odbergs "Category Classes" [Odberg 1994] spezifizieren in der Zielklasse die möglichen Ausgangsklassen einer Migration[17] sowie die über der jeweiligen Ausgangsklasse zu erfüllenden Prädikate. In [Su 1991] werden die erlaubten Klassenkombinationen als "role sets" bezeichnet und Abfolgen dieser zu "migration patterns" zusammengefaßt, die somit verschiedene Pfade durch die erlaubten Zustände eines Objekts darstellen. Es wird dann gezeigt, daß sich die Pattern–Mengen (und damit indirekt die erlaubten Migrationen) durch reguläre Ausdrücke beschreiben lassen, wenn es nur die fünf unbedingten Migrationsoperatoren "create", "delete", "modify", "generalize" und "specialize" gibt. Einen ähnlichen Weg geht [Wieringa et al. 1995], das die möglichen Übergänge zwischen Klassen als Transitionen eines endlichen Automaten, hier "migration diagram" genannt, erfaßt. Da die Klassen den Zustände des Automaten entsprechen[18], können an den Transitionen die jeweils für den Übergang verantwortlichen Methoden aufgeführt werden. Strukturell lassen sich die Klassenübergänge begrenzen, wenn man wie bei den Rollen oder Chambers "Predicate Classes" nur Übergänge von Ober- zu Unterklasse und umgekehrt zuläßt.

Aufgrund von Zustandsrestriktionen lassen sich in jedem Fall Zusicherungen über die Objekte einer Klasse ableiten. Zwar gilt dies auch für Transitionsrestriktionen, doch hier sind die Zusicherungen nicht immer offensichtlich, da es möglich sein kann, eine Restriktion für den Wechsel von A nach B durch vorherigen Wechsel nach C zu umgehen. Daß dennoch hauptsächlich Transitionsbeschränkungen zum Einsatz kommen, dürfte daran liegen, daß nur so Attribute der Ausgangsklassen in den Restriktionsprädikaten auftauchen können und nur so der sich der

[17] Genauer: einer Klassenhinzugewinnung; Da in [Odberg 1994] nur Mechanismen zur Hinzunahme von Instanziierungsbeziehungen vorgestellt werden, muß ein Wechsel durch das nachfolgende explizite Löschen der Instanziierungsbeziehung zur alten Klasse vorgenommen werden.

[18] und nicht, wie bei der Konvertierung von Sus Ansatz in endliche Automaten, den Transitionen.

aus den erlaubten Transitionen ergebende Graph einfach mit den erlaubten Übergängen der realen Objekte vergleichen läßt.

Objektkonversion: Der Übergang in eine neue Klasse führt fast immer auch zu neuen, geänderten oder gelöschten Attributen. Verfahren der Schemamodifikation bieten unter gleichen Umständen benutzerdefinierte Methoden zur Objektumwandlung an [Ferrandina et al. 1994; Monk und Sommerville 1993]. Objektmigrationsansätze vertrauen hingegen meist darauf, daß der Benutzer migrierte Objekte nachbearbeitet bzw. initialisiert [Hartmann et al. 1994; Li und Dong 1994] oder mit vorgegebenen Initialisierungsroutinen/-werten [Chambers 1993; Odberg 1994] zufrieden ist. Allerdings lassen sich transitonsbeschränkende Verfahren leicht um Konversionsmethoden erweitern. Besonders einfach fiele dies bei dem Vorschlag aus [Li und Dong 1994], wo lediglich die bestehende Notation der Transitionsvorgaben vom Dreitupel <start,ziel,art> zu <start,ziel,art,konvertierung> erweitert werden müßte.

Objektrepräsentation: Nicht alle vorgestellten Konzepte machen Angaben, wie Objekte intern dargestellt werden sollen. Migriert ein Objekt nämlich in eine neue Klasse bzw. erhält es eine zusätzliche Klasse, so muß im Normalfall die Speicherrepräsentation des Objekts angepaßt werden, da Attribute hinzugekommen oder weggefallen sind. Dazu kann natürlich das alte Objekt in ein zu der neuen Klasse passendes kopiert und anschließend das alte gelöscht werden. Hierbei ergeben sich nicht nur Performance–Nachteile, auch die Beibehaltung der Objektidentität bzw. die Verhinderung von ins Leere zeigenden Verweisen wird zum Problem [Li und Dong 1994; Zdonik 1990]. Hierfür existieren mehrere Lösungsansätze, die anhand des Beispiels von Bild 4.10 erläutert werden sollen:

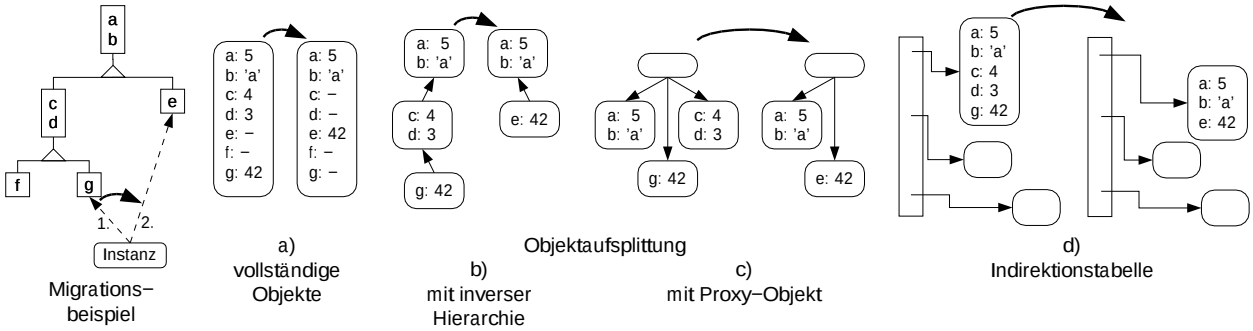


Abbildung 4.10: Verschiedene Repräsentationsarten der links dargestellten Objektmigration

Bei der einfachsten Lösung [Chambers 1993] umfaßt die Struktur eines Objekts von vorneherein sämtliche, in allen möglichen Varianten vorkommenden Attribute (s. Abbildung 4.10 a). Der Speicherbedarf dieser Vorgehensweise wird jedoch zu groß, wenn die Unterschiede zwischen den Varianten oder die Zahl der Klassenkombinationen, die ein Objekt annehmen kann, zunehmen. In diesem Fall werden Objekte häufig mit Hilfe von den jeweiligen Klassen entsprechenden Teilobjekten gespeichert. Existiert eine Ordnung auf den Objektteilen, wie etwa bei Rollen (S. 56) oder der Objektspezialisierung (S. 60), so kann man die Teile hierarchisch aufeinander und schließlich auf das Identität tragende Objekt verweisen lassen (vgl. Bild 4.10 b) [Sciore 1989; Gottlob et al. 1996; Li und Dong 1994; Bardou und Dony 1996]. Methodenaufrufe müssen an spezielle Teilobjekte gerichtet werden und werden wenn nötig delegationsartig die Hierarchie hinauf weitergeleitet. Umgekehrt kann auch ein Proxy–Objekt existieren, das auf alle Teile verweist und die Methodenaufrufe entsprechend weiterleitet [Odell 1992; Davis 1992]. Wegen des erhöhten Aufwands beim Methodenaufruf wird diese Lösung nur selten verwendet.

Will man trotz der vorhandenen Nachteile die Aufteilung der Objekte verhindern, so müssen Mechanismen genutzt werden, die es erlauben, alle Referenzen vom alten auf das neue Objekt umzuleiten. [Davis 1992, S.29] schlägt dazu Indirektionstabellen (s. Abbildung 4.10 d)

oder Rückverweise auf alle referenzierenden Objekte vor, um die dort vorhandenen Referenzen anpassen zu können.

Bei der Verwendung der Objektmigration zur Aufweichung der strikten Templates in klassenbasierten Objektmodellen müssen für die vorgestellten Bereiche Lösungen gewählt werden. Bezüglich der Konversion kann davon ausgegangen werden, daß der Benutzer das Objekt anpaßt (damit er das Objekt modifizieren kann, wird schliesslich die Migration ausgeführt). Interessanter erscheinen die Bereiche der Migrationsbeschränkungen, die sich mit den bei multipler Instanziierung erwähnten Instanziierungsrestriktionen überschneiden sowie die Frage einer effizienten Repräsentation.

4.1.5 Objektmodelle, eine Übersicht

Das Kapitel hat bisher die große Zahl verschiedener, im Spannungsfeld zwischen klassen- und prototypbasierter Objektorientierung liegender Ansätze aufgezeigt. Es stellt sich heraus, daß offensichtlich ein Zielkonflikt zwischen der Instanzanpaßbarkeit, der Typisierbarkeit und der Effizienz der Instanzspeicherung und -generierung eines Objektmodells besteht. Dieser Zielkonflikt kann hauptsächlich auf die Verwendung strikter vs. loser Templates zurückgeführt werden. Dies zeigt sich auch in Tabelle 4.4, die die besprochenen Objektmodelle anhand der am Anfang dieses Kapitels definierten Vergleichskriterien zusammenfaßt.

Die von den **prototypbasierten** Modellen verwendeten losen Templates erlauben es beliebige Änderungen an Objekten quasi nebenher vorzunehmen, verhindern aber auch einfache Typisierungskonzepte. Erlaubt zudem die dynamische Empathy (also die Delegation) das Löschen von Attributen ("Cancelation"), so erschwert sich auch der Aufbau einer Ordnungshierarchie. Bei namens- oder gar signaturkompatibler Delegation zwingen andererseits Objektänderungen möglicherweise zur Suche nach einem neuen Delegationspartner. Jede dieser Alternativen hat im betrachteten Anwendungsgebiet Nachteile: Ohne Attributtypisierung fällt die frühzeitige Erkennung von Modellierungsfehlern schwerer, da Attribute, die auf ein Objekt der falschen Klasse zeigen, zur Laufzeit (bei der Prozeßmodellierung also etwa bei der Generierung der Gleichungen) entdeckt werden. Ohne Ordnungshierarchie kann eine Übersicht über vorhandene Konzepte nur durch zusätzlichen Aufwand (z.B. Bausteinpaletten oder Suchverfahren) sichergestellt werden. Die bei namens- oder signaturkompatibler Delegation notwendige Suche nach einem neuen, passenden Delegationspartner kann erfordern, bei lokalen Editieroperationen einen globalen Überblick über vorhandene Objekte zu besitzen.

Klassenbasierte Sprachen erhalten durch die strikte Template–Funktion der Klassen in Verbindung mit der Signatur–Kompatibilität der Vererbungsrelation Ordnungshierarchie und abstrakte Templates quasi geschenkt. Eine Attributtypisierung ist durch sie einfach möglich, wenn auch Sprachen wie Smalltalk darauf verzichten. Um die in klassenbasierten Systemen fehlende Flexibilität der Objekte zu erhalten, behalten verschiedene Erweiterungen zwar die Striktheit der Templates bei, erlauben es einem Objekt jedoch, die Templatebeziehung während seiner Existenz zu ändern. Die Unterscheidung aus [Stein et al. 1989] in "anticipated" und "unanticipated sharing" von Eigenschaften wird durch Änderbarkeit der eigentlich "anticipated sharing" ausdrückenden Instanziierungsbeziehung sozusagen erweitert um das *pre-planed unanticipated sharing*. Zwar können Objekte durch diese Verfahren (wie etwa Multiple Instanziierung, Rollen oder Objektmigration) diverse, auch unvorhergesehene, Eigenschaftskombinationen erben, die Variationsmöglichkeiten werden jedoch durch die verschiedenen Restriktionen gemäß den Wünschen des Modellierers vorhersagbar eingeschränkt. Von den vorgeschlagenen Restriktionen sticht besonders die wenig genutzte Methode ins Auge, eine Klasse in mehrere orthogonale Partitionen mit disjunkten Subklassen aufzuteilen. Nicht zufriedenstellend gelöste Probleme stellen die Behandlung von Namenskonflikten mehrerer gleichzeitig verwendeter Templates dar, sowie die teilweise hohe Komplexität des entstehenden Formalismus, insbesondere einiger Restriktionsarten.

Über die verschiedenen **Mischformen** klassen- und prototypbasierter Objektorientierung läßt sich nur schlecht ein generelles Urteil fällen. Sciores "Object Specialization" und in schwächeren Ausmaß auch Hybrid ermöglichen verschiedenste Verwendungen ihrer Objekte, sind aber nicht einfach

Ansatz	Empathy: Wann	Empathy: Wie	Empathy: Empfänger	Empathy: -einschränkung[a]	Templates: Was	Templates: Wie	Ordnungshierarchie	Typisierung
prototypbasiert								
DELEGATION	dynamisch	ex- und implizit	Objekt	–	Objekt	lose, statisch ggf. Cancelation	–	nein
SELF (traits)	dynamisch	implizit	Objekt	–	Objekt	lose,statisch	–	nein
OMEGA	statisch	implizit	Typ	–	Objekt	strikt,kann selber neues Template werden	Typhierarchie	ja
BOS	dynamisch	ex- und implizit	Objekt	–	Objekt	lose, nur erweiterbar	Delegationslinks begren. Wirksamkeit	jain
klassenbasiert	statisch	implizit	Typ	1	Klasse	strikt, statisch	Klassenhierarchie	mögl.
multiple Vererbung	statisch	implizit	Typ	–	Klasse	strikt, statisch	Klassenhierarchie	mögl.
multiple Instanziierung	statisch änderbar	implizit	Typ	2	Klasse	strikt, dynamisch	Klassenhierarchie mehrfache Einordnung	mögl.
Mixins	statisch	implizit	Typ	–	Klasse	strikt, statisch	Klassenhierarchie	mögl.
KSL	statisch	implizit	Typ	–	Klasse	lose, statisch	Klassenhierarchie begren. Wirksamkeit	jain
Rollen	statisch, änderbar	implizit	Typ	3	Klasse	strikt, dynamisch	Klassen-/ Rollenhierarchie	mögl.
Objektmigration	statisch, änderbar	implizit	Typ	diverse s.S.61 f	Klasse	strikt, dynamisch	Klassenhierarchie	mögl.
Mischformen								
Objektspezialisierung	dynamisch	implizit	Objekt	4	Objekt	lose, nur erweiterbar	Klassenhierarchie, ggf. unvollständig	ja
HYBRID	statisch[b]	ex- und implizit	Typ und Objekt	–	Objekt, Klasse	lose, nur erweiterbar	Klassenhierarchie, unvollständig	nein
Split Objects	statisch	implizit	Objekt	–	Objekt	lose	nur objektintern	nein

Tabelle 4.4: Vergleich verschiedener Objektmodelle

[a]Empathyrestriktionen: 1=nur ein Empathy–Geber erlaubt; 2=Änderungen nur in Einklang mit Templateänderungen; 3=strukturelle Restriktionen; 4=durch die Klassenhierarchie

[b]dynamisch laut [Stein et al. 1989]

verständlich (Objektspezialisierung) und nur ineffizient zu implementieren (beide). “Split Objects“ haben analoge Eigenschaften und Kritikpunkte wie Rollen, zudem bleibt die Methodensuchstrategie unklar, wenn ein Split Object als ganzes betrachtet werden soll.

Bevor die für verschiedene verfahrenstechnische Modellierungsaufgaben genutzten Objektmodelle betrachten werden, untersucht der folgende Abschnitt Operationen für Suche und Einordnung von Klassen und Instanzen. Solche Operationen können ggf. Schwächen ausgleichen, die insbesondere prototypbasierte Modelle aufgrund der fehlenden Ordnungsrelation in diesen Bereichen haben.

4.2 Operationen für Suche und Einordnung

Kapitel 3.3.2 führte als für die Wiederverwendung von Modellen wichtige Operationen die Suche nach zu einer Aufgabenstellung passenden Datenbankeinträgen sowie die Einordnung neu erzeugter Modelle in die Datenhierarchie auf. Diese Verfahren sind teilweise unabhängig vom verwendeten Objektmodell und fanden darum in den vorhergehenden Abschnitten keine Beachtung. Dies soll nun nachgeholt werden.

Da, wie in Kapitel 3.3.2 dargelegt, auf Anfragesprachen beruhende Suchverfahren nur begrenzt den Anforderungen genügen, geht Abschnitt 4.2.1 kurz auf die beispielgetriebene Suche und Variationen der hierbei häufig zur Anwendung kommenden Ähnlichkeitsfunktion ein, zumal diese auch für die Einordnung von Objekten in eine Hierarchie verwendet werden können. Abschnitt 4.2.2 geht dann auf verschiedene Ansätze zur Klassifikation von Objekten und zur (Re-)Organisation der Klassenhierarchie ein, bevor 4.2.3 das “Case–Based Reasoning“ als spezielle Wiederverwendungsform mit seinen Such- und Einordnungsmethoden beschreibt.

4.2.1 Beispielbasierte Suche und Ähnlichkeitsfunktion

Beispielgetriebene Suche kann in DBMS mit festem, vorgegebenem Schema und atomaren Typen zum Einsatz kommen, indem, wie auf S.40 beschrieben, Beispiele in Anfragen über die Datenbasis umgewandelt werden. Wird ein durch den Benutzer änderbares Schema oder dynamische Datentypen verwendet, so können die Attributwerte des vorgegebenen Beispiels nicht einfach per Namensgleichheit bijektiv auf Attribute der verfügbaren Vergleichswerte abgebildet und verglichen werden. Im besonderen Maße tritt dies natürlich bei prototypbasierten Objektmodellen und denjenigen klassenbasierten Modellen auf, die Abweichungen der Instanz von der Klasse erlauben.

Ein Ansatz in diesem Fall besteht darin, die Unterschiede und Gemeinsamkeiten der zu vergleichenden Objekte zu einem Zahlenwert zusammenzufassen, der **Ähnlichkeitsfunktion**. Die Ähnlichkeitsfunktion $AF : O \times O \rightarrow]0, 1]$ bildet zwei Objekte $o_i \in O$ auf ein Zahl zwischen 0 und 1 ab, so daß gilt[19]

$$\begin{aligned} AF(o_1, o_2) = 1 &\quad\Leftrightarrow\quad o_1 = o_2 \\ AF(o_1, o_2) < AF(o_1, o_3) &\quad\Leftrightarrow\quad o_1 \text{ und } o_3 \text{ sind einander 'ähnlicher' als } o_1 \text{ und } o_2 \ (\textit{Monotonie}) \end{aligned}$$

Ähnlichkeitsfunktionen beruhen meist darauf, daß die Zahl gleicher Eigenschaften zweier Objekte ins Verhältnis zur Zahl aller oder der ungleichen Eigenschaften gesetzt wird [van Rijsbergen 1979]. Rijsbergen zählt verschiedene Koeffizienten hierfür auf. Im einfachsten Fall, der allerdings die Null als Ähnlichkeitswert zuläßt, ergibt sich:

$$2\frac{|X \cap Y|}{|X| + |Y|} \qquad \textit{(Dices Koeffizient)}$$

[19] Die Definitionen in der Literatur unterscheiden sich stark: [Maarek et al. 1991] fordert Symmetrie statt Monotonie, [Schwanke 1991] definiert die Monotonie formaler mittels Mengen gemeinsamer und unterschiedlicher Eigenschaften und [van Rijsbergen 1979] leitet die Ähnlichkeitsfunktion als Kehrwert eines, häufig metrischen, *Un*ähnlichkeitsmaßes ab: $AF(x) = (1 + UF(x))^{-1}$.

d.h. die Kardinalität der Schnittmenge der Eigenschaftsmengen X und Y von o_1 und o_2 wird durch die Gesamtzahl der Eigenschaften dividiert. Wären o_1 und o_2 also bezüglich der betrachteten Eigenschaften völlig ungleich, so ergibt sich 0, sind sie völl ig gleich, so ergibt sich 1. Die durch Dices Koeffizienten und andere Koeffizienten betrachteten Eigenschaftsmengen X und Y können die Objektidentität, die Klassenzugehörigkeit, die Oberklassenbeziehungen, die Attribute, Methodenaufrufe oder das dynamische Verhalten der Objekte enthalten [Spanoudakis 1994; Castano et al. 1994; Schwanke 1991]. Bei Attributnamen oder ähnlichen mehrdeutigen Eigenschaften kann durch einen Thesaurus oder die Erzwingung eines gemeinsamen Vokabulars [De Antonellis und Zonta 1990] die Synonymproblematik und durch eine Gewichtung nach Informationsgehalt die Gefahr zufälliger Übereinstimmung [Schwanke 1991; Maarek et al. 1991] teilweise gelöst werden. [Spanoudakis 1994, S.46ff, S.111ff] bewertet die **Wichtigkeit von Attributen** für eine Unterscheidung anhand der Kriterien "characticity", "abstractness" und "determinance". Characticity gibt an, inwieweit ein Attribut verschiedene Werte/Typen in unterschiedlichen Ähnlichkeitsmengen annimmt. Abstractness gibt an, inwieweit die Existenz eines Attributs entscheidend für das es besitzende Objekt ist. Und Determinance bestimmt in welchem Grade Werte/Typen andere Attribute von dem betrachteten abhängen[20].

Gruppiert man ähnliche Objekte zu Clustern zusammen und ordnet diese hierarchisch wie etwa in [Maarek et al. 1991; Wegner 1987], so kann man diese Cluster als Klassenextensionen innerhalb einer Extensionshierarchie auffassen und somit mittels eines Ähnlichkeitsmaß die Objekte bzw. Klassen in eine Hierarchie einordnen. Ob allerdings auch Vererbungs- oder andere Beziehungen zwischen den so entstandenen Klassen gelten, hängt von der verwendeten Ähnlichkeitsfunktion ab (vgl. S. 72).

4.2.2 Klassifikation und Klasseneinordnung

Im Verlauf einer Modellierung neu entstandene Modelle und Modellbausteine müssen — sollen die Suchverfahren effizient arbeiten können bzw. soll der Benutzer einfach stöbern können — in Ordnungsstrukturen eingebunden werden. Dies kann zum einen durch Materialisierung der wechselseitigen Beziehungen wie der oben erwähnten Ähnlichkeitsfunktion oder `ARE_ALIKE` und `ARE_THE_SAME` in ASCEND (s. S. 75) geschehen. Wir betrachten hier jedoch die Einordnung in die in Kapitel 2.3.2 und diesem Kapitel definierten, hierarchischen, durch das Auftreten gleicher Eigenschaften induzierten Strukturierungsbeziehungen: die Spezialisierungs-, die Vererbungs- und die Delegationshierarchie. Da dieses Kapitel auch klassenlose Objektmodelle vorstellte, müssen die Einordnungsaufgaben unterschieden werden in:

Klassen(re)organisation: Einordnung einer neuen Klasse (die etwa im Rahmen der Objektmigration oder Objektklassifikation erzeugt wurde) in die Klassenhierarchie

Objektklassifikation: Zuordnung eines Objekts zu einer Klasse, bzw. Erzeugung einer zu einem Objekt passenden Klasse

Delegations(re)organisation: Einordnung eines neu erzeugten oder geänderten Objekts in die Delegationshierarchie

Überaschenderweise finden sich in der Literatur aus dem Bereich der Objektorientierung zwar zahlreiche Vorschläge zur automatischen (Re-)Organisation von Klassenhierarchien, aber bereits die Objektklassifikation wird nur in wenigen Artikeln behandelt und auf die automatische Delegationsorganisation geht keine der gefundenen Arbeiten ein. Im folgenden werden darum auch Verfahren aus anderen Bereichen vorgestellt.

[20] Die Zahlenwerte der ersten beiden Kriterien werden dabei durch folgende Formeln bestimmt: $Charac_a = \frac{|\text{Wertebereiche von } a|}{|\text{Klassen, die } a \text{ enthalten}|}$ und $Abstract_a = \frac{|\text{Direkte Instanzen von } c|}{|\text{Alle Instanzen von } c|}$, wobei c die Klasse ist, die a erstmals definiert. Ähnlich, aber weitaus komplexer, kann auch ein Zahlenwert für die Determinance gewonnen werden. Diese Zahlen gehen dann normiert als Gewichtung in die Berechnung des Abstands zweier Attribute und damit zweier Objekte ein.

Klassen(re)organisation

Mit dem Begriff Klassenorganisation bezeichnen wir im folgenden sowohl das Einfügen einer neuen Klasse in eine Klassenhierarchie als auch die eng damit zusammenhängende Reorganisation einer Klassenhierarchie. Eingabeparameter sind also eine Klassenhierarchie sowie ggf. eine neue Klasse. Als Ergebnis wird eine Klassenhierarchie gewünscht, die aufgrund mehr oder weniger formal definierter Eigenschaften optimal ist sowie ggf. die eingefügte Klasse enthält. Die meisten Verfahren hierfür beruhen darauf, gemeinsame Eigenschaften (meist Attribute und Methoden) mehrerer Klasse zu erkennen und in einer neuen Klasse zusammenzufassen (*Faktorisieren*), zu deren Subklassen dann die ursprünglichen Klassen werden [Dicky et al. 1996, S.252]. Die Algorithmen unterscheiden sich jedoch in den berücksichtigten objektorientierter Features wie Methodenüberladung und -redefinition oder Attributspezialisierung.

Die Einordnung einer Klasse in die Klassenhierarchie ist in ihrer allgemeinen Form unentscheidbar, da dazu unter Umständen Methodenverhalten verglichen werden müßte [Rundensteiner 1992, S.27]. Alle Verfahren benutzen darum **Einschränkungen** bezüglich der beachteten Beziehungen zwischen den Klassen. In [Casais 1992] und [Rundensteiner 1992] werden Klassen als Mengen **atomarer Attributdefinitionen** angesehen, d.h. Attribute werden als unteilbare Einheit aus Name und Typ bzw. Implementierung gesehen. Attributspezialisierung oder Methodenzerlegung sind darum bei ihnen nicht möglich. Dies erlaubt verhältnismäßig einfache Verfahren. Zum einen werden durch lineare Dekomposition der Subklassenbeziehung mehrere gleichzeitige Spezialisierungsschritte (bei Casais) bzw. die gleichzeitige Extensions- und Typverfeinerung (bei Rundensteiner) durch Einführen von Zwischenklassen in ihre Einzelschritte getrennt. Zum anderen wird faktorisiert, um Superklassen für eine schlecht einordbare Klasse (bei Casais auch wegen Attribut–Cancelation) zu schaffen. Bild 4.11 zeigt ein Beispiel, wobei die grauen Klassen vom Verfahren automatisch erzeugt werden.

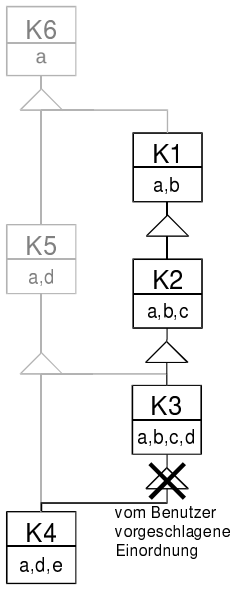


Abbildung 4.11: Einfügen der Klasse K4 (nach [Casais 1992])

Rundensteiners Algorithmus arbeitet dabei global, d.h. er untersucht alle Klassen der Hierarchie auf Gemeinsamkeiten mit der einzufügenden Klasse und bildet entsprechende Äquivalenzklassen. Er besitzt darum eine quadratische Zeitkomplexität bezüglich der Zahl der Subklassenbeziehungen in der Klassenhierarchie [Rundensteiner 1992, S.50]. Zudem besteht lineare Komplexität zur Anzahl der Attribute der Klassen, da diese bei einem Vergleich zweier Klassen betrachtet werden müssen. Casais Verfahren erwartet dagegen als Eingabe eine ungefähre Vorgabe für die Position der einzuordnenden Klassen (die durchgestrichene Subklassenrelation in Bild 4.11) und beginnt von dort aus mit der lokalen Reorganisation der Klassenhierarchie. Dadurch müssen weniger Klassen betrachtet werden, so daß eine niedrigere Komplexität zu erwarten ist (der Autor macht hierzu keine Aussagen). Dafür kann aber nicht wie bei Rundensteiner garantiert werden, daß jedes Attribut nur in einer Klasse neu eingeführt wird. Da Rundensteiner ihr Verfahren zur Einordnung von Sichten in die Klassenhierarchie entworfen hat, zeigt sie, daß der verwendete Algorithmus bei einer entsprechende Wahl des Subklassentests sowohl die Typ- als auch die Extensionsspezialisierung in der erweiterten Hierarchie sicherstellt.

[Lieberherr et al. 1991] verwendet eine ähnliche Vorgehensweise zur Reorganisation der gesamten Klassenhierarchie, unterscheidet aber zwischen Typ und Namen eines Attributs. Da auch er **keine Attributspezialisierung** zuläßt, ändert sich der Aufwand für den Test auf Attributgleichheit nur unwesentlich. Daß Lieberherrs Verfahren NP–vollständig ist, liegt vielmehr an seinem Minimalitätsziel: Gleichzeitig die Zahl der Attributdefinitionen und der Spezialisierungskanten minimieren. Die Attributdefinitionen verringert Liebermann, indem er, wie Rundensteiner, gleiche Attributdefinitionen in jeweils einer Klasse bündelt, so daß jedes Attribut nur einmal definiert wird. Hierbei

erzeugt der Algorithmus allerdings Mixin–Klassen und erhöht so die Zahl der Spezialisierungskanten. Diese können wegen der Komplexität der optimalen Lösung nur durch eine Heuristik in den Klassengraphen eingeordnet werden. Da das Verfahren im Zusammenhang mit der Erzeugung eines Klassengraphen aus Beispielobjekten entwickelt wurde (siehe unten) und somit vorhandene Strukturen nicht ausnutzt oder erhält, kann es nur schlecht zur inkrementellen Einordnung einer einzelnen Klasse verwendet werden.

Die Verfahren aus [Dicky et al. 1996] und [Moore 1996] setzen an der fehlenden Repräsentation von **Attributspezialisierung** und Methodenzerlegung bei anderen Verfahren an. Moore zieht im Rahmen der Klassenrestrukturierung gemeinsame Unterausdrücke aus Methoden heraus und erzeugt neue Methoden aus ihnen. Dies stellt laut Moore zum einen sicher, daß nicht nur jedes Attribut sondern auch jeder Teilausdruck nur an einer Stelle der Hierarchie definiert wird. Zum anderen erlaubt das Herausziehen von Teilmethoden es, neue Gemeinsamkeiten zwischen Klassen und damit neue Oberklassen zu entdecken. Moores Algorithmus verwirft im Gegensatz zu denen von Rundensteiner und Casais die bestehende Klassenhierarchie völlig, indem zu Anfang alle instanziierten Klassen um ererbte Attribute erweitert werden und Subklassenbeziehungen sowie die abstrakten Klassen vom nachfolgenden Algorithmus ignoriert werden. Aus den so erzeugten vollständigen Klassen werden die Methoden bzw. Methodenteile faktorisiert und anschließend eine neue Hierarchie erzeugt. Durch den völligen Neuaufbau der Klassenhierarchie kann das Verfahren nur schlecht inkrementell eingesetzt werden. Die schlechte Performance wird in [Moore 1996, S.247] allerdings auf die gleichzeitige Untersuchung aller Methoden zum Zwecke der Faktorisierung zurückgeführt. Als problematisch erweist sich bei diesem Ansatz aber insbesondere, daß die neuentstandenen Zwischenklassen und Methodenfragmente aufgrund ihrer systemgenerierten Namen das Verständnis der erzeugten Hierarchie und ihrer Methoden deutlich erschweren.

Dickys Verfahren [Dicky et al. 1996] ähnelt von der Vorgehensweise dem Rundensteiners, wenn auch die neu zu plazierende Klasse mit jeder anderen Klasse einzeln verglichen wird, ohne daß Äquivalenzgruppen gebildet werden. Der wesentliche Unterschied besteht darin, daß die Vergleichsroutine, die gleichzeitig auch die aus den Klassen herauszufaktorisierenden Attribute und Methoden bestimmt, Spezialisierung zuläßt. Dicky unterscheidet hierbei zwischen Signatur-/Typspezialisierung und Implementierungsspezialisierung: Unterschiedliche Implementierungen können nicht automatisch unterschieden werden und führen lediglich zur Faktorisierung einer abstrakten Methode[21]. Bei unterschiedlichen Typen bzw. Signaturen hingegen werden Gleichheit, Spezialisierung und Unvergleichbarkeit sowie das Auftreten von “self“–Parametern (siehe S.27) unterschieden und passende Generalisierungen erzeugt. Die Entscheidung über die Beziehung der beiden Klassen zueinander trifft das Verfahrens dann aufgrund der Ergebnisse aller verglichenen Attribute/Methoden der beiden Klassen. Durch die von Dicky verwendete Vorgehensweise können wesentlich anwendungsnähere Klassenhierarchien betrachtet werden als bei den Verfahren mit atomaren Attributen. Allerdings stellt Dicky selbst fest, daß bei einigen Kombinationen aus unterschiedlichen Signaturen, Code und “self“–Typen noch menschliche Eingriffe notwendig sind und daß im schlimmsten Fall unzusammenhängende Attribute nur aufgrund ihren gleichen Namens zu einer gemeinsamen Generalisierung faktorisiert werden.

Verweise auf weitere, ähnliche Verfahren finden sich in [Lieberherr et al. 1991, S.224f] und [Dicky et al. 1996, S.257f]. Letzteres erläutert insbesondere die mathematischen Grundlagen (die Gitter–Theorie). Zu erwähnen bleibt noch, daß anscheinend keine Verfahren existieren um aufgesplittete Klassenhierarchien, wie sie etwa bei der Verwendung von Rollen (S. 56) oder Multipler Klassifizierung (S.55) auftreten, automatisch zu (re)organisieren, ohne vorher alle möglichen Klassenkombinationen zu materialisieren.

Eine aus einem anderen Forschungsgebiet stammende Alternative zur Anordnung von Klassen stellen **Terminologische Wissenrepräsentationssysteme** (TWR) dar. Terminologische Konzepte, die im weitesten Sinne den objektorientierten Klassen entsprechen, werden von ihnen in eine sogenann-

[21] Also einer Methode, deren Implementierung erst in den Unterklassen spezifiziert wird.

te Subsumtionshierarchie (analog zu der Spezialisierungshierarchie aus Kapitel 2.3.3) eingeordnet [Nebel 1990, Kap. 4.4]. Im Gegensatz zu den oben aufgeführten Verfahren, die mittels Heuristiken und der Vernachlässigung objektorientierter Eigenschaften eine polynomiale Komplexität zu erreichen versuchen, beachtet die Berechnung der Subsumtionsbeziehung Typen und Restriktionen von Attributen und läßt auch deren Spezialisierung zu. Allerdings findet bei der Subsumtion keine Faktorisierung gemeinsamer Eigenschaften zweier Konzepte statt. Die Komplexität der Subsumtionsbestimmung richtet sich nach der Mächtigkeit der verwendeten terminologischen Sprache, so daß diese Sprache, um entscheidbare Subsumtionsalgorithmen zu erhalten, schwächer als etwa VeDa ist. Ein auf TWR basierendes Verfahren zur Verbesserung von VeDas Modellbausteinhierarchie beschreibt die in einer Zusammenarbeit des Lehr- und Forschungsgebiets Theoretische Informatik mit dem Lehrstuhl für Prozeßtechnik entstandene Dissertation [Sattler 1998]. Hierbei werden die Modellbausteine und Modelle zuerst von VeDa in eine terminologische Sprache transformiert. In dieser werden anschließend Subsumtions- oder Instanzbeziehungen berechnet und dem Modellierer als mögliche Hierarchie oder Instanzeinordnung vorgeschlagen, die er dann in der Datenbank nachvollziehen kann. In der weiterführenden Arbeit [Molitor 2000] werden zudem Verfahren vorgestellt, mit dem sich mehrere Klassen refaktorisieren lassen und neue Unterklassen einer Klasse für eine Instanzauswahl bestimmt werden können. Die entwickelten Verfahren lieferten in einem Praxistest verwertbare Ergebnisse und die von R. Molitor erwähnten möglichen Probleme im Bereich der Rückübersetzung und der exponentiellen Größe der berechneten Konzepte traten nicht auf. Da aber teilweise Heuristiken zum Einsatz kamen müssen diese Probleme zumindest theoretisch als noch nicht endgültig gelöst betrachtet werden. Zudem kann aufgrund der oben erwähnten Komplexitätsproblematik nicht die komplette in VeDas Klassen enthaltenen Information in der gewählten terminologische Sprache repräsentiert werden, so daß die Ergebnisse der Berechnungen nicht in jedem Fall zu den vorhandenen Objekten und Klassen passen müssen und ggf. einer Überarbeitung durch den Modellierer bedürfen.

Die vorliegende Arbeit beschäftigt sich mit Ansätzen zur *vollständigen* Repräsentation der in VeDas Hierarchien enthaltenen Information in einer Datenbank und der möglichst vollständigen Nutzung dieser Information zur Wiederverwendung. Dementsprechend liegen die weiter oben beschriebenen, auf objektorientierten Formalismen beruhenden Verfahren näher als die TWR–Ansätze.

Objektklassifikation

Die Aufgabe der Objektklassifikation besteht darin, zu einem gegebenen Objekt eine passende Klasse zu finden bzw. zu generieren. Die benötigten Eingabeparameter variieren weit: von einem prototypischen Objektnetz in [Lieberherr et al. 1991] über ein möglicherweise nur implizit vorhandenes Objekt beim TWR-Ansatz der “most–specific classes“ bis hin zu weitgehend unstrukturierten Daten bei den Clusterungsmethoden des Information Retrieval. Ergebnis sind aber immer ein oder mehrere Objekte, die in die zugehörige Klassenhierarchie eingeordnet sind.

Im **naiven Ansatz** erscheint die Objektklassifikation einfach auf die oben aufgeführten Verfahren zur Klassenorganisation rückführbar zu sein. Aus den Typinformationen des prototypbasierten Objekts[22] wird eine entsprechende Klasse erzeugt und diese von den im vorherigen Unterkapitel beschriebenen Algorithmen in die Klassenhierarchie eingeordnet bzw. eine gleiche Klasse identifiziert. Leider treten hierbei zwei Probleme auf: Übermäßige Spezialisierung der Klasse und unzureichende Typinformation im Objekt. Die Tatsache, daß die aus einem Objekt erzeugte Klasse genau dem Typ dieses Objekts entspricht, und damit unter Umständen spezieller als erwünscht ist, kann durch zusätzliche Information (etwa mehrere Objekte gleichen Namens aber leicht unterschiedlichen Typs in [Lieberherr et al. 1991]) oder eine Nachbearbeitung nach der Einordnung in die Klassenhierarchie

[22] Lediglich bei prototypischen Objekten kommt eine automatische Klassifikation in eine objektorientierte Klassenhierarchie hinein überhaupt in Betracht. Bei nur aus Wertinformationen bestehenden, klassenbasierten Objekten kann die sie prägende Struktur- und Typinformation nicht im erforderlichen Maße geschlußfolgert werden.

behoben werden. Schwerwiegender dürfte hingegen sein, daß in den meisten prototypbasierten Objektmodellen die Objekte nur unzureichende Typinformation tragen (vgl. die Diskussionen auf Seite 47 f). Insbesondere wenn ein Objektnetz, also mehrere verknüpfte Objekte, klassifiziert werden muß, kann dies zu Schwierigkeiten führen, da die Klassifikation eines Objektes die möglichen Klassen der anderen Objekte beeinflußt.

Diese Probleme mögen der Grund dafür sein, daß sich der Aufgabenstellung, prototypbasierte Objekte in eine Klassenhierarchie einzuordnen, anscheinend niemand direkt gewidmet hat. Verwandte Ansätze finden sich allerdings in der objektorientierten Analyse (OOA), wo in [Lieberherr et al. 1991] und [Chen und Lee 1996] Verfahren vorgeschlagen werden, um aus den bei der Analyse gewonnenen Objekten Klassenstrukturen zu erzeugen. Chen und Lee machen es sich hierbei recht einfach, indem sie nur **atomare Attribute** betrachten, also weder Attributspezialisierung noch verknüpfte Objekte. Ein invertierter Index von den Attributen auf die Objekte erlaubt es dann, gleiche Attributmuster zu erkennen und aus den so gewonnenen Klassen eine Subtyphierarchie zu erstellen.

[Lieberherr et al. 1991] erlaubt hingegen **verknüpfte Objekte** und nutzt dies auch um die Beziehungen zwischen den Klassen zu generieren und die referenzierten Objekten zugewiesenen Klassen zu generalisieren[23]. Das entstandene Klassennetz — von einer Hierarchie kann man noch nicht sprechen — minimiert Lieberherr dann mit seinen oben beschriebenen Klassenorganisationsalgorithmen. Allerdings geht der Algorithmus von Voraussetzungen aus, die zwar bei der OOA eingehaltenen werden, nicht aber bei allgemeinen Modellierungsaufgaben: Zum einen müssen alle Objekte den Namen der Klasse tragen, zu der sie gehören (vgl. die beiden `Student`–Instanzen in der Fußnote). Zum anderen benötigt der Klassifizierungsalgorithmus, um die Überspezialisierung einer Klasse zu vermeiden, mehrere, geschickt gewählte Beispiele für sie. Auch kann der Algorithmus nicht inkrementell ausgeführt werden.

Terminologische Wissensrepräsentationssysteme erlauben es, zu einem terminologischen Objekt seine "**most-specific concepts**" zu berechnen [Nebel 1990, S.104ff], was der Klassifizierung entspricht[24]. Das Objekt besteht hierbei im Normalfall aus einigen Attribut- und Konzeptzuordnungen, aus denen der Klassifizierungsalgorithmus neue oder spezialisierte Konzeptzuordnungen generiert. Das Ergebnis ist allerdings keine minimale Überdeckung der Attribute des Objekts, wie bei einem objektorientierten Algorithmus zu erwarten, sondern eine maximale Überdeckung, d.h. ein Objekt wird einem Konzept zugeordnet, selbst wenn alle Eigenschaften des Objekts, die dieses Konzept enthält, auch schon von anderen Konzepten, denen das Objekt zugeordnet wurde, beschrieben werden. Genauso wie bei der oben beschriebene Subsumtion liegt das Problem der TWR–Systeme bei der Klassifikation in dem Spannungsfeld zwischen Mächtigkeit der Repräsentationssprache und Komplexität bzw. Berechenbarkeit des Algorithmuses. So kann etwa das von Nebel vorgestellte Verfahren Vollständigkeit nur dann garantieren, wenn alle Beziehungen zwischen zwei Objekten bekannt sind, also nicht vom System hergeleitet werden müssen [Nebel 1990, S.113ff]. In objektorientierten Datenbanken, die bekanntlich der "closed-world assumption" folgen, ist diese Voraussetzung allerdings immer erfüllt. Der bereits auf Seite 70 erwähnte Ansatz von Molitor erlaubt es zudem die speziellste Klasse für eine vom Modellier bestimmte Menge von Instanzen zu berechnen. Da das verwendete Verfahren auch Teilmengen betrachtet, können Überspezialisierungen vom erkannt und verhindert werden. Allerdings gelten die dort beschriebenen Probleme auch für die Verwendung des Ansatzes zur Objektklassifikation.

Die Propagation eines teilweise klassifizierten Objektes zu seinen "most–specific classes" kann mittels der in Abschnitt 4.1.4 beschriebenen impliziten **Objektmigrationsverfahren** (etwa Chambers

[23] Aus den beiden Objekten

```
instance:Student                instance:Student
   fach: Informatik                fach: Elektrotechnik
```

folgert er etwa, daß Informatik und Elektrotechnik Spezialisierungen einer (durch den Benutzer "Fachbereich" zu nennenden) Klasse sind.

[24] Man beachte, daß TWR–Systeme diese Operation als "Realization" bezeichnen, während "Classification" dort die oben beschriebene Einordnung eines neuen Konzepts in die Konzepthierarchie benennt.

"Predicate Classes") auch in objektorientierten Formalismen erfolgen, sofern die Migrationsbedingungen entsprechend gewählt wurden. Die Notwendigkeit, diese Bedingungen aufzustellen und für jedes neue Objekt eine oder mehrere passende Startklassen zu wählen, erschwert jedoch die Automatisierung.

Verschiedene **Cluster–Verfahren** werden im "Information Retrieval" genutzt, um schlecht oder gar nicht strukturierte Objekte zu Klassen zusammenzufassen und so die Suche in ihnen zu vereinfachen. Je nach dem Vorhandensein von Strukturinformation können das Objekt charakterisierende Attribute zur Klassifizierung genutzt werden (wie z.B. auch beim unten beschriebenen "Case–Based Reasoning") oder lediglich eine Ähnlichkeitsfunktion. [van Rijsbergen 1979, S.31ff] nennt verschiedene „ad–hoc"–Cluster–Verfahren, die mit einer möglichst geringen Zahl an Durchläufen über der Objektmenge versuchen, die Einteilung der Objekte vorzunehmen, wodurch aber die ursprüngliche Anordnung der Objekte Einfluß auf das Ergebnis gewinnt und inkrementelle Erweiterungen unmöglich werden. Das auf einer Ähnlichkeitsfunktion beruhende Verfahren des "Single–Link Clustering", das auch in dem bei den Äquivalenzfunktionen erwähnten [Maarek et al. 1991] benutzt wird, besitzt zwar quadratische Komplexität bezüglich der Zahl der Objekte, liefert aber eine gegenüber Umsortierungen und kleinen Änderungen an den Objekten stabile, hierarchische Ordnung. Bei allen aufgeführten Verfahren entsprechen die entstandenen Klassen aber lediglich der Extensionsfunktionalität objektorientierter Klassen, d.h. die Zuordnung eines Objekts zu einer Klasse läßt keine Rückschlüsse auf Typ oder Spezialisierungsbeziehung zu. Die hierfür benötigte Information geht nämlich bei der Beschränkung auf charakterisierende Attribute bzw. auf die Ähnlichkeitsfunktion verloren.

Delegations(re)organisation

Die Delegationsorganisation befaßt sich mit dem Einordnen neuer prototypbasierter Objekte in eine Delegationshierarchie bzw. der Neuordnung eines bestehenden Hierarchie, wobei Delegationstiefe, neu zu definierende und/oder zu überschreibende Attribute minimiert werden. Da gemäß [Stein 1987] Delegation und Vererbung äquivalent sind, müßten die oben aufgeführten Verfahren zur (Re-)Organisation einer Klassenhierarchie auch für die Delegation anwendbar sein. Hier zeigt sich aber erneut der schon auf Seite 50 erwähnte Unterschied zwischen der theoretischen Äquivalenz und der praktischen Differenz aufgrund unterschiedlicher Typrestriktionen. Die angeführten Klassenorganisationsverfahren betrachten nämlich alle die Subtyp- bzw. Spezialisierungshierarchie, die zwar meist mit der Vererbungshierarchie verbunden sind (vgl. 2.3.2), im Gegensatz zu dieser aber eine Teilmengenbeziehung zwischen den Klassen implizieren. Das bedeutet, daß in Subtyp- und Spezialisierungshierarchien ganze Teilbäume als Einordnungsstellen ausgeschlossen werden können, wenn ihre Wurzel nicht mit der einzuordnenden Klasse konsistent ist (sprich: sie nicht subsumiert). Bei der Vererbungs- und Delegationsbeziehung hingegen können einmal getroffene Änderungen bei späteren Klassen-/Objektübergängen wieder zurückgenommen werden, so daß jede Klasse/jedes Objekt mit dem einzuordnenden Exemplar verglichen werden muß, um den besten Platz für dieses zu finden. Die Delegation ist zwar meist mit der Namenskompatibilität (vgl. S. 53) verknüpft, die einige Teilbaumausschlüsse erlaubt, die bei der Delegation aber ebenfalls 'vererbte' Wertinformation ist jedoch keiner monotonen Einschränkung unterworfen.

Eine Lösung dieser Problematik wird in [Light 1993] vorgestellt. Das dort vorgeschlagene Verfahren zur Einordnung von Wörtern in linguistische Klassen, die sich in einer Default–Vererbungshierarchie befinden, kann analog auf die Delegationsorganisation übertragen werden. Da diese Default–Vererbungshierarchie der Delegation *ohne* Namenskompatibilität entspricht[25], bleiben Light keine Möglichkeiten das Durchsuchen aller Klassen zu vermeiden, um die von ihm angestrebte Minimierung der Summe aus den Anzahlen der Vererbungskanten, der neu hinzukommenden und der überschriebenen Attribute zu erreichen. Da zudem Multiple Vererbung erlaubt ist, zeigt Light, daß die Suche

[25] Vorhandene Attribute können zwar nicht entfernt, wohl aber in den Zustand 'unbekannt' gesetzt werden.

nach der optimalen Lösung NP–vollständig wäre und setzt eine Greedy–Heuristik ein, ohne jedoch eine Approximationsabschätzung geben zu können. Die Tatsache, daß Lights Objekte nur ternäre Attribute besitzen können, also insbesondere kein Beziehungen zwischen den Objekten existieren, hat für die Übertragbarkeit von Lights Verfahren auf die allgemeine Delegationsreorganisation nicht die negativen Auswirkungen wie bei den oben aufgeführten Verfahren zu Klassifikation. Während dort die geänderte Klassifikation eines assoziierten Objekts auch zu die Reklassifizierung seines Gegenparts führen kann, tritt dies bei der Delegation wegen der fehlenden Typisierung der Attribute nicht auf.

4.2.3 Case–Based Reasoning

Im Fallbasierten Schließen (Case–based Reasoning; CBR) treffen die Probleme der Suche und der Klassifikation aufeinander. Es stellt als Spezialfall des Analogieschließens [Spanoudakis 1994, S.16, S.24] ein Anwendungsgebiet für Ähnlichkeitsfunktionen dar und benötigt zur Einordnung neuer Fälle entsprechende Organisationsverfahren. [Kolodner 1993] definiert CBR als eine Methode in der sich ein Problemlöser an frühere Situationen ähnlich der betrachteten erinnert und sie nutzt, um diese neue Situation einfacher zu lösen[26]. Durch die Verwendung von Beispielen zur Problemlösung anstelle von fakten-/regelbasierter Inferenz kann CBR auch in nur mangelhaft verstandenen oder maschinell unentscheidbaren Situationen wie dem verfahrenstechnischen Entwurf (vgl. etwa [Surma und Braunschweig 1996; Lohmann 1994]) Lösungsvorschläge liefern.

In seiner Grundstruktur ähnelt CBR dabei dem in dieser Arbeit vorgestelltem Wiederverwendungsproblem. [Goel und Chandrasekaran 1992] identifizieren als Teilaufgaben Fallzugriff (retrieval), Falladaption und Fallspeicherung analog zu Suchen, Anpassen, Einordnen (siehe S. 10). Hinzu kommt in einigen Ansätzen noch das Verbessern der obigen Prozesse aufgrund gelungener oder fehlerhafter Fallanwendungen. In der Realisierung dieser Teilaufgaben bestehen jedoch nicht unwesentliche Unterschiede zwischen CBR und dem in dieser Arbeit verfolgten Ansatz: Die **Adaption** der gefundenen Beispiele an die durch die Anforderungen gegebenen Randbedingungen soll im CBR (weitgehend) automatisch erfolgen. Ein Modell für eine ähnliche Aufgabenstellung zu adaptieren, erfordert aber ein hohes Maß an Kontextwissen (vgl. [Kolodner 1993, Kap.11]), so daß entsprechende Ansätze sich etwa auf Case–Based Retrieval beschränken und die Adaption dem Benutzer überlassen [Surma und Braunschweig 1996; Lohmann 1994], für jede Teiltransformationsaufgabe explizite Modifikationspläne vorgeben [Goel und Chandrasekaran 1992], die Adaption auf die Generalisierung innerhalb der Subsumtionshierarchie begrenzen [Napoli et al. 1996] oder umfangreiches Domänenwissen benötigen [Kolodner 1993, Kap.11]. Die **Fallspeicherung** neuer Fälle in Zugriffsstrukturen, Indizierung genannt, erfordert häufig die Vorgabe aussagekräftiger zu indizierender Attribute [Kolodner 1993, S.257ff],[Lohmann 1994, S.48ff] und/oder ihrer Gewichtungen durch einen Menschen. Die Gewichtungen sind zudem teilweise abhängig vom einzuordnenden Fall [Surma und Braunschweig 1996] oder von dem zu lösenden Problem [Lohmann 1994, S.53ff],[Kolodner 1993, S.349ff]. Der **Zugriff** auf einen zum Problem passenden Fall in der Fallbasis geschieht durch Matching von Indexwerten oder strukturellen Eigenschaften ggf. variiert durch die vorgegebenen Gewichtungen oder dynamisch abgeleitete Eigenschaften und Abstraktionen [Kolodner 1993, Kap.9]. Je nach dem, welche Selektionskriterien gewählt werden, ob mehrere oder nur ein Zielfunktion vorliegen und ob die erhaltenen Fälle nach ihrer Ähnlichkeit mit dem betrachteten Problem sortiert werden, nähern sich die verwendeten Verfahren der Ähnlichkeitsfunktion an.

Im Gegensatz dazu soll der in dieser Arbeit vorgestellte Ansatz die Anpassung lediglich für den Benutzer vereinfachen, die Einordnung in ein objektorientiertes Schema vornehmen und die Suche nach passenden Modellbausteinen nur auf die in diesem Schema vorhandenen Informationen basieren. Der Ansatz entspricht somit den in [Spanoudakis 1994, S.3f,S.24f] geäußerten Kritikpunkten, daß die

[26] „In case–based reasoning, a reasoner remembers previous situations similiar the current one und uses them to help solve the new problem.“ [Kolodner 1993, S.4]

Erlangung des beim CBR benötigte Domänenwissen einen Flaschenhals darstellt, daß vom Benutzer während der Modellierung einzugebende Zusatzinformation die Akzeptanz verringert und daß die strukturellen Vorgaben der meisten Fallrepräsentationen die Anwendbarkeit einschränken.

In einigen betrachteten verfahrenstechnischen CBR–Beispielen kommen **Ähnlichkeitsfunktionen** zum Einsatz. [Lohmann 1994] benutzt das gewichtete arithmetische Mittel über die Ähnlichkeitswerte der einzelnen Indizes (gegeben durch Tabellen oder Fuzzy–Funktionen), um die gespeicherten Fälle im Verhältnis zum betrachteten Fall zu bewerten. Würde das Mittel über den Gesamtwert der Gewichte, die der Anpassung der Bewertungsfunktion an verschiedene Problemstellung dienen, normiert, so wäre die Bewertungsfunktion ein Ähnlichkeitsmaß. In [Surma und Braunschweig 1996] werden Fließbilder anhand von Jaccards Koeffizient (vgl. [Maarek et al. 1991]) verglichen, indem sie mit Hilfe der charakterisierenden Attribute "Typ" und "Funktion" die Komponenten der beiden Fließbilder aufeinander abbilden[27] und deren jeweilige Ähnlichkeit anhand lokaler Ähnlichkeitsfunktionen und einer vom Benutzer angegebenen Gewichtung berechnet. Die lokalen Ähnlichkeitsfunktionen können allgemein sein (wie von den Autoren vorgeschlagen) oder domänenbasiert, wie im Verfahren aus [Lohmann 1994]. Anschließend werden die Ähnlichkeit der Verknüpfungen in den Fließbilder ebenfalls mittels Jaccards Koeffizient beurteilt, allerdings erweitert um einen Term, der den Zusammenhang des gemeinsamen Schnittgraphen bewertet, um Abbildungen mit hoher lokaler Entsprechung zu bevorzugen. Alle verwendeten Ähnlichkeitsfunktionen sind unsymmetrisch, da der vermutete Anpassungsaufwand in die Tabellen bzw. Funktionen integriert wurde.

4.2.4 Zusammenfassung

Zusammenfassend läßt sich festhalten, daß Ähnlichkeitsfunktionen sowohl in der Suche als auch der Einordnung Verwendung finden können. Bei ihrer Auswahl muß entschieden werden, ob Domänenwissen integriert werden soll, welche Objekt- und Klasseneigenschaften wie in die Funktion einfließen sollen und welche mathematischen Eigenschaften sie erfüllen soll. Die Klassifikation hingegen leidet unter grundsätzlichen Problemen bezüglich ihrer Komplexität sowie der berücksichtigen objektorientierten Eigenschaften und wechselseitigen Beeinflussung verschiedener Objekte, so daß häufige, komplette Reorganisationen vermieden werden sollten. Ein Ansatz, wie der von Casais verfolgte (S. 68), der die vorläufige Einordnung einer Klasse nutzt, um die weitere Betrachtung auf einen lokalen Bereich zu beschränken, dürfte darum von Vorteil sein. Die Probleme der Reorganisation von Delegationshierarchien zeigen, daß auch bei Verwendung prototypbasierter Objektmodelle noch nicht alle mit der Modifizierbarkeit verbundenen Schwierigkeiten beseitigt sind. Die beschriebenen CBR–Ansätze können zwar gute Klassifikationsergebnisse erreichen, benötigen dafür aber ein hohes Maß an vorgegebenem Domänenwissen, dessen Gewinnung in einem durch den Benutzer erweiterbaren System, ggf. ständig vorgenommen werden müßte.

4.3 Einordnung spezieller ingenieurtechnischer Objektmodelle

Dieses Unterkapitel soll aufzeigen, inwieweit die in Abschnitt 4.1 beschriebenen Objektmodellvariationen in den verschiedenen für verfahrens- und ingenieurtechnische Anwendungen entwickelten Objektmodellen enthalten sind. Hierbei muß nach der Zielrichtung der Sprachen unterschieden werden. ASCEND, OMOLA, MODELICA und MOSES konzentrieren sich auf die abstrakte, objektorientierte Repräsentation von Gleichungssystemen. Ihre Modelle können nach vergleichsweise geringer Übersetzung zur Simulation oder als Eingabe eines Simulators genutzt werden. MODEL.LA, VEDA, SHOOD und das AE&C–Datenmodell von Blaha et al. hingegen sind oder enthalten generische

[27] Die Autoren gehen davon aus, daß sich alle Komponenten eines Fließbildes in Typ (Reaktor, Pumpe, Destillationskolonne, ...) oder Funktion (verschiedene Reaktionen, verschiedene Ströme, verschiedene Trennvorgänge, ...) unterscheiden, so daß eine einfache Abbildungsvorschrift definiert werden kann.

Modellierungssprachen, die zur Modellierung beliebiger, also auch nichtmathematischer Sachverhalte genutzt werden können. KBDS und CLOOD befassen sich hauptsächlich mit der Alternativen- und Versionsverwaltung bzw. deren automatischer Generierung. Schlußendlich versuchen KBMoSS, N–dim und teilweise auch Épée gezielt die explorative Modellierung, also das Modifizieren und Anpassen bestehender Modelle zu vereinfachen. Das jeweils intendierte Anwendungsgebiet zeigt selbstverständlich Auswirkungen auf das genutzte Objektmodell sowie den Umfang der beschriebenen Modellierungsmethodologie, so daß neben der klassischen klassenbasierten sowie der prototypbasierten Objektorientierung auch diverse Variationen auftauchen.

4.3.1 Klassische Objektmodelle

Als klassische Objektmodelle bezeichnen wir im folgenden klassen- oder framebasierte Modelle ohne die auf den Seiten 52 bis 56 beschriebenen Erweiterungen. Eine früher Vertreter mit einem 'lupenrein' klassenbasierten Objektmodell ist das **AE&C–Datenmodell** aus [Blaha et al. 1989]. Es verwendet als Sprache die von [Rumbaugh et al. 1991] definierten OMT–Objekt–Diagramme bzw. eine textuelle Repräsentation von ihnen und befaßt sich hauptsächlich mit der Definition von vier Teilmodellen für Komponenten, Verknüpfungen, Fließbilder und blockorientierte Simulationen. Diese Modelle werden auf eine relationale Datenbank abgebildet. Da eine blockorientierte Simulation als Endergebnis angestrebt wird, beschränken sich die als Klassen definierten Modellbausteine auf fest vorgegebene Standardelemente, die sich auf gleichem Niveau wie die Methodologie- und Bausteinebene VeDAs befinden (vgl. S. 15). Das führt dazu, daß viele Klassen, die Methodologiekonzepte repräsentieren, keine Attribute haben. Maßgefertigte Bausteine müssen in der relationalen Datenbank mühsam durch Eintrag von <Ausrüstung, Attributname, Wert>–Attributtripeln definiert werden [ebd., S.757r]. Interessanterweise beherrscht die verwendete Sprache zwar keine Delegation, diese wird aber im Datenmodell trotzdem in Form von Defaultvererbung benutzt[28] und muß somit von den Applikationsprogrammen erbracht werden. Die Vorkehrungen für die Wiederverwendung von Modellbausteinen beschränken sich auf die Definition einer Library–Datenstruktur, in der einzelne Prozeßlinien ("Pipelines") gespeichert werden können [ebd., S.763l].

Ascend [Piela et al. 1991; Westerberg und Piela 1994], **Omola** [Andersson 1994] und **Modelica** [Boudaud et al. 1997], gleichungsnahe Modellierungssprachen, benutzen alle eine sehr interessante Variante des methodenlosen, klassenbasierten Objektmodells: Während des eigentlichen Modellierungsvorgangs existieren bei ihnen keine Instanzen. Sie werden erst im Verlauf der Vorbereitung auf den Simulationsvorgang durch den Compiler [Andersson 1994, S.164ff] oder den Interpreter aus den Klassen erzeugt und speichern dann das mathematische Modell, insbesondere die Variablenwerte und freien Parameter. Um bei der Modellierung ohne Instanzen auszukommen, werden Defaultwerte für Attribute in Klassen intensiv genutzt, so daß jeder Klasse in Form der Defaultwerte ihrer Attribute quasi eine Instanz zugeordnet ist. Diese Defaults können nicht nur im Zuge der Vererbung geändert werden, sondern alle drei Sprachen bieten mehr oder weniger prägnante Möglichkeiten Defaultwerte von Attributen einer Unterkomponente aus der Oberkomponente heraus zu überschreiben[29]. Damit realisieren sie den in [Gil und Lorenz 1996] "Environmental Acquisition" genannten Einfluß eines Behälterobjekts auf seine Teile, hier allerdings getrieben durch den Behälter statt durch die Teile. Neben einigen Unterschieden — so beherrschen Ascend und Omola nur einfache Vererbung, lediglich Omola läßt beliebige Änderungen geerbter Attribute zu und Modelica trennt Typ und Klasse — ist allen drei Sprachen gemein, daß sie ihre Modellierungmethodologie implizit in der Sprachdefinition enthalten. Explizite Basisklassendefinitionen, wie sie etwa Omola besitzt [Andersson 1994, Anhang B], erscheinen dadurch etwas mager.

Projektübergreifende Wiederverwendung findet bei allen drei Sprachen nur mit Hilfe von Modellbibliotheken statt. Dies ergibt sich aus zwei Gründen. Zum einen handelt es sich bei allen drei Spra-

[28] Etwa indem im "Plant–Section–Equipment"–Baum einige Attributwerte höherer Knoten auch von ihren Unterknoten genutzt werden, sofern sie nicht von ihnen überschrieben werden [Blaha et al. 1989, S.757l]

[29] Pfadausdrücke in Ascend, WITH–Spezialisierungen lokaler Klassen in Omola und Wertparameter in Modelica

chen um textuelle Eingabesprachen für die Simulation, nicht aber um Entwicklungsumgebungen mit einem variablen Datenschema. Wird ein solches benötigt, müssen darum andere Repräsentationsformen zum Einsatz kommen. Innerhalb der Modellierungsumgebung OmSim entspricht etwa das zur Darstellung der Omola–Klassen genutzte Objektmodell [Andersson 1994, S.159ff] nur entfernt Omola, vielmehr aber, abgesehen von seiner Domänenbezogenheit, einem klassischen klassenbasierten Modell. Insbesondere die in Omola expliziten Modifikatoren werden in OmSim aufgelöst und implizit in neu kreierten Klassen gespeichert [Andersson 1994, S.88]. Zum anderen geht der spezielle Modifikationsmechanismus der drei Sprachen nur in seiner Prägnanz, nicht aber in seiner Modellierungsmächtigkeit über die Vererbung hinaus[30]. Auch Modelicas an [Abadi und Cardelli 1996] angelehnte "Subclassing implies Subtyping"–Typ–Klassen–Trennung (vgl. Seite 26) kann die Mächtigkeit nicht wirklich erhöhen, da sie hauptsächlich dazu dient, die Nichtspezialisierbarkeit der Gleichungen in den Klassen zu umgehen, d.h. Interface und Implementierung stärker zu trennen. Wie alle klassenbasierten Modelle eignen sich somit auch diese drei vor allem für das "anticipated sharing" [Stein et al. 1989, S. 35f], was die genannten Modellbibliotheken realisieren.

MOSES [Maffezzoni und Girelli 1998] zielt ähnlich den obigen drei Sprachen auf die gleichungsnahe Modellierung ab. Als Beispiel dient allerdings die mechanische Simulation eines Roboters. Anders als Modelica und deren Vorgänger läßt MOSES keine Vererbungshierarchie zu, ordnet seine prototypischen Modellbausteine dafür aber baumartig in einer Bibliotheksdatenbank an. Den Verzicht auf Vererbung zwischen den Bausteinen begründen die Autoren mit der besseren Lesbarkeit von Klassen, wenn zu ihrem Verständnis nicht zahlreiche Oberklassen betrachtet werden müssen, sowie mit der Unmöglichkeit eine gute Klassenhierarchie zu definieren, wenn — wie im Anwendungsfall — fortlaufend zusätzliche Klassen hinzukommen. Die vorkommenden Konzepte teilt MOSES ähnlich VeDAs Hierarchie (s.S. 15) in die drei Ebenen Model–Type, Model–Prototype, Model–Instance. Erstere enthält die Basiskonzepte wie Model, Connection und Terminal, die als erweitertes Entity–Relationship–Diagramm mit Vererbung definiert sind und das Datenbankschema bilden. Die Model–Prototypes sind Instanzen[31] dieser Basiskonzepte. Zwischen den Modell–Prototypes besteht aus den oben erläuterten Gründen keine Vererbung und neue Prototypen können nur durch einen Bibliotheksverwalter in die öffentliche Bibliothek eingetragen werden. Die eigentlichen Modelle bestehen dann aus 'Kopien' der Prototypen, die allerdings nur mittels der vorhandenen Parameter angepaßt werden können, so daß sie sich nicht wie prototypbasierte Objekte sondern wie Instanzen der Modellbausteine verhalten.

Im Bereich der Wiederverwendung leidet MOSES stark unter dem restriktiven Objektmodell, insbesondere dem Verzicht auf eine Klassenhierarchie zwischen den Modellbausteinen, sowie den Änderungsbeschränkungen der Bausteinbibliothek. Der Vererbungsverzicht erscheint besonders unnötig, da die gegebenen Argumente eher wenig stichhaltig sind: Zwar wird etwa in [Dvorak 1994] eine ähnliche Auffassung bezüglich der Überschaubarkeit tiefer Klassenhierarchien vertreten, dort wird dagegen aber eine automatische Restrukturierung der Hierarchie eingeführt. Eine vollständige Ansicht einer Klasse läßt sich zudem auch bei Vorhandensein geerbter Methoden und Attribute durch entsprechende Werkzeugunterstützung erreichen. Und daß auch flexible oder dynamische Klassenschemata hierarchisch strukturiert werden können, zeigen andere aufgeführte Ansätze.

Model.La [Stephanopoulos et al. 1990a] und **LCR** [Nagel et al. 1995] geben die meisten Probleme bei der Einordnung in die anfangs erwähnten Zielkategorien auf. So dient Model.La zur Modellierung verfahrenstechnischer Prozesse und LCR zur Beschreibung chemischer Reaktionen. Beiden Sprachen liegt jedoch das selbe, allgemein verwendbare Objektmodell zugrunde, dessen Grammatik lediglich für jede Domäne modifiziert wird (vgl. [Nagel et al. 1995, S.34f]). Es handelt sich hierbei um ein framebasiertes Datenmodell, das neben Spezialisierung und Instanziierung noch fünf weitere semantische Beziehungen (plus vier Inverse) zwischen Objekten und ihren Attributen ermöglicht

[30] [Andersson 1994, S.90ff] zeigt anhand eines Beispiels die Abbildung auf normale Vererbung mit und ohne lokale Klassendefinitionen.

[31] Die Autoren reden von Vererbung zwischen den einzelnen Ebenen. Es handelt sich aber offensichtlich um die 'Vererbung', die die Instanziierung zwischen Klasse und Instanz hervorruft, und nicht diejenige zwischen Klassen.

und Attribute zusätzlich mit Facetten versieht. Mit Hilfe dieser vielfältigen Beziehungen sollen alle einem Modellierungsobjekt zugrunde liegenden Annahmen und Vereinfachungen erfaßt und die Vermischung deklarativem und prozeduralem Wissens, wie es beispielsweise in den gleichungsnahen Modellierungssprachen stattfindet, vermieden werden [Stephanopoulos et al. 1990a, S.813f]. Zwar wird die Einordnung des Objektmodells von MODEL.LA und LCR durch die etwas lückenhaften Semantikdefinition erschwert, es handelt sich aber anscheinend um ein konventionelles, aus Klassen *und Objekten* bestehendes Modell. Die Objekte enthalten jeweils wie in VEDA die Komponenten eines verfahrenstechnischen Modells, die Klassen die Modellbausteine. Die Modellierungsmethodologie verteilt sich wie bereits erwähnt auf die Sprache und die oberen Ebenen der Klassengraphens. Anscheinend beruht diese Aufteilung auf zwei Ebenen allerdings auf den Beschränkungen des verwendeten zweistufigen Objektmodells. Die Entwicklungsumgebung Design-Kit, die MODEL.LA verwendet, speichert ihre Informationen nämlich in drei Wissensbasen mit einer fast analogen Aufteilung zu VEDAs drei Ebenen: `Working` enthält die in einem Modell enthaltenen Instanzen, `Library` die zusammengesetzten Modellbausteine und `Network--Model` die grundlegenden Abstraktionen, zu denen MODEL.LA auch die elementaren Modellbausteine zählt.

Interessant erscheint noch ein kurzer Blick auf zwei der verwendeten semantischen Beziehungen: `is-attached-to` und `is-characterized-by`. `is-attached-to` verbindet in MODEL.LA die dortigen Interfaces mit den Devices und Connections und in LCR verschiedene Atome zu Molekülen. Nach Aussagen der Autoren ermöglicht diese Beziehung den Informationstransfer zwischen verschiedenen Modellteilen[32], so daß es sich möglicherweise, eine genauere Definition fehlt leider, um einen ähnlichen Mechanismus wie die oben beschriebene Komponentenmodifikation in OMOLA, ASCEND und MODELICA handelt. `is-characterized-by` erlaubt es anscheinend, ähnlich den Mixin–Klassen, orthogonal zur Klassenhierarchie definierte Modifikatoren auf eine Klasse anzuwenden. So erreicht etwa die Beziehung `Reactor-x is-characterized-as \adiabatic\`, daß in das Attribut `pressure` der Klasse `Reactor-x` ein konstanter Wert eingetragen wird [Nagel et al. 1995, S.76]. Leider erwähnen die Autoren nicht, ob die möglichen Modifikationen in der Sprache fest eingebaut sind oder ob vom Benutzer eigene definiert werden können.
Zweifel an dem von den Autoren vorgestellten Facetten- und Relationenkonzept ergeben sich allerdings durch die Möglichkeit Klassen 'natürlichsprachlich' zu definieren. Die entsprechende Sprache [Stephanopoulos et al. 1990a, S.844ff] enthält nämlich anscheinend keine Konstruktoren um die Facetten und semantischen Beziehungen zu setzen.

Neue Konzepte werden in MODEL.LA erzeugt, indem zuerst die benötigten Klassen aus vorhandenen oder neu zu erzeugenden Klassen zusammengesetzt und anschließend instanziiert werden [Nagel et al. 1995, S.50ff], so daß die direkte Wiederverwendung von Modellen schwierig ist. Allerdings unterstützt MODEL.LA die Erzeugung von Modellalternativen durch — außerhalb des Objektmodells definierte — Kontexte [Stephanopoulos et al. 1990b], die Versionierungs- und Konfigurationsaufgaben erledigen und es dadurch ermöglichen, Alternativen effizient anzulegen und zu speichern.

4.3.2 Fortgeschrittene Modelle

Die folgenden Ansätze verwenden prototypbasierte Objektmodelle oder erweitern das klassenbasierte Objektmodell. Wir werden sie darum — wertungsfrei — als fortgeschrittene Modelle bezeichnen.

Die Modellrepräsentation der Modellierungsumgebung **ModKit** [Bogusch et al. 1996] beruht auf der in VEDA entwickelten Modellierungsmethodologie. Bei ihrer Realisierung geht ModKit allerdings etwas andere Wege als die im letzten Kapitel aufgeführten Sprachen, auch wenn es wie diese ein konventionelles, klassenbasiertes Objektmodell nutzt. Statt nämlich, wie etwa OMOLA oder MODEL.LA, die Modellbausteine als Klassen zu speichern und die Modellierungsmethodologie in die

[32] „establishes linkages between different modelling elements, allowing flow of information from one to the other“ [Nagel et al. 1995, S.29]

Sprache zu verlegen, wird die Methodologie in der Klassenebene repräsentiert und Bausteine zusammen mit den Modellen weitgehend in der Instanzebene. Dies erlaubt es auf die Eigenentwicklung eine Objektmodells zu verzichten und die kommerzielle Entwicklungsumgebung G2 zu verwenden.

Dieser Ansatz kann als gemischt klassen-/prototypbasiert klassifiziert werden, da die in der Instanzebene gespeicherten Modellbausteine als Prototypen dienen, während ihre übergreifende Struktur durch die Klassen der Methodologieebene festgelegt wird. Allerdings bestehen auf der Instanzebene keine Sharing–Beziehungen wie etwa Delegation, so daß es hier unter Umständen zu redundanten Modellen kommt. Die Repräsentation von Modellen und Modellbausteinen in einer Ebene hat natürlich auch hier die in Kapitel 2.3.5 vorgestellten Nachteile: Verlust der Bausteinhierarchie mit ihrer Typisierungsinformation und ihren anwendungsnahen Konzepten. Zur Behebung dieses Mißstandes wurden einige Elementarbausteine auf der Klassenebene realisiert und vorgegebene Modellbausteine in Paletten (also Bibliotheken) abgelegt. Zudem wird das TWR-System CRACK verwendet, um aufgrund der bestehenden Klassenhierarchie und den vorhandenen Objekten eine Navigationshierarchie für den Benutzer zu berechnen [Sattler 1998]. Aufgrund der nur eingeschränkt möglichen Modifikation der Klassenhierarchie in MODKIT zur Laufzeit und wegen der Unvollständigkeit der nach CRACK übertragenen Information wird die resultierende Hierarchie derzeit allerdings nicht zur Reorganisation der von MODKITs Klassenhierarchie oder zur Objektklassifikation genutzt.

Wesentlich weiter in Richtung prototypbasierte Modellierung geht die auf dem in Kapitel 4.1.1 beschriebenen BOS aufbauende Modellierungsumgebung **n-dim** [n-dim Group 1995b; Levy et al. 1993]. N–DIM unterscheidet nur zwischen atomaren und zusammengesetzten Objekten ("Modelle" genannt), intrinsischen Attributen, Relationen und Operationen. N–DIMs 'Modelle' sind eine Samm-

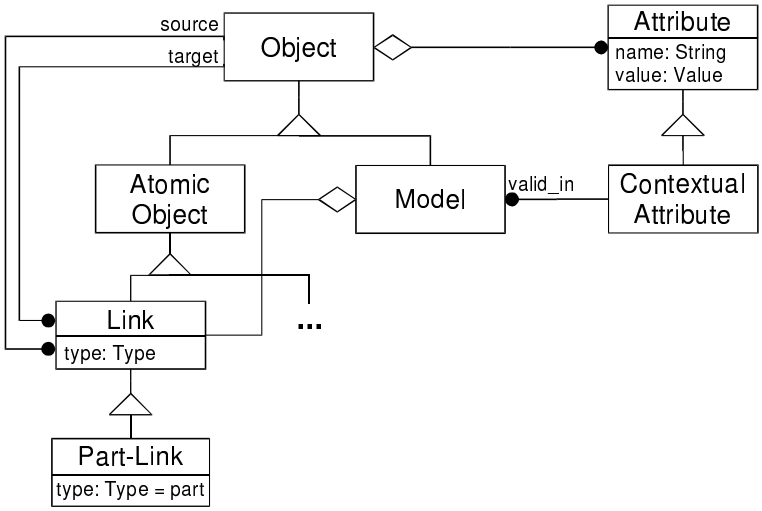


Abbildung 4.12: N–dims Metamodell (nach den Angaben aus [Levy et al. 1993])

lung von Verbindungen zwischen Objekten, wobei spezielle 'part'–Verbindungen das Enthaltensein eines Objekts in einem Modell anzeigen sollen und auch entsprechend repräsentiert werden. Bild 4.12 zeigt das wahrscheinliche Metamodell. Durch ein View–ähnliches Konzept kann ein und dasselbe Objekt unter verschiedenen Namen in mehreren zusammengesetzten Objekten erscheinen und dort jeweils eine andere Sicht auf seine "kontextuellen" Attribute zeigen. Die genaue Funktionsweise dieses Sichtenmechanismus wird allerdings nicht erläutert [Levy et al. 1993, S.11,13ff]. Neben dieser Nutzung der Modelle zur Beschreibung verschiedener Objektkontexte kann jedes Modell auch die Definitionssprache für ein anderes sein. Hierzu wird das Modell als (Teil einer) Grammatikspezifikation aufgefaßt (vgl. Bild 4.13), der die erzeugten Objekte entsprechen müssen. Modelle dienen in N–DIM somit also sowohl der Objektgruppierung (=Extension) als auch der Strukturbeschränkung neuer Objekte (=striktes Template) und erfüllen somit teilweise Klassenaufgaben.

Die Objekt–Klassen–Integration einer Sprache wie HYBRID, in der Objekte problemlos in Klassen

umgewandelt werden können, bzw. selbst bereits Klassen sind, erreicht N–DIM durch diese Grammatikobjekte allerdings nicht. Nur falls die durch eine Sprache zu beschreibenden Objekte vollständig durch Links verknüpft sind und keine alternativen Produktionen existieren, stimmen nämlich Modell und Grammatik'modell' eines Objekts überein. Ansonsten treten Unterschiede auf wie im Beispiel von Bild 4.13. Die durch dieses Modell definierte Grammatik beschreibt Objekte, die aus einem oder mehreren, per preceeding-Link verknüpften Basistasks bestehen. Interpretiert man dieses Modell jedoch nicht als Sprache sondern als Objekt, so handelt es keineswegs um ein Beispielobjekt für diese Sprache. Vielmehr wird ein immer (also ohne Wahlmöglichkeit) aus drei Objekten zusammengesetztes Objekt repräsentiert, wobei lediglich zwei Objekte durch eine precedes–Relation verknüpft sind. Wie bei klassenbasierten Systemen muß hier also eine Abstraktionsleistung erbracht werden, um eine Klasse (bzw. ein Grammatikmodell) aus einem Objekt zu erstellen. Ein Nachteil der Verwendung

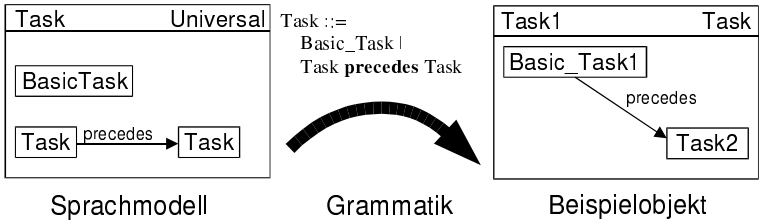


Abbildung 4.13: Sprachdefinition per Objekt (nach [Levy et al. 1993])

dieser Grammatikmodelle gegenüber Klassen ist zudem, daß die grafische Darstellung — zumindest soweit von den Autoren vorgestellt — keine Konstrukte zur Darstellung der EBNF–Abkürzungoperatoren * und + sowie des Grammatiksymbols λ enthält. Dies könnte in komplizierteren Grammatiken die Übersichtlichkeit der Darstellung weiter verschlechtern, die bereits unter der Aufspaltung einer Grammatik in ein Fenster pro Produktion leidet. Klassenbasierte Definitionen können entsprechende Zusammenhänge, wie bereits Abbildung 4.12 zeigt, wesentlich komprimierter repräsentieren.

N–DIM besitzt ein Workspace–basiertes Versionierungskonzept (vgl. etwa [Barghouti und Kaiser 1991, S.297ff]) mit privaten, öffentlichen und Bibliotheksarbeitsbereichen [n-dim Group 1995b] und kann beispielgetrieben suchen [Levy et al. 1993, S.28ff]. Das illustrierende Beispiel zu letzterem demonstriert gleichzeitig auch den bereits auf Seite 51 aufgeführten Einwand Zdoniks gegen klassenlose Sprachen: Da das verwendete Beispielmodell für die Definition von Produktentwürfen in n-dims Universal–Sprache notiert ist, die beliebige Beziehungen erlaubt (und damit prototypbasiert ist), kann nicht garantiert werden, daß ähnliche Objekte die zur Suche verwendeten Eigenschaften ("property" und "function") überhaupt besitzen.

Insgesamt stellt N–DIM sicher eine geeignete Umgebung für den explorativen Entwurf dar, was nach Angaben der Autoren auch eine der Hauptzielrichtungen ist. Zur interaktiven Suchen nach Wiederverwendungskandidaten und Unterstützung der frühzeitigen Fehlererkennung fehlen aber noch Verfahren die Generierung und Interpretation der Grammatikobjekte vereinfachen und diese somit klassenähnlicher machen.

Épée geht ähnlich wie N–DIM über die bloßen Aufgaben der Prozeßmodellierung hinaus und sieht sich als Integrations-, Werkzeug- und Archivierungsrahmenwerk für Modellierungsanwendungen. ÉPÉE ist somit in gewissen Grenzen ein CORBA–ähnlicher "Object Request Broker". In dieser Arbeit interessiert allerdings vor allem das Objektmodell, das in [Fraga et al. 1994; épée Group 1994] beschrieben wird.

ÉPÉEs Objektmodell kennt neben prototypischen Objekten noch hierarchisch angeordnete Templates (die quasi als Klassen dienen), Attribute (Slots genannt) und Methoden. Im Gegensatz zu allen anderen besprochenen Objektmodellen ist der Gültigkeitsbereich von Attributnamen in ÉPÉE nicht auf die Klasse oder das Objekt beschränkt, in dem das Attribut definiert wurde. Vielmehr existiert eine globale Liste aller vorhandenen Attribute mit ihren Typen, die jeden Attributnamen global auf

einen Typ festlegt[33]. Diese Maßnahme erlaubt eine ungewöhnlich einfache automatische Klassifizierung der Instanzen. Ein Objekt ist genau dann Instanz eines Templates, wenn es mindestens alle Attribute desselben besitzt. Templates werden hier also im Sinne von minimalen Templates benutzt, wie sie etwa in [Stein und Zdonik 1989; Stein 1987] vorgeschlagen werden. Die aus einem Template generierten Objekte können nachträglich um neue Attribute erweitert werden. Das System erlaubt somit multiple Instanziierung, da ein Objekt mehreren Templates zugeordnet sein kann, wenn es jeweils alle Attribute von diesen besitzt. Namenskonflikte der durch die Templates bereitgestellten Methoden können nicht auftreten, da Methodennamen, wie die Attributnamen, global definiert sind.

Die globale Definition der Attributnamen vereinfacht zwar die automatische Klassifizierung von Objekten beträchtlich, da nur noch die Namen der Attribute in Template und Instanz verglichen werden müssen und man somit Verfahren, die atomare Attribute voraussetzen (vgl. S. 71), verwenden kann. Sie hat aber einen schwerwiegenden Nachteil: Ein Untertemplate ist eine strikte Erweiterung seiner Obertemplates, das heißt, es kann geerbte Attribute und Methoden nicht verfeinern. Dies schränkt die Modellierungsmächtigkeit und Wiederverwendbarkeit der Templates erheblich ein.

Das Versionierungs- und Kooperationskonzept von Epee beruht ähnlich dem n–dims auf dem Workspaceparadigma mit einer privaten "scratch database" pro Nutzer und beliebig vielen öffentlichen "permanent databases", aus denen Objekte nicht gelöscht werden können. Auch in den "scratch databases" können Objekt nicht geändert, sondern nur geändert neu erzeugt werden. Ein Link zwischen alter und neuer Version des Objekts[34] wird mit der diese Änderung ausführenden Methode versehen, so daß die Nachvollziehbarkeit der Entwicklungshistorie eines Objektes in begrenztem Umfang gegeben ist.

Épées Ansatz der globalen Attributnamen zeigt Vorteile bei Speichereffizienz und Klassifizierbarkeit im Vergleich zu reinen prototypbasierten Ansätzen. Umso mehr überrascht es, daß Épée die hierdurch vorhandenen prototypbasierten Fähigkeiten nur wenig ausnutzt: Objektmodifikation kann nur durch sequentielles Hinzufügen einzelner Attribute erfolgen. Weder das Entfernen von Attributen noch die Objekt'migration', was in diesem Modell die Erweiterung eines Objekts um die Attributmenge eines neuen Templates bedeutet, werden unterstützt.

KBDS [Bañares-Alcántara 1995] sieht seine Aufgabenstellung in zwei Punkten: Erstens, Erfassung aller Entscheidungen, die im Verlauf des Modellierungsprozeß getroffen werden, und zweitens, iterative Abgrenzung und Generierung des Alternativenraumes der möglichen Modelle einer Problemstellung im Sinne der explorativen Erstellung eines Designs [ebd., S.269f]. Auf das Objektmodell wirkt sich besonders das zweite Ziel aus, das zu einem einstufigen Und/Oder–Graphen führt. KBDS kennt neben den acht auf Sprachebene definierten Basisklassen (Schema, Einheit, Ausrüstung, Metaschema und andere) nur Objekte, es handelt sich also fast um ein vollständig prototypbasiertes System. Allerdings kommt keine Delegation zum Einsatz. "Sharing" wird stattdessen wesentlich grobgranularer erreicht, indem die verschiedenen Alternativen eines Modells gemeinsame Teilobjekte verwenden.

Wie Bild 4.14 zeigt, werden die verschiedenen Fließbilder aus Einheiten ("Units") zusammengesetzt, die wiederum verschiedene Implementierungen in Form verschiedener Ausrüstungsgegenständen ("Equipment") haben können. Dieser einfache Und/Oder–Graph, wie er sich in VeDa mit Hilfe der Composite–Implementations auf Klassenbasis in beliebiger Höhe bilden läßt, dient dazu alternative Implementierungen des Modells automatisch zu generieren. Hierzu werden alle möglichen Kombinationen der Ausrüstungen erzeugt (im Bild also vier), die jeweils ein vollständiges und konsistentes Fließbild ergeben. Die Konsistenz wird hierbei durch ein "Assumption–based Truth Maintenance System"[35] aufgrund von Inkompatibilitätsbeziehungen [ebd., S.287f] und anderen Annahmen gesichert.

[33] Um es nochmals ganz explizit zu machen: In Épée wird nicht über den Typ von `Klasse.attribut` geredet, wie in anderen Sprachen, sondern über den Typ von `attribut`, der für jedes Template und Objekt gilt, in der das Attribut auftaucht.

[34] Eigentlich müßte von altem und neuem *Objekt* die Rede sein, da jede Version einen anderen Objektidentifikator besitzt.

[35] zu übersetzen in etwa mit "Verwaltungssystem für Wahrheitswerte auf der Basis von Annahmen"

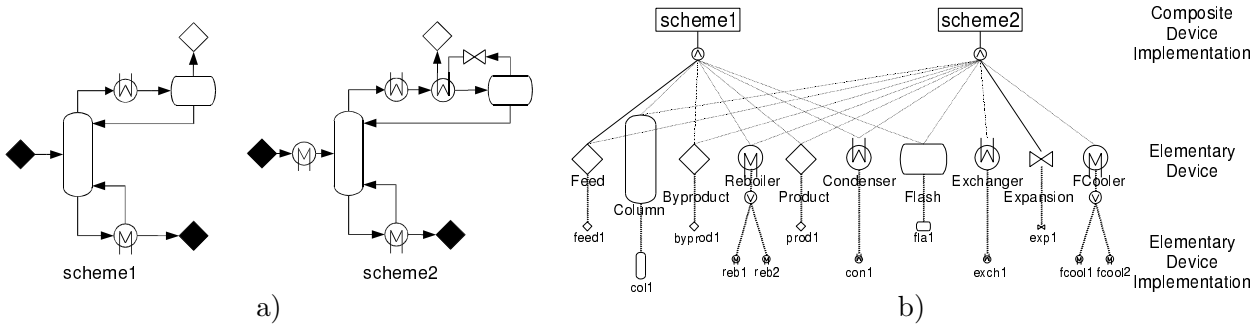


Abbildung 4.14: Die beiden Prozeßvarianten aus a) repräsentiert KBDS wie in b) [Bañares-Alcántara 1995]

Auch KBDS unterstützt Wiederverwendung lediglich innerhalb eines Projektes und auch hier nur im Rahmen der Weiterentwicklung eines Fließbildes. Weitergehende Wiederverwendung wäre auch mangels entsprechender beliebiger Aggregationen nur schwer möglich.

CLOOD [Gross-Hardt und Vossen 1993] beschäftigt sich in der Tradition der VLSI–Datenmodelle hauptsächlich mit der Versionierung von Objekten. Aufbauend auf einem prototypbasiertem Objektmodell werden Objektversionen definiert, Typeinschränkungen verschiedener Versionen des gleichen Objekts festgelegt und die Versionen ähnlich KBDS in einem, hier allerdings beliebig tiefen, Und/Oder–Baum angeordnet. Letzterer wird genutzt, um durch zwei Sichten auf ihn (Konstellations- und Konfigurationsbaum) die Übersicht über die verschiedenen Versionen eines Objekts zu erhöhen. Als reine prototypbasierte Sprache leidet CLOOD unter den ab Seite 47 aufgeführten Problemen. Auch werden die Wechselwirkung von Versionierung und Delegation nicht betrachtet, obwohl sich durch diese wahrscheinlich aufgrund der in [Bardou und Dony 1996] unterschiedenen Konzepte des "value–sharings" und des "property–sharings" die Probleme der Delegation bei Wertänderungen delegierter Attribute lösen ließen.

[Nguyen et al. 1992] definiert mit **SHOOD**, einem generischen Objektmodell für den ingenieurtechnischen Entwurf, ein sehr umfassendes Modell, das Metaklassen, mit deren Hilfe semantische Beziehungen definiert werden können, Objektmigration, multipler Instanziierung und automatische Klassifikation umfaßt. Multiple Instanziierung kann nach Ansicht der Autoren nur angewendet werden, wenn die instanziierten Klassen keine gemeinsamen Verbindung haben [ebd., S.245], wodurch sie Namenskonflikte vermeiden. Multiple Vererbung lehnen sie als Alternative bei sich beeinflussenden Klassen — zu Recht — ab und schlagen als Lösung eine Kombination von Aggregation und Multipler Instanziierung vor. Die Details dieser Lösung werden leider weder aus [Nguyen et al. 1992] noch aus ihren anderen Artikeln zu diesem Thema klar. Dies gilt ebenso für einige Details des in [Nguyen und Rieu 1992] genauer erläuterten Klassifizierungsalgorithmus: SHOOD teilt die Klassen der Klassenhierarchie bezüglich einer Instanz in mögliche und verbotene ein. Hat eine Klassifizierung erstmals stattgefunden (möglicherweise mit Hilfe der bei der Instanziierung angegebenen Klassen), so strebt der Klassifizierer nur noch möglichst spezielle Klassen unterhalb dieser Stammklasse an. Die hierbei genutzte Bewertungsfunktion bleibt im Unklaren, stützt sich aber anscheinend auf die Instanzstruktur sowie den Inkonsistenz-Grad (Anteil der verletzten weichen Restriktionen) und den Unvollständigkeitsgrad (Anteil der nicht vorhandenen optionalen Attribute) der Instanz bezüglich der Klasse. Das Ergebnis entspricht der durch implizite Objektmigration erreichbaren Klassifizierung, ohne jedoch die dort erforderliche explizite Angabe der Migrationsregeln zu erfordern.

Es hat den Anschein als biete SHOOD gute Voraussetzungen für Wiederverwendung, selbst wenn die Autoren hauptsächlich auf die Modellierung des Lebenszyklus und semantischer Relationen abzielten, wie zum Beispiel die fehlende Datenbankanbindung erkennen läßt. Die tatsächliche Eignung läßt sich allerdings aufgrund der offenen Fragen nicht beurteilen.

In [Syvertsen et al. 1992] werden Datenbank-, Management- und Ingenieurwelt verglichen und davon

ausgehend wird für ingenieurtechnische Entwurfsanwendungen ein objektorientiertes Datenmodell mit Klassen und Relationen, gemeinsamer Subtyp- und Vererbungsbeziehung, multipler, dynamischer Instanziierung mit Hilfe von Rollen sowie einem Kontext genannten Sichtenmechanismus gefordert. Leider finden sich keine genaueren Ausführungen zur Realisierung oder Formalisierung.

Kurz erwähnt werden soll hier noch EXPRESS (s. [Schenk und Wilson 1994]), handelt es sich dabei doch um die Modellierungssprache die von der ISO auch in verfahrenstechnischen Projekten wie pdXi (s. [ISO 1998]) verwendet wird. EXPRESS bietet ein objektorientiertes Modell ohne Kapselung mit diversen unspektakulären Erweiterungen wie Constraints, Modulen und abgeleiteten Attributen. Interessant ist jedoch die Verwendung der Subklassenbeziehung. Eine Klasse kann nämlich spezifizieren, daß ein Instanz zusätzlich alle, genau eine oder beliebig viele ihrer Subklassen instanziieren muß. Es handelt sich also um Multiple Instanziierung, mit einer strukturellen Restriktion über die Subklassenbeziehung. Der Eigenschaft von EXPRESS als Metamodellierungssprache gemäß werden aber Namenskonflikten einfach dadurch vermieden, daß eine Instanz in einer solcher Vererbungshierarchie aus eine Sammlung einzelner Instanzen besteht, die alle getrennt betrachtet werden müssen [ebd., S.267f & S.305ff]. Die Instanzebene wurde lediglich zur Modellierung einzelner Beispiele zur Verifikation der Informationsmodelle eingefügt, so daß auch eine Wiederverwendung nicht vorgesehen ist.

4.3.3 Verfahrenstechnische Datenmodelle und objektorientierte Wiederverwendung

Wie nach den Ausführungen in Kapitel 2.3 nicht anders zu erwarten, betonen fast alle hier betrachteten objektorientierten Datenmodelle Struktur- und Restriktionsaspekte in Form von Attributen und Constraints gegenüber dem Verhaltensaspekt in Form von Methoden. Eine Ausnahme stellt lediglich Épée dar, das aufgrund seiner Zielsetzung als komplette, werkzeugintegrierende Modellierungsumgebung natürlich Objektmethoden vollständig unterstützt. Aufgrund dieser Untergewichtung der Methoden verwundert es auch nicht, daß sich Verfahren wie die Methodenfaktorisierung von Moore (s.S. 69) oder Objektmigrationsverfahren wie das von Chambers (S. 61), dessen Hauptziel die Zuordnung neuer Methoden ist, hier nicht wiederfinden.

Für die Wiederverwendung von Objekten/Konzepten beschränken sich die meisten Verfahren auf altbekannte Ansätze. Eine objektorientierte Vorgehensweise mit Spezialisierung und Instanziierung wird zwar als wesentlich erachtet, aufgrund der meist nicht vorhandenen Datenbankanbindung kann projektübergreifende Wiederverwendung jedoch häufig nur durch Eintrag von (ggf. speziell bearbeiteten) Bausteinen in Bibliotheken oder Dateien erfolgen. Von den betrachteten Modellierungsumgebungen (Épée und n–dim) ermöglicht lediglich n–dim eine beispielgetriebene Suche über den gespeicherten Objekten.

Zur Anpassung wiederzuverwendender Objekte werden neben der objektorientierten Spezialisierung vorhandener Bausteinklassen verschiedene Ansätze verfolgt. Die Parametrisierung von Bausteinen und ihre Modifikation durch die sie enthaltenden Objekte (“Environmental Aquisition“) findet sich in Ascend, Omola, Modelica und möglicherweise Model.La. KBDS generiert modifizierte Objekte durch die automatische Komposition wechselnder Teilkomponenten. Von den beiden prototypbasierten Ansätzen unterstützt n–dim beliebige Objektmodifikationen nur ihm Rahmen der jeweils verwendeten ’Sprache’, und Épée läßt prinzipiell nur Modifikationen zu, die global bekannte Attribute verwenden. SHOOD schließlich nutzt die aus Seite 54 besprochene Multiple Instanziierung, um verschiedene, unabhängige Aspekte eines Bausteins getrennt modellieren und anschließend kombinieren zu können. Ähnlichen Zwecken dient wohl die `is-characterized-as`–Facette von Model.La.

Einordnung von Objekten in eine Hierarchie bzw. Reorganisation dieser Hierarchie unterstützen nur wenige Ansätze. ModKit verwendet für Klassifikation und Klassenreorganisation ein TWR–System mitsamt den auf den Seiten 70 und 71 beschriebenen Problemen. Épée ordnet Objekte aufgrund seiner globale Attributnamen selbständig den passenden Templates zu und SHOOD verwendet einen spezialisierenden Klassifizierer.

Eine abschließende Bemerkung zum Verhältnis von ingenieurtechnischer Modellierungssprachen und der Objektorientierung sei erlaubt. Das Beispiel KBDS zeigt, daß die alleinige Verwendung einer objektorientierten Sprache noch keine hinreichende Voraussetzung für die Wiederverwendung ist. Vielmehr ist die zugrundegelegte Modellierungs*methodologie* ebenso entscheidend. Nutzt diese die durch die Objektorientierung gegebenen Möglichkeiten nicht aus, wie etwa KBDS indem nur eine Aggregationsstufe erlaubt wird, kann die Fähigkeit zur Wiederverwendung darunter leiden.

4.4 Zusammenfassung

Dieses Kapitel hat einige grundlegende Ansätze zur Repräsentation objektorientierter Daten sowie verschiedene Realisierungen verfahrenstechnischer Objektmodelle aufgezeigt und ihre Eignung gemäß den in Kapitel 3 definierten Modifikations- und Klassifikationsoperationen untersucht. Hierbei fiel auf, daß sowohl klassenbasierte als auch prototypbasierte Ansätze prinzipielle Probleme bei der Erfüllung aller Anforderungen haben. Klassenbasierte Ansätze erlauben die Modifikation von Objekten nicht in ausreichendem Maße, prototypbasierte Ansätze besitzen keine suchunterstützende Ordnungshierarchie und benötigen deutlich mehr Platz zur Speicherung der Objekte. Diese Defizite verschieben sich bei Abwandlung der reinen Ansätze in Richtung des jeweils anderen entsprechend. Es liegt darum die Vermutung nahe, daß die Verwendung strikter bzw. loser Templates sowie die von der Delegations-/Vererbungsbeziehung sichergestellte Kompatibilität (keine Kompatibilität, Namenskompatibilität oder Signaturkompatibilität) Auslöser für die jeweiligen Probleme sind.

Die vorgestellten Erweiterungen des klassenbasierten Systems gehen in zwei Richtungen. Zum einen durch Objektmigration die Änderung der Templatebeziehung zur Laufzeit zulassen, zum anderen durch verschiedene Variationen der Subklassen–Beziehung die Generierung neuer Templates vereinfachen. Ersteres erhöht die Modifizierbarkeit, letzteres senkt die Zahl notwendiger Klassen und hebt damit die Übersichtlichkeit. Multiple Instanziierung und Rollen erlauben das 'Zusammenstellen' neuer Klassen aus Teilklassen und sehen damit am vielversprechendsten aus, wenn auch die Namenssichtbarkeitsproblematik bei Multipler Instanziierung nicht und bei Rollen nicht entsprechend dem Anwendungsgebiet gelöst ist.

Die untersuchten ingenieurtechnischen Modelle zeigten alle eine mangelnde Unterstützung der ungeplanten Wiederverwendung und insbesondere auch der Skalierbarkeit. Ein Vorgehen, wie in Kapitel 3 beschrieben, ließe sich bestenfalls begrenzt mittels ModKit, Épée und, je nach tatsächlicher Realisierung, SHOOD erreichen. Alle anderen Systeme bzw. Sprachen scheinen für ungeplante Wiederverwendung ungeeignet zu sein.

Das nächste Kapitel wird sich dementsprechend mit einem Objektmodell und zugehörigen Diensten beschäftigen, das die geforderten Anforderungen eher erfüllt, als die aufgeführten Systeme.

Kapitel 5

Klassenbasierte Modellierung und prototypbasierte Modifikation vereint

Die im letzten Kapitel vorgestellten Objektmodelle aus Informatik und Ingenieurwissenschaften zeigen alle deutliche Probleme, wenn man einerseits weitgehende Änderungen auf der Objektebene und andererseits eine strukturierende, hierarchische Anordnung der auftretenden Konzepte wünscht. Wie wir im folgenden Abschnitt zeigen werden, sind diese Probleme darauf zurückzuführen, daß die strikte Template–Beziehung, die für die hierarchische Strukturierung notwendig ist, die Änderungsmöglichkeiten der Objekte einschränkt, und daß die für die Suche vorteilhafte Signaturkompatibilität der Ordnungsbeziehung die Erweiterung der Templatehierarchie erschwert. Deswegen wird anschließend ein Konzept vorgestellt, in dem aufbauend auf einem klassenbasierten Objektmodell prototypbasierte Änderungsmöglichkeiten allein mittels entsprechender Dienste ermöglicht werden.

5.1 Die Antagonisten der Vereinigung — Strikte Templates und Signaturkompatibilität

Zur Motivation des in späteren Abschnitten vorgestellten Konzepts soll als erstes aufgezeigt werden, warum eine naive Kombination prototypbasierter und klassenbasierter Mechanismen in einem Objektmodell scheitern muß. Wie in den Kapiteln 3 und 4 ausgeführt, wäre ein solche Kombination wünschenswert, also ein Objektmodell das die besten Eigenschaften prototypbasierter und klassenbasierter Objektorientierung kombiniert: 1) Von der prototypbasierten Objektorientierung sollte die Modifizierbarkeit der Objekte erhalten bleiben, ohne sich um zugehörige Klassen kümmern zu müssen. 2) Von der klassenbasierten Objektorientierung soll die ordnende, Suche und Durchstöbern vereinfachende Hierarchie verfügbar sein. Den in Kapitel 4.1 vorgestellten Ansätze gelingt diese Kombination jedoch nicht, so daß sich die Frage stellt, ob sie prinzipiell unmöglich ist. Wie im folgenden gezeigt wird, ist dies tatsächlich der Fall, da Modifizierbarkeit und Ordnunghierarchie auf alternativen, sich wechselseitig ausschließenden Realisierungen der Templatebeziehung und Signatur/Namenskompatibilität beruhen.

Informell läßt sich folgendermaßen argumentieren: Für die vollständige Modifizierbarkeit eines Objekts o muß eine etwaig vorhandene Templatebeziehung lose sein und zu Objekten, von denen das Objekt o erbt, soll keine Kompatibilität notwendig sein (es muß also 'Cancelation' möglich sein, vgl. S. 53). Damit andererseits eine abstrakte Ordnungshierarchie Sinn macht, bedarf es einer strikten Templatebeziehung, damit sich die in der Abstraktion gefundenen Eigenschaften auch in den Objekten wiederfinden. Zudem muß innerhalb der Hierarchie Signaturkompatibilität existieren, damit unpassende Teilbäume bei Suchvorgängen abgeschnitten werden können. Dies läßt sich in Tabelle 5.1 sehen, in der die für Modifizierbarkeit wünschenswerten Eigenschaften in der rechten Spalte zu finden, sind während sich diejenigen für Ordnungshierarchien in der mittleren Spalte befinden.

Template-Art	strikt	lose
Auswirkung	Ordnung auf Templates impliziert Ordnung auf Objekten	einfache Strukturveränderung eines Objekts
Ordnungsrelationart	signaturkompatibel	höchstens namenskompatibel
Auswirkung	Suchraum läßt sich einschränken	Einordnung neuer Konzepte einfach
wünschenswert für	Suche/Stöbern	Modifikation

Tabelle 5.1: Auswirkungen unterschiedlicher Objektrealisierungen

Obige Aussagen lassen sich formalisieren, indem man sie auf die Attributbeziehung *Attr*, die Hierarchiebeziehung $\leq$ und die Template–Beziehung $Templ$ zurückführt:

Definition 5.1 (Grundlegende Beziehungen) *Sei Klasse die Menge aller Klassen, Name die Menge aller Attributnamen und Objekt die Menge aller Instanzen bzw. die Menge aller Objekte, wenn es keine Klassen gibt. Zeit enthalte diskrete Zeitpunkte. Es gelte* $Objekt \cap Klasse = \emptyset$. *Desweiteren existieren folgende Relationen:*
Die Attributbeziehung

$$Attr : Klasse \times Name \times Klasse$$
$$\textit{bzw.} \quad Attr : Objekt \times Name \times Objekt$$

mit $Attr(k, m, t)$ *: die Klasse bzw. das Objekt* k *hat ein Attribut* m*, das auf die Klasse bzw. Objekt* t *verweist. Die Attributbeziehung ist eindeutig, das heißt* $Attr(k, m, t) \wedge Attr(k, m, s) \Rightarrow t = s$

Die Templatebeziehung

$$Templ : Klasse \times Objekt$$
$$\textit{bzw.} \quad Templ : Objekt \times Objekt$$

mit $Templ(k, i)$ *: die Klasse bzw. das Objekt* k *ist Template für das Objekt* i*. Jedem Objekt ist hierbei mindestens ein Template zugeordnet.*

Die Ordnungsrelation

$$\leq: Klasse \times Klasse$$
$$\textit{bzw.} \quad \leq: Objekt \times Objekt$$

wobei $\leq$ *die Klassen bzw. die Objekte in einem zyklenfreien (außer reflexiven Zyklen) Graphen anordnet.*

In einigen der folgenden Formeln ist der Gültigkeitszeitpunkt der Beziehungen von Bedeutung. Wir verwenden der Einfachheit halber diskrete Zeitpunkte $z \in Zeit$ *und erweitern obige Relationen jeweils um eine zusätzliche Zeitkomponente (also z.B.* $Attr : Klasse \times Name \times Klasse \times Zeit$*) die besagt, daß die entsprechende Relation zum Zeitpunkt* z *gültig ist.*

Zur Vereinfachung der Schreibweise definieren wir zudem $type(j) = l :\Leftrightarrow Templ(l, j)$.

Ausgehend von diesen Relationen lassen sich die Striktheit der Templatebeziehung sowie signaturkompatible Hierarchiebeziehungen definieren[1]. Im folgenden werden sie nur für klassenbasierte Systeme angegeben, die Definition für prototypbasierte Systeme ergibt sich analog durch die Ersetzung von $Klasse$ durch $Objekt$.

[1] Leider geben die Schöpfer dieser Begriffe weder in [Stein et al. 1989] noch in [Wegner und Zdonik 1988] eine formale Definition an, so daß hier eine Definition aufgrund der textuellen Beschreibung und der Beispiele in beiden Quellen vorgenommen wird.

Definition 5.2 (Strikte Templatebeziehung) *Die Templatebeziehung Templ ist genau dann strikt, wenn ein Objekt exakt die Attribute hat, die ihm durch sein Template vorgegeben sind, d.h. wenn gilt*

$$\forall k \in Klasse, o, p \in Objekt, m \in Name : Templ(k, o) \wedge Attr(o, m, p) \Rightarrow \quad (5.1)$$
$$\exists l \in Klasse : Attr(k, m, l) \wedge type(p) \leq l$$
$$\forall k, l \in Klasse, o \in Objekt, m \in Name : Templ(k, o) \wedge Attr(k, m, l) \Rightarrow \quad (5.2)$$
$$\exists p \in Object : Attr(o, m, p) \wedge type(p) \leq l \ [2]$$

Eine analoge Bedingung gilt, falls Objekte als Templates dienen.

Formel (5.1) fordert also, daß Attribute, die in einem Objekt vorkommen, schon in seinem Template (hier: der Klasse) passend definiert sind. Umgekehrt fordert (5.2), daß alle Attribute des Templates auch entsprechend in den Instanzen des Templates realisiert sind.

Definition 5.3 (Signaturkompatible Hierarchiebeziehung '$\leq$') *Damit die Hierarchiebeziehung $\leq$ die Signatur-Kompatibilität erhält, muß sie der Spezialisierungs- bzw. der Untertyprelation entsprechen. In einer Klassenhierarchie mit Attributspezialisierung muß gelten*

$$\forall k, l, t \in Klasse, m \in Name : k \leq l \wedge Attr(l, m, t) \Rightarrow$$
$$\exists u \in Klasse : Attr(k, m, u) \wedge \quad (5.3)$$
$$u \leq t \quad (5.4)$$

Die Bedingung für eine signaturkompatible Hierarchie zerfällt somit in die Teile (5.3) und (5.4). Der erste Teil entspricht der von Signaturkompatibilität implizierten Namenskompatibilität, der zweite Teil (5.4) enthält die über die Namenskompatibilität hinausgehende Forderung der Signaturkompatibilität. Je nach Definition dieser Kompatibilität (vgl. S. 25 und [Wegner und Zdonik 1988]) handelt es sich entweder um die Attributspezialisierung zulassende Restriktion oder um die Untertyprestriktion. In obiger Definition wurde die Attributspezialisierung gewählt. Für die Untertyprestriktion, muß 5.4 $u = t$ lauten.

Wie man sieht, ähnelt Definition 5.3 der Forderung (5.2) aus Definition 5.2. Der Unterschied zwischen den beiden Definitionen liegt in den betroffenen Beziehungen ('Clone' bzw. 'InstanceOf' bei den strikten Templates, 'Delegation' bzw. 'Inheritance' bei der Signaturkompatibilität) und der bei der Signaturkompatibilität nicht auftretenden Umkehrung der Existenzforderung (vgl. (5.1)), welche die Existenz zusätzlicher Attribute in der Instanz gegenüber dem Template verbietet.

Definition 5.4 (Most–specific Template) *Ein Template k ist dann 'most–specific' für ein Objekt o, wenn gilt:*

$$Templ(k, o) \wedge$$
$$\forall l \in Klasse : l \leq k \Rightarrow \neg Templ(l, o)$$

5.1.1 Vereinfachung der Anfrageauswertung

Es soll nun gezeigt werden, daß die Eigenschaften in der mittleren Spalte von Tabelle 5.1 hinreichende und notwendige Bedingung für eine vereinfachte Suche sind.

Sind in einem Objektmodell mit strikten Templates und Signaturkompatibilität die Objekte jeweils ihren most-specific Templates zugeordnet, so lassen sich Suchoperationen auf den Objekten vereinfachen, indem die dann geltenden, oben definierten Restriktionen genutzt werden.

[2] Der Einfachheit halber sei angenommen, daß optionale Attribute grundsätzlich über die Kardinalität des Typs realisiert werden und nicht über die An-/Abwesenheit des Attributs im Objekt.

Vereinfachung durch strikte Templates

Strikte Templates implizieren die Äquivalenz der Attribute von Klasse und Objekt und eine Untertypbeziehung zwischen den Typen der äquivalenten Attribute. Dadurch kann die typische Suchoperation

$$\text{Suche alle Objekte } o \text{ mit } Attr(o, m, x) \text{ und } Attr(o, n, y).$$

(wobei x und y typischerweise nicht als feste Objekte sondern als Klassen t und u ($type(x) = t$ und $type(y) = u$) angegeben werden[3]) vereinfacht werden zu

$$\text{Suche Klassen } k, \text{ mit } Attr(k, m, t) \text{ und } Attr(k, n, u).$$

Man sucht also Klassen, die Template für das gesuchte Objekt o sein könnten. Die Instanzen dieser Klassen und aller in der Hierarchie unter ihnen stehenden Klassen sind dann mögliche Kandidaten für o. Im weiteren muß also nur noch innerhalb dieser Objekte gesucht werden, wodurch sich der Suchraum gegenüber einer auf Objekten basierenden Suche einschränken läßt.

Im folgenden wird gezeigt, daß die Menge der Objekte der auf obige Art erstellten Kandidatenmenge, alle Kandidaten für die Beantwortung der ursprünglichen Anfrage enthält.

Satz 5.5 *Sei* $A^*_{m,t} = \{o \in Objekt | Templ(k, o) \wedge Attr(k, m, r) \wedge q \leq r \wedge q \leq t$[4]$; k, q, r \in Klasse\}$ *mit* $m \in Name, t \in Klasse$. $A^*_{m,t}$ *ist also die Menge der Instanzen, die sich bei der Suche nach Klassen mit Attributen* m *mit Typ* t *ergibt. Sei weiterhin* $B_{m,t} = \{o \in Objekt | Attr(o, m, p) \wedge type(p) = s \leq t, p \in Objekt, s \in Klasse\}$ *die Menge der Objekte, die sich bei der Suche nach Objekten mit Attributen* m *vom Typ* t *ergibt. Wenn* $Templ$ *eine strikte Template–Beziehung ist und* $\leq$ *eine signaturkompatible Hierarchiebeziehung dann gilt* $B_{m,t} \subset A^*_{m,t}$.

Beweis:
Sei $o \in B_{m,t}$. *Aufgrund der Definition von* $B_{m,t}$ *gibt es dann ein* $p \in Objekt$ *mit* $type(p) = s, s \leq t$. *Aufgrund der Striktheit der Templatebeziehung (Gleichung (5.1)) gibt es dann eine Klasse* k *mit* $Templ(k, o)$ *und* $Attr(k, m, l)$ *wobei* $s \leq l$. *Da außerdem* $s \leq t$, *gilt* $o \in A^*_{m,t}$. □

Was wären nun gute Ausgangsklassen zum Stöbern? Sei zusätzlich $A_{m,t} = \{k \in Klasse | Attr(k, m, r) \wedge q \leq r \wedge q \leq t, q, r \in Klasse\}$, also die Menge der zu $A^*_{m,t}$ gehörigen Klassen. Wählt man aus $A_{m,t}$ die Suprema bezüglich $\leq$, also die Menge $C_{m,t} = \{k \in A_{m,t} | \neg\exists l \in A_{m,t} : k \leq l\}$, so enthält $C_{m,t}$ gerade die Kandidatenklassen, die als Ausgangspunkt für eine Suche über den Objekten dienen können.

Lemma 5.6 *Ohne strikte Templates kann es hingegen Klassen* k *und Objekte* i *geben, so daß*

$$\begin{aligned} &\exists o, p \in Objekt, m, l \in Name, t, s \in Klasse : Templ(k, i) \wedge \\ &\quad Attr(i, m, o) \wedge \neg Attr(k, m, t) \wedge && (5.5) \\ &\quad Attr(k, n, s) \wedge Attr(i, n, p) \wedge s \neq type(p) && (5.6) \end{aligned}$$

In diesem Fall kann ein Objekt also mehr Attribute haben, als sein Template vorgibt (5.5) oder der Klasse angehörige Attribute mit anderem Typ besitzen (5.6). Eine Suche nach Instanzen über die Klassen wird somit unmöglich.

[3] Etwa "Suche alle Prozeßkomponenten, deren Geometrie zweidimensional spezifiziert ist und in denen eine Reaktion vorkommt".

[4] Bedeutung und Folgen der ungewöhnlich aussehenden Bedingung $q \leq s \wedge q \leq t$ werden in Kapitel 7.1.2 genauer erläutert.

Vereinfachung durch Signaturkompatibilität

Bei der Suche nach den Klassen, die ein Template des gesuchten Objekts sein können, kann aufgrund der Signaturkompatibilität die hierarchische Anordnung der Klassen genutzt werden, um die Zahl der zu betrachtender Klassen einzuschränken: Da für alle Oberklassen der gesuchten Klasse(n) Forderung (5.4) gilt, braucht die Klassenteilhierarchie unter einer Klasse k' nicht mehr durchsucht zu werden, sobald in ihr ein Attribut m existiert mit $Attr(k', m, t')$ und t' und t keine gemeinsame Unterklasse besitzen. Formal:

Satz 5.7 *Für gegebene $m \in Name$ und $t \in Klasse$ sei k eine Klasse für die gilt:*

$$\forall r \in Klasse\ \neg\exists q \in Klasse : Attr(k, m, r) \wedge (q \leq r \wedge q \leq t) \tag{5.7}$$

Dann gilt (5.7) auch für alle Klassen $l \leq k$, d.h. für alle solche l gilt $l \notin A_{m,t}$.
Beweis:
Angenommen es existiert ein $l \leq k$ für das (5.7) nicht gilt, d.h. $\exists q' \in Klasse : Attr(l, m, r') \wedge q' \leq r' \wedge q' \leq t$. Wegen (5.4) gilt $r' \leq r$ für r aus $Attr(k, m, r)$. Damit gilt aber auch $q' \leq r$, womit im Wiederspruch zu (5.7) für k ein $q \in Klasse$ existierte. □

Die Signaturkompatibilität ist für dieses Vorgehen eine notwendige Voraussetzung, da ohne sie keine Aussagen über den Zusammenhang von Attributtypen in Ober- und Unterklassen möglich sind. Namenskompatibilität, die nächst niedrigere Kompatibilitätsstufe, kann nur noch zur Einschränkung der Suche benutzt werden, wenn *alle* Attribute des gesuchten Objekts bekannt sind. In diesem Fall lassen sich die Unterklassen aller Klassen ausschließen, die Attribute besitzen, die im gesuchten Objekt nicht vorhanden sind. Allerdings dürfte dieser Fall bei der Suche nach wiederverwendbaren Objekten eher selten auftreten. Ist in der Ordnungshierarchie 'Cancelation' erlaubt, so lassen sich keinerlei Einschränkungen bei der Templatesuche aufgrund der Ordnungshierarchie durchführen.

5.1.2 Modifizierbarkeit

Als nächstes wird gezeigt, daß die Eigenschaften der mittleren Spalte von Tabelle 5.1 mit einer einfachen Modifizierbarkeit unvereinbar sind.

Während strikte Templates und Signaturkompatibilität die Suche vereinfachen, behindern sie andererseits die einfache Änderbarkeit von Objekten, indem sie Strukturänderungen eines Objekts von der Struktur anderer Objekte abhängig machen. Am deutlichsten tritt dies bei strikten Templates hervor, bei denen aufgrund von Def. 5.2 zu allen diskreten Zeitpunkten[5] gilt:

Lemma 5.8 *Sei $Templ$ eine strikte Templatebeziehung. Dann gilt*

$$\begin{aligned}
&\forall y, z \in Zeit, k, t \in Klasse, m \in Name, o \in Objekt : \\
&\quad y < z \wedge Templ(k, o, y) \wedge Templ(k, o, z) \wedge Attr(k, m, t, y) \wedge Attr(k, m, t, z) \Rightarrow \\
&\quad (\exists h, i \in Objekt : Attr(o, m, h, y) \wedge Attr(o, m, i, z) \wedge && (5.8) \\
&\quad type(h) \leq t \wedge type(i) \leq t) && (5.9)
\end{aligned}$$

Beweis:
Folgt direkt durch Einsetzen des um die Zeitkomponente erweiterten (5.2). □

Die möglichen Änderungen an o sind also eng begrenzt. Attribute können weder hinzukommen noch gelöscht werden (5.8), und auch die einem Attribut zuweisbaren Objekte bleiben immer durch das strikte Template beschränkt (5.9). Das Lemma zeigt allerdings auch die Ansatzpunkte, um Änderungen an den Objekten durchführen zu können, ohne auf strikte Templates verzichten zu müssen:

[5] Vgl. die Zeiterweiterung der Basisrelationen am Ende von Def. 5.1

1. Die Attributbeziehung $Attr(k, m, t, z)$ kann für verschiedene z unterschiedliche t (einschließlich "undefiniert") annehmen, d.h. es können Änderungen in den Klassen vorgenommen werden. Die Möglichkeiten hierfür sind durch die Signaturkompatibilität der $\leq$–Beziehung eingeschränkt (vgl. unten).
2. $Templ(k, i, z)$ kann für verschiedene z unterschiedliche k annehmen, d.h. einem Objekt werden im Zeitablauf unterschiedliche Templates zugeordnet. Dies entspricht der in Kapitel 4.1.4 beschriebenen Objektmigration.

Die Signatur- oder Namenskompatibilität der $\leq$–Beziehung schränkt die möglichen Änderungen auf zwei Arten ein. Erstens werden durch sie, wie oben erwähnt, Templateänderungen eingeschränkt, sobald das zu ändernde Attribut bereits in einem übergeordneten Template definiert ist. Wegen (5.3) kann bei Namens- und Signaturkompatibilität ein in einem Obertemplate vorkommendes Attribut im Untertemplate nicht gelöscht werden, durch (5.4) schränkt das Obertemplate die möglichen Typen des Attributs im Untertemplate ein.

Die zweite Änderungseinschränkung ergibt sich in prototypbasierten Sprachen auch ohne strikte Templates. Bei ihnen dient die Delegationsbeziehung als Ordnungsbeziehung und fordert dann meist Namenskompatibilität (vgl. S. 48 f). Dies bedeutet aufgrund (5.3), daß in einem Objekt manche Attribute nicht gelöscht werden können, ohne daß ein neuer Delegationspartner gesucht wird.

5.1.3 Unvereinbarkeit der Paradigmen

Abschnitte 5.1.1 und 5.1.2 haben also gezeigt, daß maximale Modifizierbarkeit im Widerspruch zu strikten Templates und einer signaturkompatiblen Ordnungsrelation steht, während andererseits die Such- und Stöbervorteile einer durch die Ordnungsrelation induzierten Hierarchie nur wirksam werden, wenn die Relation signaturkompatibel und die Templatebeziehung strikt ist. Somit kann also prinzipiell ein Objektmodel nicht durch Kombination klassen- und prototypbasierter Mechanismen beide Anforderungen erfüllen. Allerdings schließt diese Unvereinbarkeit nicht aus, daß für den Benutzer der Eindruck erweckt wird, ein Objektmodell würde beiden Paradigmen angehören. Dies läßt sich etwa durch die Emulation eines Modells auf der Grundlage des anderen erreichen.

5.2 Emulation eines prototypbasiertem auf einem klassenbasierten Modell

Dieser Abschnitt stellt als Lösung der oben aufgeführten Unvereinbarkeit der Vorteile von prototypbasierten und klassenbasierten Objektmodellen eine Emulation ein. Hierbei werden prototypbasierter Änderungsoperationen in einem klassenbasierten Modell emuliert. Dies verbindet für den Benutzer die Möglichkeit beliebiger Änderungen mit den Vorteilen abstrakter Wissensbeschreibung. Die Schnittstellen, Funktionsweise und Probleme der Emulation werden im folgenden beschrieben.

5.2.1 Benutzerinteraktion

Bei der Emulation eines prototypbasierten Objektmodells in einem klassenbasierten entsteht das Problem, daß Operationen für strukturelle Änderungen sowohl auf der Objekt- als auch auf der Klassenebene existieren. Um jedem Benutzer ein konsistentes, redundanzfreies Interface zu bieten, werden darum zwei getrennte Interfaces angeboten: Ein klassenbasiertes und ein prototypbasiertes, je nachdem welche Arten von Änderungsoperationen bevorzugt werden. Operationen zum Durchsuchen der Klassenhierarchie stehen in beiden Schnittstellen zur Verfügung.

Abbildung 5.1 zeigt Ausschnitte aus beiden Schnittstellen und einige Zusammenhänge zwischen ihnen. Angeboten werden sowohl modifizierende als auch rein betrachtende Operationen. Ein auf dem prototypbasiertem Modell arbeitender Benutzer erhält die volle Kontrolle über Attributwerte

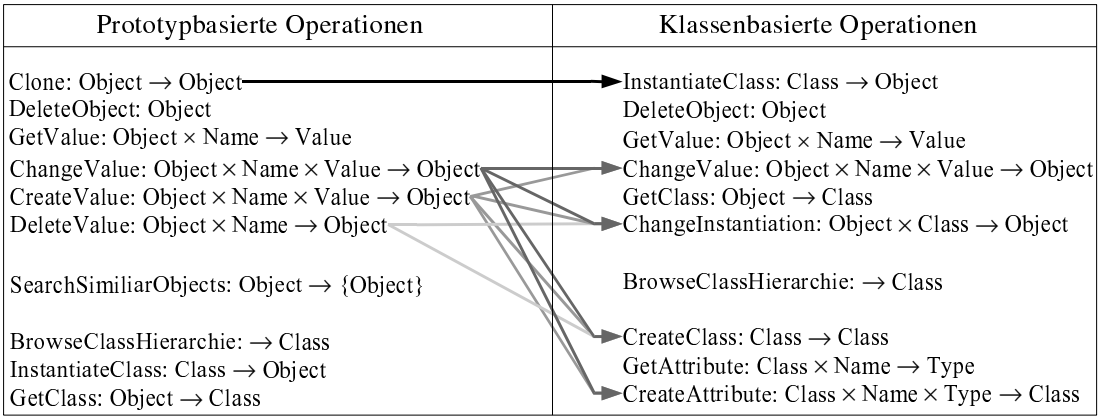


Abbildung 5.1: Abbildung prototypbasierter Operation auf klassenbasierte

und -struktur der Objekte. Neue Objekte kann er nicht nur aus existierenden Objekten generieren sondern auch aus Klassen instanziieren, da die vorhandene Klassenhierarchie für ihn sichtbar bleibt. Das bereitgestellte prototypbasierte Modell kennt keine Delegationsbeziehung, für den Benutzer erscheinen deshalb alle Attribute im Objekt definiert und mit einem Wert versehen. Der Verzicht auf Delegation wird möglich, da die durch die Delegation hervorgerufene Speicherplatzersparnis (vgl. Tabelle 4.3) durch das klassenbasierte Grundmodell in ähnlicher Weise sichergestellt wird. Der Verzicht auf Delegation vermeidet die auf Seite 46 aufgeführten Probleme mit der bei größeren Änderungen unter Umständen notwendigen Nachführung der Delegationsbeziehung.

Die klassenbasierte Schnittstelle bietet direkten Zugriff auf das zugrundeliegende klassenbasierte Objektmodell (vgl. auch Abbildung 5.2). Wie bei der prototypbasierten Schnittstelle werden betrachtende und modifizierende Operationen angeboten. Im Gegensatz zur prototypbasierten Schnittstelle muß jedoch zwischen den auf Klassen und den auf Objekten arbeitenden Operationen unterschieden werden. Erstere erlauben nur einen Zugriff auf Wertinformation, letztere nur auf Typinformation. Deutlich wird dieser Unterschied z.B. bei der `ChangeValue` Operation. Die prototypbasierte `ChangeValue` Operation erlaubt es, beliebige Werte in das spezifizierte Attribut des Objekts einzutragen, selbst wenn diese nicht mit dem derzeitigen Typen des Attributs übereinstimmen.

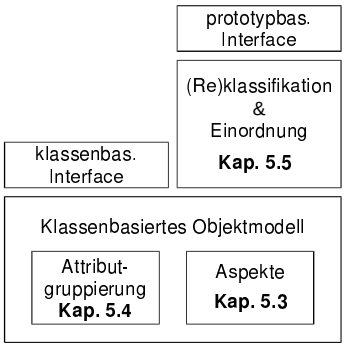


Abbildung 5.2: Interfaces und Emulationskomponenten

In letzterem Fall wird der Typ das Attributs und damit des Objekts implizit mitgeändert, d.h. es werden automatisch `ChangeInstantiation` und ggf. `CreateClass` aufgerufen. Die klassenbasierte Version von `ChangeValue` hingegen überprüft die Typkorrektheit des eingefügten Attributs und erlaubt somit nur Wert- aber keine Typänderungen. Typänderungen erfordern im klassenbasierten Interface entweder Schemamodifikationsoperationen wie `ChangeAttributeType`, die für normale Benutzer nicht verfügbar sind, oder das Erzeugen einer neuen Klasse mittels `CreateClass` mit den entsprechend geänderten Attributen und das anschließende Umhängen des Objekts.

Die Pfeile in Abbildung 5.1 repräsentieren die Abbildung prototypbasierter Operationen auf klassenbasierte. Selbstverständlich müssen *alle* Operationen des prototypbasierten Interfaces auf klassenbasierte abgebildet werden und nicht nur die mit Pfeilen versehenen. Die strukturverändernden Operationen in der Mitte der Abbildung sind jedoch besonders interessant. Ihnen widmet sich der nächste Abschnitt.

5.2.2 Emulation durch Migration der Objekte

Da alle Information intern in einem klassenbasierten System gespeichert wird, müssen prototypbasierte Operationen wie das Modifizieren der Attributstruktur eines Objektes auf klassenbasierte Operationen abgebildet werden. Strukturveränderungen, die in prototypbasierter Objektorientierung durch direkte Manipulation der Objekte durchgeführt werden, müssen in klassenbasierten Systemen durch eine Kombination von Objekt- und Klassenmanipulationen realisiert werden. Dies ist notwendig, da Objekte nur im Rahmen der durch die Klasse vorgegebenen Struktur verändert werden dürfen.

Änderungsoperationen auf den Objekten erfordern, daß das Objekt reklassifiziert wird, also einer neuen Klasse zugeordnet (migriert) wird [Gottlob et al. 1996]. Dies entspricht der im letzten Abschnitt motivierten Lösung des Modifikation vs. Änderbarkeit–Dilemmas durch strikte aber auswechselbare Templates (vgl. S.89). Problematisch ist, daß viele Objektänderungen jeweils Klassen erfordern, die noch nicht in der Klassenhierarchie enthalten sind, und somit neu zu erzeugen sind. Durch Klassifizierungs- und Klassenorganisationsmethoden kann, wie in Kapitel 4.2.2 beschrieben, zwar erreicht werden, daß redundante Information in den Klassen minimiert wird. Die Zahl der entstehenden Klassenvarianten und Vereinigungsklassen kann dennoch schnell unüberschaubar werden, da ähnliche oder gar äquivalente Prozeßkomponenten auf eine Vielzahl unterschiedlicher Arten modellierbar sind. Der nächste Abschnitt geht auf eine Lösung dieses Problems ein.

5.2.3 Multiple Instanziierung statt kombinatorischer Klassenexplosion

Die Erläuterungen auf Seite 54 zeigten, daß die durch Vereinigungsklassen hervorgerufene kombinatorische Explosion der Klassenzahl vermieden werden kann, wenn Multiple Instanziierung zugelassen wird, ein Objekt also mehrere Templates besitzen kann. Abbildung 5.3 zeigt diesen Übergang ab-

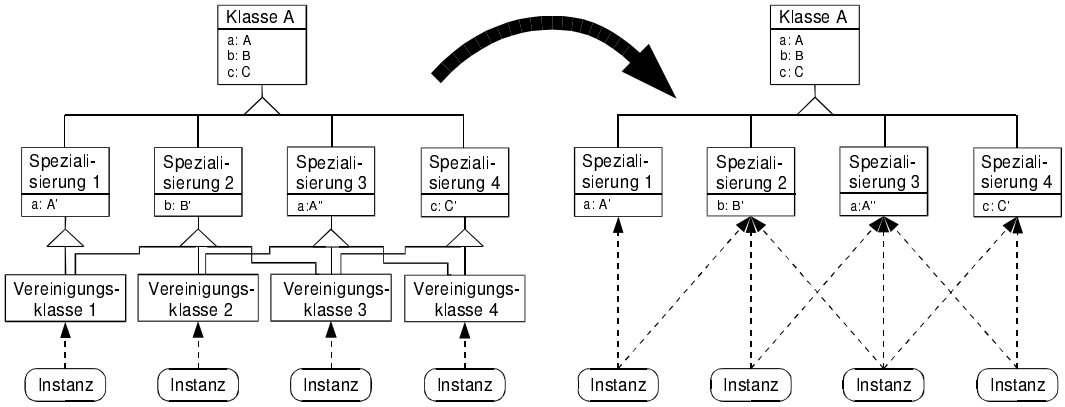


Abbildung 5.3: Übergang von Vereinigungsklassen zu Multipler Instanziierung

strakt. Die Transformation erfolgt, indem man auf die Vereinigungsklassen verzichtet und die Kombination der Spezialisierungen auf Instanzebene ausgeführt wird. Diese Verlagerung läßt sich auch bei der Emulation prototypbasierter Operationen in einem klassenbasiertem System nutzen. Strukturelle Änderungen an Objekten müssen bei der Emulation jeweils durch entsprechende neue Attribute in den Klassen nachgebildet werden, etwa indem gemeinsam hinzugefügte Attribute jeweils zu einer Spezialisierungsklasse zusammengefaßt werden. Alternative Änderungen benötigen dann wie in Abbildung 5.3 links zahlreiche Vereinigungsklassen. Wird multiple Instanziierung zugelassen, können hingegen wie rechts in Abbildung 5.3 die Vereinigungsklassen entfallen.

Auf Seite 54 wurde gezeigt, daß Multiple Instanziierung Restriktionen notwendig macht, damit die Konsistenz der erstellten Objekte gewahrt wird. Abschnitt 4.1.4 stellte hierfür Transitions- und Zustandsrestriktionen vor.

Transitionsrestriktionen legen fest, auf welche Weise ein Objekt seine Instanziierungsbeziehungen zu Klassen ändern darf, also welche Übergänge zwischen Klassen erlaubt sind. Die als konsistent betrachteten Kombinationen von Klassen, also die Zustandsrestriktionen, lassen sich daraus ableiten. Versuche mit einer den 'Category Classes' aus [Odberg 1994] ähnlichen Erweiterung von VeDa führen zu Modellen wie in Abbildung 5.4. Hierbei werden mögliche Transitionen in der Klasse angegeben, in die sie hinein führen. `gain_from K` bezeichnet hierbei eine Transition, die ein Objekt die Klassen zusätzlich annehmen läßt, wenn es bereits Instanz der Klasse `K` ist, während es bei `move_from K` die Instanziierungsbeziehung zu `K` aufgeben muß. Im Modell von Abbildung 5.4 sind sowohl die Transitionen gemäß der Spezialisierungsbeziehungen als auch die Hinzunahme der Implementierung zum Device–Modell durch Transitionsrestriktionen modelliert. Allerdings führt dieses bereits auf abstrakten Niveau zu nur schwer verständlichen Modellen: Die Gleichwertigkeit von Unterklassen- und Transitionsbeziehungen erschwert die Entscheidung, ob ein Klasse Unterklasse einer anderen sein soll oder mit ihr zusammen instanziiert werden soll. Zudem können ererbte Transitionsbeschränkungen für den Benutzer überraschend wirken. So kann in Abbildung 5.4 ein `REACTOR` nur dann eine `REACTOR_COMP_IMPLEMENTATION` annehmen, wenn er nicht bereits Instanz von `REACTOR_ELEM_IMPLEMENTATION` ist. Diese Restriktion wird von den Klasse `COMPOSITE_IMPLEMENTATION` und `ELEMENTARY_IMPLEMENTATION` geerbt und kann für den Benutzer also überraschend kommen, da sie in `REACTOR_COMP_IMPLEMENTATION` nicht explizit ist. Eine Materialisierung aller geerbten Übergangsvorschriften und -restriktionen führt hingegen zu Redundanzen und einer Aufblähung der Klassen. Als weiteres Problem erscheint zudem die Notwendigkeit alle erlaubten Übergänge zwischen Klassen angeben zu müssen. In Beispielen, denen ein Objektlebenszyklus zugrunde liegt, wenn also etwa ein Objekt der Klasse Person erst zusätzlich die Klasse Schüler, dann Student und schließlich Angestellter annimmt, stellt dies aufgrund der meist linearen Struktur und der relativen geringen Zahl der Übergänge kein Problem dar. Verwendet man Multiple Instanziierung jedoch zur Emulation von Änderungsoperationen auf Objekten so müssen weitaus mehr Übergänge erlaubt werden, sollen nicht strukturelle Änderungen an den Objekten praktisch unmöglich gemacht werden. Entsprechend wächst natürlich auch die Zahl der anzugebenden Transitionen.

Zustandsrestriktionen kennzeichnen bestehende Klassenkombinationen einer Instanz als konsistent oder inkonsistent. Daraus lassen sich die erlaubten Übergänge zwischen den Klassen ableiten. Um die Instantiierungsmöglichkeiten bei der Emulation prototypbasierter Änderungsoperationen einzuschränken, eignen sie sich aus Gründen der strukturelle Repräsentation und der Konsistenzwiederherstellung besser:
Zustandsrestriktionen können ähnlich den Transitionsrestriktionen aus einer Aufzählung aller erlaubten Kombinationen bestehen, aber meist werden sie wesentlich kompakter repräsentiert: durch logische Formeln oder indem *strukturelle Zusammenhänge* der zugrundeliegenden Klassenhierarchie genutzt werden (vgl. Abschnitt 4.1.4). Im Normalfall bedeutet dies, daß nur Ober- und Unterklassen gemeinsam instanziiert werden dürfen. Statt Übergänge zwischen allen Unterklassen einer Klasse explizit zu repräsentieren, wie es bei Transitionsrestriktionen notwendig ist, müssen die Unterklassen lediglich mit der Oberklasse verbunden werden. Dadurch daß konsistente Zustände und nicht die Übergänge zwischen ihnen repräsentiert werden, kann das Objektsystem zudem versuchen, einen *neuen konsistenten Zustand* zu finden, wenn der Benutzer bei seinen Modifikationen auf einer Konsistenzverletzung besteht, etwa indem er einem Reaktor einen Kolonnenkopf hinzufügt.

Die für eine einfache strukturelle Repräsentation der Zustandsrestriktionen und eine Lösung des Namenskonfliktproblems der Multiplen Instanziierung entworfenen Erweiterungen des objektorientierten Modells werden in Kapitel 5.3 beschrieben.

```
class:                  DEVICE_IMPLEMENTATION
metaclasses:            concept
superclasses:           SUBSTANTIAL_MODELING_CONCEPT
gain_from:              DEVICE

class:                  ELEMENTARY_IMPLEMENTATION
metaclasses:            elementary-concept
superclasses:           DEVICE_IMPLEMENTATION
gain_from:              DEVICE :if :not COMPOSITE_IMPLEMENTATION
move_from:              COMPOSITE_IMPLEMENTATION
move_from:              IMPLEMENTATION
...-attributes:         ...

class:                  COMPOSITE_IMPLEMENTATION
metaclasses:            composite-concept
superclasses:           DEVICE_IMPLEMENTATION
gain_from:              DEVICE :if :not ELEMENTARY_IMPLEMENTATION
move_from:              ELEMENTARY_IMPLEMENTATION
move_from:              IMPLEMENTATION
...-attributes:         ...

class:                  REACTOR
metaclasses:            composite-concept
superclasses:           DEVICE
move_from:              DEVICE
...-attributes:         ...

class:                  REACTOR_COMP_IMPLEMENTATION
metaclasses:            composite-concept
superclasses:           COMPOSITE_IMPLEMENTATION
gain_from:              REACTOR
...-attributes:         ...

class:                  REACTOR_ELEM_IMPLEMENTATION
metaclasses:            composite-concept
superclasses:           ELEMENTARY_IMPLEMENTATION
gain_from:              REACTOR
...-attributes:         ...
```

Abbildung 5.4: Mögliche Transitionsrestriktionen in VeDa

5.2.4 Lokale und globale Einordnung und Reorganisation

Beim zugrundeliegenden Objektmodell des vorgeschlagenen Ansatzes handelt es sich um ein klassenbasiertes Modell ohne 'Cancelation'[6]. Objekte müssen also migrieren, will man ihre Attributstruktur ändern. Wie Abschnitt 4.2.2 zeigte, darf die Problematik dieser Migration nicht unterschätzt werden. Die dort vorgestellten Verfahren zur Einordnung von Objekten und Klassen in eine Klassenhierarchie gehen alle Kompromisse ein, um die Komplexität der Aufgabe zu begrenzen. Entweder werden nicht alle objektorientierten Fähigkeiten berücksichtigt oder es werden Minimierungsziele vereinfacht. Für ein bei dieser Art von Emulation zu verwendendes Verfahren bieten sich dementsprechend folgende Einschränkungen an:

- Es wird zwischen lokaler Einordnung und globaler Reorganisation unterschieden. Im Verlauf von Änderungen wird ein Objekt ausgehend von den vorhandenen Klassen nur lokal neu eingeordnet. Auch wird die bestehende Klassenhierarchie nur eng begrenzt reorganisiert. Zum einen kann dadurch die bei Änderungen notwendige Neueinordnung schneller ausgeführt werden, so daß dem Benutzer ständig die aktuelle Einordnung angezeigt werden kann. Zum anderen lassen sich so den Benutzer irritierende Änderungen der Klassenhierarchie während der Benutzung vermeiden. Die lediglich lokale Einordnung von neuen Objekten und Klassen kann aber auch zu lokalen Optima führen und erfordert daher eine regelmäßige globale Reorganisation, um Redundanzen und Fehleinordnungen zu beseitigen. Dies entspricht dem in Bild 5.5 hinzugekommenen Reorganisationskreis oberhalb des Wiederverwendungszykluses.

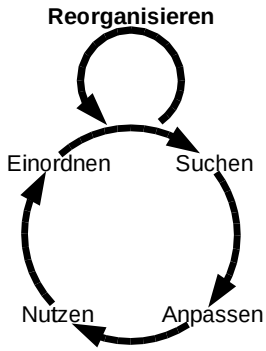


Abbildung 5.5: Wiederverwendung & Reorganisation

- Als Ausgangspunkt für die Neueinordnung eines Objektes nach einer Änderung wird eine vorhandene Einordnung des Objekts (vor der Änderung) benötigt. Diese Voreinordnung erlaubt es, bevorzugt im lokalen Umfeld der alten Einordnung nach passenden Alternativklassen zu suchen. Eine entsprechende Voreinordnung ist immer vorhanden, da alle existierende Objekte — und nur diese können manipuliert werden — eine Klassenzuordnung besitzen.
- Es existieren die Anwendungsdomäne allgemein beschreibende Metaklassen. Die Zuordnung zu ihnen kann nicht geändert werden (sie sind also 'essential types' im Sinne von [Zdonik 1990]) und schränkt somit die möglichen Neueinordnungen eines Objektes ein. Diese Metaklassen erlauben zudem eine grobe semantische Interpretation der Attribute und machen so unsinnige Zuordnungen unwahrscheinlicher (vgl. das 'Principle of Semantic Homogeneity' in [Spanoudakis 1994, S.50f]).
  Die Klassen der auf Seite 15 f dargestellten Methodologieebene dienen als Metaklassen für die verfahrenstechnische Prozeßmodellierung. Sie verhindern etwa Umwandlungen von Devices in Connections oder Interfaces. Die Trennungslinie zwischen diesen essentiellen und normalen Klassen muß allerdings nicht mit der Trennungslinie zwischen Methodologie- und Modellbausteinebene übereinstimmen. Da beide Ebenen mit Hilfe der Attributgruppierung (siehe Kapitel 5.4) in einer Instanziierungsebene zusammengefaßt werden, kann die Eigenschaft 'essentiell' als eigene Klassenfacette modelliert werden.

[6] vgl. S. 53

- Die Methoden und Restriktionen der Klassen werden als unveränderbar angesehen und beeinflussen nicht die Einordnung von Objekten in Klassen. D.h. die Einordnung eines Objekts hängt ausschließlich von seinen Attributen ab, die Methoden eines Objekts ergeben sich aus den zugeordneten Klassen. Diese Abweichung vom objektorientierten Modell in Richtung der strukturellen Objektorientierung kann hingenommen werden, weil die zentrale Speicherung der Daten das Hauptziel der Arbeit darstellt, während sich Methoden weitgehend in unabhängigen Werkzeugen befinden. Berechnete Attribute sind von dieser Einschränkung nicht betroffen, da sie in VeDa als normale typisierte Attribute mit Trigger–Facetten modelliert werden. Lediglich die Trigger–Methoden dürfen keine Namenskollisionen verursachen.

Aufgrund dieser Voraussetzungen kann der Suchraum für ein geändertes Objekt/eine neue Klasse erheblich eingeschränkt werden, bzw. es brauchen einige Vergleichskriterien nicht beachtet zu werden. Zwar vereinfacht und beschleunigt dies die Einordnungsalgorithmen, kann aber auch zu an mehreren Stellen definierten Attributen oder Klassen führen, da die lokale Einordnung lediglich einer Heuristik folgt. Eine optimale Einordnung kann erst während der globalen Reorganisation erreicht werden. Sollen allerdings bei der Reorganisation neue semantische Strukturen entdeckt werden, etwa für den Benutzer verständliche Aspekte, so kann sie, zumindest nach dem derzeitigen Stand der Informatik, nicht vollständig automatisiert werden[7].

5.2.5 Eine Datenbankarchitektur

Insgesamt ergibt sich also eine Datenbankarchitektur wie in Abbildung 5.6. Der rechte obere Quadrant zeigt, wie sich das DBMS aus Sicht eines normalen Modellierers wie eine prototypbasierte Datenbank mit einer klassenbasierten Suchhierarchie darstellt. Zudem befinden sich dort die in Abschnitt 5.2.4 angerissenen Mechanismen zur lokalen Einordnung neuer Objekte und Klassen, die benötigt werden, um die Illusion der prototypbasierten Modellierung aufrecht zu erhalten.

Abschnitt 5.2.4 wies darauf hin, daß auf die lokale Einordnung neuer Objekte und Klassen eine globale Reorganisation folgen muß, um aus globaler Sicht fehlerhafte Einordnungen zu beseitigen. Wie dort und auch in Kapitel 4.2.2 erwähnt wurde, kann eine Reorganisation aufgrund syntaktischer Strukturen keine domänengerechte Einordnung garantieren. Darum findet sich links in Abbildung 5.6 ein “Experte“ genannter Benutzer, der die von Reorganisationsunterstützung erstellten Reorganisationsvorschläge[8] ggf. modifiziert oder verwirft. Da die Reorganisation hauptsächlich die Klassen betrifft, kann sie nicht über die prototypbasierte Schnittstelle stattfinden. In der linken Hälfte der Abbildung wird deshalb direkt auf das klassenbasierte Datenmodell der Datenbank zugegriffen.

Als Grundlage zur Speicherung von Objekten und Klassen dient ein relationales DBMS. Die Notwendigkeit zur Verwendung einer relationalen Datenbank ergibt sich zum einen durch die Kooperation mit dem ProArtCE–Projekt [Dömges et al. 1996], das eine entsprechende Datenbank verwendet, und zum anderen durch den Datenumfang realer Prozeßmodelle, der objektorientierten Datenbanken Schwierigkeiten bereiten könnte.

Zusätzlich sind noch weitere Module erforderlich: etwa zur Auswertung von VeDas Regeln (Konsistenzkontrolle) und Methoden (Methodenaufrufweiterleitung), zur Versionierung von Schema und Objekten, zur Handhabung paralleler Zugriffe und ggf. zur Umwandlung in andere Datenmodelle, um fremde Werkzeuge integrieren zu können. Bereits in Kapitel 1 wurden die entsprechenden Themen jedoch als nicht in das Aufgabengebiet dieser Arbeit fallend gekennzeichnet, so daß sich die folgenden Abschnitte ganz auf das Objektmodell und seine Erweiterungen konzentrieren, die für die Realisierung der vorgestellten Emulationsmethodik notwendig sind.

[7] Selbstverständlich können neue Aspekte aufgrund *syntaktischer* Strukturen automatisch entdeckt werden. Ob diese dann allerdings immer eine Entsprechung in der Domäne haben, ist zweifelhaft.

[8] Diese können etwa durch die in [Sattler 1998] und [Molitor 2000] vorgestellten Verfahren gewonnen worden sein.

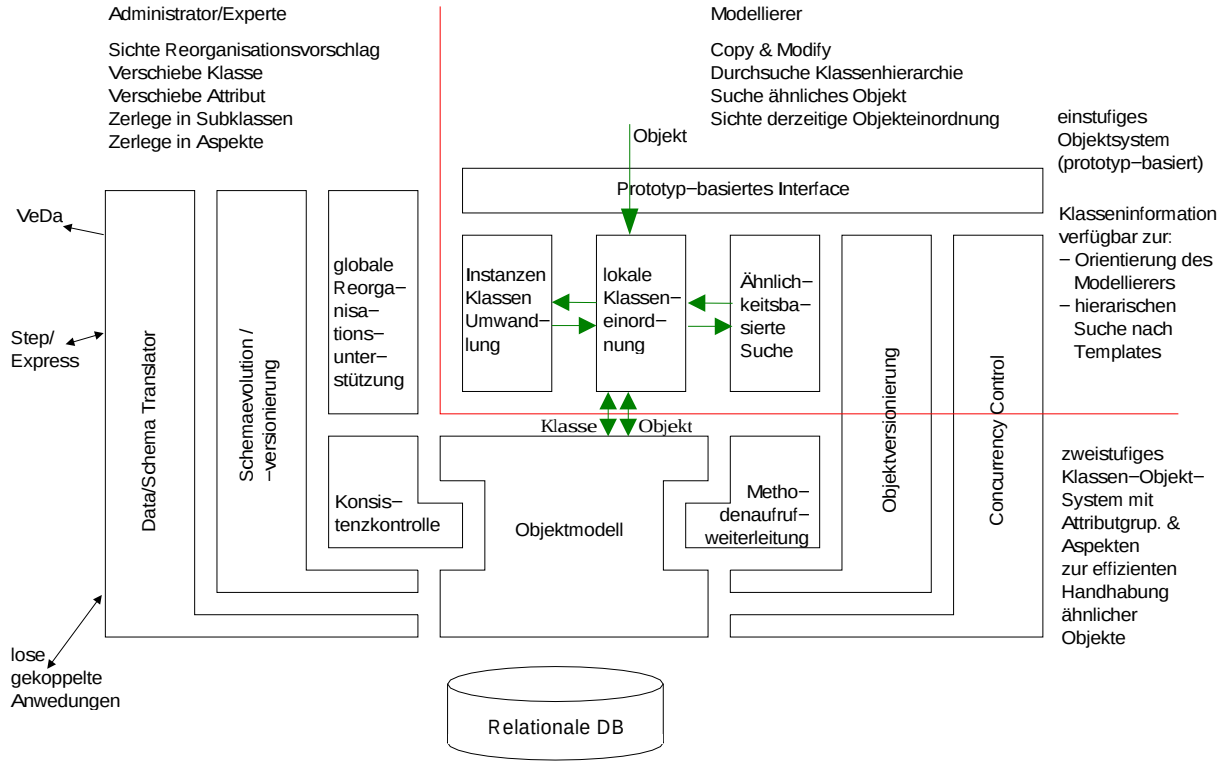


Abbildung 5.6: Architekturübersicht und Anwenderrollen

5.3 Aspekte: Ordnung im Dschungel Multipler Instanziierung

Die oben vorgestellte Emulation erlaubt es, Objekte typbeeinflußenden Änderungen zu unterwerfen. Zu diesen Änderungen müssen allerdings passende Klassen gefunden bzw. generiert werden. Die Generierung hatten wir bereits in Kapitel 5.2.3 durch Multiple Instanziierung der Objekte aus verschiedenen Klassenteilen diskutiert und gezeigt, daß ein Restriktionsmechanismus für die Multiple Instanziierung benötigt wird. Ein solcher Restriktionsmechanismus, der zudem die Multiple Instanziierung und die Behandlung auftretender Namenskonflikte integriert, wird in diesem Unterkapitel vorgestellt.

5.3.1 Motivation und Definition für Aspekte

Betrachten wir als Beispiel die Modellierung der Modell–Interface–Klassen in VeDa. Interfaces repräsentieren, wie auf Seite 23 erwähnt, Ein- und Austrittsmöglichkeiten von Strömen verschiedener Art in Devices und Connections. Interfaces unterscheiden sich aber, wie [Souza und Marquardt 1998, Kap. 7] aufzählt, nicht nur anhand der Frage, ob ein Strom durch sie in ein Device bzw. eine Connection hinein- oder herausfließt. Zusätzlich gibt es Unterschiede in der Art der Ströme, die durch ein Interface hindurchfließen, etwa ob tatsächlich Materialströme existieren, wie in einem `VALVE-INTERFACE` oder lediglich Informationen wie in einem `SIGNAL-INTERFACE`. Interfaces können im Rahmen einer Prozeßdekomposition mit anderen Interfaces im dekomponierten Prozeß gleichgesetzt werden oder im Rahmen des Durchspielens von Prozeßalternativen aktiviert bzw. deaktiviert werden. Da sich diese Eigenschaften eines Interfaces auf seinen Typ und seine möglichen Attribute auswirken, können Interfaces entlang verschiedener Achsen klassifiziert werden. Das VeDa–Modell realisiert dies durch die Benutzung von multipler Vererbung und Vereinigungsklassen (siehe [Souza und Marquardt 1998, Kap. 7]). Viele in dieser Klassenhierarchie einzeln vorgenommenen Modifikationen sind sich ähnlich. So spezialisiert die Klasse `DEVICE-SIGNAL-INTERFACE-IN`

das Attribut `coupled-interface`[9] der Klasse `DEVICE-INTERFACE` vom Typ `CONNECTION-INTERFACE` auf den Typ `CONNECTION-SIGNAL-INTERFACE-OUT`. Analog spezialisiert die Klasse `CONNECTION-SIGNAL-INTERFACE-IN` das gleiche Attribut von `DEVICE-INTERFACE` auf `DEVICE-SIGNAL-INTERFACE-OUT`. In beiden Fällen findet also eine Spezialisierung des Typs um '`SIGNAL`' und '`OUT`' statt.

Benutzt man nun Aspekte, so lassen sich die verschiedenen Spezialisierungsmöglichkeiten der Klasse `Interface` entlang verschiedener Dimensionen aufspalten (vgl. Abb. 5.7): entlang der Art der ankoppelbaren Verbindung (`IFType`), entlang der Richtung des Flusses durch das Interface (`IFDirection`), entlang dem Modellierungsblock, dem das Interface zugeordnet werden kann (`IFCarrier`), entlang der Position des Interfaces bezüglich eines aggregierten Modellbausteins (`IFComposition`) und entlang der Einordnung des Interfaces in eine Alternativenverwaltung (`IFVersioning`). Diese verschiedenen Spezialisierungsrichtungen nennen wir Aspekte. Dabei sind `IFType`, `IFCarrier` und `IFVersioning` notwendige Aspekte, da ein Interface immer den Typ des hindurchgehenden Stroms, den Typ seines Trägers (Device oder Connection) und seinen Standardaktivierungszustand angeben muß. Dementsprechend sind `IFDirection` und `IFComposition` optionale Aspekte, da sich über sie nicht in jedem Fall aussagen machen lassen, etwa weil das Interface ungerichtet ist oder weil sich das Interface in keinem aggregierten Modellbaustein befindet. Ein Modellierer, der `Interface` instanzieren will, *muß* also eine Klasse aus den Unterklassen jedes notwendigen Aspektes auswählen und *darf* eine Unterklasse jeden optionalen Aspektes selektieren.

Die Aspekte `IFType`, `IFCarrier` und `IFDirection` sind einfach verständlich. Zu jedem von ihnen muß eine Entscheidung getroffen werden, welche der vorhandenen Unterklassen die gewünschten Eigenschaften der Instanz bezüglich dieses Aspektes der Klassendefinition am besten repräsentiert. Der optionale Aspekt `IFDirection` erfordert zudem eine Entscheidung, ob überhaupt eine seiner Unterklasse instanziiert werden soll. Ein Objekt könnte gleichzeitig Instanz von `Interface`, `SignalInterface`, `DeviceInterface` und `InInterface` sein. Allerdings treten in diesen Klassen Attribute gleichen Namens und somit eine horizontale Namenskollision auf (vgl. S. 53). Das Attribut `coupled_to` etwa wird in drei der Klassen definiert, so daß sich die Frage stellt, welchen Typ dieses Attribut im oben erwähnten Objekt hat, das alle drei Klassen instanziiert. Dieses Problem wird auf Seite 102 f genauer behandelt werden. Kurz gefaßt muß das Attribut in der Instanz einen Wert annehmen, der mit allen in den drei Klassen definierten Typen übereinstimmt. Im obigen Beispiel muß das Attribut `coupled_to` auf ein Objekt verweisen, das Instanz von `DeviceInterface`, `SignalInterface` und `OutInterface` ist.

Der Aspekt `IFComposition` demonstriert, daß Aspekte direkt anderen Aspekten zugeordnet werden können und nicht immer direkt einer Klasse untergeordnet sein müssen. Dies kann etwa benutzt werden, wenn mehrere Wahlmöglichkeiten eines Aspekts auch gleichzeitig selektiert werden können[10]. Im Beispiel kann somit eine Instanz der Klasse `Interface` sowohl ein inneres als auch ein äußeres Interface sein, etwa wenn es sich in einem Teil eines dekomponierten Prozesses befindet wobei dieser Teil selber wieder dekomponiert ist.

Unterhalb von `IFVersioning` ist eine Möglichkeit sichtbar, wie Unterklassen verschiedener Aspekte gegenseitig ihre Instanziierung beeinflussen können. `IFVers.State` spezifiziert hierbei, ob ein Interface Objekt gerade aktiv, d.h. im Diagramm sichtbar und benutzbar, oder passiv ist, während `IFVers.Options` angibt, ob ein Interface für ein bestimmtes Device notwendig ist oder nicht. Die Interfacespezifikation eines Reaktors könnte beispielsweise folgendermaßen aussehen.

edukt_in:	`:dom VALVE-INTERFACE, IN-INTERFACE, DEVICE-INTERFACE, REQUIRED`
produkt_out:	`:dom VALVE-INTERFACE, OUT-INTERFACE, DEVICE-INTERFACE, REQUIRED`
temp_line:	`:dom SIGNAL-INTERFACE, OUT-INTERFACE, DEVICE-INTERFACE,OPTIONAL`

[9] `coupled-interface` gibt an, mit welchem anderen Interface innerhalb des Modells eine Verbindung besteht, wohin also der von einem Interface beispielsweise ausgehende Strom gelangt. Für frühzeitige Konsistenztests ist es darum nötig, `coupled-interface` entsprechend zu spezialisieren, so daß etwa ein Interface mit einem ausgehenden Strom nur mit einem Interface mit einem eingehenden Strom verbunden werden kann.

[10] Wie oben erwähnt, kann ohne eine solche Konstruktion nur eine Möglichkeit in jedem Aspekt ausgewählt werden

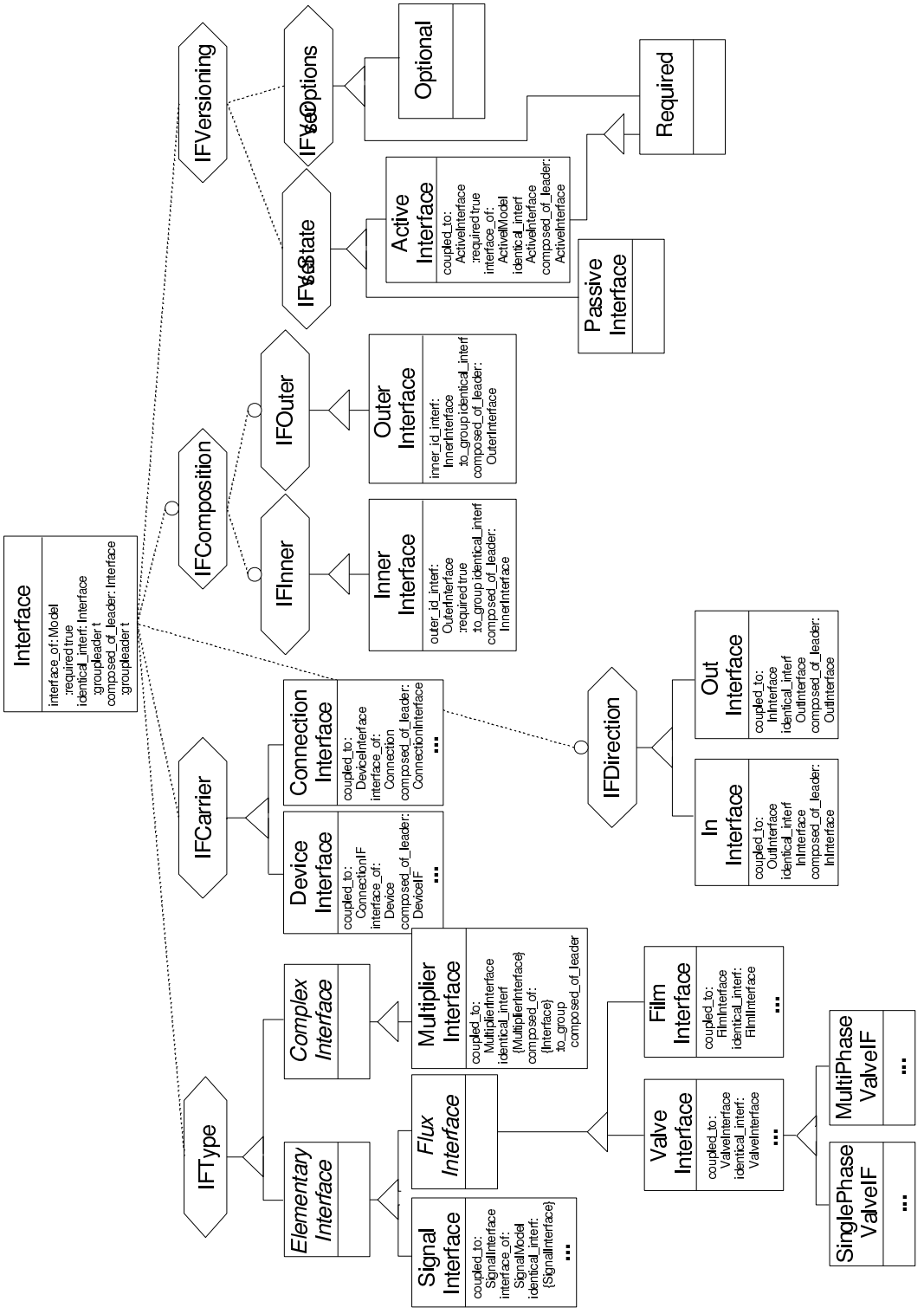


Abbildung 5.7: Modellierung der Interface–Hierarchie mit Aspekten

Wie man sieht werden hier in der Domänenfacette mehrere Klassen pro Attribut angegeben, die in einer entsprechenden Instanz unter Verwendung der multiplen Instanziierung alle instanziiert werden müssen. In einem Modell ohne multiple Instanziierung müßte die Klasse des Zulaufs etwa `REQUIRED-DEVICE-VALVE-IN-INTERFACE` lauten, was zum einen die notwendige Klassenzahl erhöht und zum anderen verhindert, daß Teile des Typen von der Oberklasse geerbt werden.

Zurück zur Beeinflussung von Instanziierungsbeziehungen: Im Beispiel sind Zu- und Abfluß ein notwendiges Interface, der Meßausgang für die Temperaturmessung ein optionales. Die Eigenschaft "notwendig" impliziert, daß das entsprechende Interface immer "aktiv" sein muß. Ein durch den Slot `edukt_in` referenziertes Objekt muß also immer auch `ActiveInterface` instanziieren. Diese Implikation zwischen den beiden Unterklassen verschiedener Aspekte wird durch die Unterklassenbeziehung zwischen ihnen repräsentiert. Diese und weitere Abhängigkeiten werden genauer in Abschnitt 5.3.2 behandelt. Ein weiteres, mehr an der Schnittstelle zwischen konzeptuellem und mathematischem Modell angesiedeltes Beispiel findet sich in [Baumeister und Jarke 1999].

Die Verwendung von Aspekten führt zu zwei offensichtlichen Vorteilen in obigen Beispiel. Die bei der Instanziierung einer Klasse notwendigen Entscheidungen können in weitgehend beliebiger Reihenfolge getroffen werden und können auch — solange ein unter Umständen inkonsistentes System in Kauf genommen wird — beliebig verzögert werden. Dies ist vorteilhaft gegenüber anderen Modellierungsmöglichkeiten: Bei einer Modellierung mit geschachtelten Spezialisierungen wird die Reihenfolge der Entscheidungen durch die Anordnung der Klassen vorgegeben und bei eine Modellierung mit Multipler Vererbung und Vereinigungsklassen müssen alle Entscheidungen bei der Auswahl einer passenden Vereinigungsklasse auf einmal getroffen werden. Desweiteren werden bei der Verwendung von Aspekten häufig weniger Klassen benötigt als ohne Aspekte. Die drei linken Aspekte[11] in Abbildung 5.7 benötigen 17 Klassen und Aspekte, während die Modellierung der gleichen Information in [Souza und Marquardt 1998] 34 Klassen benötigt.

Definition von Aspekten

Wie aus dem obigen Beispiel ersichtlich, fügen sich Aspekte leicht in die bekannten Konzepte der Objektorientierung wie Klassen, Objekte, Instanziierung und Oberklassenbeziehung ein. Aufbauend auf diesen werden Aspekte darum hier informell definiert, indem das Konzept 'Aspekt' und die Relation 'istAspektVon' eingeführt wird. Eine formale Definition, die dann einige Änderungen an den Definitionen der Objektorientierung erfordert, findet sich in Abschnitt 6.2.

Definition 5.9 (Aspekt) *Ein Aspekt ist eine abstrakte Klasse, die Ausgangspunkt mindestens einer istAspektVon–Relation ist. Den Endpunkt einer istAspektVon–Beziehung nennen wir auch 'Aspekthalter' und Unterklassen eines Aspekts 'Aspektkinder' (siehe auch Abbildung 5.8 a).*

Definition 5.10 (istAspektVon) *IstAspektVon ist eine gerichtete, azyklische, transitive und asymmetrische Relation, die von einem Aspekt ausgeht und in einer Klasse[12] endet. Zwischen zwei Klassen darf nicht sowohl eine istAspektVon–Relation als auch eine Unterklassenbeziehung bestehen.*

Definition 5.11 (Instanziierung von Aspekten) *Wenn ein Objekt einen Aspekt indirekt[13] instanziiert, so muß es auch einen der Aspekthalter oder alle Superklassen des Aspekts instanziieren. Wenn ein Objekt einen Aspekthalter instanziiert, so muß es auch alle notwendigen Aspekte des Aspekthalters instanziieren. Diese Implikationen zeigt Abbildung 5.8 b.*

[11] Die anderen beiden Aspekte werden in [Souza und Marquardt 1998] nicht über eigenständige Klassen modelliert und können somit nicht verglichen werden.

[12] Man beachte, daß gemäß obiger Definition ein Aspekt eine Klasse ist. IstAspektVon kann also auch in einem Aspekt enden.

[13] Die indirekte Instanziierung einer Klasse erfolgt bekanntlich, wenn eine Unterklasse dieser Klasse instanziiert wird. Im Weiteren meint Instanziierung immer direkte und indirekte Instanziierung.

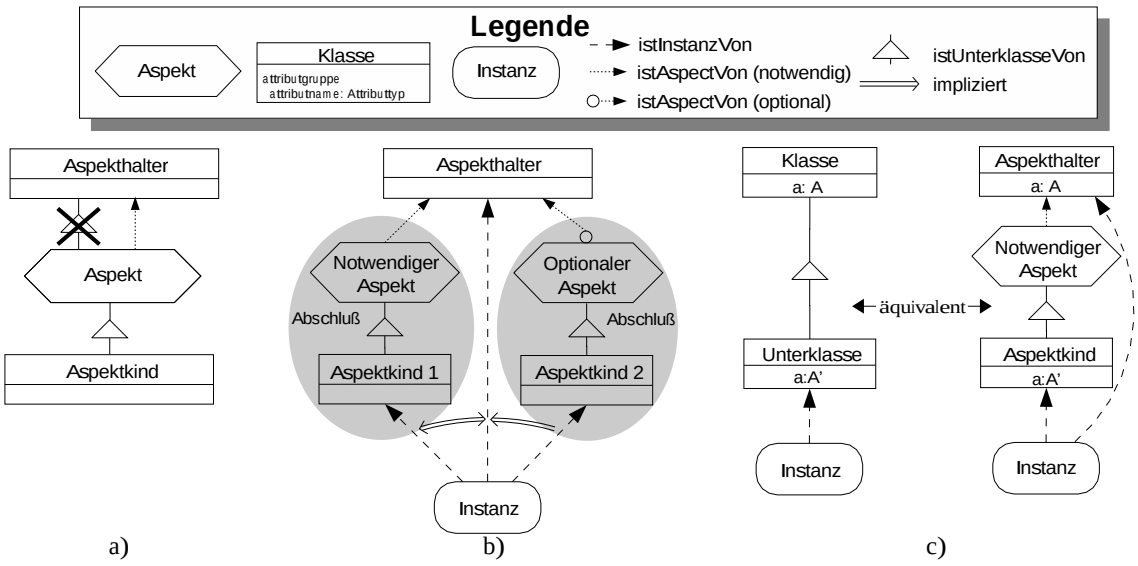


Abbildung 5.8: Illustration der Aspektdefinition

Definition 5.12 (Multiple Instanziierung und Restriktion) *Ein Objekt kann mehr als eine Klasse direkt instanziieren. Ist dies der Fall, so dürfen die transitiven Hüllen der Oberklassenbeziehung ausgehend von den instanziierten Klassen keine gemeinsamen Klassen enthalten. Entsprechende transitiven Hüllen sind in Abbildung 5.8 b sichtbar.*

Definition 5.13 (IstAspektVon und die Unterklassenbeziehung) *Attribute, die sowohl in einem Aspekt oder einer seiner Unterklassen als auch im Aspekthalter vorhanden sind, müssen den selben Restriktionen entsprechen, als wenn eine Unterklassenbeziehung zwischen dem Aspekthalter und der entsprechenden Klasse bestünde.*

Die Definitionen 5.9 und 5.10 stützen sich stark auf die schon vorhandenen objektorientierten Konzepte ab. Man beachte, daß ein Aspekt eine normale Klasse ist, in der Attribute, Methoden, ... definiert werden können, und die auch Unterklassenbeziehungen zu anderen Klassen haben kann. Lediglich die direkte Instanziierung ist verboten. Die Definitionen 5.12 und 5.11 schränken die Multiple Instanziierung ein, so daß nur am Übergang Aspekthalter–Aspekt mehrfach instanziiert werden kann. Es sei erwähnt, daß die Forderung aus Definition 5.12 nur dann ausreicht, wenn die Klassentaxonomie eine gemeinsame Wurzelklasse besitzt. Im anderen Fall, d.h. wenn die Klassen einen Wald bilden, könnten Klassen aus verschiedenen Bäumen gleichzeitig instanziiert werden, obwohl sie nicht über istAspektVon und einen Aspekthalter zusammenhängen. Dieser Zusammenhang über IstAspektVon- und Superklassenbeziehungen muß dann explizit gefordert werden. Definition 5.13 ist für das korrekte Funktionieren von Aspekten nicht unbedingt erforderlich. Sie stellt sicher, daß ein einzelnes Aspektkind und eine einzelne Unterklasse für den resultierenden Typ einer Instanz äquivalent sind (vgl. Abbildung 5.8 c). Dies fördert eine intuitivere Klassenhierarchie.

5.3.2 Abhängigkeitsmodellierung mit Aspekten

Obwohl Abbildung 5.7 eine Abhängigkeit zwischen den Aspektkindern `ActiveInterface` und `Required` enthielt, gehen die bisherigen Definitionen nicht auf solche Abhängigkeiten ein. Dabei wird ihr Auftreten dadurch wahrscheinlich, daß Aspekte vorhandene Klassen entsprechend von Gemeinsamkeiten zerteilen. Es kann nicht davon ausgegangen werden, daß alle entstehenden Teile unabhängig von einander sind. Im weitverbreiteten "Personen mit Rollen"–Szenario (siehe etwa [Wieringa et al. 1995; Gottlob et al. 1996; Wong et al. 1997]) kann man solche Abhängigkeiten leicht aufzeigen: Die Rollen"verheiratet" und "katholischer Priester" von Person schließen sich beispielsweise gegenseitig

aus, die Rolle “im Gefängnis“ schränkt mögliche Berufsrollen stark ein, und die Rolle “Dozent“ erfordert die Rolle “Dr.“. Modelliert man diese Rollen als Aspektkinder verschiedener Aspekte wie “Ausbildung“, “Anstellung“ oder “Familienstand“, so ergeben sich Abhängigkeiten zwischen den Kindern verschiedener Aspekte. Eine weitere Art der Abhängigkeit enthält das Beispiel von Abbildung 5.7 in Form von Namenskonflikten zwischen Attributdefinitionen. Zusammenfassend lassen sich folgende Abhängigkeitsarten identifizieren:

1. Abhängigkeiten zwischen Attributen in verschiedenen Klassenteilen
   (a) Wertabhängigkeiten zwischen Attributen gleichen Namens (wenn etwa das Attribut “temperatur“ in den Aspektkindern “metrische Maßeinheiten“, “amerikanische Maßeinheiten“ und “veraltete Maßeinheiten“ existiert)
   (b) Typabhängigkeiten zwischen Attributen gleichen Namens (wie in Abbildung 5.7)
   (c) Abhängigkeiten zwischen Attributen verschiedenen Namens
2. Instanziierungsabhängigkeiten von Klassenteilen
   (a) Wechselseitiger Ausschluß von Teilen (wie im “Priester vs. verheiratet“–Beispiel)
   (b) Implikation verschiedener Teile (wie im “Dozent ⇒ Dr.“–Beispiel)
   (c) Einschränkung möglicher Auswahlen (wie im “Gefängnis–Anstellungs“–Beispiel)
3. Abhängigkeiten zwischen Attributwerten und Instanziierbarkeit verschiedener Klassenteile (z. B. daß die Rolle “Kind“ nur angenommen werden kann, wenn das Attribut “alter“ des entsprechenden Objekts kleiner als zehn ist)

Nicht für alle diese Abhängigkeiten werden im folgenden Repräsentationsformen mit Hilfe von Aspekten angegeben. Abhängigkeiten zwischen Werten verschiedener Ausprägungen desselben Attributs in einem Objekt werden nicht unterstützt, da es nur wenige Anwendungsbeispiele gibt. Zudem widersprechen verschiedene Werte eines Attributs der Idee, daß Aspektkindern verschiedene Teile einer Klasse repräsentieren, die in einem Objekt vereint werden. Wir fordern darum, daß in einer Instanz jedes Attribut nur einmal vorkommt, so daß keine unterschiedlichen Werte möglich sind. Allgemeine Abhängigkeiten zwischen Attributen verschiedener Namen lassen sich nur durch Restriktionen etwa in Form von PL1–Regeln erfassen. Ihr Spezialfall, die Unifikation von Attributen verschiedener Namen, wie sie etwa bei der Schemaintegration auftritt, kann durch berechnete Attribute beziehungsweise in VeDa durch entsprechende Trigger gelöst werden.

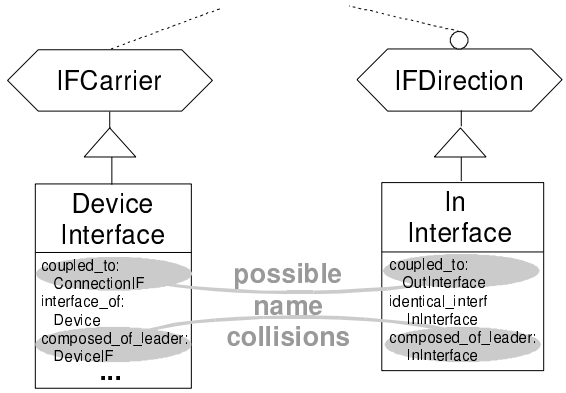


Abbildung 5.9: Ein Namenskonflikt aus Abb. 5.7

Auflösung von Namenskonflikten

Aspekte lassen es gemäß obigen Definitionen zu, daß mehrere Klassen von einem Objekt aus instanziiert werden. In diesen Klassen können Attribute gleichen Namens definiert sein. Hierdurch ergibt sich ein horizontaler Namenskonflikt. Da Namenskonflikte in diesem Ansatz als explizites Modellierungselement benutzt werden, um die Aufspaltung von Attributdefinitionen zu ermöglichen, muß definiert werden, was eine solche Situation für Typ und Wert des Attributs bedeuten. Wie auf Seite 54 beschrieben, verbieten Modellierungssprachen meist horizontale Namenskonflikte oder schränken

ihre Anwendung stark ein. Dies wird vermutlich durch die Annahme verursacht, daß gleichnamige Attribute in unterschiedlichen Klassen nur zufällig gleich benannt sind und in Wirklichkeit unterschiedliche Dinge beinhalten. Da Aspekte auf der Idee basieren, daß eine Klasse in einzelne semantisch jeweils zusammengehörige Teile aufgespalten wird, trifft eher die gegenteilige Annahme zu, daß nämlich gleichnamige Attribute in verschiedenen, durch Aspekte zusammenhängenden Klassen, die gleiche Sache bezeichnen.

Betrachtet man Abbildung 5.10 a, so befindet sich in ihr Attribut `a` als Attribut mit horizontalem Namenskonflikt. Um die natürlichste Form der Konfliktlösung zu bestimmen, soll angenommen werden, daß Objekt `o` nicht durch Multiple Instanziierung sondern durch einfache Instanziierung von einer virtuellen Vereinigungsklasse `D` entstanden ist (Abbildung 5.10 b). Der Typ von `a` in dieser Vereinigungsklasse bestünde dann aus `T` *und* `S`, wäre also die Vereinigung der Typen des Attributs in den beiden Oberklassen. Die meisten Objektsystem fordern in diesem Fall die Existenz einer gemeinsamen Oberklasse von `S` und `T` in der `a` existieren muß, so daß die Struktur von Abbildung 5.10 b nicht erlaubt wäre. Da in unserem Ansatz jedoch multiple Instanziierung verwendet wird, kann dieser Typkonflikt dadurch gelöst werden, daß auch das Objekt, auf das `a` verweist, mehrere Klassen (eben `S` und `T`) instanziiert[14]. In Abbildung 5.10 c ist `x` ein solches Objekt, auf das das Attribut `a` verweist, und das somit Instanz von `S` und `T` sein muß. Wegen der Definitionen 5.11 und 5.12 müssen `S` und `T` einen gemeinsamen Aspekthalter haben, so daß bereits bei der Definition der Klassen Typtests durchgeführt werden können.

Diese Lösung für die Typkonflikte zwischen gleichbenannten Attributen impliziert auch eine Lösung für die Wertabhängigkeiten zwischen ihnen. Da alle gleichnamigen Attribute der Klassen einer Instanz in der Instanz durch nur noch ein Attribut repräsentiert werden, müssen sie alle den selben Wert haben. Dies stimmt mit der Auffassung überein, daß gleichnamige Attribute die gleiche Sache beschreiben.

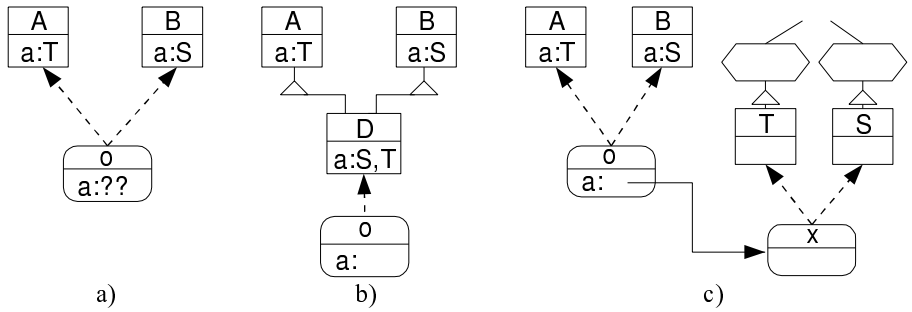


Abbildung 5.10: Namenskonflikt, abstrakt gesehen: a) mit multiple Instanziierung, b) mit (virtueller) Vereinigungsklasse, c) gelöst mit multipler Instanziierung des im Attribut enthaltenen Objekt

Instanziierungsabhängigkeiten

Die Instanziierbarkeit eines Aspektkindes kann davon abhängen, ob das instanziierende Objekt bereits andere Aspektkinder instanziiert oder noch nicht. Zwischen Aspektkindern *desselben* Aspekts herscht als Abhängigkeit Ausschluß, der automatisch durch Definition 5.12 erzwungen wird. Im Folgenden werden darum nur noch die auf Seite 102 identifizierten Instanziierungsabhängigkeiten zwischen Kindern *unterschiedlicher* Aspekte berücksichtigt.

Wechselseitiger Ausschluß Der wechselseitige Ausschluß von Aspektkindern kann nicht nur innerhalb eines Aspekts stattfinden, sondern auch zwischen verschiedenen Aspekten. Abbildung 5.11

[14] Werden Klassen und primitive Typen (z.B. int, char) unterschieden, so müssen spezielle Bedingungen eingeführt werden, falls `S` oder `T` von primitivem Typ sind.

zeigt einen Ausschnitt aus einer möglichen (wenn auch verbesserungswürdigen) Modellierung eines Kolonnenbodens als elementares Device aus [Baumeister und Jarke 1999]. Da ein variabler Dampffluß eine variable Dampfspeicherung impliziert, besteht ein wechselseitiger Ausschluß zwischen `TrayConstantVaporHoldup` und `TrayForceDrivenVaporFlow`, obwohl sie sich in verschiedenen Aspekten (`TrayVaporFlow` und `TrayVaporHoldup`) befinden. Wie im Bild sichtbar kann dies durch Einführung eines weiteren Aspekts (in der Abbildung der grau hinterlegte) modelliert werden. Da von den Kindern eines Aspekt nur eines pro Objekt instanziiert werden kann, schließen sich `TrayConstantVaporHoldup` und `TrayForceDrivenVaporFlow` dann gegenseitig aus. Da es zu diesem Aspekt jedoch kein äquivalentes Konzept in der Domäne gibt, bleibt er unbenannt.

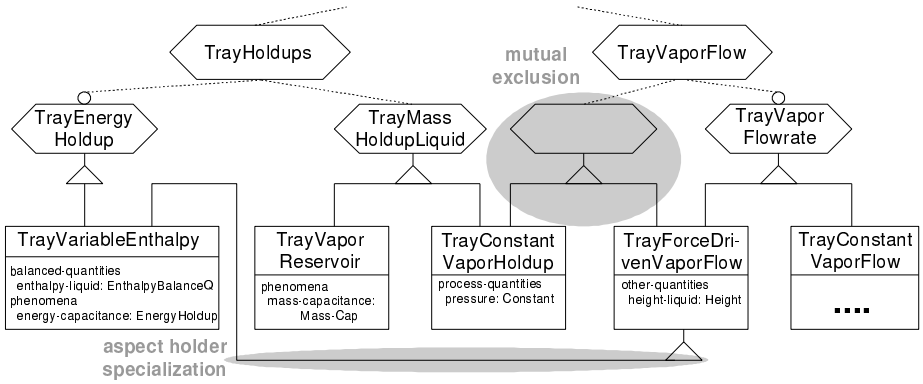


Abbildung 5.11: Wechselseitiger Ausschluß und Aspekthalterspezialisierung

Implikation anderer Aspekte — Aspekthalterspezialisierung Ein Aspektkind kann nicht nur, wie im letzten Absatz beschrieben, inkompatibel zu einem anderen sein, sondern — im Gegenteil — auch die Instanziierung einer anderen Klasse *erfordern*. Wie im "Dozent $\Longrightarrow$ Dr."–Beispiel impliziert also ein Aspektkind ein anderes. Gilt diese Implikation in beide Richtungen, sprich beide Aspektkinder können immer nur zusammen instanziiert werden, so besteht vermutlich ein Fehler im Modell und beide Klassen sollten zu einer zusammengeführt werden. Ansonsten kann die Implikation modelliert werden, indem das implizierende Aspektkind zusätzlich zu seinen sonstigen Beziehungen eine Unterklasse des implizierten Aspektkindes wird. Dadurch wird das implizierte Aspektkind automatisch instanziiert, wenn das implizierende instanziiert wird. Da durch diesen Mechanismus für den Aspekthalter (also etwa die Person) zusätzliche Eigenschaften erzwungen werden, er also spezialisiert wird, nenn wir diesen Mechanismus *Aspekthalterspezialisierung*.

Diese Vorgehensweise stimmt mit den ab Seite 100 definierten Eigenschaften der Aspekte überein, da diese nicht verbieten, daß ein Aspektkind mehreren Aspekten zugeordnet wird[15]. Es wird lediglich durch die Wahl eines Aspektkindes gleichzeitig die Auswahl in zwei Aspekten vorgenommen.

In Abbildung 5.11 soll die Klasse `TrayVariableEnthalpy` nur instanziiert werden dürfen, wenn gleichzeitig auch der Dampffluß variabel modelliert wird, da andernfalls ein Indexproblem in den mathematischen Gleichungen entstehen kann. Dadurch daß `TrayVariableEnthalpy` Unterklasse von `TrayForceDrivenVaporFlow` wird, kann sichergestellt werden, daß das Problem nicht auftritt.

Einschränkung von Auswahlmöglichkeiten — Aspektspezialisierung Die Auswahl einer Möglichkeit (also eines Aspektkindes) in einem Aspekt kann die Auswahlmöglichkeiten in anderen Aspekten auf einen Teilbereich einschränken. Oben wurde das "Gefängnis–Anstellungs"–Beispiel genannt. Abbildung 5.12 zeigt wiederum ein Beispiel aus der Modellierung eines Kolonnenbodens. Da das Durchregnen (`Weeping`) der Flüssigkeit nicht bei allen Bodenarten auftreten kann, soll die Auswahlmöglichkeit unter den Bodenarten eingeschränkt werden, wenn Durchregnen modelliert wird.

[15] Allerdings muß es von allen Aspekten einen Weg zu einem gemeinsamen Aspekthalter geben. In einem Modell mit Wurzelklasse, wie dem vorliegenden, ist dies jedoch immer der Fall.

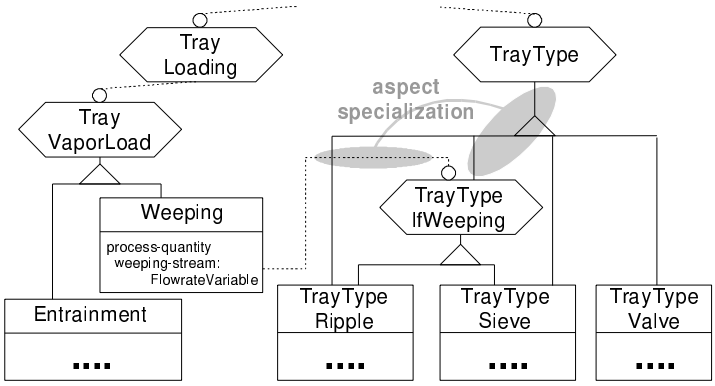


Abbildung 5.12: Aspektspezialisierung

Keine der Klassen impliziert die andere, da etwa ein Siebboden modelliert werden kann, ohne Durchregnen, das im Normalbetrieb nicht auftritt, zu berücksichtigen, und da es sein kann, daß Durchregnen modelliert werden soll, ohne das eine Festlegung über die Art des Kolonnenbodens gemacht wird.

Mit den bisherigen Modellierungsmöglichkeiten der Aspekte läßt sich diese Abhängigkeit nicht ausdrücken, allerdings haben wir bisher noch nicht definiert, was die Spezialisierung eines Aspekts durch einen anderen bedeuten soll.

Definition 5.14 (Aspektspezialisierung) *Wenn ein Aspekt A (`TrayTypeIfWeeping` im Beispiel) eine Unterklasse eines anderen (sozusagen des Oberaspekts; hier `TrayType`) ist, darf jede Instanz des Aspekthalters (`Weeping`) von A nur den Oberaspekt instanziieren, indem sie A instanziiert.*

Mit dieser Definition und dem neu eingeführten Aspekt `TrayTypeIfWeeping` läßt sich die geforderte Abhängigkeit modellieren. Werden entsprechende Abhängigkeiten nicht benötigt, so wird Def. 5.14 nicht benötigt und Spezialisierungsbeziehungen zwischen Aspekten werden verboten. In diesem Sinn ist Def. 5.14 kein notwendiger Teil der Aspektdefinitionen aus Abschnitt 5.3.1.

5.3.3 Anwendung von Aspekten

Aspekte erfüllen zwei verschiedene Funktionen: Zum einen unterstützen sie die Generierung neuer Klassen, zum anderen verbessern sie die auf den Klassen definierbare Ordnung. Dabei kann man drei Anwendungsfälle unterscheiden: Erstens den **Modellierer**, der bei weitgehend vorgegebenen Methodologie- und Bausteinebenen (vgl. S. 15) ein Modell eines verfahrenstechnischen Prozesses zu erstellen sucht und dabei ggf. die Bausteinebene erweitert. Zweitens den **Metamodellierer**, der die Bausteinebene (teilweise) definiert und dabei die Methodologieebene u.U. erweitert. Drittens erneut den Metamodellierer der vorgegebene, aber ohne Aspekte modellierte Methodologie- und Bausteinebenen in eine Modellierung mit Aspekten überführt. Im Folgenden werden die Effekte der beiden Funktionen in den Anwendungsfällen untersucht.

Mehr Ordnung in Klassenhierarchien

Prinzipiell spannt jeder Aspekt einen Entscheidungsraum auf, in dem die Aspektkinder die verschiedenen, sinnvollen Auswahlen repräsentieren. Ein einzelner Aspekt entspricht somit einer einfachen Vererbunghierarchie, in der der Modellierer ebenfalls eine Auswahl treffen muß. Durch die Kombination mehrerer Aspekte können alle Entscheidungen strukturell repräsentiert werden, die notwendig sind, um eine Instanz einer Klasse vollständig zu repräsentieren. Diese Entscheidungen können zwar auch in einer Hierarchie mit stufenweiser einfacher Vererbung oder mehrfacher Vererbung ausgedrückt werden, allerdings nur mit einer deutlich höheren Zahl von Klassen (vgl. Abbildung 5.7 und

zugehörige Erläuterungen). Dies führte, wenn diese Modellierungsart auf mehreren Hierarchiestufen wiederholt wird, zu einer unüberschaubaren Klassenzahl, in denen sich zudem Entscheidungen an verschiedenen Stellen wiederholen und somit die Wartungsfreundlichkeit mindern.

Die explizite Modellierung von Entscheidungsräumen und Auswahlmöglichkeiten durch Aspekte hat Vorteile sowohl für den eigentlichen Modellierer als auch für den Metamodellierer: Der Modellierer kann bei der Erzeugung eines neuen Objekts aus der Klassenhierarchie anhand der Aspektkinder schnell sehen, welche Wahlmöglichkeiten er hat. Diese Information ist zwar auch in einer Klassenhierarchie ohne Aspekte enthalten, zumindest sofern entsprechend modelliert wurde, muß dort allerdings erst innerhalb der Oberklassenhierarchie identifiziert werden. Anhand der Anordnung der Aspekte in der Hierarchie kann der Modellierer zudem erkennen, welche Entscheidung welche Einschränkungen zur Folge hat, etwa indem ein Aspekt Unteraspekt eines anderen ist oder weil, wie oben gezeigt, Abhängigkeitsbeziehungen zwischen Aspekten und Aspektkindern modelliert wurden.

Der Metamodellierer hingegen erhält durch die Entscheidungsbaum–Metapher eine Grundstruktur an die Hand, um die Klassenhierarchie aufzubauen. Dadurch kann er etwa anstreben, zumindest die möglichen Entscheidungen (also Aspekte) vollständig zu modellieren, auch wenn dies bei den einzelnen Alternativen dieser Entscheidungen (also den Aspektkindern) aufgrund deren Anzahl nicht im Vorhinein möglich ist. In Projekten mit mehreren Metamodellierern sollte sich zudem das Verständnis der Modelle der anderen verbessern. Der Preis hierfür besteht allerdings im Verzicht auf ad hoc definierte Klassen und Attribute, die zwar weiterhin möglich sind, aber eine Einordnung in die bestehende Entscheidungsstruktur erfordern. Um die Modellierung etwas zu vereinfachen, werden auf S. 107 einige Anwendungsregeln genannt.

Bei der Umwandlung einer bestehenden Klassenhierarchie ohne Aspekte in eine mit solchen verursacht die ordnende Funktion der Aspekte zusätzliche Arbeit, da die bestehende Hierarchie gemäß dem Domänenwissen des Metamodellierers durch Aspekte ergänzt werden muß. Da die möglichen automatischen Verfahren alle rein syntaktisch sind, kann eine Nachbearbeitung notwendig sein, damit Aspekte wirklich domänenrelevante Entscheidungen repräsentieren.

Aspekte als Klassenmodifikatoren

Bei der Verwendung von Aspekten als Klassenmodifikatoren (also als $\triangle \mathrm{M(K)}$ gemäß Definition auf Seite 53) kommt die schon bei Mixins und Rollen vorhandene Eigenschaft zur Anwendung, Klassendefinitionen ändern zu können, ohne die Klassen selber zu modifizieren (vgl. Abbildung 4.7). Aspekte erreichen diese Funktionalität, da Objekte zur Instanziierung einer Klasse noch zusätzlich Aspektkinder instanziieren können, die die Typen der vorhandenen Attribute verändern oder neue Attribute einfügen können. Hierdurch können Klassen an leicht geänderte Objekte angepaßt werden oder Ausnahmen modelliert werden.

Diese Funktion der Aspekte wird hauptsächlich durch den Modellierer genutzt, wenn er ausgehend von einem vorhandenen Objekt ein neues erzeugt und ändert. Die zusätzlichen bzw. gelöschten Attribute werden durch Hinzunahme oder Entfernung von Instanziierungsbeziehungen zu entsprechenden Aspekten/Aspektkindern modelliert. Dies setzt natürlich voraus, daß der Modellierer nur solche Modifikationen durchführt, die semantisch (nach Sicht des Metamodellierers) Sinn machen, oder daß die Klassenhierarchie neben den Aspekten zum oben erwähnten Entscheidungsbaumparadigma zusätzlich auch solche aus rein syntaktischen Gründen enthält bzw. ihre Erzeugung erlaubt.

Dadurch daß alle möglichen, erwünschten oder auch nur bisherigen Änderungen in der Klassenhierarchie als Aspekte und Aspektkinder gespeichert sind, kann ein Modellierer auch Einblick in die Schwere von Änderungen gewinnen, die aus dem vorhergegebenen Rahmen herausgehen. Können sie durch ein zusätzliches Aspektkind in einem der vorhandenen Aspekte modelliert werden, so hat er vermutlich nur eine bisher noch nicht modellierte Alternative entdeckt, was aufgrund der hohen Variabilität der Domäne häufig vorkommen sollte. Muß jedoch ein neuer Aspekt eingeführt werden,

so hat er möglicherweise eine bisher in der Modellierung nie beachtete Entscheidung entdeckt (oder einfach einen vom Metamodellierer vergessenen Aspekt).

Die beim Übergang einer normalen Klassenhierarchie auf eine solche mit Aspekten möglichen automatisch erkennbaren Aspekte fallen alle in die Kategorie $\triangle$M(K), da sie automatisch nur aufgrund unterschiedlicher Attributdefinitionen und Unterklassenstrukturen erkannt werden können. Mindestens in drei Situationen lassen sich mögliche Aspekte einfach identifizieren:

1. Rautenförmige Vererbungsstrukturen (vgl. Abbildung 5.3 links, auch wenn die Strukturen dort nicht sehr rautenförmig aussehen). Die die Attribute tragenden Klassen werden hierbei zu Aspektkindern (in der Abbildung also die Klassen `Spezialisierung 1` bis `Spezialisierung 4`, die Aspekte werden aufgrund der in den Vereinigungsklassen vorkommenden Kombinationen gebildet. Die Hierarchie aus Abbildung 5.7 wurde auf diese Weise gebildet.
2. Mehrstufige Spezialisierung (vgl. Abbildung 3.8 links). Sie sollte eigentlich nur vorkommen, wenn das Objektmodell multiple Vererbung nicht unterstützt. Zur Konvertierung in Aspekte wird eine solche Hierarchie in eine Hierarchie mit rautenförmiger Vererbung umgewandelt und entsprechend Punkt 1 behandelt.
3. Ähnliche Attribute in benachbarten Klassen. Befinden sich in benachbarten[16] Klassen Attribute gleichen Namens aber unterschiedlichen Typs während der Rest der Klassen gleich ist, so kann diese Hierarchie möglicherweise in eine Hierarchie mit rautenförmiger Vererbung umgewandelt werden und dann analog behandelt werden.

Es muß sich aber immer vor Augen gehalten werden, daß die Erzeugung von Aspekten aufgrund syntaktischer Kriterien (etwa von Attributen oder Vererbungsbeziehungen) eine für die Domäne sinnvolle Hierarchie ergeben kann aber nicht muß. Der folgende Abschnitt soll dem darum immer benötigten Domänenexperten einige Leitlinien bei der Definition einer Hierarchie an die Hand geben.

Anwendungsregeln

Grundsätzlich dienen Aspekte und Aspektkinder der Abstraktion. Während Aspektkinder, ähnlich wie Klassen, Instanzen abstrakt repräsentieren, ersetzt ein Aspekt Unterklassenbeziehungen und faßt die zu einem 'Thema' gehörigen alternativen Unterklassen einer Klasse unter sich zusammen. Dies bedeutet im Einzelnen:

Aspekte: Aspekte beschreiben die **Gründe** eine Klasse zu spezialisieren (bzw. die zu treffenden Entscheidungen). In Bild 5.13 sind diese Gründe etwa, daß Effekte aufgrund zu hoher/niedriger Last auftreten, daß Einflüsse der Geometrie mitmodelliert werden sollen oder daß der Dampfstrom modelliert werden soll. Ist ein bestimmter Grund immer gegeben, so wird er durch einen notwendigen Aspekt repräsentiert, ansonsten durch einen optionalen.

Aspektkinder: Aspektkinder entsprechen den **Möglichkeiten** wie die Klasse aus dem jeweiligen Grund spezialisiert werden kann. Aspektkinder enthalten genau die Attribut- und Restriktionsdefinitionen, die für ihre Möglichkeit der Spezialisierung aus dem jeweiligen Grund nötig sind.

Aspekthalter: Der Aspekthalter, also die Klasse die einen oder mehrere Aspekte besitzt, enthält nur noch Definitionen, die nicht bereits in Aspektkindern vorkommen. Ausnahme: Definitionen, die von Aspektkindern spezialisiert werden. Da eines der Kinder eines notwendigen Aspekts immer Teil einer Aspekthalterinstanz ist, kann dies insbesondere bedeuten, daß der Aspekthalter als Klasse alleine unvollständig ist.

Namenskonflikte: Namenskonflikte zwischen Aspektkindern verschiedener Aspekte sind erlaubt und zur gegenseitigen Beeinflussung der Spezialisierungsrichtungen auch erwünscht. Ein solcher Konflikt ist erlaubt, wenn das Attribut in allen beteiligten Aspektkindern das gleiche Konzept repräsentiert und das Attribut in allen Aspektkindern in eine für das jeweilige Kind

[16] Z.B. Klassen mit der gleichen Oberklasse

notwendigen Weise spezialisiert wird. Die Definition eines solchen Attributs bereits im Aspekthalter ist vorteilhaft (da dadurch die Repräsentation *eines Konzeptes sichergestellt wird) aber nicht notwendig.*

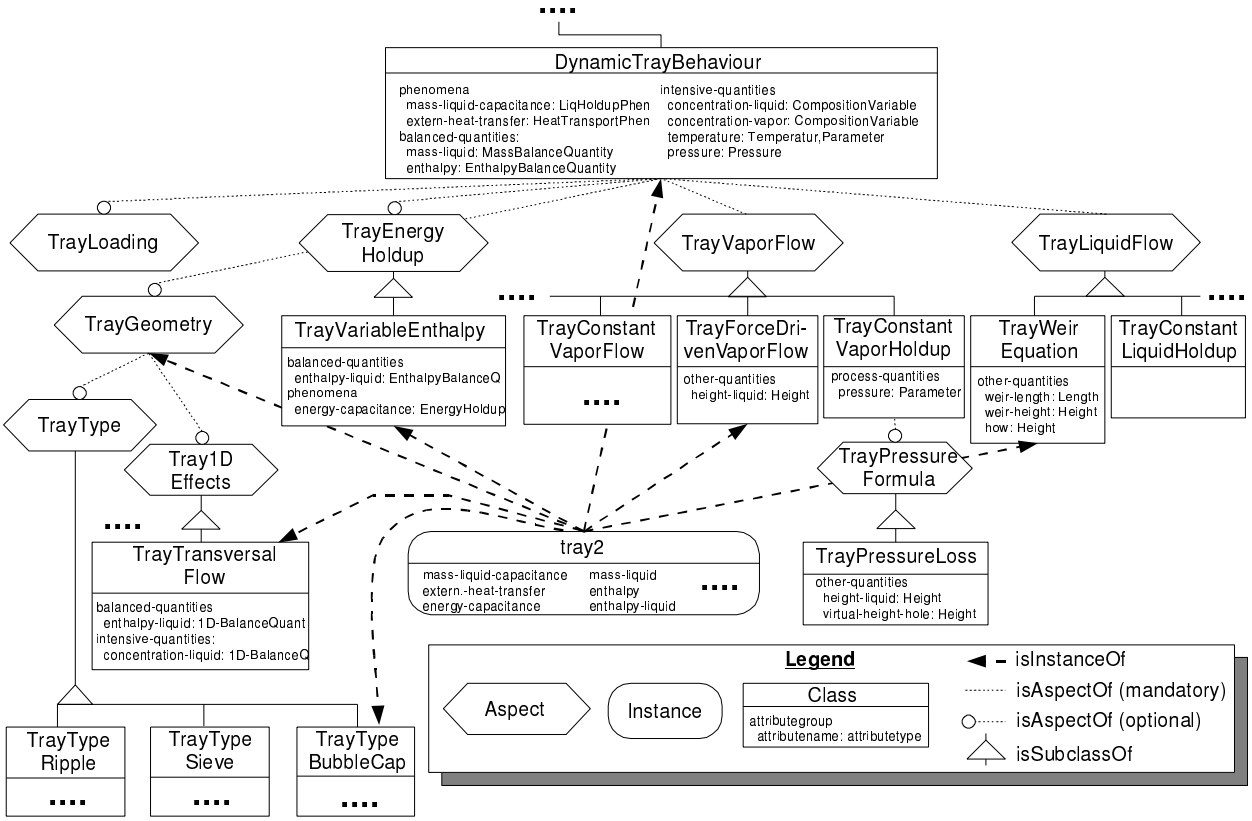
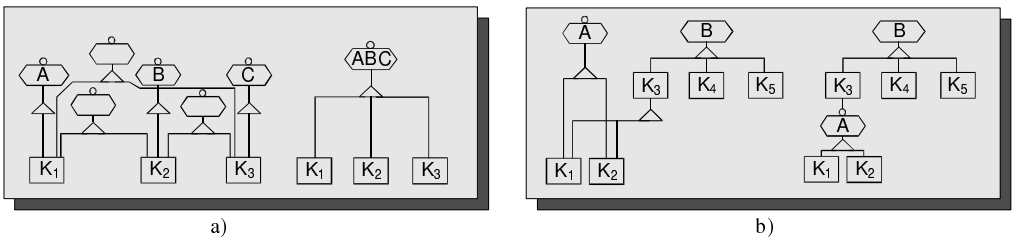


Abbildung 5.13: Elementares Modell eines Destillationskolonnenboden (erweiterte Version von [Baumeister und Jarke 1999])

Neben diesen elementaren Erklärungen existieren noch einige Regeln, die bei der Modellierung mit Hilfe von Aspekten beachtet werden sollten:

Abhängigkeiten vermeiden: Häufig kann ein Sachverhalt auf mehrere Arten mit Hilfe von Aspekten modelliert werden (siehe etwa die Flow–Modellierung in den Abbildungen 5.13 und 5.11); zum einen wegen leicht unterschiedlicher Interpretationen der realen Welt, zum anderen, da einige der Abhängigkeitsmodellierungen (vgl. Abschnitt 5.3.2) sich in redundanter Weise einsetzen lassen. Bild 5.14 a zeigt links ein unübersichtliches Modell, das insbesondere drei künstliche Aspekte zum wechselseitigen Ausschluß enthält. Das Modell im gleichen Bild rechts ist vollständig äquivalent und sollte bevorzugt werden. Das gleiche gilt für den linken und rechten Teil von Bild 5.14 b. Da der rechte ohne zusätzliche Abhängigkeitsmodellierung auskommt, sollte er bevorzugt werden.



a) b)

Abbildung 5.14: Modelle mit und ohne Abhängigkeiten

Unterklassenbeziehung oder Aspekt? Eine Unterklasse und ein notwendiger Aspekt mit Aspektkind sind weitgehend äquivalent (siehe Abbildung 5.8 c). Hat eine Klasse darum zusätzlich zu Aspekten noch Unterklassen, so entspricht dies einem Modell, in dem alle diese Unterklassen Aspektkinder eines weiteren Aspekts sind. Für die generierbaren Klassen hat die Wahl der Modellierung zwar keine Auswirkungen, das Modell mit den direkten Unterklassen sollte trotzdem nur gewählt werden, wenn der den Unterklassen entsprechende Aspekt die Hauptspezialisierungsrichtung vorgibt, andere Aspekte also nicht gleichberechtigt sind.

Aspektpatterns: Einige Arten der Aspektverwendung treten häufiger auf, ohne das sie von vorneherein offensichtlich sind. Bild 5.15 a zeigt einen Aspekt, der nicht der Auswahl aus mehreren Alternativen dient, sondern das Vorhandensein einer Klasse modelliert, die Klasse also einzeln anwählbar macht, wenn der entsprechende Aspekthalter instanziiert wird.

Abbildungen 5.15 b und 5.15 c zeigen zwei auf den ersten Blick ähnliche Modelle mit in Wirklichkeit sehr unterschiedlichen Bedeutungen. In 5.15 b definieren die Aspekte jeweils einen neuen Bereich in dem Spezialisierungen vorgenommen werden können, etwa mit `A`=Lebewesen, `B`=Landbewohner, `C`=Wüstenbewohner sowie `Aa`=Skelettarten und `Ac`=Wassergewinnungstrategien. In dieser Verwendungsform wird die Fähigkeit von Aspekten zur Verminderung der Klassenzahl optimal ausgenutzt.

Im Modell von Abbildung 5.15 c definieren die Aspekte nicht neue Spezialisierungsbereiche sondern schränken einen schon vorhandenen ein (vgl. Aspektspezialisierung auf Seite 105). Als Beispiel aus der Prozeßmodellierung könnte etwa `A`=Reaktor, `B`=Flüssig–Flüssig-Reaktor, `C`=EOEG–Reaktor sein und `Aa`=Reaktorbauart und `Ac`=Bauart bei exothermer Reaktion. Es ist zu vermuten, daß diese Art der Aspektverwendung bei VEDA–ähnlichen Modellierungen häufiger auftritt als die aus 5.15 b

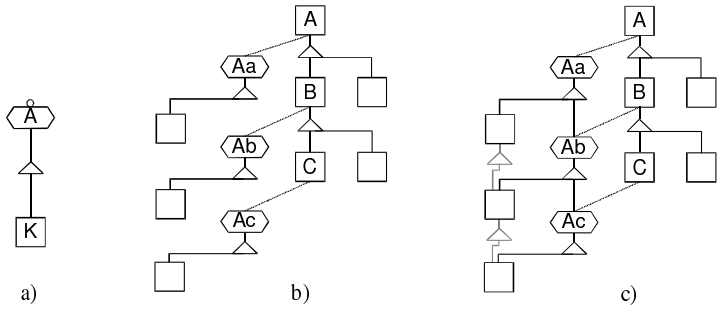


Abbildung 5.15: Aspektpattern

Die Kenntnis der in diesem Abschnitt angegebenen Regeln ist aber nur für Personen notwendig, die neue Aspekte definieren wollen, d.h. für den Metamodellierer und für fortgeschrittene Modellierer. Das Hinzufügen einer weiteren Entscheidungsalternative (also eines Aspektkindes) kann auch ohne Kenntnis dieser Regeln erfolgen.

5.3.4 Diskussion

Aspekte bieten die in Abschnitt 5.2.3 geforderte Möglichkeit der multiplen Instanziierung bei gleichzeitiger struktureller Restriktion der erlaubten Instanziierungen. Im Gegensatz zu Rollen (vgl. S. 56 ff) erlauben Aspekte die wechselseitige Sichtbarkeit der Attribute von multiple instanziierten Klassen und definieren einen Mechanismus zur Auflösung dadurch auftretender Namenskonflikte. Erst hierdurch kann die Aufspaltung der verschiedenen Teilsichten von Klassendefinitionen bzw. -spezialisierungen auf unabhängige Konzepte durchgeführt werden, da erst diese Namenskonfliktauflösung die spätere Rekombination der Teildefinitionen gestattet. Diese Fähigkeiten bietet keines der in Kapitel 4.1.2 vorgestellten Verfahren.

Ein Grund hierfür könnte sein, daß viele der in Kapitel 4.1.2 erwähnten Verfahren darauf abzielen, einfach auf bestehende objektorienterte Standardsprachen wie C++ oder Java abbildbar zu sein (etwa durch einen Präcompiler). Dies ist mit Aspekten nicht möglich, da solche Sprachen grundsätzlich die eindeutige Typisierung von Attributen fordern. Die Implementierung einer VeDa–artigen Sprache mit Aspekten mittels C++ ist dementsprechend aufwendig.

Vorteile	Nachteile
• Aufteilung der Spezialisierungsmöglichkeiten einer Klasse in verschiedene Entscheidungsdimensionen • weniger Klassen als bei multipler Instanziierung • alle Attribute gleichzeitig sichtbar und beeinflußbar • Aspekte fügen sich natürlich in Objektmodell ein	• Unbrauchbar bei Existenz vieler Instanziierungsabhängigkeiten zwischen verschiedenen Aspekten • nicht als Aufsatz auf Standard–OO–Modelle implementierbar

Tabelle 5.2: Vor- und Nachteile der Modellierung mit Aspekten

Wie in Abschnitt 5.3.2 gezeigt wurde, können zusätzlich zu Abhängigkeiten zwischen Attributen — auf nichts anderes werden Namenskonflikte ja abgebildet — auch Instanziierungsabhängigkeiten zwischen verschiedenen Aspekten bzw. Aspektkindern modelliert werden. Die Verwendung dieser Möglichkeit erhöht zwar die Mächtigkeit der Sprache, allerdings wächst im gleichen Maß die Unverständlichkeit der Modelle und das Verhältnis Aspektklassen zu eingesparten Vereinigungsklassen verschlechtert sich. Während in der Methodologieebene von VeDa solche Abhängigkeiten selten vorkommen, sind in der Bausteinebene häufig nicht alle möglichen Kombinationen von Aspektkindern erlaubt, so daß zusätzliche, abhängigkeitsmodellierende Aspekte benötigt werden. Dort kommen auch verstärkt Modelle der Art wie in Abbildung 5.15 c vor, etwa in Form der parallelen Spezialisierungshierarchien von Devices und ihren Implementierungen. Somit könnte es eine Grenze in der Hierarchie von VeDa geben, ab der die Verwendung von Vereinigungsklassen tatsächlich speicherplatzsparender ist als die von Aspekten. Zur Bestimmung dieser Grenze müßten allerdings wesentlich weitgehendere Modelle auf der Bausteinebene untersucht werden als bisher.

Die Erstellung des Modells für Abbildung 5.13 läßt zudem vermuten, daß die Verwendung von Aspekten eine erhebliche Einarbeitung verfahrenstechnischer Metamodellierer erfordert. Das Problem liegt dabei weniger in den neuen Sprachkonstrukten als in der geänderten Modellierungszielrichtung. Statt einen oder wenige Modellbausteine vollständig zu erstellen, müssen die einem Baustein innewohnenden Variationsmöglichkeiten erkannt, auf Zusammenhänge untersucht und in passende Aspekte aufgeteilt werden. Es wäre zu untersuchen, ob darum ein Vorgehen vorzuziehen ist, daß zuerst Standardmodelle erzeugt und aus diesen anschließend Aspekte gewinnt. Zur Zeit ist eine solche Untersuchung allerdings noch nicht möglich, da die Anwendung VeDa–artiger Modellierungsmethoden in der Industrie noch in den Kinderschuhen steckt und darum kaum entsprechende Modelle vorhanden sind.

5.4 Attributgruppierung: Attributkategorien ohne Metamodellierung

Bei der Definition der vier Ebenen von VeDa (vgl. S.15 f) wurde das Konzept der Metamodellierung genutzt und ab Seite 21 genauer vorgestellt. Die Metamodellierung erlaubt es dabei, übergreifende Strukturierungs-, Methoden- und Restriktionsinformation zu erfassen (etwa die Zusammenhänge zwischen Devices, Connections und ihren Interfaces) und diese Information auf der Klassenebene

nochmals zu variieren (indem etwa für Kolonnen mindestens ein Zufluß und zwei Abflüsse verlangt werden). Allerdings treten bei dieser Verwendung der Metamodellierung konzeptuelle und implementierungstechnische Probleme auf, die der nächste Unterabschnitt behandelt. Wie Abschnitt 5.4.2 dann zeigt, lassen sich diese Probleme beheben, indem man sich auf einen Teil der Metamodellierungsfunktionalität beschränkt und diesen auf Klassenebene realisiert.

5.4.1 Fähigkeiten und Probleme der Metamodellierung

Metaklassen nehmen als Klassen der Klassen Einfluß auf Struktur und Verhalten von Klassen sowie gelegentlich auch auf deren Instanzen. Diese Auswirkungen lassen sich auf die folgenden Funktionen von Metaklassen zurückführen (die Beispiele beziehen sich auf Abbildung 2.1).

Funktion 1 —Klassen zusammenfassen: Metaklassen fassen Klassen gemäß ihrer Zugehörigkeit zu Grundkonzepten der Domäne zusammen. Neben der so erreichten Gruppierung der Klassen erlaubt dies auch, die folgenden Funktionen auf alle entsprechend angeordneten Klassen gemeinsam auszuführen. In VeDa werden beispielsweise die Klassen `Column`, `Evaporator` und `Reactor` als Instanzen der Metaklasse `Device` zusammengefaßt.

Funktion 2 — Attribute gruppieren: Die Attribute zusammengefaßter Klassen können ebenfalls nach semantischen Kriterien in Attributkategorien, auch Metaattribute genannt, vereinigt werden. Der Typ von Attributen einer Kategorie kann beschränkt werden, und auf Attribute einer Kategorie kann gemeinsam, d.h. unabhängig von ihrem jeweiligen Namen, zugegriffen werden. In VeDa werden etwa verschiedene Interface-Attribute (wie `Product_out`, `Mixture_in`) oder charakterisierende Attribute (wie `phenomena`, `fluxes`) mit Hilfe von Attributkategorien zusammengefaßt.

Funktion 3 — Methoden, Restriktionen und Attribute der Klassen definieren: Im Rahmen ihrer Eigenschaft als Klassen übernehmen Metaklassen die normalen Aufgaben einer Klasse, allerdings für Klassen statt für Instanzen. Durch Metaklassen definierte Funktionen und Restriktionen können genutzt werden, um Klassen zu erzeugen und zu modifizieren, bzw. um ihre Erzeugung und Modifizierung einzuschränken. Klassenattribute enthalten variable, aber für alle Instanzen einer Klasse gleiche Werte. In VeDa treten Klassenmethoden und -restriktionen praktisch nicht auf. Abbildung 2.1 enthält darum als etwas ausgefallene Beispiele eine Methode `create()`, mit der sich neue Composite–Device–*Klassen* erzeugen lassen, und eine Restriktion `deviceattrname_neq_connection_attrname`, die gleiche Attributnamen in den Attributkategorien `subdevice` und `subconnection` verbietet.

Funktion 4 — Methoden und Restriktionen auf Instanzen definieren: Teilweise existiert auch der Wunsch, bereits in Metaklassen Methoden und Restriktionen der Instanzen zu definieren, z.B. weil eine Methode nur auf den Attributkategorien arbeitet und für alle Klassen, die die Metaklasse instanziieren, gleich ist. In VeDa existiert etwa eine Restriktion `all_interfaces_eq_subdevice_interface`, die verlangt, daß in allen Instanzen von Klassen, die Instanz von `CompositeDevice` sind, alle Interfaces des Gesamt–Devices eine Entsprechung an einem Unter–Device haben müssen. Eine solche direkte Instanzbeeinflussung durch Metaklassen gehört allerdings nicht zu deren Standarddefinition. Sie wird durch einige Zusätze (etwa 'Metarules' [Jeusfeld 1992] in ConceptBase oder 'instance–instance–types' [Klas und Schrefl 1995] in VODAK) erreicht.

Zudem besitzt eine Metaklasse natürlich noch die übrigen im Einschub auf Seite 25 aufgeführten Klassenfunktionen, da sie ja eine Klasse darstellt. Sie interessieren für die folgenden Betrachtungen aber nicht weiter.

Auf Seite 21 wurden verschiedene Anwendungen der Metamodellierung vorgestellt, nämlich Sprachadaptierbarkeit, Introspektion und Modellierungsmodellierung. Die aufgeführten Funktionen treten nun in den verschiedenen Anwendungsgebieten unterschiedlich gewichtet auf: Introspektion beruht stark auf den beiden Gruppierungsfunktionen (Funktionen 1 und 2), die die zu betrachtenden Konstrukte aufsammeln. Sprachadaption legt das Schwergewicht mehr auf die Definition von Verhalten

und Attributen von Klassen, sowie zu einem geringeren Teil, auch von Instanzen (Funktionen 3 und 4). Modellierungsmodellierung, also die Definition eines Domänenmetamodells, benutzt die Funktionen ausgeglichener, allerdings mit einer Betonung der Gruppierungsfunktionen und einem eher schwach ausgeprägten Bedarf für die Definition von Klassenattributen und Methoden. Diese mehr oder weniger starke Beschränkung auf Teilfunktionalitäten wird interessant, wenn in Abschnitt 5.4.2 eine veränderte Repräsentation für Metaklassen vorgeschlagen wird, die nicht die gesamte oben aufgeführte Funktionalität realisiert.

Der durch Metaklassen zusätzlich ermöglichten Funktionalität stehen allerdings einige **Probleme** gegenüber, die sich bei der Verwendung von Metamodellierung zur Domänenmodellierung ergeben.

1. Kommerzielle objektorientierte Datenbanken unterstützen Metaklassen nicht oder nur sehr begrenzt. Ähnliches gilt für verbreitete Programmiersprachen wie C++, Smalltalk oder Java und Objektmodelle, z.B. OMT und UML.
2. Gelegentlich sollen in Metaklassen Attribute definiert werden anstelle von Attributklassen, d.h. Attribute, die durch die Instanzen instanziiert werden und nicht durch die Klassen. Ein solches ebenenüberspringendes Attribut ist etwa das Attribut `occurs_in` der Metaklasse `Elementary-DeviceModel`. Es verweist von jeder Instanz einer `ElementaryDeviceModel`–Klasse zurück auf diejenige Instanz einer `CompositeDeviceModel`–Klasse, von der die Instanz ein Teil ist. Ein weiteres Beispiel ist das Attribut `OID` der Metaklasse `Class`.
3. Teilweise fällt die Entscheidung schwer, ob ein Konzept durch eine Klasse oder eine Metaklasse ausgedrückt werden soll, insbesondere, wenn es gleichzeitig Attribute und Attributkategorien enthält.

Wie schon auf Seite 29 f dargelegt, stellt es keine Lösung der Metamodellierungsproblematik dar, einfach auf die Metamodellierung zu verzichten. Im folgenden wird gezeigt, daß man sich die Ähnlichkeiten zwischen Instanziierung und Spezialisierung zunutze machen kann, um innerhalb der Klassenebene Teilfunktionalitäten der Metamodellierung zu implementieren.

5.4.2 Attributgruppierung als Metamodellierung 'light'

Normalerweise werden Metamodelle durch Instanziierungsbeziehungen zwischen Klassen (Attributen) und Metaklassen (Metaattributen) repräsentiert, wie in Abbildung 5.16 a sichtbar. Die Abbildung zeigt aber auch den doppelten Instanziierungschritt von Metaklassen zu Instanzen, der für die oben aufgeführten Probleme verantwortlich ist. Zum einen sind zwei sukzessive Instanziierungsschritte inkompatibel mit den Zwei–Ebenen–Objektmodellen der verbreiteten objektorientierten Datenbanken und Sprachen, die nur einen Instanziierungsschritt erlauben (obiges Problem 1). Außerdem erzwingen die getrennten Ebenen die Einordnung eines Konzepts als Klasse oder Metaklasse (Problem 3). Schließlich verursacht die obere Instanziierungbeziehung Problem Nummer 2, da Instanziierungsbeziehungen nicht transitiv sind, ein in der Metaklasse definiertes Attribut also nicht durch eine Instanz instanziiert werden kann. Es bietet sich also an die obere Instanziierungsbeziehung durch eine andere Beziehung zu ersetzen.

Ersatz durch Multiple Spezialisierung

Zwischen den Einschränkungen, die ein Superattribut auf ein Attribut ausüben kann, und den durch ein Metatattribut hervorgerufenen existieren — je nach Definition — nur geringe Unterschiede. Bei beiden werden Typ und Ausgangsklasse des Attributs eingeschränkt. Eine alternative Realisierung für Metaklassen besteht darum darin, die oberen Instanziierungsrelationen durch Spezialisierungsrelationen zu ersetzen. Dadurch werden alle Metaklassenebenen (also Meta-, Metameta-, ...) in die Klassenebene abgebildet (vgl. Abbildung 5.16 b). Da in unserem Modell der Metamodellierung ein Attribut mehrere Metaattribute besitzen kann, müssen hier auch mehrere Superattribute eines Attributs, also "Multiple Spezialisierung", zugelassen werden. Die so entstehenden Superklassen

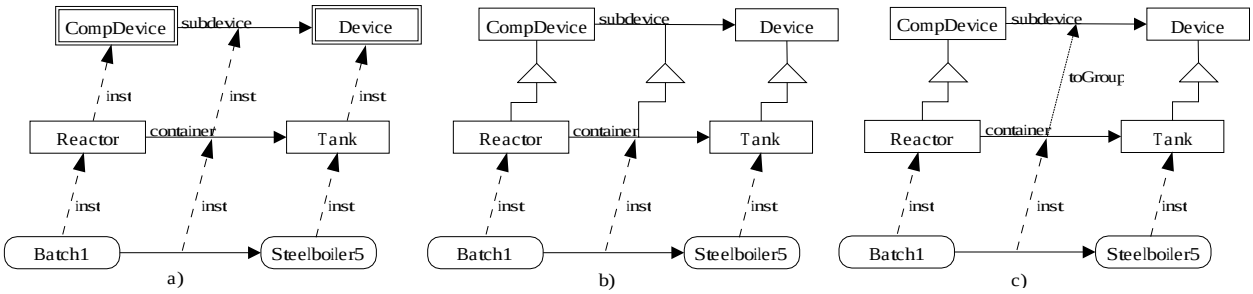


Abbildung 5.16: Metamodellierung und Alternativen: a) Metaklassen; b) Multiple Attributspezialisierung; c) Attributgruppierung

und -attribute werden wir im folgenden 'virtuelle' Metaklassen/-attribute nennen. Multiple Spezialisierung beinhaltet eine explizite Spezialisierungbeziehung zwischen Attributen. Standardobjektmodelle verwenden häufig die implizite Spezialisierung bei der aufgrund von Namensgleichheiten auf eine Spezialisierungsrelation geschlossen wird. Hier können sich hingegen spezialisierendes Attribut und spezialisiertes Attribut (also das virtuelle Metaattribut) im Namen unterscheiden. Die implizite Spezialisierung von Attributen über Attribute gleichen Namens läßt sich einfach auf die multiple Spezialisierung abbilden, umgekehrt ist dies natürlich nicht der Fall.

Die Spezialisierungsrelation übernimmt in dieser Modellierungsalternative die Gruppierungsfunktionen der Metamodellierung. Die Möglichkeit direkt Attribute und Methoden für Instanzen zu definieren ergibt sich auf natürliche Weise dadurch, daß die 'Metainformation' tragenden Konzepte in der Instanziierungshierarchie direkt über den Instanzen liegen. Lediglich Methoden- oder Restriktionsdefinitionen über Klassen (Metaklassenfunktion 3) können durch dieses Konzept nicht verwirklicht werden.

Allerdings zeigt bereits [Klas und Schrefl 1995, S.41ff] Unterschiede der Verwendung von Metaklassen mit 'instance–instance–types' gegenüber der Verwendung von Superklassen auf. Zwar entspricht dies nicht genau unserer Verwendung, läßt aber die Vermutung zu, daß die oben aufgezeigte Verwendung von Superklassen und -attributen keine vollständig äquivalente Semantik zu Metaklassen und -attributen hat. Tatsächlich erlaubt das vorgestellte Modell zumindest in zwei Fällen *mehr* Relationen als die Instanziierung, was mit einer verringerten Semantik der entsprechenden Relationen einhergeht. Zum einen können 'virtuelle' Metaattribute wie `subdevice` direkte Instanzen auf der Instanzebene haben, was bei 'echten' Metaattributen nicht der Fall ist. Zum anderen erlauben die normalerweise verwendeten Definitionen der Spezialisierung, daß sich die von Attributen und von Klassen ausgehenden Spezialisierungsbeziehungen auf verschiedene Ebenen der Spezialisierungshierarchie beziehen (siehe Abbildung 5.17).

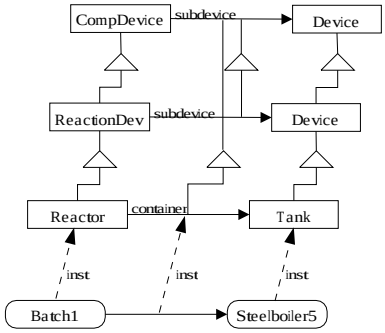


Abbildung 5.17: Multiple Spezialisierung ≠ Metaklassen

Diese Probleme ergeben sich daraus, daß eine Beziehungsart (die Spezialisierungsrelation) zwei verschiedene Beziehungen repräsentieren muß (Spezialisierung von Klassen sowie Metaklassenbeziehungen zwischen Klassen). Zudem läßt sich Multiple Spezialisierung nicht einfach auf die implizite Spezialisierung abbilden. Diese wird jedoch in Objektmodellen ohne eigenständige Attributobjekte[17], wie etwa den C++, Smalltalk, und ähnlichen Sprachen zugrundeliegenden Modellen, eingesetzt, so daß Multiple Spezialisierung dort nicht zur Verfügung steht.

Die Spezialisierung löst also einige der oben aufgeführten Probleme der Metamodellierung, kann aber die ursprüngliche Semantik nicht erhalten. Eine neue, an die genauen Bedürfnisse angepaßte Relation wird darum eingeführt.

[17] Engl.: attributes as first class citizens.

Attributgruppierung statt Instanziierung

Um eine alternative Repräsentation für Metaklassen zu erhalten, die deren Verhalten besser entspricht als die Multiple Spezialisierung, wird eine neue Relation “toGroup“ und eine Menge “groupleader“ eingeführt. toGroup ersetzt die Spezialisierungsbeziehung zwischen Attribut und virtuellem Metaattribut. groupleader enthält alle virtuellen Metaattribute und dient der Konsistenzsicherung, indem nur explizit als groupleader gekennzeichnete Attribute als virtuelle Metaattribute genutzt werden dürfen. Der Einsatz der Relation läßt sich grob in Abbildung 5.16 c ersehen. Die Semantik der neuen Relation läßt sich informell folgendermaßen beschreiben (eine formale Definition befindet sich in Kapitel 6.3): Die Beziehungen zwischen Attributen und Metaattributen werden durch toGroup–Relationen repräsentiert. toGroup wird hierbei ähnlich definiert wie die Spezialisierungsrelation. Dies hat zur Folge, daß eine Klasse mit einem gruppierten Attribut eine Spezialisierung der Klasse mit dem entsprechenden virtuellem Metaattribut sein muß. Gleiches gilt für die Zielklassen von Attribut und Metaattribut. Im Gegensatz zur multiplen Spezialisierung fordert die toGroup–Relation zusätzlich, daß immer die speziellste Version eines Metaattributs referenziert wird. Zudem sind direkte Instanziierungen eines in der Menge Groupleader befindlichen Attributs nicht erlaubt. Zusätzlich wird noch definiert, daß auf Attribute bzw. auf eine Attributmenge auch unter dem Namen ihres Groupleaders (also ihres durch toGroup zugeordneten Metaattributs) zugegriffen werden kann. Dadurch erhält die Attributgruppierung die gleichen erwünschten Eigenschaften wie die multiplen Spezialisierung, allerdings ohne die unerwünschten zusätzlichen Relationsmöglichkeiten.

Ähnlich der multiplen Spezialisierung werden auch bei der Attributgruppierung die gewünschten Metamodellierungsfunktionalitäten bereitgestellt und die Metaklassennachteile vermieden. Klassen und Attribute werden über die Spezialisierungs- bzw. die toGroup–Relation gruppiert. Die Gruppierungsfunktionalität der toGroup–Relation muß natürlich bei Ihrer Implementierung berücksichtigt werden (vgl. S. 116). Durch die Abbildung der Metaklassen in die Klassenebene ergibt sich ein zweistufiges Klassen–Instanzen–Schema, das die auf Seite 112 beschriebenen Probleme nicht kennt: Die Abbildung auf konventionelle Programmiersprachen ist problemlos möglich (s. S. 116), Instanzattribute und -methoden können auch ohne Verwendung von Hilfskonstrukten wie Metaformeln direkt definiert werden, da die virtuellen Metaklassen in der Klassenebene definiert werden, und eine Klasse kann gleichzeitig Klassen- und Metaklassenanteile enthalten. Als Nachteil gegenüber den Metaklassen ergibt sich jedoch, daß virtuelle Metaklassen keine Methoden oder Restriktionen über Klassen definieren können. Erfreulicherweise ist dies jedoch die Funktionalität, die durch die Modellierungsmodellierung am wenigsten genutzt wird.

Ein genauerer Vergleich dieser drei Repräsentationsformen für Metamodelle auf formaler Basis findet sich in [Baumeister 1996].

5.4.3 Anwendung der Attributgruppierung

Die bisherigen Ausführungen haben die Attributgruppierung zwar vorgestellt, einfache Anwendungsregeln und mögliche Implementierungen aber ausgeblendet. Dies soll nun nachgeholt werden.

Vereinfacht gesagt entspricht die Verwendung eines Groupleader–Attributs der einer Menge, deren Elemente einzeln benannt und typisiert werden müssen. Die Benennung und Typisierung der einzelnen Elemente erfolgt durch die Definition der gruppierten Attribute. Als Typen sind dabei nur Spezialisierungen des Elementtypen[18] des Groupleader–Attributs zulässig. Bei einem Zugriff auf das Groupleader–Attribut erhält man dann die gesamt Menge als Rückgabewert, bei einem Zugriff auf ein benanntes Element der Menge nur dieses. Da jeder Wert einer solchen Attributmenge einem benannten Element entsprechen muß, sind direkte Schreibzugriffe auf das Groupleader–Attribut nicht erlaubt.

[18] Elementtyp meint hier den um den äußersten Mengenkonstruktor reduzierten Typen.

Verwendung als Metaattributersatz

Die Sichtweise der Attributgruppierung als einzeln benenn- und typisierbare Mengen widerspricht nicht der Intention, mittels Attributgruppierung virtuelle Metaattribute zu repräsentieren. Schließlich umfassen Klassen, wie im Grundlageneinschub 2.2 herausgestellt, immer auch eine Extension, also die Menge aller Instanzen. In einer zweistufigen objektorientierten Sprache wie VDDL kann die Attributgruppierung somit eingesetzt werden, um Metaklassen innerhalb der Klassenebene abzubilden. Voraussetzung hierfür ist selbstverständlich, daß die Metaklassen keinen Gebrauch von von den durch die Attributgruppierung nicht unterstützten Funktionalitäten machen, d.h. keine Methoden oder Constraints enthalten, die sich auf die Klassenebene beziehen. *Attribute* für Klassen können hingegen auf die `shared-*-attributes` von VDDL abgebildet werden. Die Vorgehensweise ist dann folgendermaßen:
Für jede Metaklasse:

1. Erzeuge eine Klassen entsprechenden Namens und ordne sie in die Spezialisierungshierarchie der virtuellen Metaklassen ein.
2. Alle sich auf Klassen beziehenden Attribute werden als `shared-*-attributes` modelliert, alle übrigen Attribute als `individual-*-attributes`.
3. Alle Attribute, die Attributkategorien darstellen, erhalten eine `:groupleader t`–Facette, sie werden also ein Groupleader–Attribut.
4. Eine etwa auf der Klassenebene existierende kooperierende Klasse der Metaklasse (zur Umgehung des Problems von Seite 112), die für Instanzen bestimmte Attribute, Methoden und Constraints enthält, kann in die neue Klasse integriert werden.

Für jede Klasse:

1. Erzeuge eine Klasse entsprechenden Namens und ordne sie in die Spezialisierungshierarchie der Klassen ein.
2. Die Instanziierungsbeziehung zur Metaklasse wird durch eine Spezialisierungsbeziehung zur noch obiger Anleitung erzeugten Ersatzklasse ersetzt.
3. Attributkategorien werden durch `:to_group <kategoriename>`–Facetten ersetzt, die jedes Attribut der Kategorie erhält.
4. Alle übrigen Elemente der Klasse können eins zu eins übernommen werden.

Durch Umkehrung dieser Regeln kann ein entsprechendes Klassenschema wieder in Klassen und Metaklassen zerlegt werden.

Verwendung als verfeinerbare Mengen

Eine alternative Anwendung der Attributgruppierung besteht darin, sie gemäß der am Anfang erwähnten "benannte Menge"–Metapher zu benutzen, und zwar ohne daß ein entsprechendes Klassen/Metaklassen–Schema existiert bzw. existieren kann. Hierdurch kann eine weitergehende hierarchische Unterteilung von Mengen ähnlich einer Spezialisierungsrelation erreicht werden.

Abbildung 5.18 zeigt ein Beispiel bei der diese Verwendungsart auf die Spitze getrieben wurde, indem verschieden feine Untergliederungen der in einem Prozeß auftretenden Stoffe modelliert werden. Die Attributgruppe der charakterisierenden Attribute wird in der Abbildung weniger als Metaattribut verwendet, sondern in `Device` und `Reactor` hierarchisch untergliedert. Lediglich bei den Übergängen von `characterizing_attributes` auf `substances` und von `primary_products` auf `eg` kommt die Metaattributfunktionalität zum Vorschein.

Zwei Besonderheiten dieser Verwendungsform lassen sich zudem anhand der Abbildung demonstrieren. Zum einen stimmt die Umschreibung "Menge mit benennbaren *Elementen*" nicht vollständig. Bei dieser Verwendung werden vielmehr *Teilmengen* einer Attributmenge benannt. Es handelt sich hier also um eine Spezialisierung von Attributen. Änderungen an der Formalisierung in Kapitel 6.3

```
class:                          DEVICE
metaclasses:                    CONCEPT
individual-relational-attributes:
   characterizing-attribs:      :dom { ROOT } :groupleader t
   substances:                  :dom { SUBSTANCE } :groupleader t
                                   :to_group characterizing-attribs
END

class:                          REACTOR
metaclasses:                    CONCEPT
superclasses:                   DEVICE
individual-relational-attributes:
   products:                    :dom #>=1{ SUBSTANCE } :groupleader t
                                   :to_group substances
   educts:                      :dom #>=2{ SUBSTANCE } :groupleader t
                                   :to_group substances
   primary_products:            :dom #>=1{ SUBSTANCE } :groupleader t
                                   :to_group products
   by_products:                 :dom { SUBSTANCE } :groupleader t
                                   :to_group products
   waste:                       :dom { SUBSTANCE } :groupleader t
                                   :to_group products
END

class:                          EOEG-REACTOR
metaclasses:                    CONCEPT
superclasses:                   REACTOR
individual-relational-attributes:
   eg:                          :dom ETHYLEN_GLYCOLS :to_group primary_products
END
```

Abbildung 5.18: Verwendung der Attributgruppierung zur Strukturierung von Mengenattributen

sind deswegen allerdings nicht notwendig, da der verwendete Formalismus alle Attribute als potentielle Mengen ansieht. Bei der Implementierung der Attributgruppierung in einer Sprache, die mengenwertige und elementare Attribute unterscheidet, müssen allerdings entsprechende Vorkehrungen getroffen werden, wenn Attributgruppierung auch zur Strukturierung von Mengenattributen zum Einsatz kommen soll.

Zum anderen läßt sich für die Klassen auf Bild 5.18 kaum noch eine passende Metaklassen–Klassen–Hierarchie angeben, die den modellierten Zusammenhängen entspricht. Dieses Problem stimmt mit der Aussage aus [Motschnig-Pitrik und Mylopoulos 1992, S. 73] überein, daß bei ein Zusammenfassen aller Metalevel in einem (etwa durch die Relation `Class instance_of Class`) die konzeptuelle Klarheit abnimmt.

Implementierung in C++

Wie in den vorhergehenden Abschnitten gezeigt, kann Attributgruppierung verwendet werden, um Teilfunktionalitäten der Metamodellierung zu ermöglichen, ohne daß Metaklassen vorhanden sind. Interessant ist dies natürlich vor allem für Sprachen, die lediglich Instanz- und Klassenebene besitzen. Am Beispiel von C++ wird im folgenden aufgezeigt, wie sich Attributgruppierung nicht nur als konzeptuelles Werkzeug zur Struktierierung von Datenmodellen eignet, sondern daß sich entsprechende Modelle auch in die Implementierungssprache abbilden lassen.

Attributgruppierung kann im Gegensatz zu Multipler Spezialisierung einfach in Systeme ohne eigenständige Attribute abgebildet werden (vgl. [Baumeister 1996]). Dementsprechend bereitet auch

die Implementierung in C++ wenig Probleme. Allerdings werden Groupleader–Attribute nicht als Attribute sondern als Methoden implementiert. Eine Übersetzung der Klassen `Device` und `EOEG-Reactor` aus Bild 5.18 zeigt Abbildung 5.19. Hierbei werden die Groupleader–Attribute als virtuelle

```
class Device
{
    virtual set<Object> getcharacterizingattribs() { return(getsubstances()); }
    virtual set<Substances> getsubstances() { return(empty_set); }
    ...
}

class EOEG-Reactor : public Reactor
{   EthylenGlycols *eg;

    virtual set<Substances> getsubstances()
        { return(union(Reactor::getsubstances(),eg)) }
    ...
}
```

Abbildung 5.19: Implementierung der Attributgruppierung in C++

Methoden implementiert, die in den Klassen entsprechend den dort jeweils vorhandenen Attributen der Attributgruppe erweitert werden. Die Attribute selber, ob zu einer Attributgruppe gehörig oder nicht, werden als C++–Attribute implementiert. Bei geschachtelten Attributgruppen, wie im Beispiel in Abbildung 5.18, wird nur die jeweils innerste Groupleader–Methode verfeinert.

Die Übersetzung von einer facettenbasierten Darstellung wie in Bild 5.18 zur Implementierung läßt sich automatisieren. Ein entsprechender Präcompiler müßte auch einige der auf Seite 114 aufgestellten Restriktionen überprüfen, da nicht alle automatisch durch die Typtests des C++–Compilers sichergestellt werden[19].

Im Rahmen des in dieser Arbeit vorgestellten Objektmodells konnte diese einfache und auf Zugriffsmethoden basierende Implementierung der Attributgruppierung nicht verwendet werden. Die zusätzlich notwendige Modifizierbarkeit des Klassenschemas während der Programmausführung läßt sich in C++–basierten Sprachen so nicht erreichen, da Klassen und Metaklassen durch C++–Klassen repräsentiert würden, die zur Laufzeit nicht modifizierbar sind.

5.5 Zusammenfassung

Der Anfang dieses Kapitels hat gezeigt, daß sich gleichzeitige einfache Modifizierbarkeit von Objekten und hierarchische Suche nach ihnen nicht durch eine wie auch immer geartete 'Vereinigung' eines klassenbasierten und eines prototypbasierten Ansatzes erreichen läßt, da sich die Faktoren, die für die aus dem jeweiligen Ansatz erwünschte Eigenschaft verantwortlich sind, widersprechen. Aufbauend auf Objektmigration und multipler Instanziierung wurde darum ein Verfahren vorgestellt, das die strikte Templatebeziehung eines klassenbasierten Objektmodells durch eine eingeschränkte, dynamische ersetzt. Zudem werden in dem vorgeschlagenen klassenbasierten Modell Änderungsoperationen des prototypbasierten Objektmodells, etwa die Manipulation des Attributtyps eines Objekts, auf vorhandene Manipulationsmöglichkeiten abgebildet, also emuliert. Für dieses klassenbasierte Modell wurden zwei Erweiterungen der Objektorientierung vorgeschlagen: Aspekte und Attributgruppierung. Aspekte, eingesetzt, um die im Verlauf der Emulation entstehenden Klassen effizient repräsentieren zu können, erzielen gleichzeitig auch eine verbesserte Strukturierung der Klassenhierarchie. Die Attributgruppierung löst die Probleme der Abbildung zweier Klassenebenen VeDAs auf nur

[19] Die Existenz einer `:groupleader t`–Facette und das Nichtvorhandensein von Zwischenklassen müssen explizit überprüft werden.

eine, wobei die für die Anwendung wesentlichen Teile der Klassen/Metaklassen–Semantik erhalten werden konnten. Durch die Repräsentation in nur eine Ebene wird zudem ein fließender Übergang zwischen den beiden Ebenen möglich.

Das nächste Kapitel wird die beiden vorgestellten Erweiterungen formalisieren, bevor das abschließende Kapitel untersucht, inwieweit der entwickelte Ansatz die in den Kapitel 1 und 3 aufgeworfenen Anforderungen erfüllt.

Kapitel 6

Formalisierung der Erweiterungen

6.1 Die Notation

Zur Formalisierung wird in den folgenden Abschnitten der Formalismus verwendet, den M. Jeusfeld zur Axiomatisierung von O-Telos definiert hat (siehe [Jeusfeld 1992; Mylopoulos et al. 1990]). Dieser erlaubt es, Objekte, Klassen und Attribute (also die strukturellen Konzepte der Objektorientierung) auf Fakten, Regeln und Constraints einer deduktiven Datenbank abzubilden. Hierzu wird alle Information auf Quadrupel der Form $P(o, x, l, y)$ abgebildet, wobei o ein Identifikator und l ein Label ist, und x und y Identifikatoren anderer Tupel sind. Ein solches Quadrupel wird auch *Proposition* genannt.

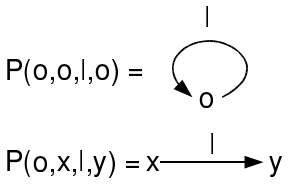


Abbildung 6.1: Propositionen

Eine Proposition der Form $P(o, o, l, o)$ beschreibt das Objekt o (siehe Abbildung 6.1), wobei o der Objektidentifikator und l der Name des Objekts ist. Eine Proposition der Form $P(o, x, l, y)$ repräsentiert das Attribut l des Objekts x mit Wert y ($o \neq x, o \neq y$). Spezialformen der Attributbeziehung sind die Instanziierung mit Label "In" und die Subklassenbeziehung mit Label "IsA". Man beachte, daß in diesem Modell jede Proposition einen Identifikator besitzt und somit auch Attribute und Instanziierungs-/Spezialisierungsbeziehungen eine eigene Objektidentität besitzen.

Um die Semantik der Propositionen zu definieren, führt Jeusfeld zusätzlich zu den Axiomen einer beweistheoretischen, relationalen Datenbanktheorie nach [Reiter 1984] 33 Axiome ein (s. [Jeusfeld 1992, S.45–50,52]), die als Regeln und Constraints ein Verhalten der Propositionen analog zu einem strukturellen Objektmodell mit Spezialisierung, beliebig vielen Instanziierungsebenen und dynamisch änderbaren Schemainformationen beschreiben. Einige dieser Axiome führen Systemklassen ein und sind weniger interessant. Die übrigen werden im weiteren vorgestellt.

6.1.1 Eindeutigkeit von Identifikatoren und Beziehungen

Die folgenden Axiome fordern die Eindeutigkeit oder korrekte Belegung von Identifikator beziehungsweise Namen einer Proposition. Zuerst wird die Eindeutigkeit des Objektidentifikators — also des ersten Elements jeder Proposition — gefordert.

$$\begin{aligned} \forall o, x_1, y_1, x_2, y_2 \in \mathrm{ID} \;\; \forall l_1, l_2 \in \mathrm{LAB} \;\; & P(o, x_1, l_1, y_1) \wedge P(o, x_2, l_2, y_2) \\ & \Rightarrow (x_1 = x_2) \wedge (l_1 = l_2) \wedge (y_1 = y_2) \end{aligned} \tag{A1}$$

Hierbei ist ID die Menge aller Objektidentifikatoren und LAB die Menge aller Labels. Aufgrund seiner in (A1) definierten Eindeutigkeit kann der Identifikator in den folgenden Axiomen zur eindeutigen Referenzierung einer Proposition verwandt werden. Zusätzlich sollen die eigentlichen Objekte — also

Propositionen der Form $P(o,o,l,o)$ — eindeutig über ihren Namen referenziert werden können.

$$\forall o_1, o_2 \in \text{ID} \;\; \forall l \in \text{LAB} \;\; P(o_1,o_1,l,o_1) \wedge P(o_2,o_2,l,o_2) \Rightarrow (o_1 = o_2) \tag{A2}$$

Namen und Identifikatoren von Beziehungen werden durch drei andere Axiome eingeschränkt. Sie stellen sicher, daß der Attributname bezüglich einer anderen Proposition eindeutig ist, das heißt, daß jedes Objekt nur eine Attributproposition mit einem bestimmten Namen besitzen kann:

$$\begin{aligned}\forall o_1, o_2, x, y_1, y_2 \in \text{ID} \;\; \forall l \in \text{LAB} \;\; & P(o_1,x,l,y_1) \wedge P(o_2,x,l,y_2) \\ & \Rightarrow (o_1 = o_2) \vee (l = in) \vee (l = isa)\end{aligned} \tag{A3}$$

Man beachte, daß Attribute nicht zwangsweise von Objekten ausgehen müssen, vielmehr kann x in obigem Axiom auch die Referenz auf ein anderes Attribut sein. Letztendlich gelangt man aber über die Verfolgung der Ausgangsreferenzen immer zu einem Objekt (siehe Axiom 33 unten).

Die Instanziierungs- und die Spezialisierungsbeziehung sind von dieser Beschränkung ausgenommen, so daß ein Objekt Instanz oder Spezialisierung von mehreren Objekten sein kann — man erinnere sich, daß auch Klassen Objekte sind. Allerdings müssen Instanziierung und Spezialisierung zwischen *zwei bestimmten* Objekten eindeutig sein, es darf also nicht zwei Instanziierungs- bzw. Spezialisierungsbeziehungen zwischen dem gleichen Paar von Objekten geben:

$$\begin{aligned}\forall o_1, o_2, x, y \in \text{ID} \; \forall l \in \text{LAB} \; & P(o_1,x,l,y) \wedge P(o_2,x,l,y) \wedge \\ & ((l = in) \vee (l = isa)) \Rightarrow (o_1 = o_2)\end{aligned} \tag{A4}$$

Man beachte, daß nicht verlangt wurde, daß die Quelle einer Instanziierungs-/Spezialisierungsbeziehung ein Objekt, also eine Proposition der Form $P(o,o,l,o)$ sein muß. Mithin können auch Attribute Instanzen oder Spezialisierungen anderer Attribute sein.

Allerdings können im bisherigen Axiomensystem noch "Schleifenbeziehungen" auftreten. Dies sind Beziehungen, bei denen die Verfolgung der "Quelle"-Referenzen niemals zu einem Objekt führt, indem die Attribute sich zyklisch als Quellreferenz benutzen. Da dies keinen Sinn macht und auch in Objektsystemen nicht auftritt, wird es durch die Definition einer Totalordnung $\leq$ auf den Identifikatoren verboten:

$$\forall o, x, y \in \text{ID} \;\; \forall l \in \text{LAB} \;\; P(o,x,l,y) \Rightarrow (x \leq o) \wedge (y \leq o) \tag{A33}$$

6.1.2 Einführung von Hilfsprädikaten

Um die Notation von Instanziierung, Spezialisierung und Attributierung in den folgenden Axiomen und den in TELOS möglichen deduktiven Regeln zu vereinfachen, werden drei Hilfsprädikate eingeführt, die diese Beziehungen von ihrem jeweiligen Objektidentifikator unabhängig machen. Dies ist möglich, da die Axiome des letzten Abschnitts gerade die Eindeutigkeit dieser Beziehungen unabhängig von ihrem Objektidentifikator fordern.

Die drei Prädikate In, Isa und A werden durch die folgenden Regeln definiert:

$$\forall o, x, c \in \text{ID} \;\; P(o,x,in,c) \Rightarrow In(x,c) \tag{A5}$$

$$\forall o, c, d \in \text{ID} \;\; P(o,c,isa,d) \Rightarrow Isa(c,d) \tag{A6}$$

$$\begin{aligned}&\forall o, p, x, y, c, d \in \text{ID} \;\; \forall l, m \in \text{LAB} \\ &P(o,x,l,y) \wedge P(p,c,m,d) \wedge In(o,p) \Rightarrow A(x,m,y)\end{aligned} \tag{A7}$$

Man beachte, daß beim Attributierungsprädikat A der Label m den in der Klasse definierten Namen des Attributs enthält, nicht den in der Instanz angegebenen. Zudem erfordert A immer einer Klasse, die das entsprechende Attribut definiert, während $P(o, x, l, y)$ beliebig zu Objekten hinzugefügt werden kann.

6.1.3 Spezialisierung und Instanziierung

Die nun folgenden Axiome definieren die Semantik der Spezialisierungs- und Instanziierungsbeziehung genauer. Zuerst wird gefordert, daß Klassen strikte Templates (s. Kap. 4.1) bezüglich des Attributierungsprädikats sind mit der Einschränkung, daß alle Attribute lediglich optional sind.

$$\begin{aligned}\forall x, y, p, c, d \in \mathrm{ID}\ \ \forall m \in \mathrm{LAB}\ \ In(x, c) \wedge A(x, m, y) \wedge P(p, c, m, d) \\ \Rightarrow \exists o \in \mathrm{ID}\ \ \exists l \in \mathrm{LAB}\ \ P(o, x, l, y) \wedge In(o, p)\end{aligned} \tag{A8}$$

Wenn also x Instanz von c ist und ein Attribut namens m hat und in c ein Attribut gleichen Namens definiert wird, so muß das Attribut von x eine Instanz des Attributs von c sein. Zusammen mit (A7) ergibt sich somit die Striktheit der Klassentemplates bezüglich A. Umgekehrt folgt aus einer Instanziierungsbeziehung zwischen zwei Propositionen, daß eine analoge Instanziierungsbeziehung jeweils für deren Quellen und Senken gelten muß:

$$\begin{aligned}\forall o, p, x, y \in \mathrm{ID}\ \ \forall l \in \mathrm{LAB}\ \ P(o, x, l, y) \wedge In(o, p) \\ \Rightarrow \exists c, d \in \mathrm{ID}\ \ \exists m \in \mathrm{LAB}\ \ P(p, c, m, d) \wedge In(x, c) \wedge In(y, d)\end{aligned} \tag{A13}$$

Die Spezialisierungsrelation wird durch drei Axiome als partielle Ordnung spezifiziert:

$$\forall c, x, y \in \mathrm{ID}\ \ \forall l \in \mathrm{LAB}\ \ P(c, x, l, y) \Rightarrow Isa(c, c) \tag{A9}$$

$$\forall c, d, e \in \mathrm{ID}\ \ Isa(c, d) \wedge Isa(d, e) \Rightarrow Isa(c, e) \tag{A10}$$

$$\forall c, d\ \in \mathrm{ID}\ \ Isa(c, d) \wedge Isa(d, c) \Rightarrow (c = d) \tag{A11}$$

Die Vererbung von Attributen zwischen Klassen wird realisiert, indem die Instanziierungsbeziehung sozusagen 'transitiv' über die Spezialisierungsbeziehung erweitert wird. Das heißt, daß aus der Existenz einer Instanziierungs- und einer passenden Spezialisierungsbeziehung auf die Existenz einer Instanziierungsbeziehung zur Oberklasse geschlossen werden kann:

$$\forall p, x, c, d \in \mathrm{ID}\ \ In(x, d) \wedge P(p, d, isa, c) \Rightarrow In(x, c) \tag{A12}$$

Der Zusammenhang zwischen Attributen und Spezialisierung gestaltet sich ähnlich zu dem zwischen Attributprädikat und Instanziierung. Aus der Spezialisierung der Propositionsquellen folgt bei gleichem Namen die Notwendigkeit einer Spezialisierung der Propositionen, und die Spezialisierung der Propositionen fordert jeweils eine Spezialisierungsbeziehung zwischen Quellen und Zielen (siehe auch Abbildung: 6.2)

$$\begin{aligned}\forall c, d, a_1, a_2, e, f \in \mathrm{ID}\ \ \forall m \in \mathrm{LAB} \\ Isa(d, c) \wedge P(a_1, c, m, e) \wedge P(a_2, d, m, f) \Rightarrow Isa(a_2, a_1)\end{aligned} \tag{A14}$$

[1]

$$\begin{aligned}\forall c, d, a_1, a_2, e, f \in \mathrm{ID}\ \ \forall m_1, m_2 \in \mathrm{LAB} \\ Isa(a_2, a_1) \wedge P(a_1, c, m_1, e) \wedge P(a_2, d, m_2, f) \Rightarrow Isa(d, c) \wedge Isa(f, e)\end{aligned} \tag{A15}$$

Schließlich wird noch die Auflösung von Konflikten bei multipler Instanziierung (und somit wegen

[1] [Jeusfeld 1992] verlangt hier unnötigerweise zusätzlich $Isa(f, e)$. Dies folgt aber aus $Isa(a_2, a_1)$ und (A15).

(A12) auch bei multipler Vererbung) behandelt:

$$\forall x, y, c, d, a_1, a_2, e, f \in \mathrm{ID}\ \ \forall m \in \mathrm{LAB}\ \ In(x,c) \wedge In(x,d) \wedge P(a_1,c,m,e) \wedge P(a_2,d,m,f) \\ \Rightarrow \exists a_3, g, h \in \mathrm{ID}\ \ In(x,g) \wedge P(a_3,g,m,h) \wedge Isa(g,c) \wedge Isa(g,d) \tag{A16}$$

Wenn es also zwei Klassen c und d gibt, die beide ein Attribut mit Namen m definieren und beide von einem Objekt instanziiert werden, so muß es eine dritte Klasse g geben, die ebenfalls eine Attribut namens m definiert und Spezialisierung von c und von d ist. Aufgrund von (A14) folgt dann, daß das in g definierte Attribute eine Spezialisierung der in c und d definierten Attribute gleichen Namens sein muß. Das in g definierte Attribut löst somit den Konflikt zwischen c und d auf.

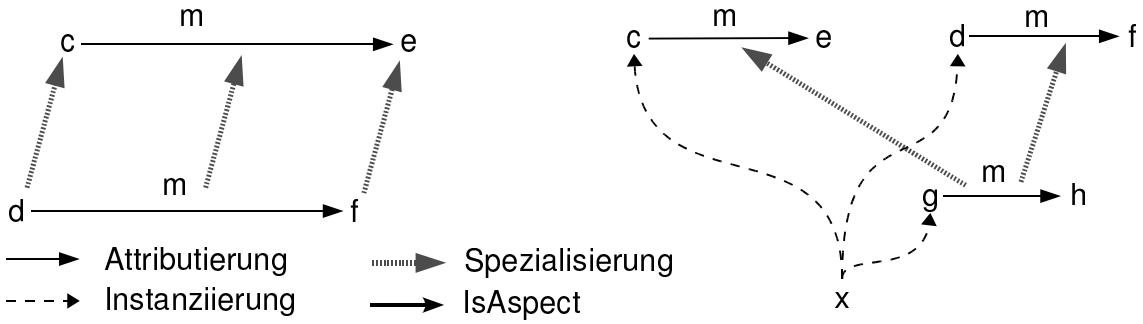


Abbildung 6.2: Axiome 14 und 15 sowie 16

Auf Grundlage dieses Axiomensystems zeigt [Jeusfeld 1992] verschiedene Eigenschaften der sich ergebenden deduktiven Objektbanktheorie, etwa die referentielle Integrität und die Eindeutigkeit von Attributklassen. Hierzu teilt er die vorgestellten Axiome auf in Regeln (A5,A6,A7,A9, A10,A12) und Integritätsbedingungen (A1–A4,A8,A11,A13–A16,A33).

6.2 Änderungen für Aspekte

Aspekte können in den Formalismus relativ einfach eingefügt werden, da O-Telos bereits die multiple Instanziierung unterstützt. Zudem muß für die Aspekte selbst kein neues Konstrukt eingeführt werden. Sie werden wie Klassen und Objekte als Individual–Objekte repräsentiert. Ihre in Kapitel 5.3 beschriebenen Eigenschaften ergeben sich dann aus den Axiomen der hier neu eingeführten 'isAspect'–Relation. Der Ausgangspunkt einer 'isAspect'–Relation ist somit ein Aspekt, der Zielknoten ein Aspekthalter. Die Unterscheidung zwischen notwendigen und optionalen Aspekten erfolgt durch einen 'required' Link von der jeweiligen 'isAspect'–Relation auf sie selbst.

6.2.1 Änderungen am Axiomensystem

Die an den O-Telos-Axiomen notwendigen Änderungen lassen sich in drei Kategorien einteilen:

1. Erweiterung bekannter Axiome, so daß sie die 'isAspect'–Relation unterstützen
2. Einfügen neuer, 'isAspect' definierender Axiome
3. Streichung alter, dem Aspektkonzept wiedersprechender Axiome.

Die Änderungen bekannter Axiome betreffen die Einführung von 'isAspect' als spezieller Relation ähnlich 'In', 'IsA' und 'A' (A3,A4) sowie die Unterscheidung zwischen der direkten Instanziierung 'In' und 'In^*', dem transitiven Abschluß der Instanziierungsrelation über die Spezialisierungsrelation (A7,A8,A12,A13). Zudem muß A15 angepaßt werden.

Gestrichen werden muß lediglich ein Axiom. A16 fordert bei Namenskonflikten aufgrund von multipler Instanziierung, daß eine gemeinsame Spezialisierung aller an dem Konflikt beteiligten Attribute

existiert und ebenfalls instanziiert wird. Da solche Namenskonflikte im Rahmen des Aspektkonzepts anders aufgelöst werden (vgl. Seite 5.3.2), muß dieses Axiom entfernt werden.

Bei den neu einzuführenden Axiomen lassen sich wieder zwei Arten unterscheiden. Zum einen solche, die Eigenschaften spezifizieren, die für andere in O–Telos vorhandenen Relationen ebenfalls definiert sind. Diese Axiome werden im folgenden gemäß der ihnen entsprechenden Axiome aus O–Telos eingeordnet. Die andere Art neuer Axiome sind solche, die zu keinem vorhandenen Axiom in Beziehung stehen. Diese werden weiter unten definiert und an der Stelle eingefügt, an der in O–Telos die Systemprädikate festgelegt werden.

Zuerst definieren wir 'IsAspect' als Abkürzung für P(_,_,isAspect,_) und legen fest, das IsAspect–Beziehungen nicht von Attributen ausgehen dürfen:

$$\forall o, c, d \in \mathrm{ID} \;\; P(o, c, isAspect, d) \Rightarrow In(c, \#Indiv) \tag{A7b}$$

$$\forall o, c, d \in \mathrm{ID} \;\; P(o, c, isAspect, d) \Rightarrow IsAspect(c, d) \tag{A7c}$$

Auf Seite 101 hatten wir gefordert, daß die Ähnlichkeit zwischen Spezialisierung und Bildung von Aspekten dadurch gewährleistet werden soll, daß in einem Aspekt auftretende Attribute gleichen Namens wie Attribute des Aspekthalters Spezialisierungen von diesen sein müssen. Wir fügen darum analog zu Axiom 14, welches diese Eigenschaft für Unterklassen definiert, ein Axiom 14b ein:

$$\begin{gathered}\forall c, d, e, a_1, a_2, f, g \in \mathrm{ID} \;\; \forall m \in \mathrm{LAB} \\ IsAspect^*(d, c) \wedge P(a_1, c, m, f) \wedge IsA(e, d) \wedge P(a_2, e, m, g) \Rightarrow Isa(a_2, a_1)\end{gathered} \tag{A14b}$$

Man beachte, daß e und d gleich sein können. Da A15 quasi die Umkehrung von A14 ist, muß nach Einführung von A14b auch A15 angepaßt werden und $IsAspect$ berücksichtigen.

$$\begin{gathered}\forall c, d, a_1, a_2, e, f \in \mathrm{ID} \;\; \forall m_1, m_2 \in \mathrm{LAB} \;\; \exists j, k \in \mathrm{ID} \\ Isa(a_2, a_1) \wedge P(a_1, c, m_1, e) \wedge P(a_2, d, m_2, f) \\ \Rightarrow (Isa(d, c) \vee (Isa(d, j) \wedge IsAspect^*(j, c)) \wedge (Isa(f, e) \vee (Isa(f, k) \wedge IsAspect^*(k, e))\end{gathered} \tag{A15}$$

Die Erweiterung um $Isa(d, h) \wedge IsAspect^*(h, c)$ ist notwendig, um der Tatsache Rechnung zu tragen, daß Attributspezialisierungen für die Quelle und das Ziel des Attributs sowohl Spezialisierungen als auch Verwendung eines Aspekts bedeuten können.

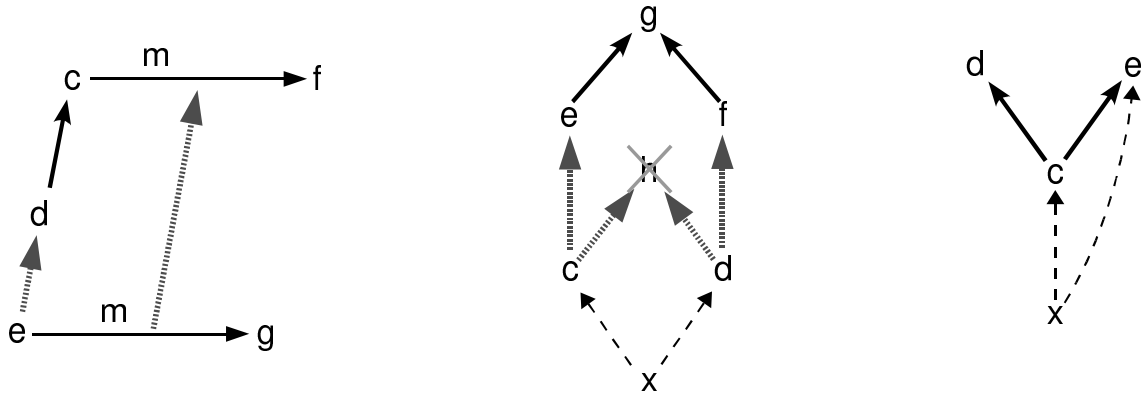


Abbildung 6.3: Axiome 14b, 22 und 23

Die weiteren Axiome entsprechen keinen O–Telos–Axiomen. Sie werden vor den O–Telos–Axiome zur Definition der Systemprädikate eingeschoben, die natürlich weiterhin benötigt werden.

Zuerst definieren vier Axiome $IsAspect^*$, den transitiven Abschluß von $IsAspect$ über sich selbst und die Spezialisierungsrelation. Dieser Abschluß wird in Axiom 22 und 14b benötigt. Man beachte, daß $IsAspect$ selbst nicht transitiv ist.[2]

$$\forall c, d \in \text{ID} \;\; IsAspect(c, d) \Rightarrow IsAspect^*(c, d) \tag{A17}$$

$$\forall c, d, e \in \text{ID} \;\; IsAspect^*(c, d) \wedge Isa(d, e) \Rightarrow IsAspect^*(c, e) \tag{A18}$$

$$\forall c, d, e \in \text{ID} \;\; IsAspect^*(c, d) \wedge IsAspect(d, e) \Rightarrow IsAspect^*(c, e) \tag{A19}$$

$$\forall c \in \text{ID} \;\; \neg IsAspect^*(c, c) \tag{A20}$$

Als weitere Eigenschaft von $IsAspect$ soll gelten, daß zwischen zwei Klassen nicht gleichzeitig ein Aspekt- und eine Subklassenbeziehung bestehen kann. Dazu ist lediglich das folgende Axiom notwendig, da sich alle übrigen, diese Forderung repräsentierenden Formeln daraus ableiten lassen (siehe Lemma 6.1 und 6.2).

$$\forall c, d \in \text{ID} \;\; IsAspect(c, d) \Rightarrow \neg Isa(c, d) \tag{A21}$$

Es werden nun kurz einige einfache Folgerungen aus obigen Axiomen eingefügt, die $IsAspect$ weiter spezifizieren. Aus A20 folgt, daß auch keine Subklassenbeziehung umgekehrt parallel zu einer Aspektbeziehung erlaubt ist.

Lemma 6.1 $\forall c, d \in ID \;\; IsAspect(c, d) \Rightarrow \neg Isa(d, c)$

Beweis: Andernfalls würde aus A18 folgen, daß $IsAspect^*(c, c)$, was A20 verbietet.

Lemma 6.2 *Aus (A21) und Lemma 6.1 folgt trivialerweise*
$\forall c, d \in ID \;\; Isa(c, d) \Rightarrow \neg(IsAspect(c, d) \vee IsAspect(d, c))$

Somit schließen sich Aspekt- und Subklassenbeziehung tatsächlich gegenseitig aus.
Aus Lemma 6.2 und (A9) folgt außerdem, daß $IsAspect$ nicht reflexiv ist, d.h. daß kein Aspekt Aspekt von sich selbst sein darf:

Lemma 6.3 $\forall c \in ID \;\; \neg IsAspect(c, c)$

Beweis: Wegen A9 gilt $\forall c \in \text{ID} \; Isa(c, c)$. Wegen Lemma 6.2 kann dann nicht $IsAspect(c, c)$ gelten. Alternativ läßt sich diese Eigenschaft auch aus A20, A19 und A18 folgern.
Ebenso folgt aus A20, A19 und A17, daß $IsAspect$ asymmetrisch ist:

Lemma 6.4 $\forall c, d \in ID \;\; IsAspect(c, d) \Rightarrow \neg IsAspect(d, c)$

In den nun folgenden drei Axiomen werden die Zusammenhänge zwischen Aspekt- und Instanziierungsbeziehung definiert. Als erstes wird die multiple Instanziierung eingeschränkt, so daß nur noch Klassen aus verschiedenen, aber dem gleichen Aspekthalter zugehörigen Aspekten gleichzeitig instanziiert werden dürfen.

$$\begin{gathered}\forall x, c, d \in \text{ID} \;\; c \neq d \wedge In(x, c) \wedge In(x, d) \Rightarrow \\ \exists e, f, g \in \text{ID} \;\; e \neq f \wedge Isa(c, e) \wedge Isa(d, f) \wedge IsAspect^*(e, g) \wedge IsAspect^*(f, g) \wedge \\ (\neg \exists h \in \text{ID} \quad Isa(c, h) \wedge Isa(d, h))\end{gathered} \tag{A22}$$

Für zwei unterschiedliche, von einem Objekt x instanziierte Klassen, muß es also zwei unterschiedliche Aspekte e und f geben, die transitiv zum gleichen Aspekthalter g gehören. Zudem darf es

[2] Gäbe es verschiedene IsAspect–Relationen für optionale und notwendige Aspekte, so wäre jede Relation für sich transitiv.

zwischen den Klassen und ihren jeweils übergeordneten Aspekten keine gemeinsame Oberklasse geben. Da $Isa(c, c)$ für alle c gilt, können e oder f natürlich auch gleich c oder d sein. Die Verwendung von $IsAspect^*$ anstelle von $IsAspect$ ist notwendig, da Aspekte selbst auch Aspekte besitzen können, so daß der gemeinsame Aspekthalter g mehrere $Isa/IsAspect$–Übergänge von e oder f entfernt liegen kann. Da es sich bei c und d um direkt instanziierte Klassen handeln muß, benötigt dieses Axiom die in A12 eingeführte Unterscheidung zwischen direkter und indirekter Instanziierung.
Als weitere Constraint kommt hinzu

$$\forall x, c, d \in \mathrm{ID} \;\; In^*(x, c) \wedge IsAspect(c, d) \Rightarrow \exists e \; IsAspect(c, e) \wedge In^*(x, e) \tag{A23}$$

Wird ein Aspekt instanziiert, so muß auch der Aspekthalter instanziiert werden. Da ein Aspekt mehrere Aspekthalter haben kann, von denen aber nur einer instanziiert werden muß, wird im Axiom zwischen d und e unterschieden. Natürlich kann $d = e$ gelten.
Vergleicht man dieses Axiom mit A14, so kann man feststellen, daß In^* analog zu A14 über $IsAspect$ erweitert wird. Allerdings nicht wie in A14, indem eine Regel definiert wird, die e in den transitiven Abschluß von In^* aufnimmt, sondern indem eine zusätzliche Instanziierungsbeziehung zu e gefordert wird.
Schließlich werden auch in der umgekehrten Richtung zusätzliche Instanziierungen gefordert:

$$\begin{aligned}\forall o_1, x, c, d, o_2 \in \mathrm{ID} \;\; In(x, d) \wedge P(o_1, c, isAspect, d) \wedge P(o_2, o_1, {}'required', o_1) \\ \Rightarrow In^*(x, c)\end{aligned} \tag{A24}$$

Wer einen Aspekthalter instanziiert, der muß auch alle notwendigen ('required') Aspekte instanziieren. Nicht durch den 'required'–Link gekennzeichnete Aspekte sind optional, und müssen darum nicht zwangsläufig angenommen werden. Da die Kennzeichnung den $IsAspekt$–Link betrifft, kann ein Aspekt bezüglich eines Aspekthalters notwendig, bezügliche eines anderen aber nur optional sein.
Wird die Definition der Aspektspezialisierung wie in Definition 5.14 gewünscht, so benötigt man noch eine weitere Einschränkung

$$\begin{aligned}\forall o, c, d, e, f, g \in \mathrm{ID} \;\; In^*(o, c) \wedge Isa(c, d) \wedge \\ IsAspect(e, c) \wedge IsAspect(f, d) \wedge Isa(e, f) \wedge Isa(g, f) \wedge In(o, g) \\ \Rightarrow Isa(g, e)\end{aligned} \tag{A25}$$

Spezialisiert also Aspekt e einen Aspekt f und instanziiert ein Objekt o den Aspekthalter c von e, dann kann o nur solche Subklassen g von f (also Aspektkinder von f) instanziieren, die auch Subklassen von e sind.

6.2.2 Ableitbare Eigenschaften

Aus den Axiomen wurden bereits im letzten Abschnitt einige einfache Eigenschaften von $IsAspect$ hergeleitet. Hier sollen nun komplexere Eigenschaften betrachtet werden: Zum einen, ob die hinzugefügten Axiome eine Auswirkung auf die Abbildbarkeit als Datalog-Datenbank haben, zum anderen, welche Einschränkungen bei Attributnamenskonflikten zwischen Aspekten zu beachten sind.

Abbildung auf Datalog

Damit auch die geänderte Menge von Axiomen auf Datalog abbildbar ist, müssen sich alle als Regeln aufzufassenden Axiome als Hornklauseln schreiben lassen. Von den neuen und geänderten Axiomen stellen lediglich A7c, A12 und A17 bis A19 Regeln dar, alle übrigen sind geschlossene Konsistenzbedingungen. Diese Axiome entsprechen aber alle der Form

$$\forall x_1, \ldots, x_n \;\; P_1(x_1, \ldots, x_n) \wedge \ldots \wedge P_m(x_1, \ldots, x_n) \Rightarrow P_0(x_1, \ldots, x_m)$$

mit gebundenen x_i und sind somit Hornklauseln. Die Abbildbarkeit auf Datalog hat sich somit durch vorgenommenen Anpassungen nicht geändert.

Typrestriktionen bei horizontalen Namenskonflikten

Auf Seite 103 wurde bereits eine Bedingung für die Typen gleichnamiger Attribute angegeben, deren Namenskonflikt im Rahmen von Aspekten gelöst werden soll: Die als Typen angegebenen Klassen müssen (indirekte) Aspektkinder derselben Aspekthalterklasse aber unterschiedlicher Aspekte sein. Diese Bedingung wurde dort anschaulich begründet. Hier wird gezeigt, daß es sich genau um die hinreichende und notwendige Bedingung für die Existenz einer multiplen Instanz zweier Klassen, zwischen denen ein Namenskonflikt existiert, handelt.

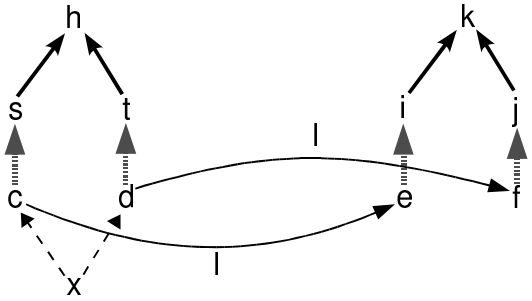


Abbildung 6.4: Bedingung für die Typen gleichnamiger Attribute (Satz 6.5)

Satz 6.5 *Für zwei Aspektkinder c und d des gleichen Aspekthalters h mit Attributen $P(x, c, l, e)$ und $P(y, d, l, f)$ kann genau dann eine gemeinsame Instanz o mit Attribut l existieren, wenn gilt:*

$$\exists i, j, k \in ID \;\; i \neq j$$
$$Isa(e, i) \wedge Isa(f, j) \wedge IsAspect^*(i, k) \wedge IsAspect^*(j, k) \wedge (\neg \exists m \in ID \;\; Isa(i, m) \wedge Isa(j, m))$$

Beweis:
1) Es wird zuerst gezeigt, daß die angegebene Bedingung hinreichend ist.
Vergleicht man die Existenzforderung obigen Satzes mit Axiom 22, so stellt man fest, daß die Existenzforderung äquivalent zum rechten Teil von A22 ist, so daß direkt folgt, daß eine Instanz p mit $In(p, e)$ und $In(p, f)$ existieren kann. Damit nun ein o mit $In(o, c)$ und $In(o, d)$ sowie mit Attribut l, also $A(o, l, \_)$, existieren kann, muß es wegen Axiom 8 ein $P(z, o, l, q)$ geben mit $In(z, x)$ und $In(z, y)$. Wegen Axiom 13 bedeutet das aber, daß $In(q, e)$ und $In(q, f)$ gelten muß. Setzt man $q = p$, so wurde die Existenzmöglichkeit einer solchen Instanz oben gezeigt. o kann also existieren.

2) Nun wird die Notwendigkeit der Bedingung gezeigt.
Es existiere ein o mit $In(o, c)$ und $In(o, d)$, das ein Attribut mit Namen l besitzt. Wie unter 1) gezeigt, muß darum auch ein q existieren mit $In(q, e)$ und $In(q, f)$. Aus Axiom 22 folgt in diesem Fall direkt die im Satz angegebene Existenzforderung. □

Korollar 6.6 *Gäbe es ein verschärftes Axiom 8, das die Vererbung von Attributen vorschreibt, also ein Axiom der Art*

$$\forall x, y, p, c, d \in ID \, \forall m \in LAB \; In(x, c) \wedge P(p, c, m, d) \Rightarrow$$
$$\exists o \in ID \, \exists l \in LAB \; P(o, x, l, y) \wedge In(o, p)$$

so ist in Satz 6.5 das Vorhandensein des Attributs l in o immer erfüllt. Es gilt dann also die verkürzte Form:

Für zwei Aspektkinder c und d des gleichen Aspekthalters h mit Attributen $A(c,l,e)$ und $A(d,l,f)$ gleichen Namens l kann genau dann eine gemeinsame Instanz o existieren, wenn gilt:

$$\exists i,j,k \in ID \;\; i \neq j Isa(e,i) \wedge Isa(f,j) \wedge IsAspect^*(i,k) \wedge IsAspect^*(j,k)$$
$$\wedge(\neg\exists m \in ID \;\; Isa(i,m) \wedge Isa(j,m))$$

In einer Sprache, in der obiges fiktives Axiom gilt (etwa C++ und Java), entspricht Satz 2 also einer allgemeinen Typrestriktion für die Verwendung von Aspektkindern. Er ersetzt somit das entfernte Axiom 16, läßt sich im Gegensatz zu diesem aber bereits auf Klassenebene überprüfen.

6.3 Änderungen für Attributgruppierung

Wie [Baumeister 1996] zeigt, ähnelt Attributgruppierung der multiplen Vererbung auf Attributen, wenn Attribute vollwertige Objekte sind (vgl. auch S. 114 f).[Baumeister 1996] definiert Attributgruppierung entsprechend ihrer Zielrichtung basierend auf einer Sprache mit an Klassen/Instanzen gebundenen Attributen. In dieser Arbeit soll jedoch auch bei der Definition der Attributgruppierung der oben aufgestellte Formalismus beibehalten werden. Dies hat unter anderem zur Folge, daß sich die Definition vereinfacht, da nur noch die Unterschiede zur Attributspezialisierung modelliert werden müssen.

Somit reichen folgende Axiome aus, um die Bedeutung der Relation *toGroup* und der Menge *Groupleader* zu definieren.

$$\forall a_1, a_2, \in \text{ID} \;\; toGroup(a_1,a_2) \Rightarrow Isa(a_1,a_2) \tag{A26a}$$

$$\forall a_1, a_2, \in \text{ID} \;\; toGroup(a_1,a_2)$$
$$\Rightarrow a2 \in Groupleader \wedge \neg\exists a_3 \in \text{ID} : Isa(a_1,a_3) \wedge Isa(a_3,a_2) \tag{A26b}$$

$$\forall a,c,d \in \text{ID} m \in \text{LAB} a \in Groupleader \wedge P(a,c,m,d) \wedge$$
$$\Rightarrow \neg\exists o \in \text{ID} : In(o,c) \tag{A27}$$

Auch hier treten wieder Regeln (A26a) und Constraints (A26b und A27) auf. A26a und b ergeben zusammen die Axiome C1 und C2 aus [Baumeister 1996], und bilden die Attributgruppierung auf eine eingeschränkte Attributspezialisierung ab. Die Äquivalenz der Bedingung $\neg\exists a_3 \in \text{ID} : Isa(a_1,a_3) \wedge Isa(a_3,a_2)$ mit der Bedingung $\neg\exists c_2 \in \text{Klasse} : Isa(c,c_2) \wedge Isa(c_2,c_1)$ aus C1 folgt hierbei direkt aus den Axiomen A14 und A15. A27 entspricht C3 und verbietet die direkte Instanziierung von Klassen, die ein Groupleader–Attribut besitzen. Bei der Attributgruppierung handelt es sich also um eine Variante der Attributspezialisierung, welche die auf Seite 5.4.2 kritisierten Eigenschaften nicht mehr besitzt.

Die Definition in [Baumeister 1996] kann sich nicht auf die Attributspezialisierung abstützen, da sie ohne eigenständige Attributobjekte auskommen muß. Dadurch steht sie dem eigentlichen Anwendungsfall der Attributgruppierung, der Abbildung von Metamodellierung auf Sprachen wie C++oder Java allerdings näher.

Diese Kapitel hat formale Definitionen für Aspekte und Attributgruppierung gegeben und so ihre Semantik eindeutig definiert. Auf Basis dieser Definitionen wurden einige Eigenschaften der Aspekte bewiesen, etwa ihre Abbildbarkeit auf Datalog und die Möglichkeit, vor der Laufzeit Typprüfungen vornehmen zu können. Damit sind sowohl Implementierbarkeit als auch frühzeitige Fehlererkennbarkeit gegeben.

Kapitel 7

Bewertung und Zusammenfassung

Die vorhergehenden Kapitel haben ein Verfahren vorgestellt, das weitgehende Modifizierbarkeit von Objekten bei gleichzeitigem Vorhandensein einer Orientierung gebenden Klassenhierarchie erlaubt. Es stellt sich die Frage, inwieweit das vorgestellte Objektmodell und die darauf durchgeführten lokalen Einordnungen modifizierter Objekte den in den Kapiteln 1 und 3 aufgestellten Anforderungen entsprechen oder ob andere Ansätze bessere Lösungmöglichkeiten bieten.

7.1 Bewertung des Ansatzes

Kapitel 1 stellte einige Kriterien vor, mit denen die Anforderungserfüllung des Ansatzes beurteilt werden kann:

Mächtigkeit der Repräsentation: Der Ansatz erlaubt aufgrund seiner objektorientierten, klassenbasierten Grundlage eine Modellierung der meisten bekannten Zusammenhänge. Hierzu zählen etwa Instanziierung, Attributierung und Vererbung. Durch Verwendung der Attributgruppierung können zudem metaklassenähnliche Beziehungen dargestellt werden. Darüber hinaus erlauben Aspekte die Dekomposition von Klassen in sich wechselseitig beeinflussende Teile.

Nicht berücksichtigt wurden allerdings Methoden und Restriktionen. Methoden erscheinen für die Repräsentation der Prozesse weniger wichtig, da sie eher in auf die Datenbank zugreifenden Werkzeugen vorkommen als in der Datenbank selbst. Restriktionen können zwar prinzipiell vom Modellierer verwendet werden, werden vom vorgestellten Wiederverwendungsansatz aber nicht unterstützt. Da sie nur schlecht automatisch aus Instanzen generieren lassen, passen sie nicht mit einem Konzept zusammen, das aufgrund von Instanzmodifikationen passende neue Klassen erzeugt. Neue Restriktionen könnten darum nur durch den Eingriff eines Metamodellierers bereitgestellt werden.

Die für eine Repräsentation der Modelle des Anwendungsgebiets erforderlichen Mechanismen sind also vorhanden.

Effizienz der Repräsentation: Durch die Verwendung einer klassenbasierten Repräsentation auf Grundlage der Multiplen Instanziierung liegt der Speicherbedarf in der normalen Bandbreite klassenbasierter Verfahren. Durch den Einsatz von Aspekten liegt zudem die Klassenzahl am unteren Ende (vgl. Tabelle 4.3). Die Anforderung einer effizienten Repräsentation wird also erfüllt.

Übersichtlichkeit: Der Ansatz erlaubt es, eine Klassenhierarchie zur Orientierung zu verwenden und hat somit inhärent eine bessere Übersichtlichkeit als prototypbasierte Ansätze. Zusätzlich können zu treffende Entwurfsentscheidungen durch Aspekte und metaklassenartige Beziehungen durch Attributgruppierung repräsentiert werden, die beide dazu beitragen, die Struktur

und Bedeutung der repräsentierten Modelle zugänglicher zu machen. Somit stehen mehr Instrumente zur Strukturierung zur Verfügung als in den meisten anderen Ansätzen.

Allerdings können bei stark verfeinerten Klassen zu viele Abhängigkeiten zwischen den Aspekten und Klassen auftreten, so daß zu viele 'synthetische' Aspekte benutzt werden müssen, um diese Abhängigkeiten auszudrücken. Eine Modellierung mit Vereinigungsklassen würde dann eine höhere Übersichtlichkeit erzielen (vgl. Kapitel 5.3.4).

Einen unangenehmen Nebeneffekt hat zudem die Tatsache, daß Aspekte und Aspektkinder vollwertige Klassen sind. Hierdurch liegen alle Aspekte im gleichen Namensraum. Ein Aspekt wie `Direction` tritt jedoch in unterschiedlichen Bedeutungen sowohl bei `Interfaces` als auch bei `Streams` auf. Aufgrund des einheitlichen Namensraums muß er durch entsprechende Maßnahmen unterschiedliche Namen erhalten (im Beispiel von Abbildung 5.7 etwa durch voranstellen von `IF` für Interface). Ähnliches gilt für die Spezialisierung von Aspekten, bei der der spezialisierte Aspekt einen anderen Namen erhalten muß als sein Oberaspekt, auch wenn er relativ zu seinen Halter die gleiche Bedeutung hat (vgl. Abbildung 5.12).

Einordnung: Die Einordnung modifizierter Instanzen in die Klassenhierarchie unter Erzeugung der dafür notwendigen Klassen kann mit einem der in Kapitel 4.2 vorgestellten lokalen Verfahren bei Verwendung der Vereinfachungen aus Abschnitt 5.2.4 durchgeführt werden.

Allerdings fehlt ein Verfahren, um auch entsprechende Aspekte automatisch zu erzeugen. Bei diesen bestehen nämlich über die bekannten Probleme der Einordnungsverfahren hinaus noch das Problem, daß eine unzureichende Überdeckung aller Aspektalternativen durch entsprechende Instanzen zu abweichenden oder sich im Verlauf ändernden Aspekthierarchien führen kann. Auch die Möglichkeit optionaler Aspekte erschwert die Bestimmung der Aspektstrukturen. Diese Schwierigkeiten werden in Abschnitt 7.1.1 noch einmal aufgegriffen und erläutert.

Adaption: Da das Verfahren beliebige Änderungen an den modellierten Objekten zuläßt, kann ein vorhandenes Modell leicht abgewandelt und angepaßt werden.

Suche: Durch Typisierung der Attribute und durch die vorhandene Klassenhierarchie, von denen einige als Metaklassen interpretiert werden können, kann die Suche nach einem Objekt besser spezifiziert und eingeschränkt werden als etwa in prototypbasierten Ansätzen. Die strikten Templates, die dies ermöglichen, können jedoch dazu führen, daß noch nicht vollständig spezifizierte Objekte an unerwarteten Stellen in der Hierarchie eingeordnet werden und somit insbesondere beim Durchstöbern der Datenbank nur schwer gefunden werden. Abschnitt 7.1.2 beschreibt dies genauer.

Das vorgestellte Verfahren erfüllt also die Anforderungen nach Mächtigkeit, Effizienz und Adaption, kann aber bei der Übersichtlichkeit und insbesondere in den Bereichen Einordnung und Suche die gesteckten Vorgaben nicht vollständig erreichen. Hinzu kommen die in Abschnitt 5.3.4 erwähnten Probleme, die in verfahrenstechnischen Modellen auftretenden Aspekte zu identifizieren. Diese Probleme lassen sich allerdings teilweise auf grundlegende Schwächen von VeDa bzw. klassenbasierter Modellierung im allgemeinen zurückführen. So lassen sich etwa in strukturellen Modellen identifizierte Teilmuster nur schlecht auf Aspekte aufteilen, da die spätere korrekte Verkoppelung der Einzelteile nicht strukturell definiert werden kann. Abschnitt 7.1.3 beschäftigt sich mit diesen objektmodellbasierten Schwächen.

7.1.1 Aspektgenerierung

Bei der automatischen Generierung von Klassen und Aspekten aus Instanzen muß normalerweise davon ausgegangen werden, daß die vorhandenen Instanzen unvollständig sind, also nicht alle Möglichkeiten späterer Modellierungen überdecken. Bei der Klassengenerierung kann dies zum Fehlen von Klassen oder zu überspezialisierten Klassen führen. Bei der automatischen Generierung von

Aspekten werden die Instanzen zusätzlich genutzt, um die Instanziierungsabhängigkeiten der Aspektkinder zu bestimmen. Dadurch können fehlende Instanzen nicht nur den Inhalt einzelner Klassen sondern den Aufbau der gesamten Hierarchie beeinflussen.

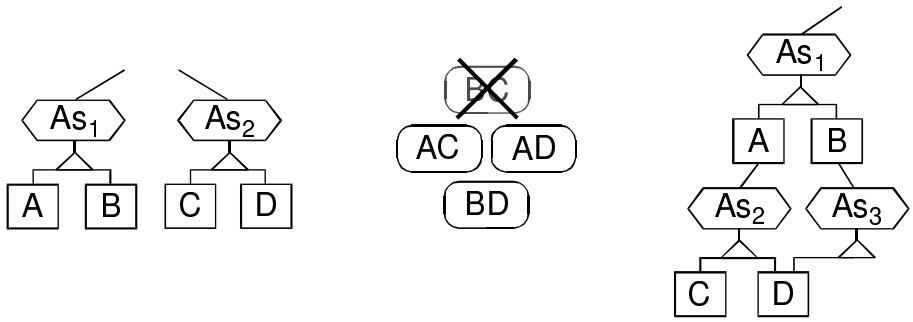


Abbildung 7.1: Unterschiedlich generierte Aspekthierarchien durch fehlende Instanzen

In Abbildung 7.1 können mit Hilfe der links dargestellten Aspekthierarchie die in der Mitte schematisch dargestellten Arten von Instanzen erzeugt werden, die jeweils Attribute aus jeweils zwei Aspektkindern enthalten. Bei der Generierung von Aspekten aus Instanzen seien aber beispielsweise keine Instanzen des 'Typs' `BC` vorhanden. Verwendet man nun eine einfache aussagenlogische Heuristik zur Erzeugung der Aspekte, so entsprechen diese Instanzen dem Ausdruck $(A \wedge C) \otimes (A \wedge D) \otimes (B \wedge D)$ mit $\otimes$ als Exklusivem Oder. Bringt man $\otimes$ soweit wie möglich nach innen, so erhält man $(A \wedge (C \otimes D)) \otimes (B \wedge D)$. "$\otimes$" verbindet hierbei gemeinsame Kinder eines Aspekts und "$\wedge$" verschiedene Aspekte Dadurch erhält man die rechts in Abbildung 7.1 dargestellte Hierarchie, die sich deutlich von der linken unterscheidet. Aus anderen Heuristiken können unterschiedliche Hierarchien resultieren, die sich aber immer mehr oder weniger deutlich von der linken unterscheiden werden.

Bei optionalen Aspekten entstehen zusätzliche Schwierigkeiten. Zum einen entspricht ein optionaler Aspekt As_1 der Formel $\ldots As_1 \vee \neg As_1 \ldots$ und kann somit durch obige Heuristik nicht erzeugt werden. Entscheidender ist aber, daß Modellierer in einer realen Modellierungssituation vermutlich Hilfsobjekte in der Datenbank speichern. Diese enthalten möglicherweise notwendige Aspekte nicht, um aus ihnen einfach neue Objekte erzeugen zu können, die sich in den ausgelassenen Aspekten unterscheiden. Zwingt man die Modellierer dann aber nicht, für jedes dauerhaft gespeicherte Objekt anzugeben, ob es zur Aspektgenerierung genutzt werden darf oder nicht, werden zu viele Aspekte als optional erkannt.

Insgesamt muß also davon ausgegangen werden, daß eine automatisch generierte Aspekthierarchie erhebliche Unterschiede zu einer Hierarchie aufweist, die von Hand erstellt wurde und die der Forderung nach Übersichtlichkeit genügt. Die Probleme mit der sich unter Umständen stark ändernden Hierarchie sind allerdings nicht spezifisch für Aspekte sondern ergäben sich bei allen Ansätzen, die zusätzliche Abhängigkeiten zwischen verschiedenen Teilen der Klassenhierarchie modellieren (wie etwa die Partitionierungen und Kardinalitätsrestriktionen der Rollen aus [Wieringa et al. 1995]), wenn diese Hierarchie aus Objekten generiert würde.

7.1.2 Sucheinschränkungen

Einer der Vorteile der Verwendung einer Klassenhierarchie liegt darin, Suchoperationen über Objekten auf Suchoperationen über Klassen abbilden und dort einschränken zu können (vgl. Abschnitt 5.1.1). Dies beschleunigt eine etwaige automatische Suche oder kann Ausgangspunkte für das Durchstöbern der Datenbank liefern.

Die möglichen Einschränkungen in nicht speziell organisierten Datenbanken sind allerdings weniger effektiv als man erwarten würde. Grundsätzlich muß in einer solchen Datenbank davon ausgegangen werden, daß nicht sämtliche Kombinationen von Attributen und Attributtypen als eigene Klassen/Aspekte vorliegen, es also beispielsweise eine Klassenhierarchie wie in Abbildung 7.2 a) geben

kann. Sucht man dort nach Objekten mit Attribut `b: B'`, so kann man sich keineswegs auf die Instanzen der Klasse `A'` und ihrer Unterklassen beschränken. Ein Objekt mit entsprechend spezialisiertem Attribut `b` aber ohne ein Attribut `d` erfüllt nämlich die Suchanfrage, wird aber dennoch `A` zugeordnet sein.

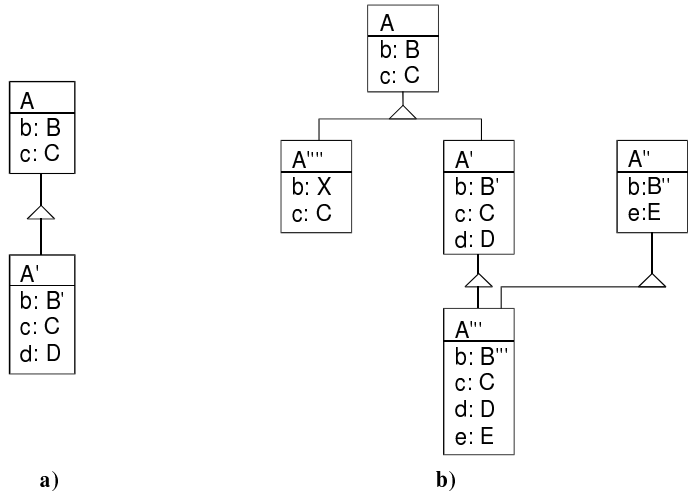


Abbildung 7.2: Beispiele zur klassenbasierten Suche nach Objekten

Abbildung 7.2 b) zeigt zudem in einem etwas umfangreicheren Beispiel, daß eine Klasse, deren Instanzen für die oben angegebene Suche relevant sind, keineswegs in direkter Verbindung zu `A'` stehen muß. Beispielsweise kann `A''` Objekte mit Attribut `b` vom Typ `B'` enthalten, da `B'''` offensichtlich Unterklasse von `B'` und von `B''` ist und somit ein Objekt mit einem Attribut `b` vom Typ `B'''` zum einen die Suchanfrage erfüllt und zum anderen als Instanz `A''` eingeordnet werden muß, falls es keine Attribute `c` und `d` besitzt. Hierdurch ergibt sich der in Kapitel 5.1.1 angegebene Ausdruck $q \leq s \wedge q \leq t$. Es muß also eine gemeinsame Unterklasse q (= `B'''` im Beispiel) des gesuchten Attributtyps t (= `B'`) und des gefundenen s (= `B''`) geben, damit die Instanzen der Klasse als Suchergebnisse in Frage kommen.

Zwar müssen auch weiterhin keine Klassen wie `A''''` durchsucht werden, die das gesuchte Attribut nicht oder nur mit inkompatiblem Typ besitzen, doch ist die Einschränkung der zu testenden Instanzen und die Möglichkeiten zur Beschneidung des Klassenbaums während der Suche geringer als ursprünglich erwartet. Der vom System zu erbringende Aufwand für die Suche erhöht sich also.

Während sich diese Aufwandserhöhung durch entsprechende Hardware kompensieren läßt, ergibt sich für einen Modellierer zum einen das Problem, daß er beim Durchstöbern der Datenbank wegen obiger Suchproblematik möglicherweise eine Vielzahl von Ausgangspunkten (vgl. Kapitel 3.3.1) vom System angeboten bekommt. Zum anderen findet er Objekte nicht dort vor, wo er sie erwarten würde: Da die Einordnung von Objekten den strikten Templates der Klassen Rechnung tragen muß, wird primär nach dem Vorhandensein von Attributen eingeordnet und erst sekundär nach der Spezialisierung der Attribute. Bei einem Modell wie VeDa, in dem einige Klassen viele mengenwertige Attribute verwenden, kann somit ein Objekt quasi bis zur kompletten Fertigstellung einer weit oben in der Hierarchie befindlichen Klasse wie `A` zugeordnet sein, sofern ein in den Unterklasse notwendiges Attribut noch nicht definiert wurde. Zum einen lassen sich dadurch Objekte schlecht durch Stöbern finden, zum anderen hat die Anzeige der derzeitigen Klasse in einem solchen Fall nur einen geringen Wert für den Modellierer.

7.1.3 Schwächen der klassenbasierten Modellierung

Die Modellierung verfahrenstechnischer Modelle mit einem Schwergewicht auf der Repräsentation des Wissens in Klassen zeigte bereits im Rahmen von VeDa einige Probleme. Zwar beheben die in

dieser Arbeit vorgestellten Ansätze einige davon, andere werden jedoch verstärkt.

Wie viele andere Objektmodelle überlädt auch das Sprachmodell von VeDa die Subklassenbeziehung. Ursprünglich wurde entlang der Subklassenbeziehung spezialisiert, vererbt, Attribute dekomponiert und in sehr frühen Versionen auch einzelne Attribute 'instanziiert'. Zum einen führte dies zu mangelnder Übersichtlichkeit der Typdefinitionen der Attribute (vgl. S. 29). Entscheidender war jedoch der Effekt, daß dadurch Instanzen durch Angabe einer um ein Attribut erweiterten Unterklasse wieder zur Klasse werden konnten[1]. Dies behinderte jedoch den Einsatz formaler Methoden und eine effiziente Speicherrepräsentation.

Im Rahmen dieser Arbeit konnten einige der parallelen Beziehungen von der Subklassenbeziehung gelöst werden. Der fließende Übergang von Klassen zu Instanzen wurde im Rahmen der Aufteilung auf die vier Ebenen entfernt und die Attributdekomposition wurde mittels der in Kapitel 5.4 vorgestellten Attributgruppierung formalisiert und mit einer neuen Relation repräsentiert. Die verbleibenden beiden Relationen Spezialisierung und Vererbung finden sich auch in den meisten anderen objektorientierten Sprachen gemeinsam wieder und ihre Aufspaltung bringt Nachteile mit sich (vgl. [Abadi und Cardelli 1996]).

Werden die Relationen, wie im vorliegenden Ansatz, nicht getrennt, so ergibt sich ein Problem bei der Modellierung struktureller Beziehungen. Bei der Repräsentation struktureller verfahrenstechnischer Modelle durch Klassen müssen nämlich nicht nur die Komponenten eines komplexen Modells in der Klasse dargestellt werden, sondern auch die Verknüpfungen der Komponenten. In VeDa werden diese Verknüpfungen letztendlich durch `Couplings` repräsentiert. Werden diese Couplings nicht in den Klassen (also den Modellbausteinen) repräsentiert, so definieren diese Klassen lediglich eine Ansammlung von Komponenten und Verknüpfungen, deren Verschaltung jedoch undefiniert ist. Speichert man diese Couplings aber als Attribute in einer einen Modellbaustein repräsentierenden Klasse, so kann die Bausteinstruktur anschließend kaum noch in Unterklassen erweitert werden. Da sich die Couplings aufgrund der Signaturkompatibilität der Subklassenbeziehung nicht mehr entfernen oder umbiegen lassen, kann ein neuer Teilbaustein nur an den Rändern bzw. an noch freien Interfaces in die zusammengesetzte Struktur eingefügt werden. Diese Schwierigkeiten treten prinzipiell auch in anderen Anwendungsfällen auf, etwa beim Flugzeugbeispiel in [Nguyen et al. 1992, S.243], wo sie allerdings wegdefiniert werden[2]. Durch die Verwendung von Aspekten für strukturelle Modelle wird dieses Problem verschärft, da nun bereits für die Bildung einer Instanz verschiedene Strukturrepräsentationen, die sich auch überlappen können, kombiniert werden müssen, also Couplings möglicherweise geändert werden müssen.

Durch die Trennung von Spezialisierungs- und Vererbungsrelation, also der Verwaltung der Klassenextensionen unabhängig von der Subklassenbeziehung, könnte dies teilweise behoben werden: Da für die Subklassenbeziehung keine Kompatibilitätsrestriktionen gelten, können geerbte Couplings beliebig überschrieben werden. Die Bedingungen für die Extensionen hingegen könnten Couplings bei der Definition der Spezialisierung ignorieren. Dabei droht jedoch die Gefahr, daß eine Subsumtion aufgrund von Objekt- und Klassenstrukturen nicht mehr betrieben werden kann (vgl. [Abadi und Cardelli 1996]). Zudem bestünde für Aspekte weiterhin das Problem, die Verkoppelung der in ihnen definierten Teilstrukturen automatisch zu definieren.

7.1.4 Alternativen

Wenn der vorgestellte Ansatz nicht alle Anforderungen erfüllt und die klassenbasierte Repräsentation weniger Vorteile als erwartet bietet, dafür aber zusätzliche Schwierigkeiten, stellt sich die Frage nach alternativen Darstellungsmöglichkeiten. An und für sich wurde diese Frage im Rahmen von

[1] Bzw. in einer alternativen Sichtweise Klassen Instanzen als Oberklassen haben konnten.

[2] „Note that these actuators are usally hydraulic or electric systems that are not described in either of the representations. There is therefore no explicit requirement for both representations to be related." [Nguyen et al. 1992, S.243]

Kapitel 4.1 beantwortet, da alle dort aufgeführten Objektmodelle die Anforderungen nicht erfüllen. Interessant sind aber vielleicht dennoch zwei dort vorgestellte Modelle, die die Wiederverwendung, wenn auch unter abgewandelten Anforderungen, unterstützen könnten.

Als erstes betrachte man ein **prototypbasiertes**, ungetyptes **Objektmodell**, das Änderungen nur an den Blättern des Delegationsbaumes zuläßt. Die oben aufgeführten Einordnungsprobleme können hierbei vollständig vermieden werden, da keine Einordnung mehr notwendig ist. Verstoßen Änderungen an einem Objekt gegen den Delegationspartner des Objekts, so können durch Delegation gewonnene Informationen kopiert und die Delegationsrelation anschließend gelöst werden. Ähnlich wie bei der oben vorgeschlagenen Trennung von Vererbung und Spezialisierung würde die Delegationsrelation also nur noch Vererbungsfunktionalität tragen aber keine Aussagen über Verwendbarkeit des Objektes oder Zugehörigkeit zu einer Gruppe mehr treffen. Weitere gewünschte allgemeine Beziehungen zwischen den Objekten könnten dann durch zusätzliche Relationen modelliert werden, bei deren Definition keine Rücksicht auf parallele Delegations- oder Subklassenbeziehungen genommen werden müßte.

So könnte zwar eines der entscheidenden Probleme des emulierenden Ansatzes gelöst werden, doch würde dies durch massive Einschränkungen in den Anforderungsbereichen Übersichtlichkeit und Suche erkauft werden:

- Die Suche wird durch die nicht mehr vorhandene Hierarchie sowie durch die Tatsache erschwert, daß völlig freie Änderungen auch zu völlig unterschiedlichen Benennungen von Attributen und Objekten führen können. Passende Suchverfahren würden also eher einer thesaurus–unterstützten Suche in schwach strukturierten Daten ähneln, als einer Datenbankanfrage. Entsprechende Verfahren mit teilweise intelligenten Strategien finden sich etwa bei der Suche in ähnlich unstrukturierten Daten im World Wide Web.

- Eine Übersicht fördernde, abstrakte Hierarchie ist bei einem solchen Ansatz selbstverständlich nicht mehr sichtbar. Wird sie dennoch benötigt, muß sie nachträglich erzeugt werden, etwa mittels TWR-Systemen oder Clustering–Verfahren. Der durch den Verzicht auf die Erhaltung der Klassenhierarchie eingesparte Aufwand muß also bei der Wiedergewinnung der dadurch verlorenen Information wieder eingesetzt werden.

Eine andere Alternative stellt der **Verzicht auf die Unterstützung ungeplanter Wiederverwendung** und die Konzentration auf den Einsatz speziell aufbereiteter Komponenten dar. Im Bereich des Software–Engineerings sehen einige Autoren (siehe etwa [Sommerville 1995, S.400ff] und [Castano et al. 1994]) einen "reuse–myth", nach dem wiederverwendbare Komponenten im Rahmen normaler Entwicklungsarbeit entstünden, während sie in Wirklichkeit nur durch zielgerichtete Entwicklung erhältlich sind. Folgt man dieser Argumentation so würden wiederverwendbare Komponenten durch interne oder externe Teams entwickelt, wobei sie ggf. die in der Datenbank vorhandenen Modelle auf Standardisierungsmöglichkeiten untersuchen. In der Datenbank bereitgestellt können sie dann durch die Modellierer verwendet und ggf. modifiziert werden. Modifizierte Komponenten brauchten durch die Klassenhierarchie nicht mehr erfaßt zu werden, da ihre Wiederverwendung durch einen Modellierer nicht unterstützt werden soll. Suchoperationen über den bereitgestellten wiederverwendbaren Komponenten müßten hingegen eine hohe Treffsicherheit haben, um die Benutzerakzeptanz zu sichern. Aufgrund der vergleichsweise niedrigen Zahl der wiederverwendbaren Bausteine, müßten vermutlich die meisten vor Wiederverwendung modifiziert werden und es bestünde bei einem hohem Benutzeraufwand für die Suche die Gefahr, daß sich Wiederverwendung für den Modellierer nicht lohnt.

Ähnlich wie im Fall der *un*geplanten Wiederverwendung muß ein Objektmodell bei diesem Ansatz sowohl Modifizierbarkeit als auch Ordnung und Suche auf den Daten unterstützen. Da sich diese konträren Forderungen nun aber auf verschiedene Teile der Datenbank beziehen[3], bieten sich Verfahren

[3] Nämlich Ordnung und Suche auf die für die Wiederverwendung erstellten Bausteine und Modifizierbarkeit auf die vom Modellierer in sein Modell eingebauten Bausteine.

wie HYBRID oder OBJECT SPECIALIZATION an (siehe Abschnitt 4.1.3), die für einzelne Objekte den Übergang von der klassenbasierten Welt in die prototypbasierte erlauben. Der erreichbare Grad der Wiederverwendung liegt allerdings niedriger als der bei ungeplanter Wiederverwendung im Prinzip mögliche.

Alternative Ansätze bieten also Verbesserungen bei einigen der Anforderungen in denen der vorgestellte Ansatz schlecht abschneidet. Dafür zeigen sie Schwächen bei anderen Anforderungen, die durch entsprechenden Aufwand ausgeglichen werden müssen, oder verfolgen eine schwächere Art der Wiederverwendung. Möglicherweise können nicht alle der in Kapitel 1 genannten Anforderungen zugleich vollständig erfüllt werden.

7.2 Ergebnisse der Arbeit

Im Ergebnis hat diese Arbeit gezeigt, daß der vorgestellte Ansatz zur wiederverwendungsfreundlichen Repräsentation verfahrenstechnischer Prozeßmodelle nicht alle Anforderungen an die Problemstellung erfüllen kann. Dabei sind Zusammenhänge zwischen verschiedenen Repräsentationsformen herausgearbeitet worden, die nicht zuletzt auch die hier vorgeschlagenen Ergänzungen des klassenbasierten objektorientierten Modells motiviert haben. Mit Hilfe dieser Ergänzungen ist das Spektrum der repräsentierbaren Beziehungen deutlich erweitert worden:

- Die Aufspaltung der Bestandteile VEDAS auf vier Ebenen und die Definition von VDDL unter Bezugnahme auf Elemente der klassischen Objektorientierung (siehe Kapitel 2) ermöglicht die Diskussion über ihre Stärken und Schwächen sowie ihre Erweiterung um wiederverwendungsunterstützende Elemente. Insbesondere konnten auch die verbleibenden Abweichungen von klassischen Objektmodellen als notwendig und vorteilhaft begründet werden. Die Aufteilung auf vier Ebenen erlaubt zudem die Zuordnung der auf den einzelnen Ebenen vorhandenen Konzepte zu verschiedenen Rollen des Modellierers.

- Das in Kapitel 3 vorgestellte Beispiel zeigt ein sehr weitgehendes Szenario für ungeplante Wiederverwendung in der Verfahrenstechnik sowie die hierfür notwendigen Operationen auf der Modelldatenbank. Zwar lassen die Einschränkungen beim vorgeschlagenen Ansatz sowie den Alternativen die Realisierbarkeit eines solchen Szenarios vorläufig in zweifelhaftem Licht erscheinen, doch repräsentiert der Ablauf dasjenige Ende des Spektrums verschiedener Wiederverwendungsarten, das maximaler Wiederverwendung bietet. Im Vergleich mit anderen, einfacher zu realisierenden Wiederverwendungsarten lassen sich somit Aufwands- und Ertragsschätzungen durchführen.

- Anhand dieses Beispiels konnte in Kapitel 4 gezeigt werden, daß vorhandene Ansätze zur objektorientierten Repräsentation nicht ausreichen die notwendigen Operationen zu realisieren, und daß verschiedene Verfahren zur Klassifizierung und Reorganisation nur eingeschränkte Fälle behandeln. Weitere Forschungen in dieser Richtung könnten die Realisierbarkeit des vorgestellten Ansatzes verbessern. Die Untersuchung spezieller ingenieurtechnischer Objektmodelle zeigte zudem, daß die ungeplante Wiederverwendung von diesen nicht oder kaum unterstützt wird.

- Das vorgestellte Verfahren zur Repräsentation metaklassenähnlicher Konstrukte in einer zweistufigen Instanzen–Klassen–Hierarchie erlaubt die einfache Repräsentation und Implementierung ordnender und gruppierender Zusammenhänge zwischen Klassen und ihren Metaklassen, ohne daß eine strikte Trennung zwischen ihnen vorgenommen werden muß und ohne das Metaklassen selbst repräsentierbar sein müssen. Das Verfahren findet bereits Verwendung bei der Spezifikation verfahrenstechnischen Wissens in VEDA(z.B. [Souza und Marquardt 1998], [Krobb et al. 1998]).

- Durch den Einsatz von "Aspekten", dem vorgestellte Organisationskonzept für Klassenhierarchien mit hoher Instanzenvariabilität, kann die zur Repräsentation der Instanzen notwendige Klassenzahl verringert werden. Im Gegensatz zu verwandten Verfahren werden darüberhinausgehend auch Namenskonflikte sowie Instanziierungsabhängigkeiten zwischen verschiedenen Aspekten behandelt. Erst durch diese Modellierungsmöglichkeiten entspricht das Konzept den Anforderungen verfahrenstechnischer Modelle.

- Die in Abschnitt 5.1 bewiesene Unvereinbarkeit von modifizierungsfreundlicher prototypbasierter Objektorientierung einerseits und such- und übersichtlichkeitsfördernder klassenbasierter Objektorientierung andererseits zeigt, daß es keine einfache Lösung des spezifizierten Problems gibt, etwa durch ein Objektmodell, welches gleichzeitig beiden Paradigmen folgt. Der Beweis liefert auch die Möglichkeiten, diese Einschränkung zu umgehen, etwa durch Objektmigration, wie im vorgeschlagenen Ansatz, oder durch den Verzicht auf gleichzeitige Klassenbasiertheit und Prototypbasiertheit wie bei den Ansätzen aus Abschnitt 4.1.3.

7.3 Ausblick

Diese Arbeit konnte selbstverständlich nicht alle Bereiche der Wiederverwendung verfahrenstechnischer Modelle behandeln. So wurde in Kapitel 4.2.2 darauf hingewiesen, daß anscheinend keine Verfahren existieren, mit den Klassenhierarchien reorganisiert werden können, die zusätzlich zu Unterklassenbeziehungen noch weitere Sprachelemente enthalten (etwa Rollen, Mixins oder eben Aspekte). Zwar kann nicht erwartet werden, daß diese Sprachelemente automatisch an semantisch sinnvoller Stelle erzeugt werden. Aber bereits die Entwicklung eines Klassenreorganisationsverfahrens, das diese Elemente beachtet und erhält, könnte die für den vorgestellten Ansatz notwendige Fähigkeit zur globalen Reorganisation verbessern.

Zudem stellt sich die Frage, welche weiteren alternativen Ansätze zur Wiederverwendung zu den in Abschnitt 7.1.4 aufgeführten sich ergeben, wenn man die Ausgangsfragestellung ausweitet. So stützte sich die Behandlung der Wiederverwendbarkeit in der vorliegenden Arbeit auf eine stark produktzentrierte Sicht und berücksichtigt mögliche Synergien mit der Gewinnung und Auswertung prozeßzentrierter Daten[4] nicht (vgl. S. 6 und [Dömges 1999]). Dabei könnte sich durch letztere eine alternative Lösung für die Suche nach passenden wiederverwendbaren Bausteinen ergeben: Anstelle oder zusätzlich zum Ergebnis einer expliziten Suchanfrage könnte das System dem Modellierer Bausteine aus Modellierungsprozessen anbieten, die ähnlich wie der derzeitige verlaufen sind. Durch die Länge der verglichenen Abläufe könnte die Selektivität dieses Vorgangs beeinflußt werden. Auch die bereits in Kapitel 7.1.4 angedeutete Verwendbarkeit der sich im Zusammenhang mit dem World Wide Web entwickelnden Suchverfahren für unstrukturierte Daten deutet darauf hin, daß die in Abschnitt 3.4 vorgenommene Beschränkung auf Standardsuchverfahren möglicherweise voreilig war.

Zweifelsohne interessant wäre auch eine empirische Untersuchung des tatsächlichen Wiederverwendungspotentials in der Verfahrenstechnik. Die von [Foss et al. 1997] durchgeführte Studie zeigt etwa, daß bei 12 untersuchten Modellierungsproblemen nur in zwei Fällen ein der ungeplanten Wiederverwendung entsprechendes Vorgehen gewählt wurde. Vergliche man diese Quote mit der theoretisch möglichen Wiederverwendung, indem man etwa die über mehrere Jahre angesammelten Modelle einer einzelnen Abteilung auf Ähnlichkeiten untersucht, so könnte zum einen der durch entsprechende Wiederverwendungskonzepte mögliche Gewinn genauer beziffert werden. Zum anderen wären Erkenntnisse über die Art der möglichen Wiederverwendung zu erwarten, so daß sich etwa die Eignung der auf den Seiten 133 f angedeuteten eingeschränkteren Wiederverwendungsverfahren fundierter beurteilen ließe.

[4] Also den durch Aufzeichnung des Modellierungsprozesses gewonnenen 'Traces'.

Anhang A

VDDLs Grammatik

A.1 Komplexe Typen

<type-def> → <compl-DT> | <primitve-DT> | <typename>

<compl-DT> → <record-DT> | <array-DT> | <set-DT> | <list-DT>

<record-DT> → **<** <record-list> **>**[1]

<record-list> → <record-ele> { **x** <record-ele>}*

<record-ele> → <rolename><type-def>

<set-DT> → <number-constr>**{**<type-def>**}** |
#{ <enum-val> { **,** <enum-val> }$^{+}$**}**

<number-constr>→ ϵ | #<int-val> | #>=<int-val> |
#<=<int-val> | #<int-val>**..**<int-val>

<list-DT> → **** <type-def> ****

<array-DT> → <type-def>[<dimension-list>]

<dimension-list>→ <dimension>**;**<dimension-list> | <dimension>

<dimension> → ϵ | <integer-expr>**..**<integer-expr>

A.2 Ausdrücke und Pfade

<expression>→ <arithm-expr> | <bool-expr>

<integer-expr>→ <arithm-expr>

<arithm-expr>→ (<arithm-expr>)|
<arithm-expr> <aribinop> <arithm-expr> |
<ariunop> <arithm-expr> |
<arithm-expr> <systembinop> <arithm-expr>|
<systemfunctionname>**(** <aparameterlist> **)** |
<path>**(** <aparameterlist> **)** |
<path> |

[1] Beachten Sie bitte den Unterschied zwischen **{** and {. Das erste Symbol bezeichnet ein *Terminal*, welches in einem Ausdruck der Sprach VDDL auftauchen kann. Das zweite Symbol ist Teil der EBNF–Notation und faßt mehrere Symbole zu einer Gruppe zusammen.

<compl-type-constant> |
<int-val> | <real-val> | <word-val>| <string-val> | **Nil**

<systembinop>→ **:intersect** | **:unite**

<systemfunctionname> → <systemname>

<aparameterlist>→ <expression> { **,** <expression> }*

<compl-type-constant>→ <record-constant> | <array-constant> |
<set-constant> | <list-constant>

<record-constant>→ **<** <record-list-const> **>**

<record-list-const> → <record-ele-const> { **x** <record-ele-const> }*

<record-ele-const> → <rolename><expression>

<array-constant>→ [<expression> { **,** <expression> }*]

<set-constant>→ **{** { <expression> { **,** <expression> }*} **}**

<list-constant>→ \ {<expression> { **,** <expression> }* } \

<bool-expr>→ **(** <bool-expr> **)** |
<bool-expr> <boolbinop> <bool-expr> |
<boolunop> <bool-expr> |
<arithm-expr> <relop> <arithm-expr> |
<OO-assertions> |
(<isunification> **)** |
<bool-val>

<boolbinop>→ **:and** | **:or** | **:xor** | **:then** | **:iff**

<boolunop>→ **:not**

<relop>→ **:eq** | **:ge** | **:gt** | **:le** | **:lt** | **:neq :ele_of :subset**

<OO-assertions>→ <objectref> (**:id** | **:nid**) <objectref> |
<objectref> **:in** <classname> |
:is_defined <path>

<objectref>→ <path>

<path> → <frame-specifier><path-tail>

<frame-specifier> → **:this** | <instancename> | <classname> | <varname>

<path-tail>→ ϵ |
<path-part><path-tail> |
<path-part>[<index-expr-list>]<path-tail>

<path-part>→ **.**<slotname> | **@**<rolename>

<index-expr-list>→ <integer-expr>{ **;** <integer-expr>}*

A.3 Frames

<veda>→ <vedaloop>

<vedaloop>→ { <class> | <aspect> | <type> | <instance> } +

<class>→ <classhead> <classbody> **END**

<classhead>→ **class:** <classname>
[**documentation:** <string-val>]
[**edited_by:** {< <string-val> **x** <date> > }$^{+}$]
metaclasses: <classnamelist>
[**superclasses:** <classnamelist>]

<classnamelist> → <classname> { **,** <classname> }*

<classbody> → [<attributes>] [<laws>] [<methods>]
[<pp-conditions>] [<sched-conditions>]

<aspect>→ <aspecthead> <classbody> **END**

<aspecthead>→ **aspect:** <classname>
[**documentation:** <string-val>]
[**edited_by:** {< <string-val> **x** <date> > }$^{+}$]
metaclasses: <classnamelist>
[**superaspect:** <classname>]
{ **aspect-of:** <classnamelist> <req-fac> }$^{+}$

<type>→ **type:** <typename>
[**documentation:** <string-val>]
typedef: <type-def>
END

<instance>→ <instancehead> <instancebody> **END**

<instancehead>→ **instance:** <instname>
[**documentation:** <string-val>]
[**edited_by:** {< <string-val> **x** <date-val> > }$^{+}$]
classes: <classnamelist>

<instancebody>→ { <attribute-value> }*

<attribute-value>→ <slotname>**:** <value>

A.4 Attribute und Facetten

<attributes> → { <si-attributes> | <sr-attributes> |
<ii-attributes> | <ir-attributes> }$^{+}$

<si-attributes> → **shared_intrinsic_attributes:** { <slotname>**:** <si-body> }$^{+}$

<sr-attributes> → **shared_relational_attributes:** { <slotname>**:** <sr-body> }$^{+}$

<ii-attributes> → **individual_intrinsic_attributes:** { <slotname>**:** <ii-body> }$^{+}$

<ir-attributes> → **individual_relational_attributes:** { <slotname>**:**
<ir-body> }$^{+}$

<si-body>→ <dom-fac> { <val-fac> | <basic-facettes> }*

<sr-body>→ <dom-fac> { <val-fac> | <basic-facettes> | <rel-facettes> }*

<ii-body> → <dom-fac> { <init-fac> | <basic-facettes> | <req-fac> }*

<ir-body>→ <dom-fac> { <init-fac> | <basic-facettes> | <rel-facettes> | <req-fac> }*

<basic-facettes>→ <readonly-fac> | <groupleader-fac> |<togroup-fac> | <doc-fac> | <trigger-facettes>

<rel-facettes> → <comp-fac> | <exc-fac> | <dep-fac> | <inv-fac>

<trigger-facettes> → <trigger-kind> <path>

<trigger-kind> → **:do_before_read** | **:do_instead_read** | **:do_after_read** | **:do_before_write** | **:do_instead_write** | **:do_after_write**

<doc-fac>→ **:doc** <string-val>

<comp-fac>→ **:comp** <bool-val>

<dep-fac>→ **:dep** <bool-val>

<exc-fac>→ **:exc** <bool-val>

<groupleader-fac>→ **:groupleader** <bool-val>

<readonly-fac>→ **:readonly** <bool-val>

<refinable-fac>→ **:refinable** <bool-val>

<req-fac>→ **:req** <bool-val>

<init-fac>→ **:init** (<path> | path **(** <aparameterlist> **)** | <value>)

<inv-fac>→ **:inv** <slotname>

<togroup-fac>→ **:to_group** <slotname>

<dom-fac>→ **:dom** <type-def>

<val-fac>→ **:val** <value>

A.5 Laws und ihre Notation

<laws> → **laws:** { <slotname>**:** <lawbody> }$^+$

<lawbody> → <PLI-formula> { <refinable-fac> | <doc-fac> }*

/each facet may appear only once, but their order is unimportant/

<PL1-formula>→ [<quantification> [<isunificationlist>][<whereclause>]] **:holds** <bool-expr>

<quantification>→ { <quantifier> <varname> **:in** <classname> }$^+$

<isunificationlist>→ isunification { ’’ isunification }*<isunification>→ <varname> **:is** <expression>

<whereclause>**:where** <bool-expr>

<quantifier>→ **:all** | **:some**

<pp-conditions> → **conditions:** { <slotname>**:** <conditionbody> }$^+$

<sched-conditions> → **scheduling_conditions:** { <slotname>**:** <s-condbody> }$^+$

<conditionbody> → (<string-val> | <groupleader-fac>) { <togroup-fac>
<refinable-fac> <doc-fac> }*

<s-condbody> → <string-val> [<refinable-fac>] [<doc-fac>]

A.6 Methoden

<methods> → **methods:** { <slotname>**:** <methodbody> }$^+$

<methodbody> → <interface-fac> [(<intern-fac> | <extern-fac>)] [<doc-fac>]

<interface-fac>→ **:interface** [<fparameterlist>] **->** (<typename> | **VOID**)

<fparameterlist>→ <varname> <typename> { **x** <varname> <typename> }*

<extern-fac> → **:extern <** <string-val> **x** <string-val> **>**

<intern-fac> → **:intern** [<quantification>] **:results**
(<expression> | <set_slot>)

<set_slot> → **:set_slot(** <path> **,** <expression> **)**

Literaturverzeichnis

Abadi, M. und L. Cardelli (1996). *A Theory of Objects*. Monographs in Computer Science. Springer, New York, Berlin, Heidelberg. 16, 25, 26, 27, 76, 133

Adler, Richard M. (1995). *Emerging standards for component software*. IEEE Computer, S. 68–77. 8

Ahmed, Shamim, A. Wong, D. Sriram und R. Logcher (1992). *Object–oriented database management systems for engineering: A comparison*. Journal of Object–Oriented Programming, 5(3):27–43. 1, 11

Albano, A., G. Ghelli und R. Orsini (1995). *Fibonacci: A programming language for object databases*. VLDB Journal, 4(3):403–444. 57, 61

Altmeyer, J., S. Ohnsorge und B. Schurmann (1994). *Reuse of design objects in CAD frameworks*. In: *1994 IEEE/ACM Intl. Conf. on Computer–Aided Design*, S. 754–761, San Jose, CA. ACM. 1, 10

Andersson, Mats (1994). *Object-Oriented Modeling and Simulation of Hybrid Systems*. Doktorarbeit, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden. 75, 76

Atkinson, M., F. Bancilhon, D. DeWitt, D. Maier, K. Dittrich und S. Zdonik (1989). *The object-oriented database system manifesto*. In: Kim, W., J.-M. Nicolas und S. Nishio, Hrsg.: *Proc. of the Int. Conf. on Deductive and Object-Oriented Databases*, S. 40–57, Kyoto. Elsevier Science Publishers. Also appeared in [Bancilhon et al. 1992]. 11, 17, 24, 25

Bancilhon, F., C. Delobel und P. Kanellakis, Hrsg. (1992). *Building an Object–Oriented Database System — The Story of O_2*. Morgan Kaufmann, San Mateo, California. 27, 143

Bardou, Daniel und C. Dony (1996). *Split objects: A disciplined use of delegation within objects*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Bd. 31, 10 d. Reihe *ACM SIGPLAN Notices*, S. 122–137, New York. ACM Press. 47, 54, 58, 59, 63, 81

Barghouti, Naser S. und G. E. Kaiser (1991). *Concurrency control in advanced database applications*. ACM Computing Surveys, 23(3):269–317. 79

Barker, H.A., M. Chen, P. Grant, C. Jobling und P. Townsend (1993). *Open architecture for computer–aided control engineering*. IEEE Control Systems, S. 17–27. 5

Baumeister, M. und M. Jarke (1999). *Compaction of large class hierarchies in databases for chemical engineering*. In: Buchmann, Alejandro P., Hrsg.: *Datenbanksysteme in Büro, Technik und Wissenschaft (BTW)*, Freiburg. Springer. 100, 104, 108

Baumeister, M. und W. Marquardt (1998). *The chemical engineering data model VeDa; Part 1: VDDL — The language definition*. Technischer Bericht LPT-1998-01, RWTH Aachen, Lehrstuhl für Prozeßtechnik. 15, 16, 18, 30

Baumeister, Markus (1996). *Attribute grouping: Emulating metamodels without instantiation*. In: Patel, D., Y. Sun und S. Patel, Hrsg.: *International Conference on Object Oriented Information Systems (OOIS)*, S. 169–179, London. Springer. 15, 21, 114, 116, 127

Beeri, C. (1989). *Formal models for object oriented databases (Invited paper)*. In: *Proc. of the Int. Conf. on Deductive and Object-Oriented Databases*, S. 405–430, Kyoto, Japan. Elsevier Science Publishers. 17

Bernstein, P. A. und U. Dayal (1994). *An overview of repository technology*. In: *Proc. of the International Conf. on Very Large Data Bases (VLDB)*, S. 705–713, Santiago, Chile. 1

Bibel, W., S. Hölldobler und T. Schaub (1996). *Wissensrepräsentation und Inferenz*. Vieweg Verlag. Auch unter http://medoc.informatik.tu-muenchen.de/Books/bibel/. 8, 13, 16, 28

Biggerstaff, T. und C. Richter (1989). *Reusability framework, assesment, and directions*. IEEE Software, 4(2):41–49. 11

Black, Andrew, N. Hutchinson, E. Jul und H. Levy (1986). *Object structure in the emerald system*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Bd. 21(11) d. Reihe *SIGPLAN Notices*, S. 78–86, Portland, Oregon, September 29 – October 2. ACM Press. 25

Blaha, M.R., N. Eastman und M. Hall (1989). *An extensible AE&C database model*. Computers and chemical Engineering, 13(7):753–766. 75

Blaschek, Günther (1991). *Type–safe object–oriented programming with prototypes — The concepts of Omega*. Structured Programming, 12:217–225. 48

Blaser, A., M. Jarke, H. Lehman und G. Müller (1987). *Datenbanksprachen und Datenbankbenutzung*. In: Lockemann, P.C. und J. Schmidt, Hrsg.: *Datenbankhandbuch*, Kap. 6, S. 563–635. Springer, Berlin, Heidelberg. 15

Boehm, B.W. (1988). *A spiral model of software development and enhancement*. IEEE Computer, 21(5):61–72. 2

Bogusch, R., B. Lohmann und W. Marquardt (1996). *Computer–aided process modeling with ModKit*. In: *Proc. of Chemputers Europe III*, Wiesbaden. 3, 4, 5, 6, 77

Bogusch, R. und W. Marquardt (1997). *A formal representation of process model equations*. Computers and chemical Engineering, 21(10):1105–1115. 6

Booch, Grady (1994). *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, Redwood City, 2. Aufl. 24, 56

Borning, A.H. (1986). *Classes versus prototypes in object-oriented languages*. In: *ACM/IEEE Fall Joint Computer Conference*, S. 30–46, Dallas, TX. 43, 45, 46

Boudaud, F., J. Broenink, D. Brück, T. Ernst, P. Fritzson, A. Jeandel, K. Juslin, M. Klose, S. Mattsson, M. Otter, P. Sahlin, H. Tummescheit und H. Vangheluwe (1997). *Modelica — A Unified Object–Oriented Language for Physical Systems Modeling*. http://www.modelica.org/current/Modelica1.html. 75

Bañares-Alcántara, R. (1995). *Design support systems for process engineering—I und II*. Computers and chemical Engineering, 19(3):267–301. 3, 4, 5, 9, 80, 81

Bracha, G. und W. Cook (1990). *Mixin-based inheritance*. In: Meyrowitz, Norman, Hrsg.: *Proc. of the Conf. on Object-Oriented Programming: Systems, Languages, and Applications / Proc. of the European Conf. on Object-Oriented Programming*, S. 303–311, Ottawa, Canada. ACM Press. 43, 53, 56

Brinkkemper, Sjaak (1990). *Formalisation of Information Systems Modelling*. Doktorarbeit, University of Nijmegen, Nijmegen, Niederlande. 21

Cardelli, Luca (1988). *A semantics of multiple inheritance*. Information and Computation, 76(2-3):138–164. 27

Casais, Eduardo (1992). *An incremental class reorganization approach*. In: Madsen, O. Lehrmann, Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, LNCS 615, S. 114–132, Utrecht, Niederlande. Springer. 68

Castano, S., V. D. Antonellis, C. Francalanci und B. Pernici (1994). *A reusability–based comparison of requirements specification methodologies*. In: Verrijn-Stuart, A.A. und T. Olle, Hrsg.: *Conference on Methods and Associated Tools for the Information System Life-Cycle*, S. 63–84. IFIP WG 8.1, Elsevier Science. 67, 134

Chambers, Craig (1993). *Predicate Classes*. In: Nierstrasz, Oscar M., Hrsg.: *European Conference on Object–Oriented Programming (ECOOP)*, LNCS 707, S. 268–296. Springer. 61, 63

Chen, J.-B. und S. Lee (1996). *Generation and reorganization of subtype hierarchies*. Journal of Object–Oriented Programming, 9(1):26–35. 71

Chiba, Shigeru und T. Masuda (1993). *Designing an extensible distributed language with a meta-level architecture*. In: Nierstrasz, O., Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, LNCS 707, S. 483–502, Kaiserslautern. Springer. 21

Costello, D.J., E. Fraga, N. Skilling, G. Ballinger, R. Banares-Alcantara, J. Krabbe, D. Laing, R. McKinnel, J. Ponton und M. Spenceley (1996). *Épée: A support environment for process engineering*. Computers and chemical Engineering, 20(12):1399–1412. 9

Cunningham, Douglas (1998). Persönliche Mitteilung. 50

Darwen, H. und C. Date (1995). *The third manifesto*. SIGMOD Record, 24(1):39–49. 11

Date, C. J. (1995). *An Introduction to Database Systems*. Addison-Wesley, Reading, Mass., 6. Aufl. 38

Davis, Stephen R. (1992). *C++ objects that change their types*. Journal of Object–Oriented Programming, 5(4):27–32. 63

Dawkins, Richard (1987). *The Blind Watchmaker*. W.W.Norton and Company, New York. 37

De Antonellis, V. und B. Zonta (1990). *A disciplined approach to office analysis*. IEEE Transactions on Software Engineering, 16(8):822–828. 67

Deutsch, Gregor (1994). *Untersuchungen zu Modellierungsvarianten von Destillationsprozessen*. Studienarbeit am Lehrstuhl für Prozeßtechnik der RWTH Aachen. 31

Dicky, H., C. Dony, M. Huchard und T. Libourel (1996). *On automatic class insertion with overloading*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Bd. 31(10) d. Reihe *ACM SIGPLAN Notices*, S. 251–267, New York. ACM Press. 68, 69

Dömges, Ralf (1999). *Projektspezifische Methoden zur Nachvollziehbarkeit von Anforderungsspezifikationen*. Doktorarbeit, RWTH Aachen, Fachgruppe Informatik. 6, 136

Dömges, R., B. Lohmann, K. Pohl, M. Jarke und W. Marquardt (1996). *PRO–ART/CE — An environment for managing the evolution of chemical process simulation models*. In: *Proc. of the 10th European Simulation Multiconference*, S. 1012–1017, Budapest, Ungarn. 3, 96

Dutoit, Allen, S. Levy, D. Cunningham und R. Patrick (1996). *The Basic Object System: Supporting a spectrum from prototypes to hardened code*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Bd. 31(10) d. Reihe *ACM SIGPLAN Notices*, S. 104–121, New York. ACM Press. 50

Dvorak, Joseph (1994). *Conceptual entropy and its effect on class hierarchies*. Computer, 27(6):59–63. 76

Ebbinghaus, H.-D., J. Flum und W. Thomas (1992). *Einführung in die mathematische Logik*. BI–Wiss.–Verl., Mannheim, Leipzig, 3. Aufl. 18, 20

ECMA und NIST (1991). *Reference Model for Frameworks of Software Engineering Environments*. NIST Special Publication 500-201, Technical Report ECMA TR/55, 2nd ed., ECMA/NIST. 3, 5, 6

Eisenecker, Ulrich W. (1997). *Generative Programmierung und Komponenten*. OBJEKTspektrum, (3):80–84. 54

Ferrandina, Fabrizio, T. Meyer und Z. Roberto (1994). *Implementing Lazy Database Updates for an Object Database System*. Technischer Bericht 9, ESPRIT-III Project GOODSTEP (6115), http://www.dbis.informatik.uni-frankfurt.de/REPORTS/GOODSTEP/GoodStepReport009.ps.gz. 63

Foss, B.A., B. Lohmann und W. Marquardt (1997). *A field study of the industrial modeling process*. In: *International Symposium on Advanced Control of Chemical Processes, ADCHEM 97*, S. 571–585, Banff, Canada. 2, 3, 136

Fowler, M. und K. Scott (1997). *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, Reading, Mass. 55

Fraga, E.S., G. Ballinger, J. Krabbe, D. Laing, R. McKinnel, J. Ponton, N. Shilling und M. Spenceley (1994). *The implementation of a portable, object-oriented, distibuted process engineering environment*. Technischer Bericht TR–94–17, Department of Chemical Engineering, Edinburgh University, Mayfield Road, Edinburgh, UK. 79

Gamma, E., R. Helm, R. Johnson und J. Vlissides (1996). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading. 54

Geoffrion, A.M. (1989). *Computer–based modelling environments*. European Journal of Operational Research, 41:33–43. 3

Gil, J. und D. Lorenz (1996). *Environmental acquisition — A new inheritance–like abstraction mechanism*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Bd. 31(10) d. Reihe *ACM SIGPLAN Notices*, S. 214–231, New York. ACM Press. 75

Goel, A.K. und B. Chandrasekaran (1992). *Case–based design: A task analysis*. In: Tong, C. und D. Sriram, Hrsg.: *Artificial Intelligence in Engineering Design*, Bd. 2, Kap. 5, S. 165–183. Academic Press, Boston, San Diego. 73

Goldberg, Adele und D. Robson (1989). *Smalltalk-80: The Language*. Addison-Wesley, Reading, MA. 21, 27

Gottlob, G., M. Schrefl und B. Röck (1996). *Extending object–oriented systems with roles*. Transactions on Information Systems, 14(3). 57, 59, 63, 92, 101

Gross-Hardt, M. und G. Vossen (1993). *Towards class-less object models for engineering design applications*. LNCS, 720:36–47. 46, 48, 81

Group, The épée (1994). *The épée Programming and Reference Manual; épée 0.2 Release*. Technischer Bericht TR–94–18, Department of Chemical Engineering, Edinburgh University, Mayfield Road, Edinburgh, UK. 79

Group, The n-dim (1995a). *The Basic Object System — An Object Model for Evolutionary Prototyping*. Research report EDRC-05-91-95, Engineering Design Research Center, Carnegie Mellon University, Pitsburgh, PA 15213. 17, 50

Group, The n-dim (1995b). *n–dim — An environment for realizing computer supported collaboration in design work*. Technischer Bericht 05-93-95, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA. 3, 9, 78, 79

Hahn, Jürgen (1997). *Entwicklung von Modellbausteinen zur Modellierung von 2–Phasen–Systemen und kaskadierenden 2–Phasen–Systemen in der Modellierungsumgebung ModKit*. Diplomarbeit, Lehrstuhl für Prozeßtechnik, RWTH Aachen. 31

Hartmann, T., G. Saake, R. Jungclaus, P. Hartel und J. Kusch (1994). *Revised Version of the Modelling Language* Troll *(Version 2.0)*. Informatik–Bericht 94–03, Technische Universität Braunschweig. 62, 63

Hemming, Werner (1991). *Verfahrenstechnik*. Vogel Verlag, Würzburg, 6. Aufl. 1, 2

Heuer, Andreas (1992). *Objektorientierte Datenbanken: Konzepte, Modelle, Systeme*. Addison–Wesley, Bonn, München, Paris. 17, 18, 24, 26, 38

Ibrahim, Mamdouh H., W. E. Nejcek und F. A. Cummings (1991). *Instance specialization without delegation*. Journal of Object–Oriented Programming, S. 53–56. 19, 52

ISO (1990). *Information technology — Information Resource Dictionary System (IRDS) framework*. ISO/IEC International Standard 10027, 1. Aufl. 15

ISO (1994). *Process data exchange using STEP, Part 1: Overview and fundamental principles*. International Standard, U.S. Product Data Association, Fairfax, VA. 5

ISO (1998). *STEP Part 231: Process Engineering Data: Process Design and Process Specification of Major Process Equipment*. Committee Draft ISO TC 184/SC 4/WG 3 N745, U.S. Product Data Association, Fairfax, VA. 5, 22, 82

Jarke, M. und W. Marquardt (1996). *Design and evaluation of computer–aided process modeling tools*. In: Davis, J.F., G. Stephanopoulos und V. Venkatasubramanian, Hrsg.: *Intelligent Systems in Process Engineering*, S. 97–109. AIChE Symposium. 3, 4, 5

Jeusfeld, M.A. und U. Johnen (1995). *An executable meta model for re-engineering of database schemas*. Intl. Journal on Cooperative Information Systems, 4(2&3):237–258. 21

Jeusfeld, Manfred (1997). Persönliche Kommunikation. 20

Jeusfeld, Manfred A. (1992). *Änderungskontrolle in deduktiven Objektbanken*. Dissertationen zur künstlichen Intelligenz. Infix–Verlag, St. Augustin. 27, 111, 119, 121, 122

Johnson, P. und C. Rees (1992). *Reusability through fine-grain inheritance*. Software — Practice and Experience, 22(12):1049–1068. 54

Johnson, Richard und M. Palaniappan (1993). *MetaFlex: A flexible metaclass generator*. In: Nierstrasz, O., Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, LNCS 707, S. 503–528, Kaiserslautern. Springer. 21

Kathuria, Ravi (1997). *Improved modeling and design using assimilation and property modeling*. Journal of Object–Oriented Programming, 10(3):32–37. 59

Kemper, A. und G. Moerkotte (1994). *Object–Oriented Database Management; Applications in Engineering and Computer Science*. Prentice–Hall International, London. 10, 11, 15, 17, 24, 26, 48

Kiczales, G., J. des Rivieres und D. Bobrow (1991). *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA. 21

Kim, Won (1990). *Introduction to Object–Oriented Databases*. MIT Press, Cambridge, MA. 19, 20

Klas, Wolfgang und M. Schrefl (1995). *Metaclasses and Their Application*. LNCS 943. Springer, Berlin, Heidelberg. 19, 21, 27, 52, 111, 113

Knudsen, Jørgen Lindskov (1988). *Name collision in multiple classification hierarchies*. In: Gjessing, S. und K. Nygaard, Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, Bd. 322 d. Reihe *LNCS*, S. 93–109, Oslo, Norwegen. Springer. 20, 53

Kolodner, Janet (1993). *Case-Based Reasoning*. Morgan Kaufmann, San Mateo, California. 38, 73

Krieger, James H. (1995). *Process simulation seen as pivotal in coporate information flow*. Chemical & Engineering News, S. 50–61. 27. März 1995. 2

Krishnan, R., P. Piela und A. Westerberg (1991). *Reusing mathematical models in ASCEND*. Technischer Bericht 06-109-91, Engineering Design Reserch Center, Carnegie Mellon University, Pittsburgh, PA. 13

Kristensen, Bent Bruun (1995). *Object–oriented modeling with roles*. In: *Proc. of the International Conf. on Object–Oriented Information Systems*, S. 57–71. 56, 57, 58

Krobb, C., B. Lohmann und W. Marquardt (1998). *The Chemical Engineering Data Model* VEDA. *Part 6: The Process of Model Development*. Technischer Bericht LPT-1998-06, Lehrstuhl für Prozeßtechnik, RWTH Aachen. 16, 22, 135

Krobb, Claudia (1997). *Entwicklung einer Spezialisierungshierarchie für Modellierungsschritte im objekt–orientierten Datenmodell* VEDA. Diplomarbeit, Lehrstuhl für Prozeßtechnik, RWTH Aachen. 6, 18

Krueger, Charles W. (1992). *Software reuse*. ACM Computing Surveys, 24(2):131–183. 7, 11, 12, 40, 45

LaLonde, W. und J. Pugh (1991). *Subclassing ≠ subtyping ≠ is-a*. Journal of Object–Oriented Programming, 3(5):57–62. 17, 20, 26

Levy, S., E. Subrahmanian, S. Konda, R. Coyne, A. Westerberg und Y. Reich (1993). *An overview of the n–dim environment*. Research report EDRC-05-65-93, Engineering Design Research Center, Carnegie Mellon University, Pittsburgh, PA. 78, 79

Li, Qing und G. Dong (1994). *A framework for object migration in object–oriented databases*. Data & Knowledge Engineering, 13(3):221–242. 61, 62, 63

Lieberherr, K.J., P. Bergstein und I. Silva-Lepe (1991). *From objects to classes: Algorithms for optimal object–oriented design*. Software Engineering Journal, 6(4):205–227. 68, 69, 70, 71

Lieberman, Henry (1986). *Using prototypical objects to implement shared behaviour in object-oriented systems*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, S. 214–223, Portland, Orgeon. ACM. 40, 43, 45, 47

Light, Marc (1993). *Classification in feature-based default inheritance hierarchies*. Technischer Bericht TR473, University of Rochester, Computer Science Department. 72

Liskov, Barbara und J. M. Wing (1993). *A new definition of the subtype relation*. In: Nierstrasz, O., Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, LNCS 707, S. 118–141, Kaiserslautern. Springer. 25

Lohmann, Bernd (1994). *Fuzzy Case–Based Reasoning bei der Modellierung verfahrenstechnischer Prozesse*. Diplomarbeit, Lehrstuhl für Unternehmensforschung, RWTH Aachen. 73, 74

Lohmann, Bernd (1997). *Ansätze zur Unterstützung des Arbeitsablaufs bei der rechnerbasierten Modellierung verfahrenstechnischer Prozesse*. Doktorarbeit, Lehrstuhl für Prozeßtechnik, RWTH Aachen. 5, 6, 8

Loomis, Mary E.S. (1994). *Hitting the relational wall*. Journal of Object–Oriented Programming. 11

Luck, Kai von (1990). *KL-One: Eine Einführung*. IWBS Report 106, IBM Wissenschaftliches Zentrum; Institut für Wissensbasierte Systeme, Stuttgart. ISSN 0938-1864. 15, 20, 28

Maarek, Y. S., D. M. Berry und G. E. Kaiser (1991). *An information retrieval approach for automatically constructing software libraries*. IEEE Transactions on Software Engineering, 17(8):800–813. 66, 67, 72, 74

MacFarlane, A.G.J., G. Grübel und J. Ackermann (1989). *Future design environments for control engineering*. Automatica, 25:165–176. 5

Maffezzoni, C. und R. Girelli (1998). *MOSES: modular modeling of physical systems in an object–oriented database*. Mathematical Modelling of Systems, 4(2):121–147. 15, 76

Maffezzoni, Claudio, R. Girelli und P. Lluka (1994). *Object–oriented database support for modeling and simulation*. In: *Proc. of the European Simulation and Modeling Conf.*, Barcelona. 10

Marquardt, W. (1991). *An object–oriented representation of structured process models*. In: *Proceedings of 1st European Symposium on Computer Aided Process Engineering (ESCAPE-1)*. In Computers and chemical Engineering 16 (1992), Suppl., S329–S336. 4, 5

Marquardt, W. (1992). *Rechnergestützte Erstellung verfahrenstechnischer Prozeßmodelle*. Chemie–Ingenieur–Technik, 64(1):25–40. 8, 9, 10

Marquardt, W. (1996). *Trends in computer–aided process modeling*. Computers and chemical Engineering, 20(6/7):591–609. 2, 4, 5, 7, 9, 15

Marquardt, W. (1997). *Modellbildung und Simulation verfahrenstechnischer Prozesse*. Vorlesungsmanuskript, Lehrstuhl für Prozeßtechnik, RWTH Aachen. 2

Marino, O., F. Rechenmann und P. Uvietta (1990). *Multiple perspectives and classification mechanism in object-oriented representation*. In: Aiello, Luigia Carlucci, Hrsg.: *Proc. of the 9th European Conf. on Artificial Intelligence (ECAI'90)*, S. 425–430. 55

Marquardt, W., A. Gerstlauer, S. Räumschüssel und B. Raichle (1992). *Objekt–orientierte Datenmodellierung für die Repräsentation von verfahrenstechnischen Prozeßmodellen*. Interner Forschungsbericht, Universität Stuttgart. 5, 17, 19, 28

Mercado Jr., A. (1988). *Hybrid: Implementing Classes with Prototypes*. Diplomarbeit, Brown University Department of Computer Science, Providence, RI. Technical Report CS–88–12. 60

Meyer, Bertrand (1986). *Genericity versus inheritance*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, S. 391–405, Portland, OR. 54

Meyer, Bertrand (1996). *The many faces of inheritance: A taxonomy of taxonomy*. IEEE Computer, 29(5):105–108. Extended version in chapter 25 of [Meyer 1997]. 21, 26

Meyer, Bertrand (1997). *Object-oriented software construction*. Prentice Hall, New York, N.Y., 2. Aufl. 27, 55, 150

Mezini, Mira (1997). *Dynamic object evolution without name collisions*. In: Aksit, Mehmet und S. Matsuoka, Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, S. 190–219. LNCS 1241, Springer. 53

Minsky, Marvin (1975). *A framework for representing knowledge*. In: Winston, P., Hrsg.: *The Psychology of Computer Vision*. McGraw-Hill. 16

Molitor, R. (2000). *Unterstützung der Modellierung verfahrenstechnischer Prozesse durch Nicht-Standardinferenzen in Beschreibungslogiken*. Doktorarbeit, RWTH Aachen, Fachgruppe Informatik. 70, 96

Monk, S. und I. Sommerville (1993). *Schema evolution in OODBs using class versioning*. SIGMOD Record, 22(3):16–22. 63

Moore, Ivan (1996). *Automatic inheritance hierarchy restructuring and method refactoring*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Bd. 31(10) d. Reihe *ACM SIGPLAN Notices*, S. 235–250, New York. ACM Press. 69

Morie, T., H. Onodera und K. Tamaru (1993). *A system for analog circuit design that stores and re–uses design procedures*. In: *Proc. of the IEEE 1993 Custom Integrated Circuits Conference*, San Diego, CA. IEEE. 8, 9

Motschnig-Pitrik, R. und J. Mylopoulos (1992). *Classes and Instances*. Intern. Journal of Intelligent and Cooperative Information Systems, 1(1):61–92. 17, 18, 116

Mugridge, W.B., J. Hosking und J. Hamer (1991). *Multi-methods in a statically-typed programming language*. In: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, Geneva. 62

Mylopoulos, John, A. Borgida, M. Jarke und M. Koubarakis (1990). *Telos: Representing knowledge about information systems*. ACM Transactions on Information Systems, 8(4):325–362. 21, 119

Nagel, C.J., C. Han und G. Stephanoupolous (1995). *Modeling languages: Declarative and imperative descriptions of chemical reactions and processing systems*. In: Stephanopolous, G. und C. Han, Hrsg.: *Intelligent systems in process engineering*, Bd. 21 d. Reihe *Advances in Chemical Engineering*, S. 1–91. Academic Press, San Diego. 15, 30, 76, 77

Nagel, Manfred (1990). *Softwaretechnik: Methodisches Programmieren im Großen*. Springer, Berlin, Heidelberg. 13

Nagl, M. und W. Marquardt (1997). *SFB–476 IMPROVE: Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik*. In: *Informatik 97 — Information als Innovationsmotor*, Aachen. GI, Springer. 2, 3, 5

Napoli, A., J. Lieber und R. Curien (1996). *Classification–based problem–solving in case–based reasoning*. In: Smith, I. und B. Faltings, Hrsg.: *Proc. of the Third European Workshop on Advances in Case-Based Reasoning*, Bd. 1168 d. Reihe *LNAI*, S. 295–308. Springer. 73

Nebel, Bernhard (1990). *Reasoning and Revision in Hybrid Representation Systems*. LNAI 422. Springer, Berlin, Heidelberg. 70, 71

Neumann, Ulrich (1995). *Entwicklung eines Systems zur Verwaltung von Konstruktionskatalogen*. Diplomarbeit, RWTH Aachen. 38

Nguyen, G. T., D. Rieu und J. Escamilla (1992). *An object model for engineering design*. In: Madsen, O. Lehrmann, Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, LNCS 615, S. 233–251, Utrecht, Niederlande. Springer. 55, 81, 133

Nguyen, G.T. und D. Rieu (1992). *Multiple object representations*. In: *Proc. 20th ACM Computer Science Conference*, S. 197–204, Kansas City (Mo). ACM Press. 81

Odberg, Erik (1994). *Category classes: Flexible classification and evolution in object–oriented databases*. In: Wijers, Gerhard, S. Brinkkemper und T. Wasserman, Hrsg.: *Proc. of the Conf. on Advanced Information Systems Engineering (CAISE)*, LNCS 811. Springer. 62, 63, 93

Odberg, Erik (1995). *MultiPerspectives: Object evolution and schema modification management for object-oriented databases*. Doktorarbeit, Norwegian Institute of Techology, Department of Computer Systems and Telematics. 59

Odell, James J. (1992). *Dynamic and multiple classification*. Journal of Object–Oriented Programming, 4(8):45–48. 54, 63

Pantelides, C.C. und H. Britt (1994). *Multipurpose process modelling environments*. In: Biegler und Doherty, Hrsg.: *Foundations of Computer Aided Design Conference (FOCAPD'94)*, S. 128–141, Snowmass, CO. CACHE Publications. 3, 4

Piela, P.C., T. Epperly, K. Westerberg und A. Westerberg (1991). *ASCEND: An object–oriented computer environment for modeling and analysis: the modeling language*. Computers and chemical Engineering, 15(1):53–72. 75

Pohl, K., R. Dömges und M. Jarke (1997). *Towards method-driven trace capture*. In: *Proc. of the 9th Conference on Advanced Information System Engineering (CAISE)*, Barcelona, Spanien. Springer. 8

Pohl, Klaus (1995). *Process–centered Requirements Engineering*. Nr. 5 in *Advanced Software Development series*. RSP, Hertfordshire, UK. 5, 6, 8

Porter, Harry H. (1992). *Separating the subtype hierarchy from the inheritance of implementation*. Journal of Object–Oriented Programming, 4(9):20–21,24–29. 26

Prieto-Diaz, Ruben (1991). *Implementing facetted classification for software reuse*. Communications of the ACM, 34(5):88–97. 38, 55

Reiter, Raymond (1984). *Towards a logical reconstruction of relational database theory*. In: Brodie, M., J. Mylopoulos und J. Schmidt, Hrsg.: *On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages*, S. 191–233. Springer-Verlag, Berlin. 119

Richardson, J. und P. Schwarz (1991). *Aspects: Extending objects to support multiple, independent roles*. In: Clifford, James und R. King, Hrsg.: *Proc. of the 1991 ACM SIGMOD International Conf. on Management of Data*, S. 298–307, Denver, Colorado. 56, 57

Rijsbergen, C.J. van (1979). *Information Retrieval.*. Butterworths. 66, 72

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy und W. Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey. 75

Rundensteiner, Elke A. (1992). *A Class Integration Algorithm and Its Application for Supporting Consistent Object Views*. Technical Report ICS-TR-92-50, University of California, Irvine, Department of Information and Computer Science. 68

Russel, S. und P. Norvig (1995). *Artificial Intelligence: A Modern Approach*. Prentice–Hall International, London, Englewood Cliffs, NJ. 16, 28

Sattler, Klaus (1988). *Thermische Trennverfahren: Grundlagen, Auslegung, Apparate*. VCH, Weinheim, Basel. 31

Sattler, Ulrike (1998). *Terminological knowledge representation systems in a process engineering application*. Doktorarbeit, RWTH Aachen. 29, 70, 78, 96

Schembecker, G., K. Simmrock und A. Wolff (1994). *Synthesis of chemical process flowsheets by means of cooperating knowledge integrating systems*. In: *Proc. of European Symposium on Computer Aided Process Engineering (ESCAPE 4)*, S. 333–341, Dublin, UK. 8

Schenk, Douglas und P. Wilson (1994). *Information Modelling the EXPRESS Way*. Oxford University Press, New York. 82

Schmidt, Joachim W. (1987). *Datenbankmodelle*. In: Lockemann, P.C. und J. Schmidt, Hrsg.: *Datenbankhandbuch*, Kap. 1. Springer, Berlin, Heidelberg. 19

Schwanke, Robert W. (1991). *An intelligent tool for re-engineering software modularity*. In: *Proc. of the 13th International Conf. on Software Engineering*, S. 83–92. 66, 67

Sciore, Edward (1989). *Object specialization*. ACM Transactions on Information Systems, 7(2):103–122. 43, 47, 54, 60, 61, 63

Shlear, S. und S. J. Mellor (1988). *Object-Oriented Systems Analysis: Modeling the World in Data*. Yourdon Press, New York. 7, 8

Smith, D. (1995). *Modelling — a tool for all seasons*. In: *International Hyprotech Users' Conference*, Noordwijk. 2

Sommerville, Ian (1995). *Software Engineering*. Addison Wesley, Harlow, England, 5. Aufl. 1, 2, 7, 8, 134

Souza, D. und W. Marquardt (1998). *The Chemical Engineering Data Model* VeDa. *Part 2: Structural Modeling Objects*. Technischer Bericht LPT-1998-02, Lehrstuhl für Prozeßtechnik, RWTH Aachen. 16, 22, 23, 97, 100, 135

Spanoudakis, George E. (1994). *Analogical Similarity of Objects: A Conceptual Modeling Approach*. Doktorarbeit, ICS, Foundation for Research and Technology — Hellas, Heraklion, Kreta. 67, 73, 95

Stefik, M. und D. Bobrow (1986). *Object-oriented programming: themes and variations*. AI Magazine, 6(4):40 – 62. 20

Stein, L. A. (1991). *A unified methodology for object-oriented programming*. In: M.Lenzerini, D.Nardi und M.Simi, Hrsg.: *Inheritance Hierarchies in Knowledge Representation and Programming Languages*, Kap. 13, S. 211–222. John Willey & Sons. 55

Stein, L.A., H. Lieberman und D. Ungar (1989). *A shared view of sharing: The treaty of Orlando*. In: Kim, Won und F. Lochovsky, Hrsg.: *Object–Oriented Concepts, Databases and Applications*, S. 31–48. Addison–Wesley, Reading, MA. 43, 44, 51, 52, 53, 54, 60, 64, 65, 76, 86

Stein, L.A. und S. Zdonik (1989). *Clovers: The Dynamic Behavior of Types and Instances*. Technical Report No. CS–89–42, Deptartment of Computer Science, Brown University, Providence, Rhode Island, USA. 57, 80

Stein, Lynn Andrea (1987). *Delegation is iInheritance*. In: Meyrowitz, Norman, Hrsg.: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, S. 138–146, New York, NY. ACM Press. 50, 60, 72, 80

Stephanopoulos, G., G. Henning und H. Leone (1990a). *MODEL.LA. A modelling language for process engineering — I. The formal framework*. Computers and chemical Engineering, 14(8):813–846. 76, 77

Stephanopoulos, G., G. Henning und H. Leone (1990b). *MODEL.LA. A modelling language for process engineering — II. Multifaceted modelling of processing systems*. Computers and chemical Engineering, 14(8):847–869. 77

Stephanopoulus, G. und C. Han (1996). *Intelligent systems in process engineering: A review*. Computers and chemical Engineering, 20(6/7):743–791. 8, 9

Steyaert, Patrick und W. D. Meuter (1995). *A marriage of class– and object–based inheritance without unwanted children*. In: Olthoff, Walter, Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, Bd. 952 d. Reihe *LNCS*, S. 127–144, Berlin, Heidelberg. Springer. 56, 59

Stonebraker, M., L. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein und D. Beech (1990). *Third-generation data base system manifesto*. ACM SIGMOD Record 19, 3. 13

Su, Jianwen (1991). *Dynamic constraints and object migration*. In: *Proc. of the International Conf. on Very Large Data Bases (VLDB)*, S. 233–242. 62

Surma, J. und B. Braunschweig (1996). *REPRO: Supporting flowsheet design by case-base retrieval*. In: Smith, Ian und B. Faltings, Hrsg.: *Proc. of the Third European Workshop on Advances in Case-Based Reasoning*, Bd. 1168 d. Reihe *LNCS*, S. 400–412, Berlin. Springer. 73, 74

Syvertsen, Tor G., F. Lillehagen und M. Løvstad (1992). *A generic object model for engineering design*. In: *Proc. of the Conf. on Technology of Object-Oriented Languages and Systems (TOOLS)*, S. 157–166. Prentice Hall, Hempstead, UK. 81

Taivalsaari, Antero (1993). *Object–oriented programming with modes*. Journal of Object–Oriented Programming, 6(3):25–32. 61

Taivalsaari, Antero (1996). *On the notion of inheritance*. ACM Computing Surveys, 28(3):438–479. 20, 25, 26, 28, 45, 46, 47

Talingent (1993). *Leveraging Object–Oriented Frameworks*. White paper, Talingent Inc. http://www.ibm.com/java/education/ooleveraging/index.html. 8

Tresch, Markus (1995). *Evolution in Objekt–Datenbanken*, Bd. 10 d. Reihe *Teubner–Texte zur Informatik*. B.G.Teubner, Leipzig. 17, 26

Ungar, D. und R. Smith (1987). *Self: The power of simplicity*. In: Meyrowitz, Norman, Hrsg.: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Bd. 22(12) d. Reihe *SIGPLAN Notices*, S. 227–242. ACM Press. 17, 43, 48

Urban, S.D. (1989). *Alice: An assertion language for integrity constraint expression*. In: *Proc. of the Comput. Software Applications Conf.*, S. 292 – 299, Orlando, FL. 19

Wegner, P. (1987). *Dimensions of object–based language design*. In: *Proc. of the Conf. on Object–Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Bd. 22(12) d. Reihe *ACM SIGPLAN Notices*, S. 168–182. 67

Wegner, P. und S. Zdonik (1988). *Inheritance as an incremental modification mechanism or what like is and isn't like*. In: Gjessing, S. und K. Nygaard, Hrsg.: *Proc. of the European Conf. on Object–Oriented Programming (ECOOP)*, Bd. 322 d. Reihe *LNCS*, S. 55–77, Oslo, Norwegen. Springer. 21, 52, 53, 86, 87

Westerberg, A.W. und P. Piela (1994). *The ASCEND language syntax and semantics*. Technischer Bericht, Carnegie Mellon University, Engineering Design Research Center, Pittsburgh, PA. 75

Wiebe, Douglas (1988). *Partial Instantiation*. Technischer Bericht 88–04–04, Department of Computer Science, University of Washington. 50

Wieringa, R.J., W. de Jonge und P. Spruit (1995). *Using dynamic classes and role classes to model object migration*. Theory and Practice of Object Systems, 1(1):61–83. 43, 56, 57, 58, 62, 101, 131

Wong, R.K., H. Chau und F. Lochovsky (1997). *Dynamic knowledge representation in DOOR*. In: *Knowledge and Data Engineering Workshop (KDEX'97)*, S. 89–96. IEEE. 58, 101

Zdonik, Stanley B. (1990). *Object-oriented type evolution*. In: Bancilhon, F. und P. Buneman, Hrsg.: *Advances in Database Programming Languages*, Kap. 16, S. 277–288. ACM. 51, 62, 63, 95

Lebenslauf

Name:	Markus Baumeister	
Staatsangehörigkeit:	deutsch	
Familienstand:	ledig	
Wohnort:	Moltkestr Str. 9; 52066 Aachen; Tel. 0241/873046	
Geburt	11.6.1968	in Münster\Westf.
Schulbildung	1974 – 1978	Besuch der Grundschule in Bad Driburg und Arnsberg
	1978 – Mai 1987	Besuch des Gymnasium Laurentianums in Arnsberg mit abschliessendem Abitur
Wehrdienst	Juli 1987 – Sep. 1988	Dienst als Transportsoldat in Hannover & Hildesheim
Hochschule	Nov. 1988 – März 1991	Studium der Informatik an der Universität Passau, dabei
	Okt. 1990	Erlangung des Vordiploms
	April 1991 – Juli 1994	Studium der Informatik an der RWTH Aachen, mit abschliessendem Diplom; Vertiefungsgebiet: Datenbanken, Nebenfach: Wirtschaftswissenschaften
Tätigkeiten	1.7.1990 – 31.3.1991	Anstellung als studentische Hilfskraft mit Aufgabengebiet Programmierung beim Lehrstuhl für Informatik mit Schwerpunkt dialogorientierte Systeme der Uni Passau
	2.5.1991 – 31.7.1992	dito beim Lehrstuhl für Informatik V (Informationssysteme) der RWTH Aachen
	15.8.1994 – 30.9.1994	Anstellung als wissenschaftliche Hilfskraft am Lehrstuhl für Prozeßtechnik der RWTH Aachen
	1.10.1994 – 30.9.1997	Graduiertenkollegsstipendium der DFG im Graduiertenkolleg “Informatik und Technik“
	1.10.1997 – 30.6.1998	Anstellung als wissenschaftlicher Angestellter am Lehrstuhl für Prozeßtechnik der RWTH Aachen im Rahmen des SFB 476 “Informatische Unterstützung übergreifender Entwicklungsprozesse in der Verfahrenstechnik“
	15.7.1998 —	Anstellungen als wissenschaftlicher Mitarbeiter im Forschungslaboratorium Aachen der Philips Deutschland GmbH