# Ein objektorientiertes Datenmodell zur Beschreibung chemischer Prozesse

Markus Baumeister

*RWTH Aachen, Lehrstuhl für Prozeßtechnik und Lehrstuhl für Informatik V*
baumeister@informatik.rwth-aachen.de

**Kurzfassung:** Die Komplexität der Modellierung chemischer Prozesse muß zur Vereinfachung und Beschleunigung von Analyse und Design chemischer Anlagen verringert werden. In diesem Artikel werden Teile des Objektmodells der unter dieser Zielsetzung entworfenen Sprache VEDA vorgestellt. Hierbei wird besonders auf eine alternative Repräsentationsmöglichkeit für Metamodelle in zweistufigen Objektsystemen, auf die gleichzeitige Verwendung prototyp- und klassenbasierter Konzepte für mehr Flexibilität bei gleichbleibenden Klassifizierungsmöglichkeiten sowie auf sich aus dem dynamischen Schema ergebende Implementierungsprobleme eingegangen.

**Schlüsselwörter:** object orientation, metamodeling, prototype-based object-orientation, roles, chemical process modeling

# 1. Das Anwendungsgebiet

Die Modellierung chemischer Prozesse stellt einen Teilschritt des Entwurfs neuer bzw. der Analyse bestehender chemischer Prozesse dar (vgl. Bild 1). Das Ergebnis der Modellierung muß ein formales Modell sein, das in weiteren Schritten ausgewertet und mit der Realität bzw. den Anforderungen verglichen wird. Die Auswertung erfolgt hierbei in der Regel durch Simulation, d.h. die wiederholte Berechnung des den Prozeß beschreibenden Gleichungssystems zur Erzeugung einer Kette von Zuständen, da die analytische Lösung des Gleichungssystems für alle nicht trivialen Probleme unmöglich ist. Der Zyklus aus Modellierung, Simulation und Evaluierung der Simulationsergebnisse wird hierbei solange durchlaufen, bis die Übereinstimmung mit den Anforderungen/der Realität ausreichend erscheint.

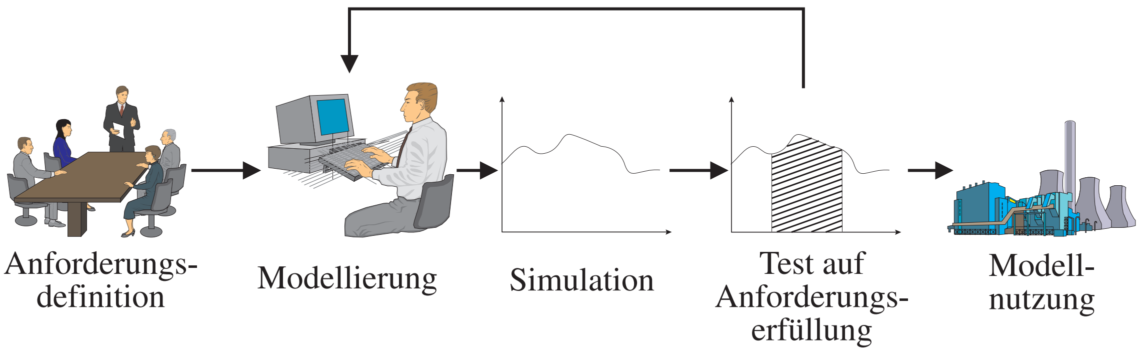


**Abbildung 1.** Ablaufschema der Modellierung chemischer Prozesse

Bei der Modellierung treten aus dem Software–Engineering bekannte Probleme auf: Komplexität, Wiederverwendbarkeit, Bestimmung von Abhängigkeiten, kreativer Prozeß u.ä. . Als Verbesserung gegenüber den konventionellen, meist gleichungsnahen Modellierungsverfahren (z.B. SpeedUp) wird am Lehrstuhl für Prozeßtechnik ein System entwickelt [18, 25, 3], das die den Prozeß beschreibenden Gleichungssysteme durch Aggregation elementarer Gleichungen entlang des strukturellen Aufbaus des Prozesses gewinnt.

Alle Modellbausteine (also sowohl strukturelle Komponenten als auch die das Verhalten der Komponenten beschreibenden Gleichungen) werden in der Modellierungssprache VeDa ([31, 19]) notiert. Strukturmodelle eines Prozesses in VeDa lassen sich, von einem abstraktem Standpunkt aus gesehen, als ungerichtete Graphen aus 'Devices' und 'Connections' auffassen (vgl. Bild 2unten), wobei beide Komponenten entweder aus Teilkomponenten zusammengesetzt oder durch Gleichungen beschrieben werden. Bild 2 zeigt einen Teil der Modellierung des EO–EG–Prozesses [16]. Als Beispiel für eine strukturelle Aggregation wird der Rührkesselreaktor aus Kessel und Kühlmantel zusammengesetzt, wobei der Kessel wiederum als elementarer Baustein durch Gleichungen beschrieben wird. Zur Vorbereitung der Simulation werden alle Gleichungen solcher elementaren Bausteine — ggf. um übergreifende Gleichungen erweitert — entlang der part–of Hierarchie aggregiert, vereinfacht und der Simulationssoftware übergeben.

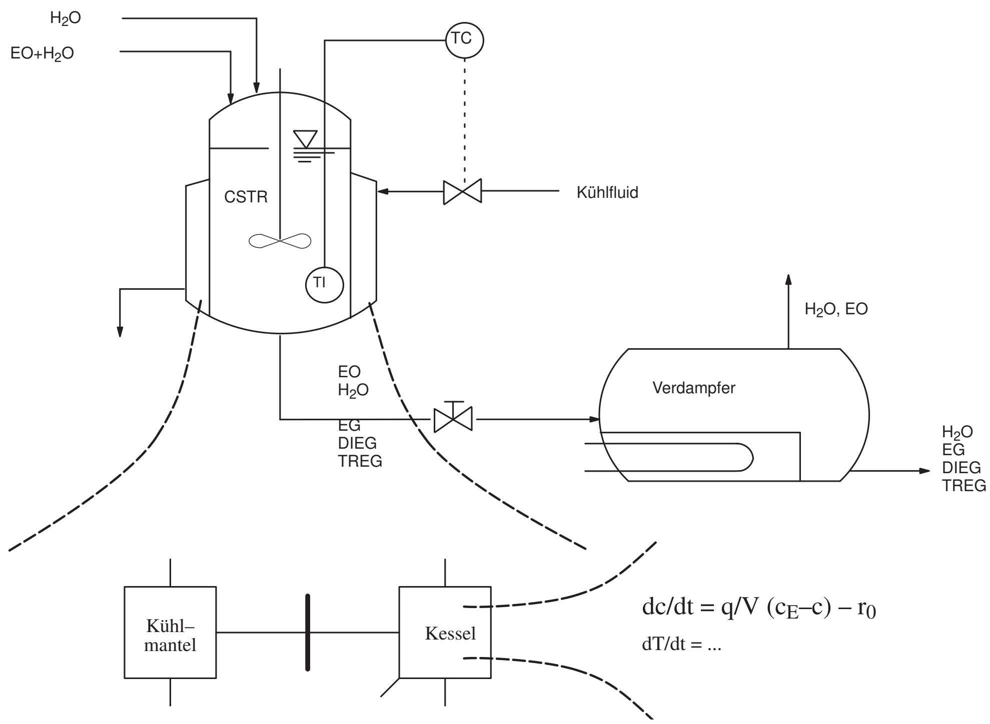


**Abbildung 2.** Ethylenoxid–Ethylenglykol–Beispielprozeß nach [16]

Das Ziels, eine Vereinfachung der Erstellung und Handhabung der umfangreichen einen Prozeß beschreibenden Gleichungssysteme, wird durch diese Modellierungsweise verwirklicht wie mit Hilfe der Entwicklungsumgebung ModKit [3] am Lehrstuhl für Prozeßtechnik demonstriert wurde. Die weiteren zu lösenden Fragestellungen sind:

- die Unterstützung unerfahrener Modellierer
- die Wiederverwendung vorhandener Teilmodelle
- die einfache explorative Modellierung

Sie erfordern allerdings ein hierfür speziell angepaßtes Datenmodell, sowie Mechanismen, um die notwendigen Informationen speichern und manipulieren zu können. Die Entwicklung dieses Datenmodells, sowie die Implementierung einer entsprechenden Datenbank sind die Ziele dieser Arbeit.

Einige der in diesem Zusammenhang auftretenden Probleme werden in den nächsten Kapiteln behandelt. Kapitel 2 untersucht die Möglichkeiten, Wissen über chemische Prozesse in VeDa in mehreren Ebenen zu strukturieren und diese Struktur dann auf nur zwei Ebenen abzubilden. Mit der Frage, ob die Verwendung lediglich einer Ebene nicht vorzuziehen sei und wie sich die Vorteile ein- und zweistufiger Modellierung kombinieren lassen, beschäftigt sich Kapitel 3. In Kapitel 4 wird auf das Problem eingegangen, das entwickelte Datenmodell mit auf konventionellen Programmiersprachen beruhenden Anwendungen zu koppeln; bis dann Kapitel 5 abschließend alles zusammenfaßt.

# 2. Metamodellierung

## 2.1. Ein Metamodell für VeDa

Die von W.Marquardt et al. an der Universität Stuttgart entwickelte Modellierungssprache VeDa [21] wird mit Hilfe von Frames notiert, ist folglich im strengen Sinne nicht vollständig objektorientiert. Sie repräsentiert Attribute, Methoden und Konsistenzbedingungen durch benannte Slots und deren Eigenschaften durch Facetten. Zusätzlich klassifiziert sie Attribute in vier, Methoden und Konsistenzbedingungen in zwei Typen. Vererbung zwischen Klassen ist auf gewohnte Weise definiert (vgl. etwa [13, Kap.10]), allerdings wird die Objekt–Substituierbarkeit nicht garantiert da Attribute spezialisiert werden dürfen (Covarianz). Obwohl KL-ONE [32] als eines der Vorbilder angesehen wird, zeigt dir ursprüngliche Sprachdefinition eine unscharfe Trennung von T-Box (also Klassen) und A-Box (also Instanzen). Zudem treten nur unzureichend definierte Metaklassen auf.

Im Verlauf der Überarbeitung und Formalisierung der Sprache ([31, Kap.2]) ergab sich neben der Definition und Überarbeitung von Typen, Typkonstruktoren, Facetten- und Methodensemantik und -kompatibilität, die Notwendigkeit, die Strukturierung der in VeDa auftretenden Objekte in verschiedene Klassen-/Instanzebenen zu überprüfen. Das Ergebnis, eine vierstufige Hierarchie gefüllt mit einigen Beispielklassen, zeigt Bild 3a. Die oberste Ebene enthält die Definition der Sprache VeDa selbst, etwa die Bedeutung der verschiedenen Facetten und Attributtypen und den Aufbau der unterschiedlichen Framearten. Im Bild befindet sich allerdings nur eine sehr unvollständige Definition eines Composite-Concepts, da die verwendete VeDa-artige Notation für die Sprachdefinition ungeeignet ist. Mittels *ConceptBase*[11] lassen sich die Klassen dieser Ebene dagegen größtenteils ausdrücken. Diese Ebene fand beispielsweise Verwendung, als durch die Einführung einer neuen Frameart mit der neuen Slotkategorie 'condition' die Modellierung von Ablaufprozessen ermöglicht wurde.

In der folgenden Ebene wird das Grundmodell der Modellierung chemischer Prozesse definiert: Ein Prozeß besteht aus mittels 'Connections' verbundener 'Devices', die wiederum in Devices und Connections dekomponiert werden können. Um eine bessere Wiederverwendbarkeit zu erreichen, werden Devices mit 'Interfaces' zum Anschluß der Connections versehen[1].

[1] Die zusätzlich noch vorhandenen 'Implementations' (zusätzliche Klassen zum einfachen Umschalten verschiedener Realisierungen eines Devices) und 'Couplings' (Relationen aus zwei Interfaces und einer Connection) werden hier aus Vereinfachungsgründen nicht berücksichtigt.

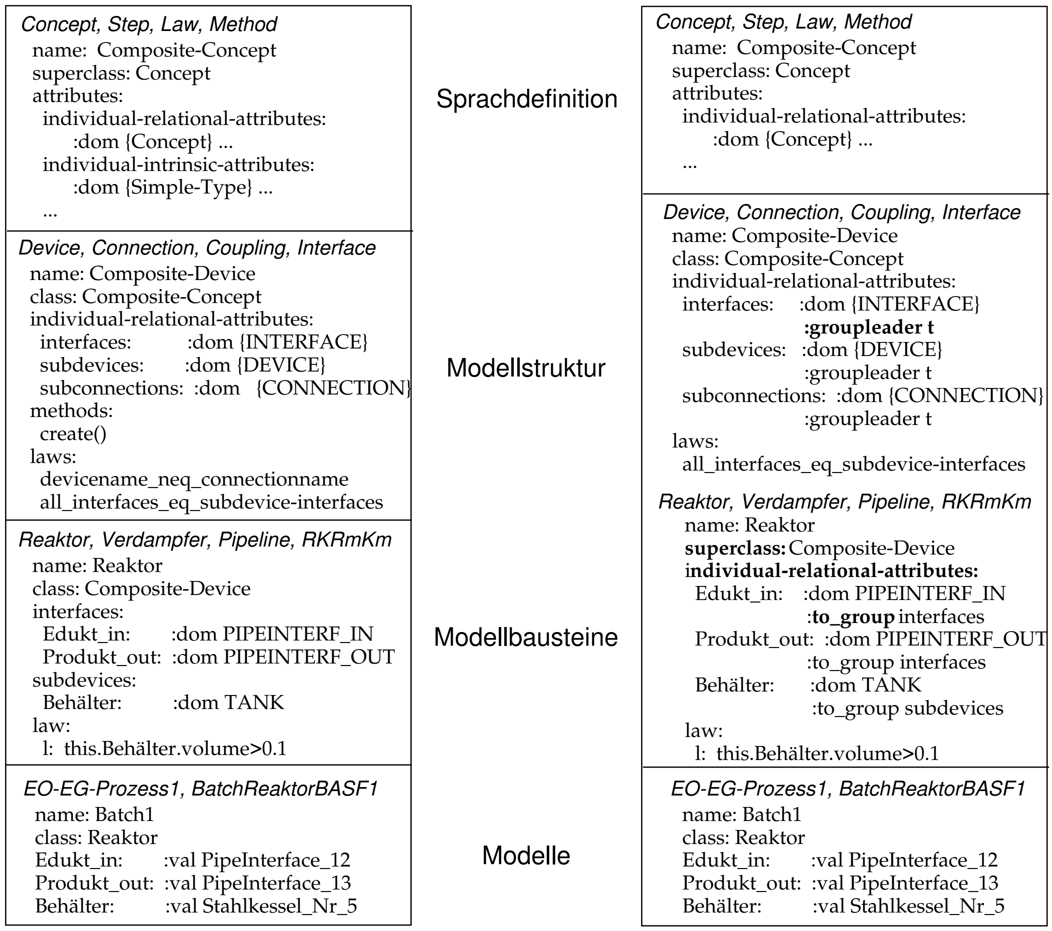


**Abbildung 3.** a Metamodell von VeDa; b) Metamodellsimulation mit Attributgruppierung

Die zweitunterste Ebene enthält die für den Modellierer relevanten Modellierungsbausteine, also z.B. verschiedene Reaktor-, Kolonnen-, Pumpen- oder Rohrtypen. Die zwischen ihnen bestehende Spezialisierungshierarchie soll einem Modellierer die Auswahl einer seinen Anforderungen am besten entsprechenden Komponente erleichtern, aber auch durch ihn erweiterbar sein[2].

In der untersten Ebene befinden sich schließlich die aus den Bausteinen zusammengesetzten Prozeßmodelle, die nach Fertigstellung in Simulatorcode umgewandelt werden.

Obwohl der Einsatz von Metaklassen/Metawissen zur Erhöhung der Ausdrucksfähigkeit von Modellierungssprachen weit verbreitet ist [14, 11, 5] (so ähnelt obige Strukturierung weitgehend der in [10] definierten), fanden sie in den bisherigen Sprachprototypen in der chemischen Prozeßmodellierung kaum Anwendung. So beschränkt sich Omola [2] auf die Klassenebene und verbirgt Metawissen in (folgenlosen) Kom-

[2]Im Gegensatz zu blockorientierten Modellierungstools, die nur vordefinierte Bausteine zur Verfügung stellen (vgl. [18]).

mentaren zur Strukturierung von Attributen wie 'terminal' oder 'submodels' sowie in den Parserroutinen. Auch Model.La [27] enthält seine Meta'klassen' (GU-Unit, Stream, ...) verborgen in der Sprachdefinition und nutzt sie nur wenig aus. Allerdings kommt Model.La durch die Vielzahl vordefinierter semantischer Beziehungen zwischen Objekten der Metamodellierung am nächsten.

Im Gegensatz zu den vorgehenden Systemen verwendet [20] ausschließlich ein Metamodell (bestehend aus 'Model', 'Variable', 'Parameter' und einigen weiteren Klassen) mit der Begründung, daß eine Strukturierung der Modellbausteine mittels Spezialisierungshierarchie den Benutzer eher verwirrt. In ähnlicher Weise instanziiert ModKit [3] zur Zeit direkt die Modellstrukturebene. Eine zukünftige Implementierung soll aber um die Modellbausteinebene erweitert werden.

Es ist ersichtlich, daß der Verzicht auf eine der mittleren Ebenen deutliche Einschränkungen der Modellierungsmächtigkeit zur Folge hat. So führt der **Verzicht auf die Modellstrukturebene** zum Verlust der Information über die grundlegende Modellierungsmethodik. Im Beispiel ließe sich etwa nicht mehr feststellen, welche Semantik das 'Behälter'–Attribut trägt, d.h. ob es sich um einen Verweis auf ein Subdevice handelt, oder das Attribut rein zufällig den Typ 'Device' hat. Da diese Informationen jedoch zur Interpretation des Modells durch Applikationen (etwa des Gleichungsaggregierers) unerläßlich ist, wird auf diese Metaebene in keinem der oben aufgeführten Modelle tatsächlich verzichtet. Dort, wo diese Ebene nicht vorhanden ist, wird versucht, die entsprechende Information durch Sprachkonstruktoren (etwa 'AT' in Omola) zu erfassen.

Modellierung **ohne die Modellbausteinebene** (d.h. eine direkte Instanziierung der Grundmodellebene) verhindert die Einordnung von Bausteinen in eine Hierarchie sowie die Benennung unterschiedlicher Attribute der gleichen Gruppe.

Durch die Metamodellierung werden somit folgende **Modellierungsmöglichkeiten** erlaubt:

1. Zusammenfassung von Klassen (in Metaklassen)
2. Gruppierung von Attributen (durch Meta-Attribute)
3. Definition von Instanzmethoden und -constraints (durch Metaformeln)
4. Definition von Klassenattributen, -methoden und -constraints (in den Metaklassen)

Der in vielen der oben aufgeführten Modellierungssprachen zum Ausdruck kommende Verzicht auf diese Ausdrucksmächtigkeit kann nur verstanden werden, wenn die nachfolgend aufgeführten **Probleme der Metamodellierung** berücksichtigt werden:

1. Mangel an Datenbanken, die Metamodellierung beherrschen
2. Unvereinbarkeit von Metamodellierung mit 'Standard'programmiersprachen wie Eiffel oder C++ (vgl. dazu auch Kapitel 4)

3. Ebenenüberspringende Attributdefinitionen (z.B. 'OID' einer objektorientierten Entity oder in VEDA das Attribut 'occurs_in' von Device)[3]

sowie im Falle VEDAs zusätzlich:

4. Einordnungsprobleme einiger Klassen in Ebene zwei oder drei (beispielsweise 'Abstract Phase')

Um mit VEDA dennoch Metaklassen nutzen zu können, wurde eine Alternative gesucht, die möglichst viele Probleme löst und dabei möglichst viele Fähigkeiten behält. Diese soll im nächsten Abschnitt beschrieben werden.

## 2.2. Alternative Repräsentationsformen für Metamodelle

Werden z.B. anhand Bild 4a die an der Metamodellierung beteiligten Objekte und Relationen betrachtet, so fällt auf, daß für einen Großteil der oben aufgeführten Probleme die doppelte Instanziierungsbeziehung zwischen Instanz und Klasse und zwischen Klasse und Metaklasse verantwortlich ist. Sie läßt sich mit den weitverbreiteten zweistufigen Objektmodellen nicht vereinbaren und verursacht Problem Nummer 3, da sie nicht transitiv ist. In den folgenden Lösungsvorschlägen werden die Instanziierungsbeziehungen darum teilweise durch Spezialisierungs- und andere Links ersetzt. Um eine Vergleichsgrundlage zu haben, wird zuerst allerdings die 'normale' Metamodellierung formalisiert.

### 2.2.1. Metaklassen

Formal kann die Verwendung der Instanziierung zur Gewinnung von Klassen aus Metaklassen mit Hilfe der folgenden Axiome notiert werden[4], falls das zugrundeliegende Datenmodell Attribute als 'First Class Objects' behandelt:

Sei $\mathcal{K}$ die Menge aller Klassen, Metaklassen, Metametaklassen, ... , $\mathcal{A}$ die Menge aller Attribute und $\mathcal{I}$ die Menge aller Identifikatoren. Ferner bedeute $c \xrightarrow{a} d$, daß das Attribut $a$ von $c$ auf $d$ verweist. Name $\subset (\mathcal{K} \cup \mathcal{A}) \times \mathcal{I}$ sei die Relation zwischen Klasse/Attribut und zugehörigem Namen. Schließlich sei Attr $\subset \mathcal{K} \times \mathcal{I} \times \mathcal{K}$ die Relation der zwischen zwei Klassen vorhandenen Attribut*namen*:

**A 1** $\forall\, c, d, mc, md \in \mathcal{K}, \forall\, a, ma \in \mathcal{A} : c \xrightarrow{a} d,\ mc \xrightarrow{ma} md,\ a \textit{ Instanz von } ma$
$\Rightarrow c \textit{ Instanz von } mc,\ d \textit{ Instanz von } md$

(D.h. wenn das Attribut $a$ Instanz des Meta-Attributs $ma$ ist, muß zwischen den Ausgangs- und Zielklassen ebenfalls eine Instanziierungsbeziehung bestehen.)

**A 2** $\forall\, c, d, mc, md \in \mathcal{K}, \forall\, a, ma \in \mathcal{A}$ :
$c \xrightarrow{a} d,\ mc \xrightarrow{ma} md,\ a \textit{ Instanz von } ma,\ Name(ma, l) \Rightarrow Attr(c, l, d)$

[3] in [14] als Instanz–Instanz–Typ bezeichnet

[4] angelehnt an [12], allerdings in anderer Notation

(D.h. wenn $a$ Instanz von $ma$ ist, so ist die Beziehung zwischen Ausgangs- und Zielklasse von $a$ zusätzlich unter dem Namen von $ma$ bekannt.)

Axiom 2 sichert die oben angeführte Eigenschaft 2, während Axiom 1 die Möglichkeiten zur Gruppierung einschränkt. Die Zusammenfassung von Klassen erfolgt durch die Instanziierung ausgehend von einer gemeinsamen Metaklasse. Klassenattribute, -methoden und -constraints werden mittels aus der Klasse–Instanz–Beziehung bekannte Verfahren definiert.

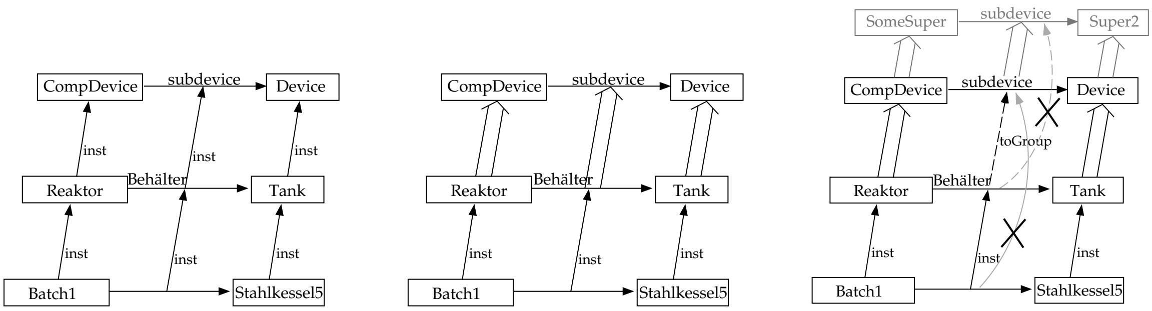


**Abbildung 4.** Metamodellierung und ihre Emulation: a) 'echte' Metamodellierung; b) Multiple Attributspezialisierung; c) Attributgruppierung

### 2.2.2. Multiple Attributspezialisierung

Ersetzt man die Instanziierungsbeziehungen zwischen Klassen und Metaklassen bzw. zwischen Attributen und Meta-Attributen durch Spezialisierungsbeziehungen,so erhält man eine Nachbildung der Metamodellierung durch multiple Attributspezialisierung (i.e. Spezialisierungsbeziehungen zwischen Attributen in Sub- und Superklasse unabhängig von ihrer Benennung) [5]. Formal läßt sich diese folgendermaßen definieren:

Sei $\mathcal{K}$ die Menge aller Klassen, $\mathcal{A}$ die Menge aller Attribute, $\mathcal{O}$ die Menge aller Instanzen und $\mathcal{I}$ die Menge aller Identifikatoren. Ferner bedeute $c \xrightarrow{a} d$, daß das Attribut $a$ von $c$ auf $d$ verweist, und Name $\subset (\mathcal{K} \cup \mathcal{A}) \times \mathcal{I}$ sei die Relation zwischen Klasse/Attribut und zugehörigem Namen. Schließlich sei Attr $\subset \mathcal{K} \times \mathcal{I} \times \mathcal{K}$ die Relation der zwischen zwei Klassen vorhandenen Attribut*namen*:

**A 1** $\forall\, c, c', d, d' \in \mathcal{K}, \forall\, a, a' \in \mathcal{A} : c \xrightarrow{a} d,\ c' \xrightarrow{a'} d',\ a\ isA\ a'$
$\Rightarrow c\ isA\ c',\ d\ isA\ d'$
(D.h. Ausgangs- und Zielklasse müssen analog zum Attribut spezialisiert werden.)

**A 2** $\forall\, c, c' \in \mathcal{K} \cup \mathcal{A}, \forall\, o \in \mathcal{O} : c\ isA\ c', o\ Instanz\ von\ c \Rightarrow o\ Instanz\ von\ c'$
(D.h. Instanzen einer Subklasse sind automatisch Instanzen der Oberklassen; dieses Axiom

[5] Die hier vorgeschlagene Spezialisierung von Attributen ist zur Erhaltung der Objektsubstituierbarkeit in streng getypten, objektorientierten Sprachen nicht erlaubt

wird selbstverständlich auch in den anderen Axiomsystemen benötigt, damit sie die objektorientierten Mechanismen vollständig beschreiben, hat dort jedoch nicht direkt etwas mit Metamodellierung zu tun.)

**A 3** $\forall\, c, c', d, d' \in \mathcal{K}, \forall\, a, a' \in \mathcal{A}, \forall\, l \in \mathcal{I}$ :

$$c \xrightarrow{a} d,\ c\ isA\ c', c' \xrightarrow{a'} d',\ Name(a,l),\ Name(a',l) \Rightarrow a\ isA\ a'$$

(D.h. Rückführung der natürlichen Attributspezialisierung auf die multiple Attributspezialisierung)

Wiederum gruppiert das zweite Axiom Attribute, indem aus ihm folgt, daß wenn Attr$(o, l, o')$, Name$(a, l)$, Name$(a', l')$ und $a$ isA $a'$ gilt, Attr$(o, l', o')$ folgt. Die Anwendungsmöglichkeiten dieser Gruppierung werden auch hier wieder durch das erste Axiom eingeschränkt.

Allerdings läßt diese Alternative Beziehungen zu, die mit der Metaklassen–Klassen–Instanzen Kombination nicht erlaubt sind (die durchgestrichenen Pfeile in Bild 4c)

- Eine Klasse, die in der Spezialisierungshierarchie zwischen $c$ und $c'$ liegt, spezialisiert das Attribut $a'$. Doch statt von dieser Spezialisierung leitet sich $a$ direkt von $a'$ her. Formal:
  $\exists c'' : c$ isA $c''$ isA $c'$, $c'' \xrightarrow{a''} d''$, Name$(a'', l)$, Name$(a', l), a$ isA $a'$.
- 'Virtuelle' Meta-Attribute (also solche Attribute, die mehrfach spezialisiert werden) können instanziiert werden, während dies für 'echte' Meta-Attribute nicht gilt.

Zudem kann in Systemen ohne 'first class' Attribute die Gleichbehandlung von natürlicher und multipler Attributspezialisierung Probleme bereiten, da erstere dort nur über Namensgleichheiten definiert wird, während für letztere zusätzliche Sprachkonstrukte (z.B. Attributfacetten) benötigt werden.

### 2.2.3. Attributgruppierung

Um eine alternative Metamodellrepräsentation zu erhalten, die der Instanziierung näher kommt als obiges Vorgehen, werden eine neue Beziehung und eine neue Menge eingeführt. Dies erlaubt es, Axiome zu formulieren, die die Mängel der multiplen Spezialisierung ausschließen, da sie nicht gleichzeitig Spezialisierung und Metamodell'simulation' beschreiben müssen.

Da außerdem eines unserer Ziele ist, mit Standard OO–Datenbanken und –Programmiersprachen kompatibel zu sein, diese aber Attribute in der Regel nicht als 'first class objects' modellieren, geht das folgende Axiomsystem von nur im Zusammenhang mit Klassen/Instanzen existierenden Attributen aus.

Die nun definierte "Attributgruppierung"sbeziehung trägt den Namen 'toGroup' und benötigt eine Menge von "Gruppenvorsitzenden"-Attributen namens 'groupleader'.

Sei $\mathcal{K}$ die Menge aller Klassen, $\mathcal{O}$ die Menge aller Instanzen und $\mathcal{I}$ die Menge aller Identifikatoren. Ferner bezeichne $c.a$ das direkte (d.h. nicht vererbte) Attribut namens $a$ der Klasse $c$, und $\mathrm{Attr} \subset (\mathcal{K} \cup \mathcal{O}) \times \mathcal{I} \times (\mathcal{K} \cup \mathcal{O})$ sei die Relation der einer Klasse/einem Objekt zugeordneten Attribute mit ihrer Zielklasse/ihrem Wert (bei Klassen handelt es sich wiederum nur um direkte Attribute[6]):

**A 1** $\forall\, c, c', d, d' \in \mathcal{K}, \forall\, a, a' \in \mathcal{I}:$
$$\mathit{Attr}(c,a,d),\ \mathit{Attr}(c',a',d'),\ c.a\ \mathit{toGroup}\ c'.a'$$
$$\Rightarrow c\ \mathit{isA}\ c',\ d\ \mathit{isA}\ d',\ c'.a' \in \mathit{groupleader},\ \neg(\exists c'' \in \mathcal{K} : c\ \mathit{isA}\ c''\ \mathit{isA}\ c' : c''.a')$$
(D.h. wenn es eine Gruppierungsbeziehung zwischen zwei Attributen gibt, müssen die 'Ausgangs–' und 'Ziel'klassen des gruppierten Attributs Spezialisierungen derjenigen des Gruppenvorsitzenden sein. Desweiteren muß das Attribut, auf das gruppiert werden soll, tatsächlich als Gruppenvorsitzender ausgezeichnet sein. Schließlich darf es entlang des Spezialisierungspfades der 'Ausgangs'klassen keine Klasse geben, die das Gruppenvorsitzendenattribut spezialisiert hat.)

**A 2** $\forall\, c, c' \in \mathcal{K}, \forall\, a \in \mathcal{I}, \forall\, o, p \in \mathcal{O}:$
$$c.a\ \mathit{toGroup}\ c'.a',\ o\ \mathit{Instanz\ von}\ c,\ \mathit{Attr}(o,a,p) \Rightarrow \mathit{Attr}(o,a',p)$$
(D.h. in Instanzen sind gruppierte Attribute auch unter dem Namen ihres Gruppenvorsitzenden bekannt.)

**A 3** $\forall\, c \in \mathcal{K}, \forall\, a \in \mathcal{I}:$
$$c'.a' \in \mathit{groupleader} \Rightarrow \neg\exists\, o \in \mathcal{O} : o\ \mathit{Instanz\ von}\ c',\ o.a'$$
(D.h. groupleader-Attribute können in Instanzen keine Werte annehmen.)

Wie schon in den vorherigen Axiomsystemausschnitten, sorgt auch hier Axiom 2 für die Gruppierung der Attribute, also dafür, daß gruppierte Attribute auch unter dem Namen des Gruppenvorsitzenden sichtbar sind und Constraints/Regeln des Gruppenvorsitzenden auch auf sie wirken. Axiom 1 schränkt die erlaubten Gruppierungen ein und vermeidet dabei das erste Problem der multiplen Attributspezialisierung, während Axiom 3 den zweiten Fehler der multiplen Spezialisierung beseitigt.

### 2.2.4. Auswirkungen der alternativen Repräsentationen

Der Ersatz der Instanziierungsbeziehung durch die Spezialisierung (2.2.2) erlaubt es, die Modellierungsmöglichkeiten (vgl. S. 6) der Metamodellierung weitgehend (bis auf Punkt vier) zu erhalten, ohne Metaklassen zu benötigen. Allerdings schränkt die Spezialisierung diese Modellierungsmöglichkeiten zu wenig ein. Durch den Übergang auf eine neue Beziehung (2.2.3) mit einer Semantik, die nicht wie bei der Spezialisierung extern vorgegeben ist, sondern selbst definiert werden konnte, können die beschreibenden Axiome so gewählt werden, daß sie weitestgehend die gleichen Effekte wie

[6] Die ausschließliche Betrachtung direkter Attribute ist ausreichend, da die Axiome 2 und 3 aus dem letzten Abschnitt in entsprechend angepaßter Form auch hier gelten.

eine Modellierung mittels Metaklassen haben. Allerdings können auch mittels der Attributgruppierung Klassenattribute, -methoden und -constraints (also Punkt vier auf S. 6) nicht dargestellt werden. Dies bedeutet insbesondere, daß Attributgruppierung nicht zur Modellierung von sich selbst (also der Sprachdefinitionsebene in Bild 3) verwendet werden kann.

Allerdings erlaubt es die Beschränkung auf zweistufige Objektsysteme ohne 'first class attributes' es, die Attributgruppierung auf einfache Art in OO–Programmiersprachen wie C++ (und damit auch in den meisten OO–Datenbanken) zu implementieren. Abbildung 5 zeigt eine Übersetzung (die auch automatisch erfolgen kann) der Klasse CompDevice aus Bild 3b in C++. Eine direkte Übertragung von Metaklassen in C++

```
class CompDevice: public Device
{
    virtual set<Interface> getinterfaces() { return(empty_set); }
    virtual set<Device> getsubdevices() { return(empty_set); }
    ...
}

class Reaktor : public CompDevice
{   Pipeline_In *Edukt_In;
    Pipeline_Out *Produkt_Out;

    virtual set<Interface> getinterfaces()
        { return(union(CompDevice::getinterfaces(), Edukt_In, Produkt_Out)); }
    ...
}
```

**Abbildung 5.** Attributgruppierung in C++

würde hingegen erfordern, die Metaklassen als Klassen darzustellen sowie Klassen und Instanzen als Instanzen der Klassen- bzw. der Instanzenklasse zu repräsentieren. Typprüfungen, Vererbung u.ä. müßte dafür selbst implementiert werden. Im Gegensatz dazu ermöglicht die hier vorgestellte Lösung die weitgehende Weiterverwendung der in der Programmiersprache vorhandenen Mechanismen.

# 3. Prototyp- vs. Klassenbasierte Objektorientierung

## 3.1. Warum Prototyping?

Die im letzten Kapitel beschriebene Attributgruppierung erleichtert es, Wissen über Struktur und Verhalten von Modellen chemischer Prozesse zu repräsentieren. Dies geschieht in Form einer Klassenhierarchie mit Spezialisierungsbeziehungen, Methoden und Konsistenzbedingungen. Modelle für konkrete Prozesse finden sich in der

Instanzebene. Einige Autoren (z.B. [17, 7, 26, 30]) erachten diese Zweiteilung als ungeeignet für Entwurfsanwendungen, weil sie den Entwurf neuer Objekte zu stark beeinträchtigt. Sie schlagen statt dessen eine klassenlose, einstufige Objektorientierung vor, das 'Prototyping'.

Bei prototypbasierter Objektorientierung [30, 28] befinden sich Typ- und Wertinformationen im selben Objekt, so daß Klassen nicht benötigt werden. Beim Editieren eines Objekts kann also neben den Werten gleichzeitig das Schema des Objekts geändert werden. Dies erlaubt dem Benutzer, ausgehend von einem vorhandenem Objekt ein von ihm benötigtes Neues zu erstellen, ohne auf von Klassen vorgegebene Attribute/-typen Rücksicht nehmen zu müssen. Dieses bei Entwurfsprozessen verbreitete Vorgehen nennt sich „Copy & Modify".

Ein Vergleich der für einen Modellierer zur Erstellung eines neuen Modellelements prinzipiell möglichen Vorgehensweisen ergibt folgendes Bild:

1. *Erzeuge ausgehend von der Metaklasse das benötigte Objekt mit zugehöriger Klasse.*

   Dies bedeutet natürlich, daß pro Instanz eine eigene Klasse generiert wird, die jeweils direkte Instanz der Metaklasse ist. Der Benutzer kann hierdurch zwar alle benötigten Modellteile genau nach seinen Vorstellungen entwerfen, verliert aber weitgehend die Möglichkeit der Wiederverwendung von Bausteinen und Wissen (vgl. allerdings die ModKit–TWR–Kopplung auf S. 13).

2. *Verfeinere eine möglichst ähnliche Klasse bis sie den Anforderungen entspricht und erzeuge dann eine Instanz von ihr.*

   Durch dieses Vorgehen wird zwar die Hierarchie der Modellbausteine bestmöglich weiterentwickelt, die Erzeugung passender Klassen erfordert allerdings vom Benutzer eine im Vorhinein zu erbringende Abstraktionsleistung.

3. *Kopiere ein ähnliches Objekt und modifiziere es bis das gewünschte Objekt erreicht ist.*

   Dies entspricht genau dem oben erwähnten „Copy&Modify". Für die explorative Erzeugung verschiedener Modellvarianten eignet sich diese Vorgehen gut. Es ergeben sich allerdings Probleme aus dem fehlenden Bezug zu einem abstraktem Konzept (also etwa einer Klasse). Konsistenzbedingungen, Methoden und Typprüfungen lassen sich nur schwer verwirklichen, neue Objekte nur generieren, wenn ein passendes ähnliches gefunden wurde, und die notwendigen Speicherstrukturen sind weniger effizient als beim Ansatz 2.

Das Ziel muß es also sein, dem Benutzer einerseits eine möglichst einfache Erzeugung neuer Objekte bzw. Änderung alter Objekte (dies insbesondere im Hinblick auf den Modellierungs–Simulations–Zyklus, s. S. 2) zu erlauben und andererseits die Vorteile von Klassen gegenüber einstufigen Objektsystemen (also Klassifikation, Templates, speichereffiziente Schemaverwaltung) zu erhalten.

## 3.2. Mischung von prototyp- und klassenbasierter OO in der Literatur

Obiges Ziel zu erreichen, gab es seit der Definition des Prototyping und verwandter Mechanismen wie Delegation verschiedene Ansätze. Während im „Treaty von Orlando“ [28] noch alle Verfahren als gleichwertig aber getrennt angesehen wurden, gab es nachfolgend Kombinationsversuche. **Self** [30], eine prototyporientierte Sprache, erhöht die Speichereffizienz durch Vererbungsbeziehungen von Methoden, Attributtypen und -werten zwischen Objekten und sogenannten 'trait' Objekten . **BOS** [22], ebenfalls prototypbasiert mit 'trait' Objekten, erhält gewisse Klassifikationsmöglichkeiten dadurch, daß die Vererbungsbeziehung zwischen Objekten zur Typisierung ausgenutzt wird: Die Objekte, von denen es Attribute/-werte und Methoden erbt, stellen den 'Typ' eines Objekts dar. Jedes Attribut kann als eine Eigenschaft einen solchen 'Typ' angeben, wodurch seine möglichen Werte beschränkt werden. Einerseits erlaubt dieser Mechanismus eine gewisse Klassifizierung von Objekten, andererseits muß der Benutzer, damit diese Klassifizierung funktioniert, sich bei jedem neu definierten Attribut Gedanken über dessen Typanforderungen machen bzw. sich bei Änderungen ggf. neue 'trait' Objekte suchen.

**epee** [7] benutzt Templates zum einen zur Erzeugung von — von da an prototypischen — Objekten, zum anderen zur Klassifikation dieser Objekte gemäß der Übereinstimmung ihrer Attributnamen mit denen der Templates. Letzteres ist nur möglich, da in epee der Name eines Attributs dessen Typ eindeutig definiert. Zudem behandelt es mehrdeutige, überlappende Zuordnungen nicht. Einen ähnlichen Weg, allerdings eher von der Klassenseite her kommend, beschreitet [6] mit seinen „**Predicate Classes**“. Hierbei wird die Mitgliedschaft in Klassen nicht nur durch Instanzierungs- und Spezialisierungsbeziehungen definiert sondern auch durch 'Predicate Classes' zugeordnete logische Ausdrücke über den Objekten. Einen Schritt weiter als die Predicate Classes geht **COCOON** [29] durch eine vollständige Trennung von Typen (also Templates) und Klassen (lediglich zur Klassifikation). Werddn in diesem Ansatz die aus Typen erzeugten Objekte durch prototypische Objekte ersetzt, so ergibt sich ein einstufiges Objektsystem mit automatischer Klassifikation aufgrund logischer Regeln. Ein solches System bietet zwar sowohl prototypische Objekte als auch Klassifikation, benötigt jedoch die ständige Auswertung einer großen Zahl deduktiver Regeln, die sich aufgrund der fehlenden Klassen nur schwer in Trigger umsetzen ließen.

Einen etwas anderen Ansatz verfolgt darum die geplante Integration von **ModKit** [3] und des von Ulrike Sattler (vgl. Bericht in diesem Band) entwickelten **TWR**[7]–Systems: ModKit erzeugt pro Objekt eine eigene Klasse, die in einer flachen Struktur angeordnet sind. Diese Klasseninformationen sollen vom TWR–System verarbeitet werden, das aus ihnen eine Spezialisierungshierarchie berechnet und diese an ModKit zurückgibt. Diese Hierarchie soll dann im Rahmen der Wiederverwendung den Benutzer in der Wahl eines seinem Modellierungsziel ähnlichen Ausgangsobjekts unterstützen. Allerdings soll sie nicht zur tatsächlichen Anordnung der Klassen in einer

[7]TWR = **T**erminologische **W**issens**R**epräsentation

Spezialisierungshierarchie verwendet werden. Hierdurch und durch die Erzeugung einer eigenen Klasse pro Objekt kann allerdings die Skalierbarkeit dieses Verfahrens für große Objektmengen beeinträchtigt werden. Zudem ist z.Z. noch unklar, inwieweit die zur Verfügung stehende Information für eine automatische Generierung einer Spezialisierungshierarchie ausreicht.

**KSL** [9] sowie **VODAK** [14] erweitern das normale klassenbasierte Konzept um optionale Typergänzungen für jedes Objekt (in KSL „traps", in VODAK „own–type" genannt). Diese erlauben es, das Schema eines Objekts unabhängig von seiner Klasse zu ändern, kommen also den Möglichkeiten der prototypischen Objektmanipulationen nah. Allerdings erlauben 'own–types' in VODAK keine Änderungen bestehender Attributtypen, ersetzen also nur Unterklassen, ohne zusätzliche Modellierungsmächtigkeit zu bieten. Die 'traps' von KSL hingegen ermöglichen beliebige Unterschiede zwischen einer Instanz und zugehöriger Klasse. Dies kommt, im Gegensatz zu VODAK, den Möglichkeiten der Objektmanipulationen in klassenlosen Systemen gleich, macht aber andererseits die Instanziierungsbeziehung zwischen Instanz und Klasse aus Klassifizierungsgesichtspunkten nutzlos.

**Sciore** verwendet in [26] drei Hierarchien, um sowohl prototypische als auch klassenbasierte Objektmodellierung zu unterstützen: Objekthierarchien, die jedes Objekt aus verschiedenen Rollen zusammensetzen; eine Klassenhierarchie, die gewisse Hierarchiestrukturen in den Objekthierarchien erzwingt; eine Templatehierarchie, in der 'typische' Objekte zum Kopieren für jede in der Klassenhierarchie vorhandene Klasse zur Verfügung gestellt werden. Auf diese Weise erreicht Sciore gleichzeitig Flexibilität, Objekttemplates und eine gewisse Klassifizierbarkeit, also drei von vier gewünschten Eigenschaften. Allerdings können Rollen vorhandene Attribute nicht verändern, bzw. die Auswirkungen widersprüchlicher Definitionen in verschiedenen Rollen sind nicht definiert. Zudem muß der Benutzer, will er Objekte tatsächlich per Copy&Modify verändern, wissen, welche Attribute in welcher Rolle definiert sind, um diese Rolle per Hand aus der Objekthierarchie zu entfernen bzw. zu ersetzen.

### 3.3. „Eigenschaften" für VeDa

Da sich offensichtlich prototypische Flexibilität und klassenbasierte Klassifizierbarkeit ausschließen, basiert der für VeDa entwickelte Ansatz für ein flexibles Modellierungssystem ähnlich wie [26] und [8] auf einem um Rollen erweiterten klassenbasiertem Vorgehen. Es beruht auf der Idee, **prototypische Objektmanipulationen** in einem klassenbasierten System **zu emulieren**. Dies läßt sich erreichen, indem bei jeder Objektänderung aus dem bestehenden Klassensystem hinaus eine neue Unterklasse der größten gemeinsamen Oberklasse erzeugt wird, die die geänderten/neuen Attribute enthält (vgl. Bild 6).

Selbstverständlich birgt dieses Vorgehen die Gefahr einer exponentiellen Klassenexplosion, so daß anstelle normaler Unterklassen Rollen die Funktion zu- und ab'schalt'-

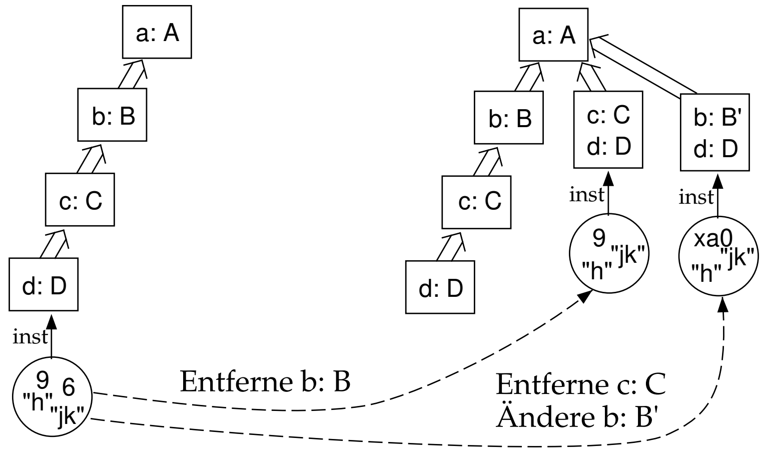


**Abbildung 6.** Emulation von Copy&Modify durch automatische Erzeugung neuer Subklassen

barer Schemabestandteile übernehmen. Rollen tauchten bereits in einigen der in 3.2 beschriebenen Ansätzen auf und erlauben es, Objekte aus Teilobjekten zusammenzusetzen und während ihrer Existenz zu modifizieren, indem Objekte als gleichzeitige Instanz mehrerer Klassen (multiple Instanziierung) behandelt werden. Somit kann die exponentiell große Zahl möglicher Kombinationen von der Klassenebene auf die Instanzenebene verlagert werden, auf der sie dann durch die Umstellung der Instanziierungsbeziehung auf Mengenwertigkeit gelöst wird, indem die Kombination der Instanziierungslinks die Kombination von Klassen zu neuen ablöst.

Zu den von [26], [14] und [8] vorgeschlagenen Verfahren ergeben sich mehrere Unterschiede:

- *Die Attribute/Methoden aller Rollen eines Objekts sind gleichzeitig sichtbar und beeinflussen sich gegenseitig.*

  In den anderen Ansätzen sind Attribute/Methoden nur dann sichtbar, wenn das Objekt explizit in der entsprechenden Rolle betrachtet wird. Im Beispiel der Rollen Angestellter und Student von Person, können die Angestellten- und die Studentenrolle (bei [14] und [8] zusätzlich noch Person) unterschiedliche Attribute 'Tel.Nr.' haben, die die Buero-, Uni- (und Privat-)Nummer angeben und je nach betrachteter Rolle einer Instanz sichtbar sind.

  Ein Anwendungsbeispiel für Rollen in VeDa stellen die Gleichungen mit Rollen Skalar-, Vektor-, Matrix-, -system, algebraisch, integrell, differentiell, partiell differentiell, ... dar. Es ist einsichtig, daß beim Betrachten einer Gleichung gleichzeitig die Attribute-/Attributänderungen aller Rollen sichtbar sein müssen, denen sie angehört.

- *Es gibt keine scharfe Trennung zwischen Rollen und Klassen.*

  [14] und [8] trennen sehr scharf zwischen Klassen und Rollen. Letztere können nur im Zusammenhang mit Klassen bzw. Klasseninstanzen existieren und nur zu

ihnen kann ein Objekt seine Instanziierungsbeziehungen während seiner Existenz ändern. Die Trennung geht teilweise so weit, daß an jeder Klasse eine eigene Rollenhierarchie hängt.

Zwar machen auch in dem für VeDa projektierten Ansatz die meisten Rollen für sich allein instanziiert keinen Sinn, allerdings muß sowohl die Zuordnung von Rollen zu Klassen als auch der Wechsel von Instanziierungsbeziehungen zu Klassen flexibler realisiert werden. So soll z.B. ausgehend von einem 'Reaktor_123' mit Rolle 'mit Kühlmantel' sowohl übergegangen werden können zu 'Reaktor_123' & 'wärmeisoliert' (Rollenwechsel) als auch zu 'Flash_123' & 'mit Kühlmantel' (Klassenwechsel).

Dies bedeutet, daß das vorgestellte Konzept in diesem Zusammenhang der multiplen Instanziierung näher steht als dem Rollenkonzept.

- *Rollenkombinationseinschränkungen durch deduktive Regeln.*

  [8] begrenzen die möglichen Kombinationen von Rollen dadurch, daß sie an jede Klasse die zugehörige Rollenhierarchie anhängen. [26] verwendet seine 'Klassen'hierarchie (in der allerdings nicht alle 'Klassen' enthalten sein müssen) sowie 'Cluster' (= disjunkte, vollständige Spezialisierung). Beide Vorgehensweisen erlauben die Kombinationen von Rollen nur in einer geringen Bandbreite einzuschränken.

  Beim Verfahren von Li und Dong [15] werden mögliche Kombinationen dadurch begrenzt, daß ein Objekt ausgehend von den Klassen, denen es angehört, nur diejenigen (zusätzliche) Klassen annehmen darf, zu denen eine Migrationserlaubnis besteht. Die hierdurch ermöglichte feinere Manipulation der gleichzeitig erlaubten Rollen wird im Verfahren von Odberg [23] noch dadurch erweitert, daß die Übergangserlaubnis von den jeweiligen Attributwerten abhängig gemacht werden kann.

  Die von [23] verwendeten booleschen Ausdrücke zur Beschränkung möglicher Rollenkombinationen werden auch im für VeDa vorgesehenen Ansatz verwendet (zu möglichen Einschränkungen siehe S. 17). Dieses Vorgehen harmoniert besser mit der im letzten Punkt erwähnten geringeren Trennung von Klassen und Rollen.

- *Jedes Objekt kann jede Rolle nur einmal annehmen.*

  Die 'qualified roles' in [8] erlauben es einem Objekt gewisse Rollen mehrfach anzunehmen (beispielweise Projektmanager für verschiedene Projekte). In der vorgesehenen Verwendung der Rollen in der chemischen Prozeßmodellierung (siehe etwa obiges Gleichungsbeispiel) macht dies wenig Sinn und ist darum nicht vorgesehen.

- *Unterstützung der ungeplanten Verwendung von Rollen.*

  Ähnlich wie [28] zwischen geplanter und ungeplanter gemeinsamer Informationsverwendung ('anticipated and unanticipated sharing') unterscheiden, können auch

Rollen geplant und ungeplant angewendet werden. Die geplante Verwendung entspricht der in allen betrachteten Ansätzen vorkommenden Definition von Rollen und Kombinationsrestriktionen im Zusammenhang mit der allgemeinen Schemadefinition. Verwendet man Rollen allerdings zur Emulation prototypischer Objektmanipulationsmöglichkeiten (vgl. S. 14), so werden neue Rollen zur Laufzeit erzeugt. Diese „ungeplanten" Rollen müssen gegenüber vorhandenen abgegrenzt, in etwa vorhandene Kombinationsbeschränkungen eingebaut, sinnvoll benannt und in etwaige Hierarchien eingeordnet werden.

Eine geringe Unterstützung dafür bietet lediglich [26], indem dynamisch erzeugte Objektspezialisierungen nachträglich in die 'Klassen'- und Templatehierarchie eingeordnet werden können und so, obwohl unabhängig generiert, den normalen Verwaltungsmechanismen unterstellt werden können.

Da eines der wesentlichen Ziele des Objektmodells für VeDa gerade die Emulation von „Copy&Modify" ist, muß die ungeplante Rollenverwendung wesentlich besser unterstützt werden.

Aufgrund insbesondere der ersten beiden Unterschiede zwischen dem hier vorgestellten Konzept und den auf Rollen basierenden Verfahren, wird im weiteren von „Eigenschaften" statt von Rollen die Rede sein. Entsprechend der sprachlichen Bedeutung von „Eigenschaft" sind alle 'Eigenschaften' eines Objekts gleichzeitig sichtbar, jede kommt maximal einmal vor, und ein Objekt setzt sich vollständig aus Eigenschaften zusammen.

Das für VeDa vorgesehene Verfahren nutzt diese 'Eigenschaften' zum einen geplant, um bisher mit multipler Vererbung und Objektverweisen modellierte Objekte zusammenzusetzen (vgl. das Gleichungsbeispiel auf S. 15). Zum anderen benutzt es sie ungeplant, um dem Benutzer die Möglichkeit zu geben, Objekte per Copy&Modify zu erzeugen. Die Auswirkungen der ungeplanten Verwendung müssen noch untersucht werden.

## 3.4. Anwendungserfahrung

Wie erwartet, zeigte eine Anwendung der Modellierungsmethodik auf ein Beispiel [4] vielfältige Möglichkeiten, Eigenschaften vorteilhaft einzusetzen. Es mußte allerdings auch festgestellt werden, daß bei einer verstärkten Verwendung von Eigenschaften das Schema an Übersichtlichkeit einbüßt, da nicht intuitiv ersichtlich ist, wann Eigenschaften und wann Subklassen benutzt werden sollen. Um hier Klarheit zu schaffen, sind Regeln zur Beschreibung sinnvoller Anwendungskontexte von Eigenschaften notwendig.

Zusammen mit den oben erwähnten Problemen müssen also Regeln für die folgenden drei Bereiche aufgestellt werden:

**Kombinationseinschränkungen** Lassen sich allgemeine, wiederkehrende Einschränkungen der möglichen Eigenschaftskombinationen feststellen und gesondert (vereinfacht) behandeln?

**geplante Eigenschaftsanwendung** In welchen Situationen soll ein Wissensingenieur Eigenschaften verwenden, wenn er Modellbausteinklassen in VeDa entwirft?

**ungeplante Eigenschaftsanwendung** Wie kann das System automatisch Eigenschaften zur Emulation des Prototypings einsetzen, ohne daß das entstehende Schema unübersichtlich oder gar sinnlos wird?

Eine wenig zufriedenstellende Beantwortung dieser Fragen ergibt sich, wenn man die anderen auf Rollen basierenden Vorgehens zur Vereinigung von prototyp- und klassenbasierter Objektorientierung untersucht. Beispielhaft soll dies hier am Ansatz von Sciore [26] erfolgen:

**Kombinationseinschränkungen** Abhängigkeit einer Rolle von der Existenz einer anderen im gleichen Objekt wird (als einziges aber einfach) gehandhabt.

**geplante Eigenschaftsanwendung** Verständlicherweise keine Antwort möglich, da der Artikel kein spezielles Anwendungsgebiet behandelt.

**ungeplante Eigenschaftsanwendung** Prototyping'emulation' per Hand durch den Benutzer. Zur Erhöhung der Übersichtlichkeit können neu erzeugte Rollen in die Klassen- und Templatehierarchie aufgenommen werden.

Andere Verfahren zeigen leichte Gewichtungsunterschiede zwischen erster und dritter Fragestellung, sind aber ähnlich unbefriedigend.

Für das für VeDa vorgesehene Konzept steht eine Beantwortung dieser Fragen (sowie eine Formalisierung der Antworten) noch aus.

Das für VeDa vorgesehene Objektmodell beruht also auf einem zweistufigem System mit Klassen und Instanzen, das mit Hilfe der Attributgruppierung 'virtuelle' Metaklassen nutzbar macht. Durch die Hinzunahme von Eigenschaften (Unterschiede zu Rollen in Kapitel 3.3) soll sowohl die Modellierung vorgegebener Modellbausteine vereinfacht werden, als auch die Verwendung einer „Copy&Modify"–artigen Vorgehensweise ermöglicht werden. Die genaue Methodik für den letzten Punkt muß allerdings noch erarbeitet werden.

## 4. Datenbank–Applikations–Kopplung

Das in den letzten beiden Kapiteln beschriebene Objektmodell eignet sich zwar gut zur Repräsentation von Wissen und (Zwischen)Ergebnissen kreativer Entwicklungsprozesse, führt aber zu Schwierigkeiten bei der Integration einer entsprechenden Datenbank mit den darauf zugreifenden Applikationen. Während nämlich in der Applikationsprogrammierung im allgemeinen von einem konstanten Schema ausgegangen wird, müssen in diesem Anwendungsgebiet Applikationen das sich regelmäßig ändernde Schema verarbeiten können und dem Benutzer zugänglich machen. Der Grund hierfür liegt in der Repräsentation der durch den Benutzer änder- und ergänzbarer Modellbausteinen als Klassen.

Erfolgversprechend ist eine integrierte Architektur wie sie Atkinson und Morrison in [1] beschreiben: Die Datenbank und die Programmiersprache sind soweit integriert, daß die Persistenz von Objekten vollständig unabhängig von den verwendeten Klassen oder dem Programmcode ist. In einem solchen System würde durch die Modifikation des Datenbankschemas die Applikationen automatisch mitangepaßt, so daß neue oder geänderte Klassen dem Programm/dem Benutzer augenblicklich zur Verfügung ständen.

Leider existiert ein derartiges System nicht, mögliche Ansätze müssen zumindest folgende Anforderungen erfüllen:

- Interpretative Auswertung
- Dynamisch änderbare Klassen in Datenbank und Programmiersprache
- Nachladen neuer Klassen und Methoden bei Bedarf
- Programmiersprache = Datenbanksprache

Smalltalk, CLOS oder Tcl mit Objekterweiterung entsprechen diesen Anforderungen mehr oder weniger (sofern ein angepaßtes DBMS Verwendung findet), während compiler–basierte Sprachen wie C++ oder Eiffel durch ihre zur Laufzeit statische Klassenstruktur für direkte Implementationen (vgl. aber unten) ausscheiden. Eine Mischform wie Java könnte hohe Performance mit dynamischen Schemaänderungen verknüpfen, allerdings beherrscht der Klassenlader der Java Virtual Machine Schemaänderungen (noch?) nicht.

Läßt sich eine solche enge Kopplung zwischen Datenbank- und Programmiersprachendatenmodell nicht verwirklichen, muß mit Leistungseinbußen gerechnet werden. Zum einen ergibt sich, selbst falls beide Modelle objektorientiert sind, ein 'Impedance Mismatch' (vgl. [13, S.100ff]) zwischen Datenbank und Applikation, zum anderen muß auf Applikationsseite das Datenmodell der Datenbank simuliert werden, was u.U. Typchecks, effiziente Zugriffsmechanismen oder andere spezielle Vorteile der Programmiersprache außer Kraft setzt. Eine mögliche Abhilfe besteht darin, das in den Applikationen sichtbare Datenmodell der Programmiersprache anzupassen und alle nur schwer darstellbaren Bestandteile (z.B. Konsistenzbedingungen, Methoden, Rollen) lediglich in der Datenbank zu handhaben. Auf Datenmodellebene verringert sich dadurch die Kopplung zwischen Anwendung und Datenbank weiter und läßt damit sprachbezogene Optimierungen zu. Als Ausgleich dafür wächst allerdings die Kopplung bei Änderungsoperationen, da nun lediglich die Datenbank über alle notwendigen Informationen verfügt, um Zulässigkeit und Auswirkungen der Änderungen zu überprüfen. Andererseits streben Verfahren der computerbasierten Kooperation sowieso eine schnelle Meldung von Änderngen an die Datenbank an, um Konflikte, Versionierungsnotwendigkeiten oder Notifikationen frühzeitig erkennen zu können.

Die vorausgegangenen Überlegungen können in der Implementierung des VeDa–Datenmodells jedoch nur begrenzt Anwendung finden, da die für VeDa zu entwickelnde Datenbank in die auf PRO–Art 2.0 [24] aufsetzende Entwurfsumgebung für chemische Prozesse integriert werden soll. PRO–Art 2.0 ist jedoch in C++ programmiert und verwendet eine relationale Datenbank als Persistenzgrundlage. Hierdurch treten die Probleme der Abbildung eines dynamischen Schemas auf C++ (verwirklicht mittels Klassen „Device–Klasse", „Device–Instanz" usw. für alle Modellstrukturklassen) und relationale Datenbanken in den Vordergrund. Der erwogene Ansatz sieht zudem die Speicherung von Schemainformationen in der Metadatenbank *ConceptBase* vor, um sie einfach verarbeiten und in ein für die relationale Datenbank verständliches Format konvertieren zu können. Die Leistungsfähigkeit eines solchen Systems mit drei verschiedenen Repräsentationen desselben Schemas muß sich allerdings noch erweisen.

Durch die Integration in eine Modellierungsumgebung sind allerdings zahlreiche Beispieldaten zu erwarten, die eine Verifikation der Ergebnisse der beiden vorangegangen Kapitel ermöglichen.

# 5. Zusammenfassung und Ausblick

Im vorliegenden Text wurden verschiedene Probleme vorgestellt, die sich beim Entwurf eines Objektmodells für die Modellierungsprache VeDa zur Modellierung chemischer Prozesse ergeben. Für sie wurden unterschiedlich ausgearbeitete Lösungsvorschläge unterbreitet.

Die Strukturierung der Modellinformationen in drei Ebenen (Modellstruktur-, Modellbaustein-, Modell-) konnte mit Hilfe der „Attributgruppierung" auf zwei Ebenen abgebildet werden, was eine Implementierung mit den meisten OO–Programmiersprachen erlaubt. Die Attributgruppierung ersetzt die bei der Metamodellierung vorkommende Instanzbeziehung zwischen Attributen und Meta-Attributen durch eine Beziehung mit selbstdefinierter Semantik und verläßt sich für die Beziehungen zwischen den Klassen und 'ehemaligen' Metaklassen auf die Spezialisierung.

Die klassenbasierten Verfahren zur Erzeugung neuer Objekte werden durch das prototypbasierte Copy&Modify sinnvoll ergänzt. Eine Vereinigung beider Ansätze durch Emulation der Modifikationsoperationen in einem klassenbasierten System mit Rollen (genauer 'Eigenschaften') wurde beschrieben und noch offene Probleme aufgezeigt.

Durch das zur Laufzeit dynamische Schema ergeben sich Probleme bei der Kopplung einer VeDa repräsentierenden Datenbank und den Anwendungsprogrammen. Diese Probleme und mögliche Lösungsalternativen wurden skizziert. Die Anwendung der vorgestellten Architekturalternativen wird jedoch durch die in großen Teilen vorgegebene PRO–Art Architektur beschränkt. Die sich ergebenden Implementierungspläne wurden kurz dargelegt.

Die im Kapitel 3.4 unbeantwortet gelassenen Fragen sowie die Implementierungsprobleme müssen im weiteren gelöst werden. Unklar ist zudem noch die Handhabung widersprüchlicher Attributdefinitionen durch verschiedene Rollen. Die sich mit der explorativen Modellierung im Modellierungs–Simulations–Zyklus ergebende Forderung nach einer Versionierung der Ergebnisse wird im Rahmen einer Diplomarbeit zur Schema-/Klassenversionierung bearbeitet. Durch die nun beginnende Implementierung können die in den Kapiteln 2 und 3 vorgestellten Konzepte zudem auf ihre Tragfähigkeit in einer realen Anwendung überprüft werden.

## Literaturverzeichnis


1. Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.
2. Mats Andersson. *Object-Oriented Modeling and Simulation of Hybrid Systems.* PhD thesis, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, December 1994.
3. Ralf Bogusch, Willi Geffers, Bernd Lohmann und Wolfgang Marquardt. Rechnergestützte Modellierung mit dem Modellierungswerkzeug ModKit. In *Modellierung komplexer Systeme in der Verfahrenstechnik*, Aachen, Februar 1996. VW–Schwerpunkt–Kolloquium.
4. Markus Baumeister und Bernd Lohmann. Modellierung des EO–EG–Prozesses mittels Attributgruppierung und Eigenschaften in VeDa. Interner Forschungsbericht, Lehrstuhl für Prozeßtechnik; RWTH-Aachen, 1996. in Fortentwicklung.
5. Sjaak Brinkkemper. *Formalisation of Information Systems Modelling.* PhD thesis, Nijmegen, 1990.
6. Craig Chambers. Predicate classes. In Oscar M. Nierstrasz, editor, *European Conference on Object–Oriented Programming (ECOOP)*, LNCS 707. Springer, 1993.
7. The épée Group. The épée programming and reference manual; épée 0.2 release. Technical Report TR-94-18, Department of chemical engineering, University of Edinburgh, September 1994.
8. Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object–oriented systems with roles. Institutsbericht 94.03, Institut für Wirtschaftsinformatik, Johannes–Kepler–Universität Linz, 1994.
9. Mamdouh H. Ibrahim, William E. Nejcek, and Fred A. Cummings. Instance specialization without delegation. *Journal of Object–Oriented Programming (JOOP)*, pages 53–56, June 1991.
10. ISO/IEC International Standard 10027. *Information technology — Information Resource Dictionary System (IRDS) framework*, first edition, 1990.
11. Matthias Jarke, Stefan Eherer, Rainer Gallersdörfer, Manfred A. Jeusfeld, and Martin Staudt. ConceptBase – a deductive object base manager. Aachener Informatik–Berichte 93-13, Fachgruppe Informatik, RWTH Aachen, 52056 Aachen, Germany, October 1993.
12. Manfred A. Jeusfeld. *Änderungskontrolle in deduktiven Objektbanken.* Infix–Verlag, St. Augustin, Deutschland, 1992.
13. Alfons Kemper and Guido Moerkotte. *Object–Oriented Database Management; Applications in Engineering and Computer Science.* Prentice–Hall International, London, 1994.
14. Wolfgang Klas and Michael Schrefl. *Metaclasses and Their Application.* LNCS 943. Springer, Berlin, Heidelberg, 1995.

15. Qing Li and Guohzu Dong. A framework for object migration in object–oriented databases. *Data & Knowledge Engineering*, 13:221–242, 1994.
16. Bernd Lohmann. Strukturierte Modellierung am Beispiel des Ethylenoxid–Ethylenglykol–Prozesses. Interner Forschungsbericht, Lehrstuhl für Prozeßtechnik; RWTH-Aachen, Februar 1995.
17. S. Levy, E. Subrahmanian, S. Konda, R. Coyne, A. Westerberg, and Y. Reich. An overview of the n–dim environment. Research report EDRC-05-65-93, Engineering Design Research Center, Carnegie Mellon University, 1993.
18. W. Marquardt. Trends in computer–aided process modeling. In *Proceedings of the Conference on Process Systems Engineering*, 1994.
19. W. Marquardt, A. Gerstlauer, and E.D. Gilles. Modeling and representation of complex objects: A chemical engineering perpective. In *Proceedings of the Conference on Intelligent Engineering Applications/Artificial Intelligence in Engineering*, Edinburgh, June 1993.
20. Claudio Maffezzoni, Roberto Girelli, and Petrika Lluka. Object–oriented database support for modeling and simulation. In *Proceedings of the European Simulation and Modeling Conference*, Barcelona, June 1994.
21. W. Marquardt, A. Gerstlauer, S. Räumschüssel und B. Raichle. Objekt–orientierte Datenmodellierung für die Repräsentation von verfahrenstechnischen Prozeßmodellen. Interner Forschungsbericht, Universität Stuttgart, 1992.
22. The n-dim Group. 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, April 1995.
23. Erik Odberg. Category classes: Flexible classification and evolution in object–oriented databases. In Gerhard Wijers, Sjaak Brinkkemper, and Tony Wasserman, editors, *Proceedings of the 6th Conference on Advanced Information Systems Engineering (CAISE)*, LNCS 811. Springer, 1994.
24. Klaus Pohl, Ralf Doemges, and Matthias Jarke. PRO–Art: PROcess based Approach to Requirements Traceability. NATURE Report Series 94–07 Appendix I, Lehrstuhl für Informatik V, RWTH–Aachen, 1994.
25. S. Räumschüssel, A. Gerstlauer, E.D. Gilles, B. Raichle, M. Zeitz, and W. Marquardt. An architecture of a knowledge–based process modeling and simulation tool. In *International Symposium on Mathematical and Intelligent Models in System Simulation*, Brussels, April 1993.
26. Edward Sciore. Object specialization. *ACM Transactions on Information Systems*, 7(2):103–122, April 1989.
27. G. Stephanopoulos, G. Henning, and H. Leone. Model.la. a modelling language for process engineering — I. the formal framework. *Comput. chem. Engng*, 14(8):813–846, 1990.
28. L.A. Stein, H. Liebermann, and D. Ungar. A shared view of sharing: The treaty of Orlando. In Won Kim and F.H. Lochovsky, editors, *Object–Oriented Concepts, Databases and Applications*, pages 31–48. Addison–Wesley, Reading, MA, 1989.
29. Markus Tresch. *Evolution in Objekt–Datenbanken*. Teubner–Texte zur Informatik, Bd.10. B.G.Teubner, Leipzig, 1995.
30. D. Ungar and R.B. Smith. Self: the power of simplicity. *LISP and Symbolic Computation*, 4(3):187–205, 1991.
31. VeDa-Gruppe. Veda: Objekt–orientierte Datenmodellierung für die Repräsentation von verfahrenstechnischen Prozeßmodellen. Interner Forschungsbericht, Lehrstuhl für Prozeßtechnik; RWTH Aachen, 1996. in Arbeit.
32. Kai von Luck. KL-One: Eine Einführung. IWBS Report 106, IBM Wissenschaftliches Zentrum; Institut für Wissensbasierte Systeme, Februar 1990.