Heft 99

H. Heß

Gestaltungsrichtlinien zur objekt-orientierten Modellierung

Dezember 1992

GESTALTUNGSRICHTLINIEN ZUR OBJEKTORIENTIERTEN MODELLIERUNG

Inhaltsverzeichnis

1 EINORDNUNG DES OBJEKTORIENTIERTEN PARADIGMAS IN DIE ARIS-ARCHITEKTUR

Die von Scheer anhand der Anforderungen an betriebliche Informationssysteme abgeleitete ARIS-Architektur kann als grundlegender Ansatz zur Systembeschreibung verstanden werden. Um Komplexität und Redundanz zu reduzieren, wird die Beschreibung eines Systems in unterschiedliche Sichten (Daten-, Funktions- und Organisationssicht) zerlegt, deren Verbindungen in einer Steuerungssicht wieder eingeführt werden. Diese Perspektiven werden jeweils zusätzlich nach der Nähe zur Informationstechnik in die Beschreibungsebenen Fachkonzept, DV-Konzept und Implementierung unterteilt (vgl. Scheer 1992).
Im letzten Jahrzehnt haben objektorientierte Ansätze eine enorme Bedeutung für die Entwicklung von Informationssystemen gewonnen. Die Verbindung von Daten und Funktionen (wie in der Steuerungssicht vorgesehen) ist auch Grundgedanke des objektorientierten Paradigmas (vgl. Abb. 1).[1]

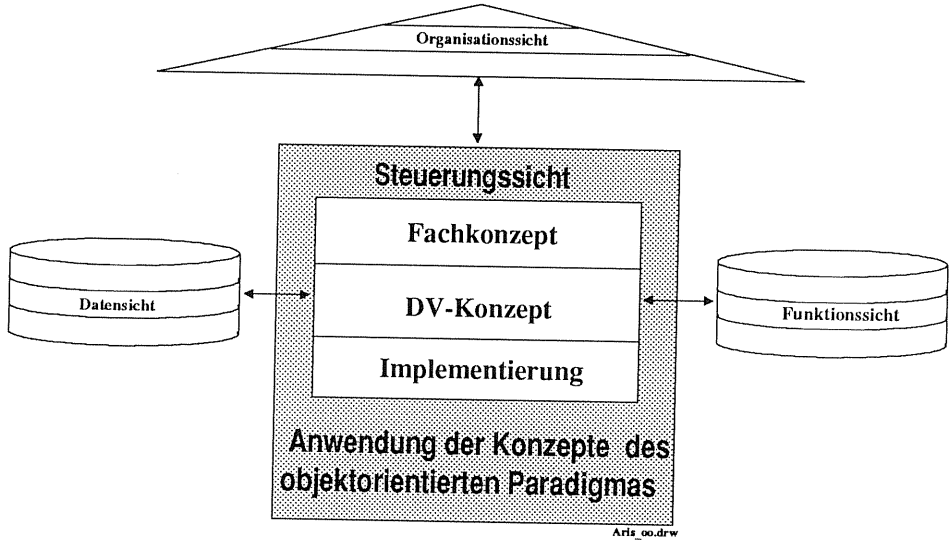

Abb. 1: Anwendung objektorientierter Konzepte innerhalb der Steuerungssicht der ARIS-Architektur

Im praktischen Einsatz bleibt das objektorientierte Paradigma augenblicklich noch sehr häufig auf die Verwendung objektorientierter Programmiersprachen beschränkt. Wie schon bei der Strukturierten Programmierung kann das Phänomen beobachtet werden, daß sich ein

---

[1] Die Grundkonzepte des objektorientierten Paradigmas können an dieser Stelle nicht vorgestellt werden; hier wird auf die umfangreiche Grundlagenliteratur verwiesen (vgl. Meyer 1990; Schlüter et al. 1990; Heß 1989).

Paradigma beginnend bei der Implementierung erst allmählich auf die vorgelagerten Phasen des DV- und Fachkonzeptes ausbreitet. Im Gegensatz dazu findet in der wissenschaftlichen Literatur eine engagierte Diskussion über die Einsatzmöglichkeiten objektorientierter Methoden in den frühen Phasen der Softwareentwicklung statt. Hier werden Fragen aufgeworfen, wie z.B.:

- Welche objektorientierten Analyse- und Designmethoden sind am geeignetsten?[2]
- Wie ist der Übergang von objektorientierter Analyse (Fachkonzept) zu objektorientiertem Design (DV-Konzept) zu bewältigen? Wo liegen die Unterschiede?
- Wie sind objektorientierte und strukturierte Methoden miteinander zu vereinbaren?[3]

Der Beitrag objektorientierter Analyse- und Designmethoden kann in zweierlei Hinsicht betrachtet werden: Zum einen werden hierbei Notationen angeboten, mit deren Hilfe die erarbeiteten Ergebnisse möglichst eindeutig und verständlich dargestellt werden sollen. Zum anderen umfassen Methoden i.a. auch Vorgehensmodelle zur Strukturierung der durchzuführenden Aufgaben und geben (meist in Form von Heuristiken) Hinweise für die einzelnen Teilaspekte der Analyse und des Designs.

Es stellt sich die Frage, in welchen Beschreibungsebenen objektorientierte Methoden sinnvollen Einsatz finden. Beschränkt sich eine Modellierung nur auf die Erstellung eines Fachkonzeptes, so hängt es im wesentlichen von den Vorlieben der damit beauftragten Experten des darzustellenden Bereichs (aber auch von den Charakteristika dieser Domäne) ab, ob wirklich objektorientierte Methoden die geeigneten Hilfsmittel sind, um zu einer für alle Beteiligten verständlichen, noch DV-unabhängigen Formalisierung der Problemstellung zu gelangen. In vielen Fällen werden traditionelle Methoden, die auf funktionaler Zerlegung beruhen oder etwa Input-Output-Beziehungen darstellen, eher vertraut erscheinen, somit effizienter nutzbar sein und den Vorrang erhalten.
Dient das resultierende Fachkonzept jedoch auch als Basis der Entwicklung eines DV-Konzepts und anschließender Implementierung, so eröffnen die Basiskonstrukte des objektorientierten Paradigmas eine Reihe von Vorteilen, die den Einsatz objektorientierter

---

2 Für einen Vergleich objektorientierter Analyse- und Designmethoden siehe Heß, Scheer 1992a; Monarchi, Puhr 1992.

3 Henderson-Sellers, Constantine diskutieren die mögliche Verbindung strukturierter und objektorientierter Techniken in unterschiedlichen Phasen der Entwicklung und zeigen sinnvolle Lifecycle-Modelle unter gegenseitiger Ergänzung der Paradigmen auf (vgl. Henderson-Sellers, Constantine 1991; siehe hierzu auch Li 1991; de Champeaux et al. 1990). Raasch konstatiert die Nachteile strukturierter Methoden gegenüber einem objektorientiertem Ansatz, macht jedoch auch Vorschläge zum Einsatz strukturierter Verfahren, die eine Migration hin zu objektorientierten Methoden offenlassen (vgl. Raasch 1991, 407-410).

Methoden auf diesen Abstraktionsebenen nahelegen: Hier sind insbesondere die leichte Änderbarkeit (insbes. Erweiterbarkeit), die Wiederverwendbarkeit sowie die Modularität der entwickelten Software zu nennen. In solchen Fällen kann es - trotz des damit i.a. verbundenen Lernaufwandes - Sinn machen, auch für die Erstellung des Fachkonzeptes objektorientierte Methoden einzusetzen, um einen fließenden Übergang über alle Phasen zu erreichen. Vielfach wird auch argumentiert, daß so eine strukturerhaltende, einfachere Abbildung der Realität ermöglicht wird. Die grundsätzliche Voraussetzung liegt natürlich in dem Einverständnis auch der an der Erstellung des Fachkonzeptes beteiligten Anwender, da auf dieser Ebene eine Sprache gefunden werden muß, mit deren Hilfe alle Betroffenen über die Anforderungen an ein zu entwickelndes System auf der fachlichen Ebene kommunizieren können.

In der Literatur wird die Grenze zwischen objektorientiertem Fach- und DV-Konzept nicht eindeutig gezogen. Analog zur generellen Differenzierung der Beschreibungsphasen nach Scheer (vgl. Scheer 1992, 16) sollten jedoch auch die bei der objektorientierten Erstellung eines Fachkonzeptes identifizierten Objekte und Klassen eindeutig der Problemdomäne zuzuordnen sein; erst in den nachfolgenden Phasen werden diese Ergebnisse um Strukturen zur DV-technischen Umsetzung erweitert. Es kann somit der Abgrenzung von Monarchi, Puhr gefolgt werden:

> "*OO Analysis* (OOA) models the problem domain by identifying and specifying a set of semantic objects that interact and behave according to system requirements. *OO Design* (OOD) models the solution domain, which includes the semantic classes (with possible additions) and interface, application, and base/utility classes identified during the design process." (Monarchi, Puhr 1992, 38)

Abb. 2 verdeutlicht diesen Zusammenhang zwischen den Phasen objektorientierter Modellierung und den dabei relevanten Aspekten eines Softwaresystems.

Um die genannten Vorteile objektorientierter Entwicklung, insbesondere die Wiederverwendbarkeit von Komponenten, Wirklichkeit werden zu lassen, ist es notwendig, der Qualität der erstellten Software besondere Beachtung zu schenken. Objektorientierte Entwicklung allein garantiert noch nicht die optimale Wiederverwendbarkeit der Komponenten. Neben einer geeigneten Form der Darstellung und Verwaltung (vgl. Heß, Scheer 1992b) wird es unabdingbar, bei der Entwicklung von Komponenten Gestaltungsrichtlinien anzuwenden, die den Weg hin zu qualitativ hochwertigen, wiederverwendbaren Komponenten weisen. Eine Übersicht über diese Richtlinien ist Zielsetzung der nachfolgenden Darstellung.

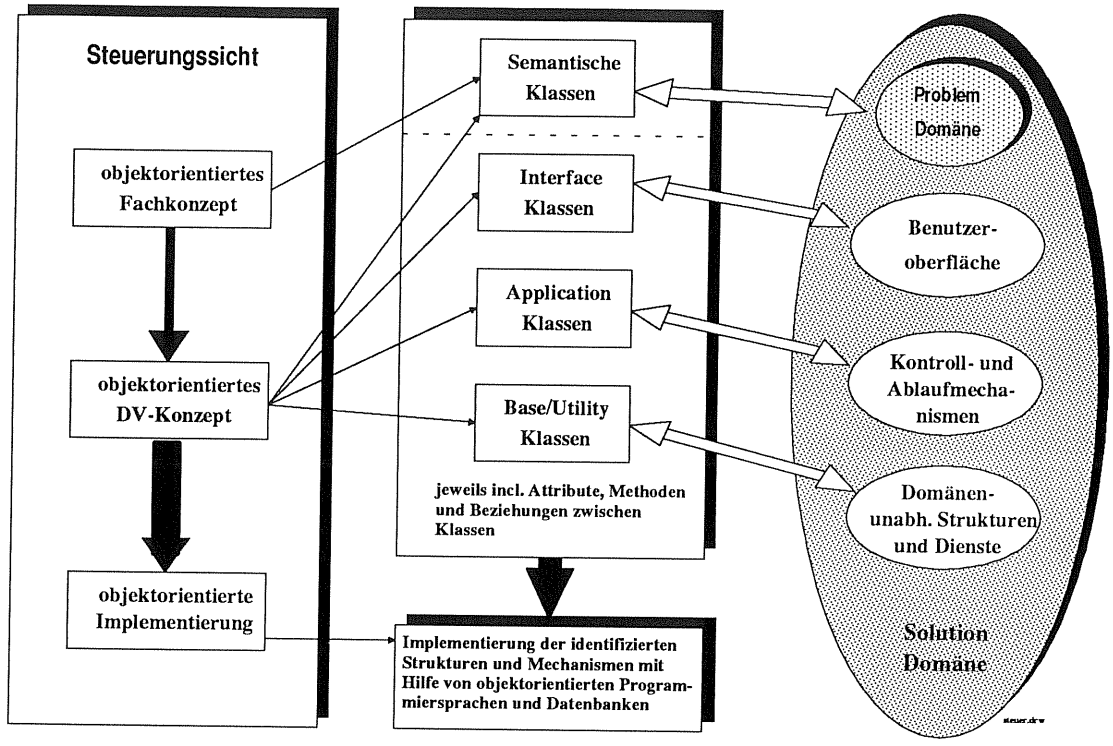

Abb. 2: Zusammenhang zwischen den Phasen objektorientierter Modellierung und den verschiedenen Aspekten eines Softwaresystems

Die folgenden Gestaltungsrichtlinien beziehen sich in erster Linie auf die Entwicklung des objektorientierten DV-Konzepts. Da ein wesentliches Kennzeichen objektorientierter Entwicklung die Verringerung des konzeptionellen Abstandes zwischen den genannten Beschreibungsebenen ist, haben sie allerdings auch direkte Aussagekraft für die Implementierung, wobei jedoch implementierungsspezifische Richtlinien, die i.a. programmiersprachenabhängig sind, nicht behandelt werden. Werden auch für die Entwicklung des Fachkonzeptes - z.B. um eine Homogenität der Beschreibung zu erzielen - objektorientierte Methoden eingesetzt, so sind viele der vorgestellten Heuristiken auch für diese Abstraktionsebene äußerst hilfreich.

2 RICHTLINIEN ZUR OBJEKTORIENTIERTEN MODELLIERUNG

Im folgenden soll zwischen den verschiedenen Teilaspekten objektorientierter Entwicklung unterschieden werden, so daß als wesentliche Schwerpunkte die Definition von Klassen, die Definition des Protokolls, der Aufbau der Klassenhierarchie sowie die Beziehungen zwischen Klassen behandelt werden.

Alle nachfolgend vorgestellten Richtlinien führen nicht zu einer unfehlbaren Methode zur Softwareentwicklung, sondern versuchen, gemachte Erfahrungen und allgemein anerkannte Ergebnisse in Form von Heuristiken darzustellen. Um die Güte eines Designs unabhängig von individuellen Vorlieben zu beurteilen, schlägt Coad vor, die Qualität eines Entwurfs an den Kosten eines Systems zu messen, wobei allerdings keine einfache operative Anwendung dieses Kriteriums möglich ist:

> "A good design is one that balances trade-offs to minimize the total cost of the system over its entire lifetime... one of the most important characteristics of a good design is that it leads to an easily maintained implementation." (Coad 1991, 68)

Ein Großteil der folgenden Gestaltungsrichtlinien hat zum Ziel, die Wiederverwendbarkeit der Software zu steigern. Um dieses Kriterium einer objektiveren Beurteilung zugänglich zu machen, wurden in den letzten Jahren eine Reihe von Maßgrößen definiert, deren Auswertung hohe Aussagekraft bei der Bewertung dieses Aspektes besitzen.[4] Da jedoch sehr viele Nebenbedingungen, wie z.B. die Retrievalmöglichkeiten und die Unterstützung der Wiederverwendung durch das Projektmanagement, die Realisierung der Wiederverwendung beeinflussen, sollten die nachfolgenden Richtlinien als eine notwendige Voraussetzung, nicht aber als hinreichend zur Erreichung dieses Ziels angesehen werden:

> "The amount of reuse of a particular class is also related to external properties such as 'usefulness' of a class in particular application domains... Thus, the quantity of reuse is not a direct function of the internal properties or internal 'reusability' of a class. Because of the numerous external factors, 'reusability' is not an attribute of a software document." (Bieman 1991, 9)

---

[4] Der Mehrzahl der definierten Maßgrößen zur Messung der Wiederverwendung liegt der Ansatz zugrunde, die Größe des wiederverwendeten Codes mit der Größe des neu entwickelten Codes zu vergleichen, wobei Modifikationen existierender Software i.a. nicht berücksichtigt werden (vgl. Boehm 1981; Conte, Dunsmore, Shen 1986; Fenton 1991). Bieman definiert dagegen eine Vielzahl von Maßgrößen, die sich speziell auf objektorientierte Software beziehen und auch Änderungsmöglichkeiten (Leveraged Reuse) umfassen; hierbei wird noch einmal in die Sichten der benutzten und der benutzenden Komponenten und in die Gesamtsicht unterschieden (vgl. Bieman 1991).

2.1 Definition von Klassen

Klassen mit ihren zugehörigen Objekten sind die Grundbausteine eines objektorientierten Systems, so daß ihre Definition zu den grundlegenden und wichtigsten Aufgaben des Entwurfs und der sich anschließenden Implementierung gehört. Obwohl objektorientierte Entwicklung Sichtweisen der Daten- und Funktionsmodellierung miteinander kombiniert, muß eine eigene Vorgehensweise zur Definition der Klassen etabliert werden.
Die generellen Unterschiede - insbesondere im Vergleich zur Funktionsmodellierung - beim Auffinden der Objekte versucht Meyer in dem grundlegenden Motto,

> "Frag nicht zuerst, was das System tut: Frag, WORAN es etwas tut !" (Meyer 1990a, 54),

zum Ausdruck zu bringen, womit auch eine strikte Bottom-Up-Vorgehensweise nahegelegt wird, die es möglichst lange unterläßt, die Funktionen auf höchster Abstraktionsebene des Systems zu untersuchen und zu beschreiben. Auch wenn dieser generelle Ansatz in der zu sehr vereinfachenden Anleitung gipfelt,

> "In der physikalischen oder abstrakten Wirklichkeit sind die Objekte modelliert und warten darauf, aufgelesen zu werden! Die Softwareobjekte spiegeln diese externen Objekte einfach wider." (Meyer 1990a, 55),

wird damit doch ein großer Vorteil des objektorientierten Entwurfs im Vergleich zu traditionellen Designmethoden angesprochen - die große Schwierigkeit der herkömmlichen Entwicklung, die komplexe Realität auf EDV-geeignete Strukturen abzubilden, wird durch die nunmehr mögliche Wahl der interagierenden Objekte als direkte Entsprechungen der komplexen Objekte der Wirklichkeit abgelöst:[5]

> "Find the objects by anthropomorphic projection; i.e., use real-world objects rather than computer science objects whenever possible." (Beck et al. 1988, 373-374)

Da die Auswahl der Objekte und Klassen nicht ganz so einfach ist, wie von Meyer beschrieben, versuchen einige Autoren, mögliche Klassen von Kandidaten anzugeben, um dem Designer diese Aufgabe zu erleichtern.[6]
Shlaer, Mellor geben fünf Kategorien vor, deren Elemente als Startideen zum Finden von Objekten in neuen Problembereichen dienen können (vgl. Shlaer, Mellor 1988, 14-19):

---

5 Scharenberg, Dunsmore beobachten bei erfahrenen Entwicklern dagegen eine Evolution der Klassendefinitionen im Laufe eines Projektes dahingehend, daß die Klassen zunächst als nicht miteinander interagierende Entsprechungen von Objekten der Domäne aufgefaßt werden können, sich allmählich aber zu kommunizierenden, anthropomorphen Klassen wandeln, die nicht unbedingt eine reale Entsprechung haben (vgl. Scharenberg, Dunsmore 1991).

6 Die im folgenden betrachteten Kategorien gelten für semantische Klassen, die bei der Fachkonzepterstellung identifiziert werden (vgl. die Unterteilung bei Monarchi, Puhr 1992, 37) und sind unpassend für die Klassen, die im Rahmen des DV-Konzeptes hinzugenommen werden.

- Gegenstände,
- Rollen (die Personen annehmen können),
- Vorkommnisse (die einen Zeitbezug haben),
- Interaktionen (die Verbindungen zwischen zwei oder mehreren anderen Objekten herstellen) und
- Spezifikationen (die häufig als Standardbeschreibungen anderer Objekte eingesetzt werden).

Die Vorschläge anderer Autoren unterscheiden sich davon im wesentlichen in der Begriffswahl, die zugrundeliegenden Ideen sind identisch, so daß die Klassifikationen von Ross (er benennt Menschen, Plätze, Gegenstände, Organisationen, Konzepte und Ereignisse (vgl. Ross 1987)) sowie von Coad, Yourdon (Strukturen, Systeme, Vorrichtungen, Ereignisse, Rollen, Plätze, Prozeduren und Organisationseinheiten (vgl. Coad, Yourdon 1991, 60-66)) in ähnlicher Weise interpretiert und verwendet werden können.[7]

Die von Budde et al. vorgestellte Metapher von Werkzeug und Material ist der Versuch, bei der Objektfindung die Benutzersicht und die systemtechnische Sicht in Einklang zu bringen (vgl. Budde et al. 1992). Entsprechend der vom Benutzer gewöhnten Sichtweise von Arbeitsmitteln und -gegenständen werden interaktive Anwendungen modelliert, indem Werkzeug-, Material- und Aspektklassen gebildet werden:

> "Werkzeugklassen fokussieren auf die Handhabung und auf Tätigkeiten; Materialklassen auf die Eigenschaften (und Widerstände!) der Arbeitsgegenstände; Aspektklassen auf das Zusammenspiel von Werkzeugen und Materialien." (Budde et al. 1992, 239)

Insbesondere die Bildung der Aspektklassen führt i.a. zur Verwendung der Mehrfachvererbung, da es sinnvoll und notwendig erscheinen kann, eine Klasse unter mehreren Aspektoberklassen anzuordnen.

Eine weitere Möglichkeit, die Objektfindung zu unterstützen, besteht darin, eine natürlichsprachliche Beschreibung der Anwendung als Grundlage zu nehmen. Alle vorkommenden Substantive sind Kandidaten für die Objekte des zu modellierenden Systems, die verwendeten Verben sollten als Ausgangspunkt genommen werden, das Protokoll jeder Klasse zu bestimmen (vgl. Abbott 1983).

---

[7] Rosson, Gold berichten von einem interessanten Experiment, bei dem die Arbeitsweise von 6 Smalltalk-Programmierern in den frühen Entwurfsphasen untersucht wird (vgl. Rosson, Gold 1989); alle waren mit dem Entwurf eines EDV-Systems für einen Lebensmittelladen beauftragt. Die Beobachtung zeigt, daß der Designprozeß als interaktiver Vorgang zu verstehen ist, währenddessen die Designer den Objekt-Metaphor nutzen, um eine neue Version des Problems zu erstellen, in der die erkannten Objekte als anthropomorphe Gebilde miteinander kommunizieren, um die gewünschte Funktionalität zu erreichen.

Eine Klassendefinition in der beschriebenen Art und Weise führt vornehmlich zu anwenderorientierten Klassen, die dessen Sicht auf das System widerspiegeln. In der Designphase wird diese Auswahl ergänzt um Klassen, die einen stärkeren Bezug zur DV-technischen Realisierung des Systems aufweisen.

Eine mögliche Gefahr der skizzierten Methoden besteht darin, zunächst zu viele Klassen zu definieren, so daß es Sinn macht, sich umgekehrt zu überlegen, was denn nicht als Klasse definiert werden sollte. Generell sollte eine Klasse als nicht geeignet erkannt werden, wenn keine besonderen Operationen auf Instanzen der Klasse angewendet werden bzw. die Instanzen nicht durch eigene Eigenschaften charakterisiert werden können (vgl. Meyer 1990a, 353). Rumbaugh et al. differenzieren diese Ausschlußkriterien noch weiter und erkennen als nicht geeignete Klassen (vgl. Rumbaugh et al. 1991, 153-156):[8]

- redundante Klassen, d.h. Klassen, deren Information schon im System zur Verfügung steht,
- irrelevante Klassen, d.h. Klassen die keine sinntragende Bedeutung für die eigentliche Funktionalität beinhalten,
- unscharfe Klassen, d.h. Klassen deren Bedeutung nicht klar abgegrenzt ist,
- Attribute, d.h. Informationen, deren Komplexität so gering ist, daß sie besser als Eigenschaften von Objekten definiert werden,
- Operationen, d.h. Vorgänge, die nicht selbst wiederum durch Eigenschaften charakterisiert werden bzw. selbst Manipulationen erfahren,
- Klassen, die die Rollen von Objekten in Beziehungen, aber nicht die Natur der Objekte selbst widerspiegeln, sowie
- Klassen, die schon Konstrukte der Implementierung widerspiegeln, aber keine Bedeutung in der Realität haben.

Im Zusammenhang mit der Definition von Klassen stehen Überlegungen, wann eine Klasse zu komplex definiert wurde und möglichst aufgespalten werden sollte. Indizien für diesen Fall sind zum einen eine sehr hohe Anzahl von Methoden, da dies darauf hindeutet, daß mehrere Abstraktionen durch die Klasse repräsentiert werden. Johnson, Foote geben als Maßzahl 50-100 Methoden an, die nur in Ausnahmefällen bei einer einzigen Klasse definiert werden sollten (vgl. Johnson, Foote 1988, 29-30).
Ein ebensolches Indiz ist es, wenn die Gesamtzahl der Methoden eine Zweiteilung aufweist und jede Hälfte im wesentlichen auf eine zugeordnete Hälfte der Instanzvariablen zugreift.

---

[8] Ähnliche Ausschlußkriterien finden sich auch bei Shlaer, Mellor, die vier Arten von Tests zur Sicherstellung der Güte der Objektwahl vorschlagen (vgl. Shlaer, Mellor 1988, 23-25).

Diese Situation taucht auf, wenn mehrere Sichtweisen auf die Instanzen in einer Klasse vereinigt wurden (vgl. Johnson, Foote 1988, 30).

2.2 Methodenprotokoll von Klassen

Der mögliche Umgang mit Objekten spiegelt sich in ihren zugeordneten Methoden wider, so daß dem Protokoll einer Klasse besondere Bedeutung geschenkt werden muß. Funktionalität und Struktur der Methoden sind äußerst wichtig auf dem Weg hin zu einem wiederverwendbaren Design.

2.2.1 Funktionalität von Methoden

Methoden eines Objektes sollten genau eine, klar abgegrenzte Funktion erfüllen (vgl. Booch 1991, 125). Maßstäbe zur Bewertung dieses Kriteriums können die Größe und auch die Schachtelungstiefe einer Methode sein. Weitere Hinweise kann die natürlichsprachliche Beschreibung der Funktionalität liefern, die bei Erfüllung des Kriteriums als einfach strukturierter Satz formuliert werden können sollte (vgl. Coad 1991, 69-70).
Die Kürze einer Methode erleichtert die Unterklassenbildung zur zugehörigen Klasse, da die Redefinition von weniger umfangreichen Methoden wesentlich einfacher ist, so daß das Splitten einer Methode sehr oft als Folge einer anstehenden Spezialisierung durchgeführt wird (vgl. Johnson, Foote 1988, 28-29; Bieman 1991, 7).

Was die Anzahl der Parameter angeht, so sollten Methoden (mit Ausnahme von Instantiierungsmethoden) möglichst wenige Parameter besitzen - Coad definiert 3 Parameter als obere Grenze (vgl. Coad 1991, 68-69). Dies erhöht die Chance, eine ähnliche Methode zu finden und bei entsprechender Klassenstruktur Vererbung bzw. Polymorphismus ausnutzen zu können. Eine Methode sollte nur unbedingt notwendige Parameter besitzen; Parameter, die als Optionen verstanden werden können, sollten bei der Initialisierung festgelegt werden und wenn nötig nur durch eigens dafür definierte Methoden geändert werden (vgl. Meyer 1990b, 83-84).
Um die Zahl der Parameter zu reduzieren, gibt es im wesentlichen die Möglichkeiten, die Methode zu splitten und ihre Funktionalität auf mehrere Methoden zu verteilen oder eine neue Klasse zu definieren, deren Instanzen gerade die Aggregation einer Gruppe von Argumenten repräsentieren. Dieser zweite Ansatz erscheint insbesondere dann sinnvoll, wenn diese Gruppe in mehreren Methoden als Parameter auftaucht (vgl. Johnson, Foote 1988, 28).

2.2.2 Zugriff auf Instanzvariablen

Gemäß des objektorientierten Konzeptes der Datenkapselung sind Methoden die einzige Möglichkeit, von außen auf die Werte der internen Daten zuzugreifen bzw. sie zu manipulieren. Doch auch für die Klasse selbst, die in ihren Methoden ja direkten Zugriff auf alle Variablen hat, gilt, daß sie sich möglichst dafür definierter Instanzmethoden bedienen sollte. Dies bedeutet, daß für jede Instanzvariable ein Methodenpaar angelegt wird, um lesenden und schreibenden Zugriff darauf zu ermöglichen (vgl. Abb. 3).
Mehrfache Vorteile resultieren aus dieser Konvention: Zum einen erhöht sich die Unabhängigkeit der übrigen Implementierung von der speziellen Form der internen Datenrepräsentation, da im Falle einer Änderung lediglich die Zugriffsmethoden anzupassen sind (vgl. Johnson, Foote 1988, 29). Zum anderen wird der Änderungsaufwand wesentlich vermindert, wenn neue Variablen zusätzlich definiert werden, deren Wert vom Inhalt der existierenden abhängt, da die Aufforderung zur Neuberechnung der neuen Variablen nicht an alle Stellen eingefügt werden muß, wo die alten Variablen geändert werden, sondern lediglich nur noch in den dafür verwendeten standardisierten Zugriffsmethoden (vgl. Wirfs-Brock, Wilkerson 1989a, 34-36).

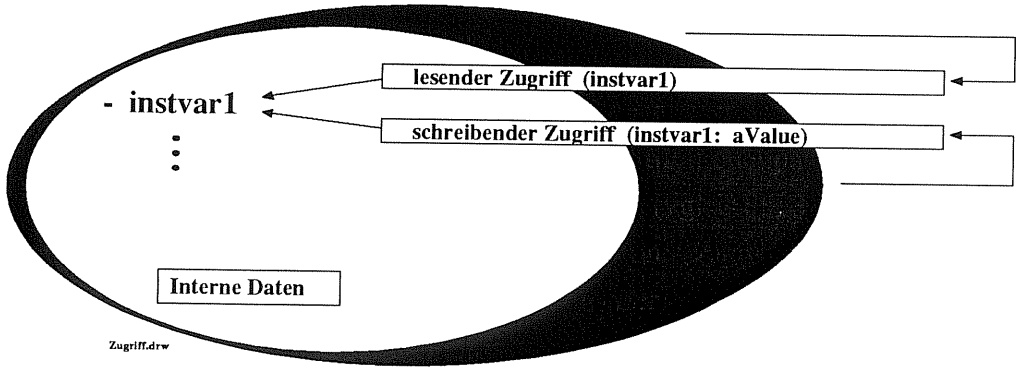

Abb. 3: Zugriff auf Instanzvariablen über Methoden

Nachteile ergeben sich aus dieser Gestaltungsrichtlinie dadurch, daß die Folge der Methodenaufrufe etwas komplexer wird und dementsprechend die Performance leidet. Zudem können Lesbarkeit und Verständlichkeit des Programmcodes vermindert sein - dies kann jedoch durch eine Namensvergabe, die den Zusammenhang zwischen Zugriffsmethoden und betroffenen Variablen deutlich macht, abgemildert werden (vgl. Wirfs-Brock, Wilkerson 1989a, 39).

2.2.3 Namenskonventionen bei Methoden

Ein wesentlicher Punkt in dem Bestreben, wiederverwendbares Design und eine wiederverwendbare Implementierung zu erhalten, liegt in der sorgfältigen Vergabe von Namen für Methoden. Nur wenn es gelingt, den Weg hin zu einem standardisierten Klassenprotokoll zu beschreiten, können die mit den Grundkonzepten Vererbung und Polymorphismus verbundenen Vorteile optimal ausgenutzt werden, da Klassen, deren Instanzen auf die gleichen Nachrichten reagieren, sehr viel einfacher in einer Hierarchie angeordnet werden können. Die Anwendung von Polymorphismus setzt ein sich überdeckendes Methodenprotokoll voraus.[9] Abb. 4 zeigt, welchen Einfluß diese Gestaltungsrichtlinie auf das Methodenprotokoll von Klassen der Eiffel-Bibliothek hatte und wie die Namensgebung für Methoden der Standardklassen *Stack*, *Array*, *Queue* und *H_Table* in späteren Versionen entsprechend modifiziert wurde. Einige Programmiersprachen erlauben die Zusammenfassung von Methoden einer Klasse nach inhaltlichen Kategorien. In Smalltalk existieren auch für die Benennung dieser Kategorien Konventionen, die erfahrenen Entwicklern den Überblick erleichtern (vgl. o.V. 1990, 261-262).Ein anderer Aspekt der Namensvergabe wird von Johnson, Foote als *Recursion Introduction* angeführt: Wenn eine Methode zum Aufruf einer ähnlichen Methode bei einer Unterkomponente des ursprünglichen Empfängers führt, sollte diese zweite Methode den gleichen (oder bei unterschiedlicher Parameterzahl ähnlichen) Namen erhalten, um die Zusammenhänge deutlich werden zu lassen (vgl. Johnson, Foote 1988, 28).

Klasse	Alte Version	Neue Version
Stack	push pop top	put remove item
Array	enter entry	put item
Queue	add remove_oldest oldest	put remove item
H_Table	insert delete value	put remove item

eiffel.drw

Abb. 4: Modifikation der Namensgebung bei Methoden der Eiffel-Bibliothek (nach: Meyer 1990b, 82)

---

9 Hopkins, Knolle demonstrieren die Vorteile von Standardprotokollen mit Hilfe von Beispielen aus der Objective-C Klassenbibliothek, die u.a. Methoden zum Instantiieren und Initialisieren, zum Kopieren, zum Vergleich, zum Speichern und Wiederauffinden von Objekten betreffen (vgl. Hopkins, Knolle 1990).

2.2.4 Kennzeichnung der Sichtbarkeit von Methoden

Die Sichtbarkeit von Methoden kennzeichnet, welchen Objekten es innerhalb des objektorientierten Systems gestattet ist, die entsprechende Methode durch Nachrichtenkommunikation anzustoßen. Nicht in allen Programmiersprachen ist hier überhaupt eine Unterscheidung möglich; die umfangreichste Differenzierung ermöglicht C++, wo Methoden als *private* (nur das Objekt selbst darf die Methode anstoßen), *protected* (nur Objekte hierarchisch untergeordneter Klassen dürfen die Methode anstoßen) oder *public* (alle Objekte dürfen die Methode anstoßen) deklariert werden können.
Auch hierbei sind einige Gestaltungsrichtlinien zu beachten: Die in Abschnitt 2.2.2 geforderten Methoden, die lediglich lesend oder schreibend auf Instanzvariablen zugreifen, sollten in jedem Fall als privat definiert werden, da andernfalls das Prinzip der Datenkapselung ausgehöhlt wird (vgl. Wirfs-Brock, Wilkerson 1989a, 36). Lediglich Methoden, deren Struktur vollkommen unabhängig von der internen Repräsentation der Daten einer Klasse ist, sollten als Bestandteile des Public Interfaces deklariert werden, um so das Prinzip der Datenkapselung möglichst in vollem Umfang zu nutzen (vgl. Korson, McGregor 1990, 53-54).

2.3 Beziehungen zwischen Klassen

Neben dem Aufbau einzelner Klassen wird ein objektorientiertes System durch die Beziehungen bestimmt, die zwischen den Klassen existieren. Nach einleitenden Betrachtungen der Richtlinien für die Modulbildung werden daher die Verwendung hierarchischer Beziehungen sowie von Kundenbeziehungen zwischen Klassen diskutiert.

2.3.1 Richtlinien der Modulbildung

Die Bildung von Modulen führt zu einer Strukturierung eines Gesamtsystems mit dem Ziel, die Übersichtlichkeit zu wahren. Wesentliche Überlegungen zur Bildung von Modulen gehen auf Parnas zurück, der auch das Geheimnisprinzip formulierte, das ein grundlegendes Kennzeichen objektorientierten Entwurfs darstellt. Dieses Prinzip fordert im wesentlichen, daß jedes Modul nach außen eine Schnittstelle anbietet, die die Benutzbarkeit des Moduls beschreibt, wobei die Realisierung dieser Funktionalität allerdings intern verborgen sein sollte (vgl. Parnas 1972).

Zwei Kennzahlen charakterisieren die individuelle Gestaltung der Zerlegung eines Systems in Module: Die *Kopplung* ist ein Maß für den Grad der Enge der Verbindungen der Module untereinander, die *Kohäsion* mißt den Grad der Verbindungen der Elemente innerhalb eines Moduls, ist also ein Maß für den internen Zusammenhalt. Ziel jeder Entwicklung sollte es sein, die Struktur so zu gestalten, daß eine möglichst lose Kopplung, aber eine hohe Kohäsion resultiert (vgl. Booch 1991, 124).

Auf objektorientierte Entwicklung übertragen muß diese generelle Aussage etwas differenzierter betrachtet werden, da sich die Kopplung eines Systems sowohl in den Vererbungs- als auch den Kundenbeziehungen zwischen Klassen und Objekten zeigt, deren Ausgestaltung in den folgenden Kapiteln ausführlich betrachtet wird (vgl. Coad 1991, 68-69). Auch wenn eine Klasse die vorherrschende Modularisierungseinheit in objektorientierten Systemen ist, kann Kohäsion für verschiedene Einheiten betrachtet werden und z.B. auch der innere Zusammenhalt einer Methode oder einer Generalisierungs-/Spezialisierungsbeziehung beurteilt werden (vgl. Coad 1991, 69-70).

2.3.2 Hierarchische Beziehungen

Mit der Definition einer hierarchischen Beziehung zwischen Klassen, d.h. der Definition einer Klasse als Unterklasse einer anderen, ist im objektorientierten Umfeld die Vererbungsbeziehung mitdefiniert, die je nach Programmiersprache allerdings unterschiedlich ausgeprägt sein kann. Diskussionspunkte sind die generelle Form der Klassenhierarchie, die Verwendung abstrakter Klassen sowie der Einsatz der Vererbungsbeziehungen gegenüber der Definition anderer Beziehungen zwischen Klassen.

2.3.2.1 Struktur von Klassenhierarchien

Offensichtlich wird Vererbung dann am stärksten ausgenutzt, wenn der Hierachiebaum der Klassen eine möglichst schmale, aber tiefgegliederte Struktur aufweist (vgl. Johnson, Foote 1988, 29). Für Vererbungsbeziehungen ist eine hohe Kopplung zwischen den beteiligten Klassen gewünscht, da damit die existierenden Gemeinsamkeiten zwischen Klassen optimal dargestellt werden, so daß der damit einhergehende Nachteil einer schwereren Verständlichkeit durch die Verteilung von Funktionalitäten über Hierarchiestufen hinweg in Kauf genommen wird:

> "Forests of classes are more loosely coupled, but may not exploit all the commonality that exists. Trees of classes exploit this commonality, so individual classes are smaller than in forests. However, to understand a particular class, it is

> usually necessary to understand the meaning of all classes it inherits from or uses. Selecting the proper shape of this class structure is highly problem dependent." (Booch, Vilot 1991, 138)

Hohe Kopplung kann daran abgelesen werden, daß möglichst alle für die Oberklasse definierten Variablen und Methoden auch sinnvolle Anwendung in der Unterklasse finden. Dagegen zeugt von einer niedrigen Kopplung, wenn Attribute der Oberklasse gar nicht benutzt werden, oder noch schlimmer, sogar Methoden der Oberklasse als gar nicht gültig erklärt werden. Gründe für solche Situationen liegen fast immer in einer fehlerhaften Modellierung, die Klassen in einer hierarchischen Beziehung anordnet, obwohl diese nicht in einer wirklichen Generalisierungs- bzw. Spezialisierungsbeziehung stehen (vgl Coad 1991, 69-70). Johnson, Foote fordern von einer korrekten Modellierung, daß eine Klasse an jeder Stelle durch eine Unterklasse ersetzt werden kann:[10]

> "There are a couple of ways that a designer can tell whether a subclass is a specialization of a superclass. An abstract definition is that anywhere the superclass is used, the subclass can be used." (Johnson, Foote 1988, 29)

Falls zwischen zwei Klassen keine vollkommene Generalisierungs- bzw. Spezialisierungsbeziehung besteht, eine Vererbungsbeziehung aber aufgrund vielfältiger Gemeinsamkeiten wünschenswert wäre, bleibt als ein gangbarer Weg, die Gemeinsamkeiten zwischen beiden Klassen auszufaktorisieren und in einer neu zu definierenden, gemeinsamen Oberklasse abzulegen (vgl. Halbert, O'Brien 1987, 78). Dies führt zur Bildung abstrakter Klassen.

2.3.2.2 Verwendung abstrakter Klassen

Abstrakte (oder auch aufgeschobene (vgl. Meyer 1990a, 252-259)) Klassen enthalten lediglich die Spezifikationen aller Methoden, die für ihre Unterklassen Bedeutung haben; die Implementierung dagegen, die durchaus für jede Klasse verschieden sein kann, wird aufgeschoben und erst in den Unterklassen festgelegt. Diese Oberklassen sind also unvollständig in dem Sinne, daß ihr Rumpf unvollständig bleibt, so daß zur Laufzeit auch keine Instanzen dieser Klassen nutzbringend erzeugt werden können, da die spezifizierten Methoden nicht ausführbar sind (vgl. Schlüter et al. 1990, 32).
Abstrakte Klassen liefern nicht nur eine Zusammenfassung der für die Unterklassen geltenden Funktionalitäten, sondern sind darüber hinaus als Handlungsanweisung an den Programmierer eventuell zukünftiger Unterklassen zu verstehen, welche Methoden auf dem

---

[10] Halbert, O'Brien demonstrieren Ausnahmefälle, in denen es sinnvoll ist, eine hierarchische Beziehung genau umgekehrt zur in der Realität existierenden Spezialisierungsbeziehung zu modellieren; in fast allen Fällen kann ein solches Auseinanderklaffen aber durch eine Änderung der mit jeder Klasse verbundenen Semantik verhindert werden (vgl. Halbert, O'Brien 1987, 77-78).

Niveau der Unterklasse spezifisch zu überschreiben sind (vgl. Deutsch 1987, 92). Abstrakte Klassen unterscheiden sich von konkreten Klassen insbesondere darin, daß sie verwendet werden, um gemeinsame Funktionalitäten von Klassen zu bündeln und Ausgangspunkt von Spezialisierungen und Erweiterungen zu sein (vgl. Wu 1991).
Aus diesen Verwendungsmöglichkeiten folgt, daß abstrakte Klassen in Klassenhierarchien die oberen Plätze einnehmen. Es kann sogar als allgemeine Richtlinie formuliert werden, daß die Spitze einer Vererbungshierarchie als abstrakte Klasse definiert werden sollte, um die Erweiterungsmöglichkeiten des objektorientierten Entwurfs möglichst optimal auszuschöpfen (vgl. Johnson, Foote 1988, 23 sowie 29). LeJacq fordert, möglichst viel der Funktionalität eines Systems schon bei der Definition abstrakter Klassen in allgemeingültiger Art und Weise zu spezifizieren:

> "To maximize the potential for refinement, the abstract state space of base classes should be as large as possible." (LeJacq 1991, 92)

Ein möglicher Weg, die Definition abstrakter Klassen zu fördern, besteht darin, beim objektorientierten Entwurf den Blick ganz bewußt weg von den internen Strukturen hin zur Definition der Verantwortlichkeit jeder Klasse zu lenken und somit zu einem Nachrichtenprotokoll zu kommen, daß vollkommen unabhängig von der Implementierung der Funktionalitäten ist (vgl. Wirfs-Brock, Wilkerson 1989b).

2.3.2.3 Konstruktionsansatz versus Unterklassenbildung

Sehr häufig steht man beim objektorientierten Entwurf vor der Entscheidung, bereits definierte Klassen in die aktuellen Aufgaben einzupassen. Hier bieten sich in vielen Fällen zwei gegeneinander abzuwägende Alternativen an (*A* sei dabei die neu zu definierende Klasse, *B* die Klasse, deren Funktionalität ausgenutzt werden soll; *A* und *B* sollen ähnliche Funktionalitäten aufweisen):
Der **Konstruktionsansatz** basiert auf dem Prinzip der Datenkapselung und geht den Weg, für die neue Klasse *A* eine Instanzvariable *b* zu definieren, die zur Laufzeit mit einer Instanz der Klasse *B* initialisiert wird. Die Klasse *B* wird sozusagen in die Klasse *A* eingenistet. Wie der Name schon sagt, wird bei der **Unterklassenbildung** die neue Klasse *A* einfach als Unterklasse von *B* definiert und erbt somit die Methoden und Variablen, sofern dies nicht für einzelne Methoden explizit ausgeschlossen wird.
Abb. 5 und 6 verdeutlichen diese beiden Vorgehensweisen am Beispiel der Definition der Klasse *Teilestamm*, die die Verwaltung der Teile eines Unternehmens ermöglichen und diese über Schlüssel zugreifbar machen soll. Es bestehen daher die Alternativen, *Teilestamm* als Unterklasse der bereits existierenden Klasse *Dictionary* zu definieren, die als Container-

klasse diese allgemeine Funktion zur Verfügung stellt, oder aber eine Instanzvariable *elemente* bei der neuen Klasse *Teilestamm* anzulegen, die auf eine Instanz der Klasse *Dictionary* verweist.

Bei der Betrachtung der Vor- und Nachteile wird deutlich, daß bei Wahl der ersten Alternative sehr viel weniger neuer Code zu implementieren ist, da automatisch eine Vielzahl von Methoden von der Oberklasse geerbt wird. So werden ohne zusätzliche Ergänzungen Statements der folgenden Art möglich:

- Teilestamm add: Teil_2314
- Teilestamm remove: Teil_2314

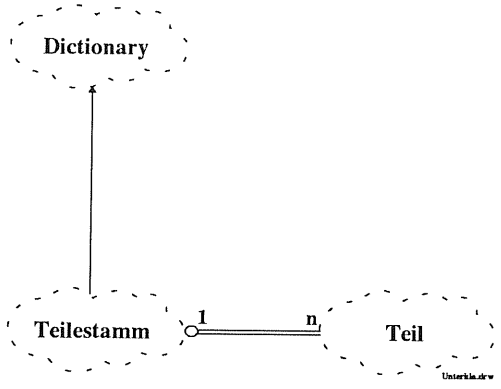

Abb. 5: Unterklassenbildung am Beispiel der Klasse *Teilestamm*[11]

Im Gegensatz dazu müssen beim Konstruktionsansatz viele Methoden neu implementiert werden, indem die entsprechenden Nachrichten an die Unterkomponente weitergeleitet werden. Falls allerdings die Situation vorliegt, daß sehr viele Methoden der Oberklasse gar nicht sinnvoll auf Instanzen der Unterklasse anzuwenden sind, so bringt der Konstruktionsansatz den wesentlichen Vorteil, das Protokoll der neuen Klasse nicht unnötig zu komplizieren und somit auch die Komplexität der gesamten Anwendung zu reduzieren, so daß die in Abschnitt 2.3.2.1 formulierte Richtlinie, eine Vererbungsbeziehung nur bei vollkommener Generalisierungs-/Spezialisierungsbeziehung zu definieren, noch einmal bekräftigt werden muß.

Dies gilt insbesondere, da bei der Unterklassenbildung sehr oft die Notwendigkeit besteht, den Code der zur Verwendung anstehenden Klassen zu erforschen (z.B. um durch Analyse die bzgl. Zeit und Speicherplatz günstigste Klasse zu ermitteln). Zum einen wird hierbei

---

11 Zur Darstellung wird die Notation der Methode *Object-Oriented Design* verwendet (vgl. Booch 1991, 154-186). Pfeile weisen hier von Unter- zu Oberklassen, Doppellinien repräsentieren Verwendungsbeziehungen zwischen Klassen, wobei der unausgefüllte Kreis deutlich macht, daß diese Instanzen der Klasse *Teil* auch von anderen Objekten aus sichtbar sind (vgl. dagegen die Verwendung der Klasse *Dictionary* in Abb. 6)

also (z.B. aufgrund von Performanceüberlegungen) gegen das Prinzip der Datenkapselung verstoßen, zum anderen taucht die Schwierigkeit auf, daß eine Verfolgung des Kontrollflusses u.U. sehr kompliziert werden kann, wenn in der Oberklasse wiederum komplexe Vererbungsbeziehungen, und insbesondere Redefinitionen von Methoden ausgenutzt werden (vgl. Taenzer, Ganti, Podar 1989).

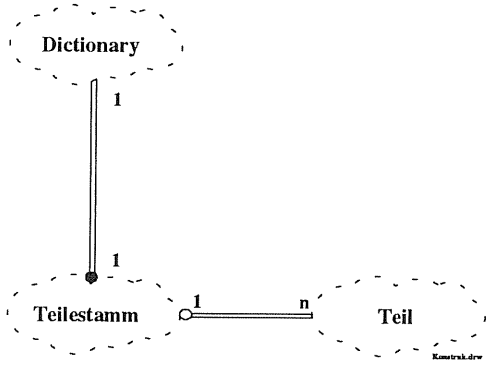

Abb. 6: Konstruktionsansatz am Beispiel der Klasse *Teilestamm*

All diese Überlegungen verdeutlichen die Reduzierung der Komplexität durch den Konstruktionsansatz im Gegensatz zur Unterklassenbildung, die jedoch bei vollkommener Spezialisierungsbeziehung durchaus ihre Vorteile hat und zu einer sehr schnellen Entwicklung führt.

2.3.3 Kunden-Beziehungen

Neben der Unterklassenbeziehung existiert in objektorientierten Systemen weiterhin die Form von Beziehungen, daß eine Klasse als Kunde einer anderen Klasse auftritt. Hier kann zudem in die Part-Of-Beziehung und den Nachrichtenaustausch unterschieden werden.

2.3.3.1 Part-Of-Beziehungen

Die Part-Of-Beziehung wurde schon als Grundlage des Konstruktionsansatzes gegenüber der Unterklassenbildung diskutiert und macht ein wesentliches Charakteristikum des objektorientierten Modellierens aus. Im Gegensatz zu anderen Modellierungsansätzen wird es möglich, komplexe, netzartige Strukturen zwischen Objekten und damit auch ihren Klassen aufzubauen, indem Attribute definiert werden, die auf andere Objekte verweisen. Hinweise zum Einsatz der Part-Of-Beziehung sind der vorangegangenen Diskussion zu entnehmen.

2.3.3.2 Nachrichtenaustausch zwischen Klassen

Der Austausch von Nachrichten zwischen Objekten und Klassen ist das grundlegende Mittel der Kommunikation in objektorientierten Systemen. Designrichtlinien hierzu geben Antworten darauf, wozu die Nachrichtenkommunikation eingesetzt werden sollte und welche Partner miteinander kommunizieren sollten.

Partner des Nachrichtenaustauschs

Das Senden von Nachrichten an andere Objekte ist eine Form der Beziehung, die den Grad der Kopplung eines objektorientierten Systems mitbestimmt. Aus dem generellen Ziel der Modulbildung, eine möglichst lose Kopplung zu erreichen, folgt, daß die Anzahl der gesendeten und empfangenen Nachrichten möglichst klein sein sollte (vgl. Coad 1991, 68-69).

Prinzipiell kann ein Objekt Nachrichten an jedes andere Objekt schicken, das innerhalb seines Sichtbarkeitsbereiches liegt, so daß Empfänger der folgenden Kategorien vorkommen können (vgl. Booch 1991, 128):

- **Same lexical scope**: Der Name des Empfängers ist explizit bekannt,
- **Parameter**: Das Objekt wurde dem sendenden Objekt als ein aktueller Parameter oder Ergebnis eines Nachrichtenaufrufs übergeben, oder
- **Field**: Eine Instanzvariable des Senders verweist auf das Objekt.

Neben diesen generellen Aussagen wird versucht, die möglichen Partner des Nachrichtenaustauschs weiter einzuengen, woraus zwei konkurrierende Gestaltungsrichtlinien resultieren, deren Anwendung gegeneinander abgewogen werden muß: Aus dem Prinzip der Datenkapselung folgt, daß Nachrichten möglichst nicht direkt zu Unterkomponenten eines Objektes geschickt werden sollten, da diese nach außen unsichtbar sein sollten (vgl. Korson, McGregor 1990, 53-54; Blake, Cook 1987). Die Anwendung dieser Vorgabe führt allerdings zu dem Phänomen des "Message-Pass-Through". Hier werden Nachrichten von Objekten zu ihren Unterkomponenten weitergereicht - es entsteht der Nachteil einer unnötigen Einbindung dieser Zwischenklassen, die im Falle einer Änderung des Nachrichtenaufbaus auch von Modifikationen betroffen sind (vgl. Coad 1991, 69).

Lieberherr, Holland, Riel versuchen mit der Formulierung des "Law of Demeter", die möglichen Empfänger von Nachrichten noch weiter einzugrenzen, wenn Datenkapselung und Modularität möglichst optimal realisiert werden sollen (vgl. Lieberherr, Holland, Riel

1988): Innerhalb des Rumpfes einer Methode *M*, die bei einer Klasse *C* definiert wird, dürfen nur Nachrichten an *C* selbst geschickt werden, an Instanzvariablen von *C* (genauer: an Objekte, auf die von Instanzvariablen referenziert wird) und an Objekte, die als Parameter von *M* auftauchen. Es wird in die starke und die schwache Form des "Law of Demeter" unterschieden, wobei die starke Form nur Instanzvariablen anerkennt, die genau bei *C* deklariert wurden, in der schwachen Form dagegen auch ererbte Instanzvariablen zugelassen sind. Als eine mögliche nachteilige Folge resultiert aus der Anwendung, daß die Zahl der Methoden, aber auch die Zahl der Parameter für einzelne Methoden stark erhöht werden muß und die Verständlichkeit stark einschränkt wird:[12]

> "We have seen that there is a price to pay. The greater the level of data hiding, the greater the penalities are in terms of the number of methods, speed of execution, number of arguments to methods and sometimes readability of the code." (Lieberherr, Holland, Riel 1988, 333)

Auch Ferstl, Sinz kommen im Rahmen der Entwicklung des Semantischen Objektmodells (SOM) zu einer Einengung der potentiellen Kommunikationspartner, da das objektorientierte Modell in wesentlichen Teilen aus dem Strukturierten Entity-Relationship-Modell (SERM) hergeleitet wird und den hier bestehenden Strukturbeziehungen hohe Aussagekraft über die Kommunikation im objektorientierten Modell zugesprochen wird (vgl. Ferstl, Sinz 1990).

Verwendung des Nachrichtenaustauschs

Der Austausch von Nachrichten ist nicht nur geeignet, Aktionen anzustoßen, sondern sollte auch verwendet werden, um Informationen zu beschaffen, so daß die Verwendung globaler Variablen möglichst reduziert werden sollte. Im Zusammenhang mit dem "Law of Demeter" gilt auch die andere Richtung, daß ein Objekt, das die Funktionalität eines anderen Objektes ausnutzt, diesem auch die benötigten Informationen möglichst schon als Parameter des Methodenaufrufs mitgibt (vgl. Korson, McGregor 1990, 53-54).
In keinem Fall sinnvoll ist die explizite Abfrage der Klasse eines Objektes. Das Problem, Objekte in Abhängigkeit von ihrer Klassenzugehörigkeit zu unterschiedlichen Aktionen zu veranlassen, sollte so geregelt werden, daß unabhängig von der Klasse die immer gleiche Nachricht an das in Frage stehende Objekt geschickt wird und dieses unter Ausnutzung von Polymorphismus gemäß seiner Klassenzugehörigkeit darauf reagiert (vgl. Johnson, Foote 1988, 28). Casais gibt eine allgemeine Vorgehensweise an, wie das explizite Überprüfen der Klasse eines Objektes vermieden werden kann (vgl. Casais 1990, 157).

---

[12] Zu einer ausführlicheren Diskussion der Vor- und Nachteile des "Law of Demeter", vgl. Sakkinen 1988.

3 ZUSAMMENFASSUNG

Ausgangspunkt der vorangehenden Darstellung war die Frage nach dem sinnvollen Einsatzspektrum objektorientierter Methoden: Aufgrund der engen Kopplung zwischen DV-Konzept und Implementierung sollte die Verwendung objektorientierter Programmiersprachen auch zum Einsatz objektorientierter Designmethoden führen. Ob allerdings auch für die Erstellung eines Fachkonzeptes, das noch vollkommen unabhängig von einer EDV-technischen Realisierung gehalten ist, auf Methoden dieses Paradigmas zurückgegriffen werden sollte, hängt neben der weiteren Verwendung des Fachkonzeptes stark von den Vorlieben und methodischen Kenntnissen der Beteiligten ab.

Zielsetzung war es, zu einer Zusammenfassung von Richtlinien zu kommen, die Unterstützung bei der Erstellung eines objektorientierten DV-Konzeptes bieten. Aufgrund des engen Zusammenhangs zur Implementierung und der strukturerhaltenden Transformation von objektorientiertem Fach- zu DV-Konzept erlangen diese jedoch auch für die übrigen Beschreibungsebenen hohe Bedeutung. Allerdings kann von dem eingenommenen, eher technisch orientierten Standpunkt aus nicht die Frage beantwortet werden, ob es gelungen ist, ein Modell zu entwickeln, das die Problemdomaine in semantisch korrekter Weise abbildet. Die erläuterten Designrichtlinien stellen größtenteils Heuristiken dar, die dem Entwickler natürlich nicht den unfehlbaren Weg hin zu optimalem Design und wiederverwendbaren Komponenten weisen; in der Summe werden sie jedoch wertvolle Hilfestellung bieten können, um die Qualität der erstellten Software zu erhöhen.

LITERATURVERZEICHNIS

Abbot, R. 1983:
Program Design by Informal English Description. In: Communications of the ACM, 26 (1983) 11, S. 882-894.

Beck, K. et al. 1988:
Experiences with Reusability (PANEL). In: Proceedings of OOPSLA '88, San Diego 1988, S. 372-376.

Bieman, J.M. 1991:
Deriving Measures of Software Reuse in Object Oriented Systems. Colorado State University, Technical Report #CS-91-112. Fort Collins 1991.

Blake, E.; Cook, S. 1987:
On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk. In: Proceedings of ECOOP '87, Paris 1987, S. 41-50.

Boehm, B.W. 1981:
Software Engineering Economics. Englewood Cliffs 1981.

Booch, G. 1991:
Object-Oriented Design with Applications. Redwood City 1991.

Booch, G.; Vilot, M. 1991:
Object-Oriented Design. In: Wiener, R.S. (Hrsg.): Journal of Object-Oriented Programming, Focus on Analysis and Design. New York 1991, S. 136-149.

Budde, R. et al. 1992:
Objektorientierte Analyse und Entwurf von Anwendungssystemen. In: Scheer, A.-W. (Hrsg.): Datenbanken 1992. Tagungsband der 4. Fachtagung Praxis und Tendenzen relationaler Datenbanken, Saarbrücken 1992, S. 231-251.

Casais, E. 1990:
Managing Class Evolution in Object-Oriented Systems. In: Tsichritzis, D.C. (Hrsg.): Object Management. Centre Universitaire d'Informatique, Genf 1990, S. 133-195.

Coad, P. 1991:
OOD Criteria, Part 1. In: Journal of Object-Oriented Programming, 4 (1991) 3, S. 67-70.

Coad, P.; Yourdon, E. 1991:
Object-Oriented Analysis. 2. Aufl., Englewood Cliffs 1991.

Conte, S.D.; Dunsmore, H.E.; Shen, V.Y. 1986:
Software Engineering Metrics and Models. Menlo Park 1986.

de Champeaux, D. et al. 1990:
Structured Analysis and Object Oriented Analysis (PANEL). In: Proceedings of ECOOP/OOPSLA '90, Ottawa 1990, S. 135-139.

Fenton, N. 1991:
Software Metrics: A Rigorous Approach. London 1991.

Ferstl, O.K.; Sinz, E.J. 1990:
Objektmodellierung betrieblicher Informationssysteme im Semantischen Objektmodell (SOM). In: Wirtschaftsinformatik, 32 (1990) 6, S. 566-581.

Halbert, D.C.; O'Brien, P.D. 1987:
Using Types and Inheritance in Object-Oriented Programming. In: IEEE Software, 4 (1987) 5, S. 71-79.

Henderson-Sellers, B.; Constantine, L.L. 1991:
Object-Oriented Development and Functional Decomposition. In: Wiener, R.S. (Hrsg.): Journal of Object-Oriented Programming, Focus on Analysis and Design. New York 1991, S. 18-23.

Heß, H. 1989:
Objektorientierter Systementwurf. In: Information Management, 4 (1989) 3, S. 76-77.

Heß, H.; Scheer, A.-W. 1992a:
Methodenvergleich zum objektorientierten Design von Softwaresystemen. In: HMD - Theorie und Praxis der Wirtschaftsinformatik, 29 (1992) 165, S. 117-137.

Heß, H.; Scheer, A.-W. 1992b:
Retrieval wiederverwendbarer Softwarebausteine. In: Wirtschaftsinformatik, 34 (1992) 2, S. 190-200.

Hopkins, J.; Knolle, N. 1990:
A Framework for Reusability. In: Journal of Object-Oriented Programming, 3 (1990) 3, S. 75-78.

Johnson, R.E.; Foote, B. 1988:
Designing Reusable Classes. In: Journal of Object-Oriented Programming, 1 (1988) 2, S. 22-35.

Korson, T.; McGregor, J.D. 1990:
Understanding Object-Oriented: A Unifying Paradigm. In: Communications of the ACM, 33 (1990) 9, S. 40-60.

LeJacq, J.P. 1991:
Semantic-based Design Guidelines for Object-Oriented Programs. In: Wiener, R.S. (Hrsg.): Journal of Object-Oriented Programming, Focus on Analysis and Design. New York 1991, S. 86-97.

Li, X. 1991:
Integration of Structured and Object-Oriented Programming. In: Wiener, R.S. (Hrsg.): Journal of Object-Oriented Programming, Focus on Analysis and Design. New York 1991, S. 54-60.

Lieberherr, K.J.; Holland, I.M; Riel, A. 1988:
Object-Oriented Programming: an Objective Sense of Style. In: Proceedings of OOPSLA '88, San Diego 1988, S. 323-334.

Meyer, B. 1990a:
Objektorientierte Softwareentwicklung. München-Wien 1990.

Meyer, B. 1990b:
Lessons Learned from the Design of the Eiffel Libraries. In: Communications of the ACM, 33 (1990) 9, S. 68-88.

Monarchi, D.E.; Puhr, G.I. 1992:
A Research Typology for Object-Oriented Analysis and Design. In: Communications of the ACM, 35 (1992) 9, S. 35-47.

o.V. 1990:
User's Guide, Objectworks Smalltalk, Release 4. Mountain View 1990.

Parnas, D.L. 1972:
On the Criteria to be Used in Decomposing Systems into Modules. In: Communications of the ACM, 5 (1972) 12, S. 1053-1058.

Raasch, J. 1991:
Systementwicklung mit Strukturierten Methoden. München-Wien 1991.

Ross, R. 1987:
Entity Modeling: Techniques and Application. Boston 1987.

Rosson, M.B.; Gold, E. 1989:
Problem-Solution Mapping in Object-Oriented Design. In: Proceedings of OOPSLA '89, New Orleans 1989, S. 7-10.

Rumbaugh, J. et al. 1991:
Object-Oriented Modeling and Design. Englewood Cliffs 1991.

Sakkinen, M. 1988:
Comments on the Law of Demeter and C++. In: SIGPLan Notices, 23 (1988) 12, S. 38-44.

Scharenberg, M.E.; Dunsmore, H.E. 1991:
Evolution of Classes and Objects during Object-Oriented Design and Programming. In: Wiener, R.S. (Hrsg.): Journal of Object-Oriented Programming, Focus on Analysis and Design. New York 1991, S. 14-17.

Scheer, A.-W. 1992:
Architektur integrierter Informationssysteme - Grundlagen der Unternehmensmodellierung. 2. Aufl., Berlin et al. 1992.

Schlüter, P. et al. 1990:
Objektorientierte Softwareentwicklung: Konzepte und Terminologie. In: Softwaretechnik-Trends, Mitteilungen der GI-Fachgruppe 'Software-Engineering', 10 (1990) 2, S. 22-44.

Shlaer, S.; Mellor, S. 1988:
Object-Oriented Systems Analysis: Modeling the World in Data. Englewood Cliffs 1988.

Taenzer, D.; Ganti, M.; Podar, S. 1989:
Problems in Object-Oriented Software Reuse. In: Proceedings of ECOOP '89, Nottingham 1989, S. 25-38.

Wirfs-Brock, A.; Wilkerson, B. 1989a:
Variables Limit Reusability. In: Journal of Object-Oriented Programming, 2 (1989) 1, S. 34-40.

Wirfs-Brock, R.; Wilkerson, B. 1989b:
Object-Oriented Design: A Responsibility-Driven Approach. In: Proceedings of OOPSLA '89, New Orleans 1989, S. 71-75.

Wu, C.T. 1991:
Benefits of Abstract Superclasses. In: Journal of Object-Oriented Programming, 3 (1991) 6, S. 57-62.