Informatik-Fachberichte 339, Hrsg: H.J. Kreowski

Organisation, Benutzer und Software

- Zur Konstituierung von Softwaresystemen -

Thorsten Spitta: Vatter Unternehmensgruppe, Rheine und Schongau

Abstract: This paper deals with the first and most important decision in the development process of software: The constitution of the system itself. This step is heavily neglected in the software engineering literature. A system is constructed by using methods of data engineering (data models, construction of terms). These methods allow to find the objects, which are the buildings stones of object oriented software, in a very early phase. Those objects can later be specified and implemented without break in the development structure. Such a break is inherent in all (very popular) flow oriented methods like SA, SADT and others. These methods are not useful to gain object oriented software structures. Therefore they have to be eliminated from teaching, tools and practice.

1. Zum Selbstverständnis des Software Engineering

Software Engineering (SE) ist eines der wichtigsten Fächer der Informatik, wenn man unter Informatik die Wissenschaft der Erstellung von Hardware und Software versteht, die praktisch benutzbar ist. Dies ist sicher eine zulässige Sicht. Praktische Verwendbarkeit der Ergebnisse ist unbestrittener Anspruch der klassischen Ingenieurwissenschaften wie Bauwesen, Elektrotechnik und Maschinenbau, auf die sich die Informatik so gerne beruft.

Software Engineering ist andererseits ein "ungeliebtes Kind" der Informatik, wie die äußerst geringe Zahl empirischer Veröffentlichungen auf diesem Gebiet in Deutschland zeigt (s. etwa Informatik-Spektrum, Informatik F&E). Das Fach gilt bei einigen Hochschulvertretern als praxeologisch und unwissenschaftlich.

Eine Klärung des oben erwähnten Widerspruchs wird umgangen, indem man die wichtigsten Forschungsfragen des Software Engineering, die der Methoden, für gelöst erklärt:

> "...die Entfaltung der Informatik als einer eigenständigen Wissenschaft mit einem sich immer mehr konsolidierenden Methodenvorrat." und: "...sich...eine solide Ingenieurdisziplin entwickelt hat." (Einladung zur GI-Jahrestagung 1990, S.4)

Ganz im Stil dieser etwas selbstgefälligen Beschreibung der Informatik als Ingenieurdisziplin geistern methodische Schlagworte wie "Top-Down-Entwurf", "Benutzerbeteiligung", "wissensbasiert", "Prototyping" u.a.m. durch die Tagungslandschaft, die Fachliteratur und vor allem durch die Werbung der DV-Industrie, speziell der Sparte Schulung.

Die Methoden sind deshalb die wichtigsten Forschungsfragen, weil Werkzeuge als die andere Seite des Software Engineering immer Methoden implizieren. Methoden sind gewissermaßen die Spezifikation der Werkzeuge. Eine falsche Methode verursacht ein falsches Werkzeug.

Der folgende Beitrag soll aufzeigen, daß

- Handlungsbedarf in der Entwicklung und Erprobung von Methoden besteht, mit denen man **Softwaresysteme**, nicht nur Programme entwickeln kann,

- eine **Methoden-Entsorgung** angesagt ist, durch die Methoden aus dem Lehrkanon entfallen, die nachweislich nicht praktisch verwendbar sind.

Als besonders wichtiges Thema der Methoden wird hier die Bildung und Abgrenzung von Systemen behandelt, nicht die Spezifikation von Programmen. Es soll eine erste Antwort gegeben werden auf die bisher kaum behandelte Frage:

Wie konstituiert man ein Softwaresystem?

Mit **Konstituierung** ist folgendes gemeint: Verschiedene Beteiligte, Auftraggeber und Auftragnehmer genannt, legen fest, was grob ein System leisten und wie es strukturiert sein soll. Dies geschieht auf Basis eines informalen Dokumentes, üblicherweise **Anforderungsdefinition** oder **Vorstudie** genannt. Dann jedoch muß ein konstruktiver Schritt folgen, der dem Softwareentwurf weit vorgelagert ist, die **Systembildung**. Über welches System spricht man? Wo grenzt es sich zu bereits existierenden Systemen ab? Was sind die Schnittstellen?

Dem Autor ist außer dem Verfahren von Jackson (83, JSD) kein Vorgehensmodell bekannt, das diesen Schritt explizit vorsieht. **Definition** (Balzert 82, Entwicklung) und **Anforderungsanalyse** (Hesse 84, Begriffssystem) sind eher intuitive Schritte, auf die unmittelbar Spezifikation und Entwurf folgen. Sprachlich wären die Begriffe **Systemdefinition** oder **Systemkonstruktion** statt "Konstituierung" zutreffender. Sie sind jedoch bereits anderweitig belegt, der eine i.S. von **Spezifikation**, der andere i.S. von **Softwareentwurf**.

Die Struktur eines Softwaresystems entsteht nicht erst beim Softwareentwurf, sondern vor Beginn der Spezifikation. Beide oben zitierten Darstellungen, stellvertretend für sehr viele andere, haben für die wichtigste Entwurfsentscheidung des Softwarentstehungsprozesses (wichtigste, weil früheste!) nicht viel mehr übrig als Nebensätze. Diese Auffassung von Software Engineering, hier "traditionell" genannt, ist die in Hochschule und Praxis noch immer vorherrschende. In der Hochschule ist sie nur ärgerlich, da sie gelehrt wird, in der Praxis ist sie jedoch exorbitant teuer.

Ein Fach Software Engineering, das sich als Konstruktionslehre für real einsetzbare und langfristig wartbare Software versteht, braucht zur Frage der Konstituierung von Systemen vieles nicht neu zu "erfinden". Es muß lediglich interdiszipinär vorgehen und auf Inhalte

- des Faches Datenbanken,
- der klassischen Ingenieurwissenschaften,
- der Betriebswirtschaftslehre

zurückgreifen und sie mit bestimmten Spezifikationsmethoden des Software Engineering kombinieren (vgl. auch Wassermann 79, Unified View):

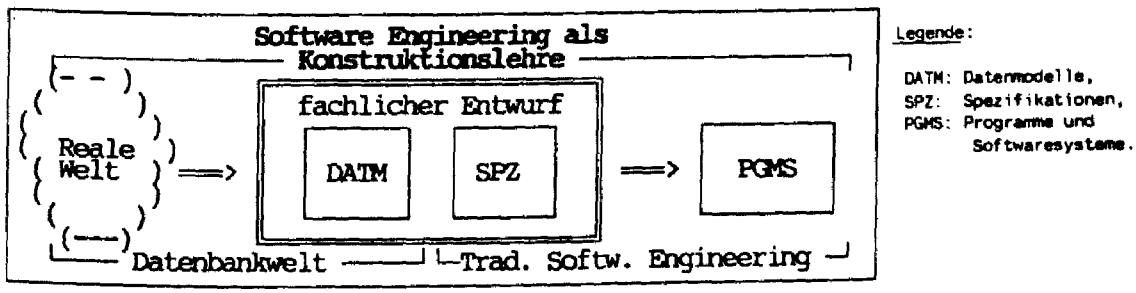

Abb.1: Software Engineering als einheitliche Konstruktionslehre

Dieser Beitrag handelt von administrativen, heute i.d.R. dialogorientierten Systemen mit nachgelagerten Batchprozessen, die in eine **Organisation** eingebettet sind. Die hier dargestellte Methode ist entstanden und wurde angewendet in über 5 Jahren Entwicklung von 12 Softwaresystemen, die alle in Betrieben eingesetzt sind oder sich in der Einführung befinden. Das Entwicklungsvolumen der letzten 4 Jahre ist in einer Datenbank abgelegt und nach vielen Kriterien auswertbar. Es betrug über 50.000 Stunden, das sind 44 MJ. Daneben sind über 30.000 Stunden Wartungstätigkeit sowohl an den Neuentwicklungen als auch an bis zu 20 Jahre alter Software belegt.

Die Konstituierung sog. Prozeßsoftware, die in Maschinensysteme eingebettet ist, wird hier nur gestreift.

Der Beitrag versteht sich als Ansatz zur **Methoden-Entsorgung**, indem sehr bekannte batch- und transformationsorientierte Methoden ausgeschlossen und für schädlich erklärt werden. Damit wird eine Hilfe gegeben, etwas gegen den schon 1983 von Schnupp ausgemachten **Methodenberg** des Software Engineering zu unternehmen, der nach wie vor nicht "abgeräumt" ist (Schnupp 83, Softwaretechnologie).

2. Anlässe zur Entwicklung von Software

Welcher Art sind die Anlässe, die zu einem Entwicklungsauftrag von Software führen? Es lassen sich fünf Fälle unterscheiden:

1. Ein manueller Prozeß ist zu langsam oder zu kostenintensiv. Eine Rationalisierung m.H. von Software ist gewünscht. Dieser Fall ist heute selten: Systeme wie Lohnabrechnung, Buchhaltung, Fakturierung u.ä. existieren in allen größeren Organisationen.
2. Prozesse sind manuell nicht abwickelbar. Es wird Software gebraucht, um eine Firma wettbewerbsfähiger zu machen, als sie ist. Diese Systeme erhalten zunehmende Bedeutung im Rahmen einer langfristigen Informationsstrategie.
3. Informationen sind nicht oder nur schwerfällig zu gewinnen, weil die Daten mit veralteten Systemen verwaltet werden. Eine Neuentwicklung auf der Basis von Datenbanken ist gefordert.
4. Ein ähnlicher Fall ist die Weiterentwicklung von Basissystemen, die getätigte Softwareinvestitionen obsolet machen.
5. Bestehende Systeme sollen verändert oder erweitert werden, weil die benutzende Organisation oder die Umwelt dieser Organisation (Kunden, politische Strukturen) sich verändert haben. Solche Erweiterungsinvestitionen verschlingen den größten Teil des Softwarebudgets von Firmen und Behörden. Sie werden auch **Wartung** genannt, obwohl Wartung eigentlich eine Pflege von Objekten ist, die durch Benutzung verschleißen.

Die Fälle 1 und 2 beinhalten Neuinvestitionen, von denen man verlangt, daß sie wirtschaftlich sind, d.h. mehr Nutzen und Einsparungen über 3 bis 5 Jahre bringen als sie kosten.

Die Fälle 3 und 4 sind Ersatzinvestitionen. Lange aufgeschobene Ersatzinvestitionen sind heute der häufigste Anlaß, Software neu zu entwickeln oder Standardsoftware zu installieren. Fall 5 soll hier nicht weiter betrachtet werden.

Den Fällen 1 bis 4 ist gemeinsam, daß

- formalisierte Informationen verwaltet werden müssen; das sind **Daten**,
- eine Softwarearchitektur entstehen soll, die diese Daten verwaltet; das sind Funktionen, die später als **Programme** ablaufen,
- eine Organisation oder ein technisches Umfeld existiert, in das die Software eingebettet werden soll und die bei der Entwicklung berücksichtigt werden muß; das sind **Benutzer**.

Wir erhalten damit folgende **Grundelemente von Software:**

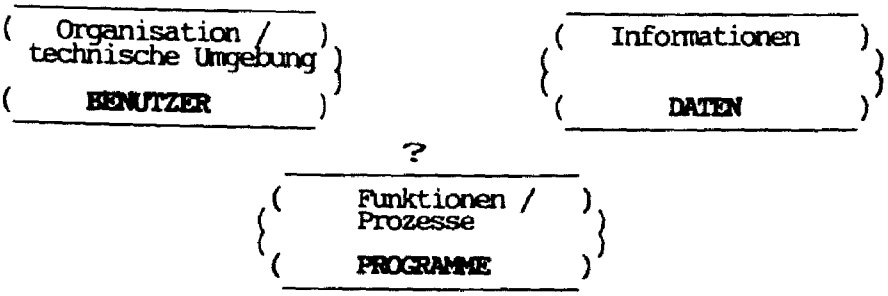

Abb.2: Grundelemente zur Konstituierung von Software

Aufgabe von SE-Methoden ist es, die Analyse und Strukturierung dieser Grundelemente so zu unterstützen, daß ein kostengünstiges und qualitativ hochwertiges Softwaresystem entsteht. Dies muß vor allem durch eine Betrachtung der Beziehungen der Elemente zueinander (s.o. '?') geleistet werden.

Im folgenden soll geprüft werden, ob und inwieweit die gängigen SE-Methoden dieser Forderung gerecht werden und ob sie überhaupt alle Grundelemente berücksichtigen. Danach werden die Grundelemente näher betrachtet, insbesondere die Annahme, daß der Benutzer konstituierend für ein Softwaresystem sei.

3. Traditionelle SE-Methoden

Alle gängigen Methoden des Software Engineering schlagen im Prinzip folgende Vorgehensweise vor:

Funktionen bzw. Prozesse werden gesammelt und geordnet. Die Ordnung erfolgt meist top down im Zuge einer schrittweisen Verfeinerung. Entweder werden die Daten beim Darstellen der Funktionen mitbetrachtet (man sieht die Funktionen als Transformationsprozesse für Daten) oder die Funktionen werden als Funktionshierarchie gesehen und die Daten als Datenhierarchie (in SADT oder in JSD). Ein Rückgriff auf die Methoden zur Datenmodellierung (Relationenmodell, Entity-Relationship-Methode) findet nicht statt.

Je nach Schwerpunkt einer Methode spricht man von datenbezogenen und von funktionsbezogenen Methoden. Ein Streit, was richtiger sei, ist müßig, da wir es mit einem typischen "Henne-Ei-Problem" zu tun haben: Funktionen erzeugen Daten, diese werden von Funktionen benutzt.

Zur Anschauung ein Beispiel, in dem eine Funktionshierarchie ohne die referenzierten Daten betrachtet wird:

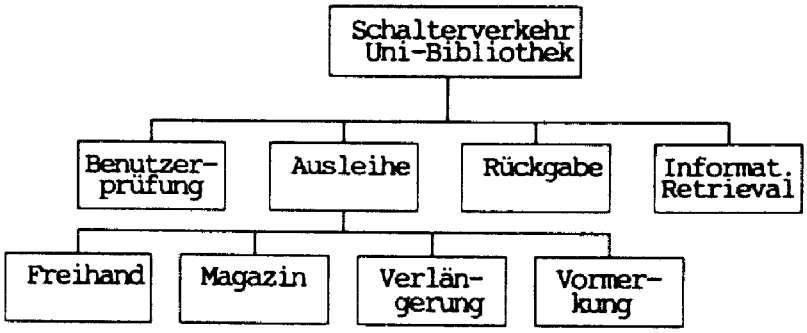

Abb.3a: Funktionshierarchie traditioneller SE-Methoden
(entnommen: Schulz 90, Entwurf, S.37)

Eine hierzu passende Datenhierarchie könnte so aussehen:

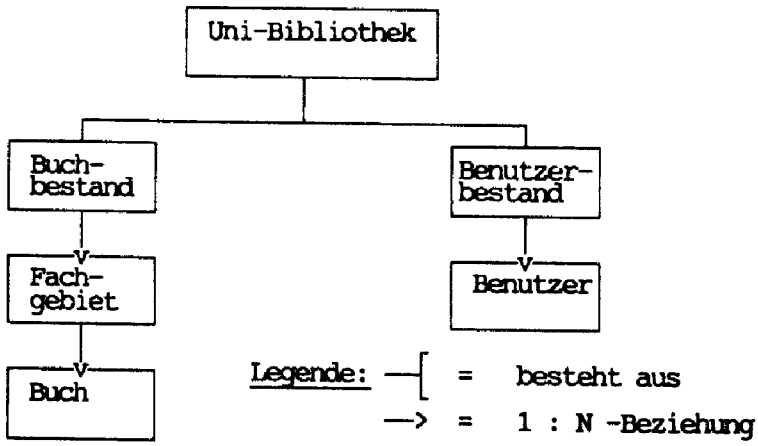

Abb.3b: Datenhierarchie

Die traditionellen Methoden des SE sind dargestellt in Balzert (81, Methoden), in neuerer Form in Schulz (90, Entwurf). Sie gehen allesamt auf Originalveröffentlichungen von 1976/77 zurück (IEEE-SE 3(1977) No.1). Ihnen ist folgendes gemeinsam:

- Entweder sie trennen Funktionen und Daten (Funktionsmodell, Datenmodell), oder sie stellen Transformationen von Daten durch Funktionen dar nach dem Schema: Eingabe —> Verarbeitung —> Ausgabe.
- Sie prüfen Konsistenz durch Input/Output-Abgleiche und verlangen damit methodisch absolute Vollständigkeit. Unbestimmtheiten sind nicht möglich.
- Sie sind aufwendig und bei Anwendung auf reale Entwicklungen kaum überschaubar. Dem Verfasser sind Dokumente von weit mehr als 1000 Seiten zugänglich, die mit IEW (information engineering workbench) erzeugt wurden.

Nun ist Softwareentwicklung ein komplexes Unterfangen und großer Aufwand alleine noch kein Argument gegen eine Methode, wenn hinterher ein qualitativ hochwertiges Ergebnis entsteht. Dieses Ergebnis ist ein Softwaresystem, das aus (Programm-)Moduln, einer Datenbasis und einer Benutzeroberfläche besteht. Es muß geprüft werden, ob dieses Ergebnis auch mit weniger Aufwand entstehen kann, als traditionelle SE-Methoden dies ermöglichen.

4. Traditionelle SE-Methoden und Objektorientierung

Der Entwurf von Software folgt zwei prinzipiell möglichen Ansätzen:
- dem **funktions-, prozedural** oder **datenfluß**orientierten,
- dem **datentyp-** oder auch -als Erweiterung- **objektorientierten.**

Die beiden grundlegenden Paradigmen Datenfluß oder Datentyp wurden schon 1972 von Parnas in ihren Auswirkungen auf die Modularisierung diskutiert (Parnas 72, Criteria). Parnas sprach zwar damals von Entwurfsentscheidungen, nicht nur von Daten, jedoch läßt sich der Ansatz, Entwurfsentscheidungen zum Gegenstand von Moduln zu machen, nur in Form von Datenstrukturen verwirklichen. Datentyp- und als Weiterentwicklung objektorientierte Ansätze beruhen auf Datenstrukturen als den Grundbausteinen von Software. Funktionen werden ihnen in Form von Operationen angegliedert.

4.1 Funktionale bzw. transformationsorientierte Methoden

Funktionsorientierte Modulstrukturen sind nach Parnas nicht wünschenswert, weil die Entwurfsentscheidungen über Modulgrenzen "verschmiert" werden. Die Systeme sind dann nur schwer und aufwendig wartbar. Weil funktionsorientiert modularisierte Systeme dem wichtigsten ökonomischen Kriterium für Softwareinvestitionen nicht genügen, einer kostengünstigen Wartbarkeit, ist diese weit verbreitete Entwurfstechnik abzulehnen. Sie findet sich nach den Beobachtungen des Autors noch überwiegend in industriell genutzten Systemen, da bei einem intuitiven, nicht explizit durchgeführten Softwareentwurf praktisch immer solche Strukturen entstehen. Explizit entstehen diese Strukturen bei Verwendung der Methode structured design (SD), die in vielen Derivaten existiert (Yourdon 76, SD).

Während die Verfechter von SD als Softwareentwurfstechnik weniger werden, feiern die funktionsorientierten "Spezifikations"-Methoden noch immer "fröhliche Urständ". Hierzu zählen structured analysis (SA), SADT, RSL, ISAC, um nur die bekanntesten zu nennen (vgl. hierzu Balzert 82, Entwicklung). Die Schulungsangebote in der Industrie sind voll von Lehrgängen, die Umschulungsinstitute verbreiten diese intuitiv eingängigen Darstellungstechniken auf breiter Front. Die Dozenten dieser Institute geben offenbar unreflektiert weiter, was sie auf Universitäten gelernt haben.

Diese Methoden modellieren allesamt einen **Daten-** oder **Steuerfluß** nach dem primitiven Schema einer erweiterten Wertzuweisung (E-V-A Prinzip). Bei SADT und bei ISAC ist ein Verfeinerungsmechanismus eingebaut , der die Darstellung bis etwa zur dritten Verfeinerungsebene besser handhabbar und übersichtlicher macht als bei SA oder RSL. Jedoch wehe dem, der zu tief verfeinert! 30 Diagramme auf Ebene 3 erzeugen 120 bis 200 Diagramme auf Ebene 4. Dem Autor ist noch niemand begegnet, der so etwas noch überblickte oder gar Änderungen durchführen konnte.

Die o.g. Methoden werden hier **transformationsorientierte Methoden** im Gegensatz zu objektorientierten genannt. Transformationsorientierte Methoden können niemals Anforderungen in einer Struktur liefern, die sich objektorientiert modularisieren läßt. Der Strukturbruch zwischen sog. Sachlogik und sog. DV-Technik, also zwischen Spezifikation und Softwareentwurf, ist zwingend.

Damit kein Mißverständnis entsteht: Als Darstellungstechnik ist eine Methode wie SADT hervorragend geeignet, nicht jedoch als Mittel für Entwurfstätigkeiten. Im Verantwortungsbereich des Verfassers wird SADT erfolgreich für die Nachdokumentation von Altsoftware eingesetzt.

Das Schema der transformationsorientierten Methoden ist:

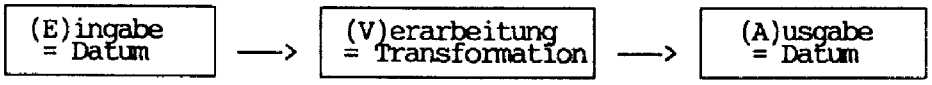

Abb.4: Schema transformationsorientierter Methoden

4.2 Objektorientierte Methoden

Objektorientierte Software besteht aus **Moduln**, die **Operationen auf Datenstrukturen** bereitstellen. Moduln interagieren durch Benutzung der Operationen anderer Moduln. Datenstrukturen repräsentieren Datenobjekte, die temporär im Hauptspeicher oder dauerhaft auf Datenträgern gehalten werden. Daher muß **objektorientiert** (genauer: objekttyporientiert) spezifiziert werden, wenn man ohne Strukturbruch zu einem objektorientierten Softwarentwurf kommen will.

Zur Beschreibung der Schemata objektorientierter Software kann man die Darstellung der Theoretiker der **abstrakten Datentypen** (Liskow 74, ADT; Guttag 77, Data Types) benutzen. Danach ist ein abstrakter Datentyp ADT, der algebraisch spezifiziert werden soll, definiert durch (vgl. Kreowski 81, Spezifikation):

ADT:= {Datentyp, Operationen, Axiome zur Benutzung}

Die Rolle der Axiome für Korrektheitsbeweise soll hier nicht weiter verfolgt werden. Die Klammerung von Datentyp und Operationen ist jedoch notwendig für eine Objektorientierung, wenn auch nicht hinreichend (Näheres s.u.). Sie liegt dem Ansatz von Parnas zugrunde und ist in der "Datenbankwelt" so selbstverständlich, daß reale Datenbanksysteme eigentlich nur nach diesem Prinzip konstruiert sind. Diese "Welt" verfügt über einen umfangreichen Methodenvorrat vom Relationenmodell über den Entity-Relationship-Ansatz bis zum Begriffskalkül (Codd, Chen, Wedekind/Ortner). Die Methoden der Datenanalyse sind hervorragend geeignet für eine objektorientierte Systemanalyse (vgl. z.B. Vetter 88, Strategie).

Die oben erwähnte hinreichende Bedingung für eine echte Objektorientierung ist ein **Vererbungsmechanismus**. Objektorientierung ist gut bekannt aus Programmiersprachen wie SIMULA 67, Smalltalk-80 und Eiffel (Meyer 90, SW-Entwicklung). Meyer spricht von **Klassen**, **Vererbung** und **Verweisen**. Hier die Analogien zu datentyporientierten Analysemethoden in knapper Form:

Begriffe der ... Programmiersprachen	datentyporientierten Analyse
Klassen	**Objekttype** mit Zugriffsoperationen
Vererbung	**Hierarchien** von Rollen eines Objekttyps
Verweise	erwünscht nur als **Operation**, nicht als direkter Zugriff ('references considered harmful')

Das Vererbungsprinzip ist in der Datenbankwelt von John und Diane Smith (77, Abstractions) in Form der Abstraktionsoperationen **Generalisierung** und **Aggregation** eingeführt und von Ortner (85, Begriffskalkül) wesentlich verfeinert worden. Die datenbankorientierten Methoden benutzen den Begriff **Vererbung** nicht oder wenig, da er den Kern des Problems in einer Analysephase nicht trifft: "Vererbung" suggeriert Hierarchie und top-down-Entwurf (s. Abb.3a). Der Entwurf von Datenmodellen geschieht jedoch fast immer bottom up durch Abstraktion. Dies ist das Gegenteil einer Verfeinerung. Der Verfasser hat ein Anfang 1989 veröffentlichtes Vorgehensmodell aus dieser Sicht "objektorientiert" genannt, obwohl das Wort Vererbung im gesamten Buch nicht vorkommt (Spitta 89, Engineering). Die Praktikabilität des Vererbungsprinzips wird weiter unten noch einmal aufgegriffen.

Objektorientierte SE-Methoden in diesem Sinne (Vererbung implizit in Datenmodellen) für die Analyse von Systemen beginnen sich erst langsam durchzusetzen (Coad 90, OOA; Spitta 89; Denert 91, (Engineering)) nachdem sie schon früh außerhalb des Einsatzbereiches realer Softwareentwicklung benutzt wurden (Ehrig 81, Stücklisten).

4.3 Bewertung

Transformationsorientierte Spezifikationsmethoden zwingen bei einer objektorientierten Modularisierung immer zu einer Umstrukturierung von Funktionen. In der **Sachlogik** sind **Daten** als Input oder Output an **Funktionen** gebunden, beim Softwareentwurf müssen Funktionen als Operationen den Datentypen zugewiesen werden. Dieser Strukturbruch führt zu

- Doppelaufwand
- Umsetzungsfehlern
- Unbestimmtheiten (mit dem Benutzer war etwas anderes abgestimmt als implementiert wird).

Daher muß objektorientiert analysiert und im Rahmen der dabei gefundenen Struktur spezifiziert werden, wenn man den Strukturbruch vermeiden will. Man muß vor der Spezifikation die zu entwickelnden Systeme von vornherein objektorientiert definieren. Dies wird in Kapitel 6 näher ausgeführt.

Die transformationsorientierten SE-Methoden haben noch weitere gravierende Mängel:
o sie berücksichtigen nur die Grundelemente Programme und Daten, nicht den Benutzer

(vgl. z.B. Hesse 84, Begriffssystem),

- o sie erhöhen durch das E-V-A - Prinzip die Komplexität der Spezifikation über das problembedingte Maß hinaus durch Erzeugung künstlicher Schnittstellen (Balzer 78, Informality).

Leider hat ein Paradigmenwechsel im Software Engineering (vgl. Floyd 83) bis heute nicht stattgefunden. Dies beweist die Aufnahme des Werkzeugs IEW in das strategische Softwarekonzept AD/cycle der Fa. IBM. Die Methodik von IEW ist transformationsorientiert. Die erschreckende Redundanz und der enorme Umfang der Grafiken lenkt den Leser vom Wesentlichen ab. Dies ist methodisch bedingt und keine Sache des Werkzeugs. Neueste Lehrgangsankündigungen von Yourdon Inc. lassen allerdings hoffen. Der "Vermarkter" von SA/SD hat eine radikale Kehrtwendung vollzogen. Die in diesem Papier geäußerten Kritikpunkte an SA/SD lassen sich inzwischen auch bei Coad/Yourdon nachlesen (Coad 90, OOA).

5. Grundelemente zur Konstituierung von Software

Wenn man objektorientierte Systeme erhalten will, müssen folgende Fragen zu den Grundelementen konstruktiv geklärt werden:

- o wie entstehen (Daten-)Objekte?
- o welche Operationen auf Objekten sind gemeint?
- o wer führt diese Operationen aus; wer entscheidet, welche ausgeführt werden?

5.1 Primäre und sekundäre Daten

Daten und Funktionen/**Programme**/Operationen werden allgemein als Grundelemente von Software betrachtet, sind also unstrittig. Alle Methoden legen sie zugrunde. Auch der **Benutzer** kommt heute immer mehr ins Gespräch, insbesondere bei Dialogsoftware. Er wird jedoch an die traditionellen Methoden angehängt und mehr oder weniger als Auftraggeber und Gesprächspartner für Abstimmungen betrachtet. Neben dieser Rolle ist der **Benutzer** jedoch **konstituierendes Element** von Software. Um diese Behauptung zu belegen, muß man die Entstehung von Daten näher betrachten.
In einer Datenbasis finden sich immer zwei grundverschiedene Kategorien von Daten:

- **primäre** oder auch originäre **Daten**
- **sekundäre** oder auch abgeleitete **Daten**.

Keine dem Autor bekannte SE-Methode oder auch Datenmodellierungstechnik trifft diese elementare Unterscheidung, obwohl sie sich wirkungsvoll bei der Konstituierung von Softwaresystemen verwenden läßt.

Primäre Daten sind solche, die von außen in eine Datenbasis eingebracht werden. Sie repräsentieren Fachwissen der Benutzer oder Meßwerte aus Maschinensystemen und werden dauerhaft gespeichert. Primäre Daten bilden die Grundlage für alle Prozesse, die in Form von Programmen abgebildet werden und für alle durch Programme daraus

abgeleiteten Daten.

Beispiel:

Teil#	Teil-Bezeichnung	MG-Basis	MG-Einheit
1324	Drehständer	1	Stück
2761	Mutter M6 Nirosta	1000	Stück

Alle logistischen Prozesse eines Unternehmens bauen auf solchen Teile-Stammdaten auf. Die Daten entstehen beim Benutzer.

Primäre Daten muß man dauerhaft speichern; sie sind gewissermaßen die "Wissensbasis" eines Softwaresystems.

Sekundäre Daten sind alle übrigen Daten, die durch Auswertung, Berechnung o.ä. aus primären entstehen. Im Prinzip bräuchte man gar keine sekundären Daten zu speichern, wenn man genügend Hardwareleistung zur Verfügung hätte, um sie jederzeit aktuell zu berechnen. Es ist sogar gefährlich, sekundäre Daten zu speichern, da immer die Gefahr der Inkonsistenz zu den primären besteht. Es ist jedoch offensichtlich nicht praktikabel, jederzeit alles neu zu berechnen, wenn man z.B. bedenkt, daß eine Vertriebsstatistik, abgeleitet aus 15 Mio Datensätzen, selbst auf sehr großen Rechnern mehrere Stunden Laufzeit beansprucht. Manchmal ist dies auch fachlich nicht erwünscht, z.B. wenn man mit der oben erwähnten Vertriebsstatistik Wochenwerte und nicht zufällige tägliche Schwankungen sehen will. In jedem Fall hat die Speicherung sekundärer, meist verdichteter Daten unerwünschte Nebeneffekte. Man kann z.B. eine fehlerhafte Buchung nach der Verdichtung nicht mehr korrigieren, ohne die Verdichtung zu wiederholen.

Ein großer Teil des Aufwandes in Softwaresystemen entsteht durch die Aufgabe, im Falle von Störungen die Konsistenz der sekundären zu den primären Daten sicher- oder wiederherzustellen.

Für die Konstituierung einer Datenbasis sind also nur die primären Daten relevant und zwar nur deren **Erzeugung**. Erzeugt werden Daten durch folgende vier Operationen:

- **anlegen** von Objekten (auf Schlüsselbasis),
- **ergänzen** von Attributen zu Objekten,
- **ändern** von Objekten und ihren Attributen,
- **löschen** von Attributen oder ganzen Objekten.

Alle **lesenden Operationen** sind für die Entstehung von Systemen irrelevant, da sie die Integrität der primären Daten nicht beeinflussen können.

5.2 Benutzer und Organisation

Nach der Differenzierung der Daten und der dazugehörenden Operationen muß auch "der Benutzer" genauer betrachtet werden.

Es ist nicht möglich, alle fachlichen Fehler, die ein Benutzer beim Erzeugen primärer Daten machen kann, durch Prüfungen in der Software zu verhindern. Gibt etwa ein Benutzer versehentlich für die Nirosta-Mutter im Beispiel die Mengenbasis 100

statt 1000 Stück ein, wird jede darauf aufbauende Berechnung falsch, etwa eine Nettobedarfsrechnung mit Stücklistenauflösung. Daher werden an den Benutzer Ansprüche bezüglich seiner Qualifikation und seiner Bereitschaft gestellt, verantwortlich zu handeln.

So wie **Software Engineering** nicht auf ein Einplatz-PC-System für ein single tasking Betriebssystem abzielt, ist hier mit Benutzer nicht der dazugehörige Individualanwender gemeint. Wenn dieser einen Fehler macht, hat er den Schaden selbst. Also wird er meist (?) versuchen, einmal gemachte und erkannte Fehler zu vermeiden, also im Eigeninteresse verantwortlich zu handeln.

Wenn ein Mitarbeiter in einem Unternehmen, etwa in der Entwicklung oder im Marketing, einen Teilestammsatz falsch anlegt, hat er keinen unmittelbaren Schaden, obwohl er möglicherweise eine logistische Katastrophe auslöst. Mit Benutzer meinen wir den in eine Organisation eingebundenen, verantwortlich und fachkundig handelnden Benutzer, kurz: den **organisierten Benutzer**. Datenerfasser früherer (?) Zeiten aus der Lochkartentechnik sind in unserem Sinne keine Benutzer.

5.3 Zusammenfassung

Grundelemente zur Konstituierung von Software sind:

- primäre Daten,
- erzeugende Operationen,
- organisierte Benutzer.

Die Wichtigkeit sog. **Erzeugungsprozesse** ist schon sehr lange bekannt in der sog. IBM-Verfahrenstechnik (IBM 76, VT). Sie lebt auch in neuesten Veröffentlichungen fort (Vetter 88, Strategie). Leider wurde die entscheidende Einschränkung auf primäre Daten und ihre Entstehung nicht gesehen und daher die Technologie sehr aufwendig. Nur ganz wenige Firmen treiben den damit verbundenen Aufwand, die extrem großen Tabellen für diese Erzeugungsprozesse zu erstellen und maschinell zu verwalten. Sie werden so groß, weil die Erzeugung primärer und sekundärer Daten nicht getrennt wird.

Durch die Beschränkung der Grundelemente auf das Wesentliche wird es möglich, einer der wichtigsten Forderungen des Software Engineering nachzukommen, nämlich der **Komplexitätsreduktion** durch Methoden.

Eine objektorientierte Technologie kann sich für die Konstituierung von Systemen, d.h. für das Strukturieren in Teilsysteme und das Minimieren von Schnittstellen zwischen ihnen, auf wenige Elemente beschränken, die sich **konstruktiv statt intuitiv** verknüpfen lassen.

6. Die Konstituierung von Softwaresystemen

Die **Konstituierung** von Softwaresystemen ist als **Phase Systemabgrenzung** vor der Spezifikation einzuordnen:

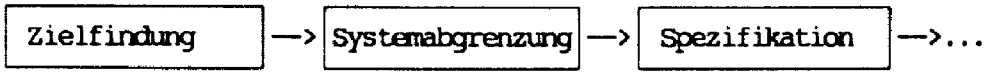

Abb.5: Konstituierung eines Systems als Phase Systemabgrenzung

Nach einer informalen Phase **Zielfindung**, in der ermittelt und abgestimmt wird, was man überhaupt will und ob man überhaupt Software braucht, muß in einer Phase **Systemabgrenzung**, früher Grobkonzept genannt, folgendes geleistet werden:

- o **Objekttypen für die relevanten primären Daten** finden,
- o **verantwortliche Benutzer** bzw. Organisationseinheiten (= Stellen) finden, die diese Daten erzeugen,
- o **je Stelle die erzeugenden Operationen** auf Typen und Attributen feststellen.

Es muß die Erzeugung aller primären Daten (= Objekttype mit Attributen) gefunden werden, damit das System keine Importschnittstellen primärer Daten aufweist. Daher wird der Name Systemabgrenzung der ebenfalls denkbaren Bezeichnung Systembildung vorgezogen.

Wie man Objekttypen bzw. Relationen findet, kann hier nicht ausgeführt werden. Es gibt anerkannte und praktisch bewährte Darstellungen zur Datenmodellierung (vgl. Chen 76, Entity-Relationship; Codd 79, Extending RM; Wedekind 81, DB-Systeme I; Ortner 83, Konstruktionssprache), auf die verwiesen wird. Man beachte, daß Codd mit seiner Normalisierungslehre genau dieselben Ziele verfolgt (und auch konstruktiv erreicht!), die mit den softwaretechnischen Zielen **maximale Bindung** und **minimale Kopplung** (= minimale Schnittstellen) verfolgt werden: Objekte sind änderungsfreundlich und fernwirkungsarm. Ein ohne normalisiertes Datenmodell entworfenes Softwaresystem kann niemals objektorientiert werden, da helfen weder Smalltalk, noch -mit Einschränkungen- ADA, noch C++.

Ein objektorientiertes System besteht aus einem oder mehreren Objekttypen einer zentralen Datenbasis, die zunächst nur aus primären Daten besteht. Das System ist eingebettet in diejenige Organisationseinheit, die die Objekte der Objekttypen erzeugt. Da häufig einzelne Objekttype keine sinnvolle Basis für Anwendungen (= erzeugte Daten) ergeben, muß man zusammengehörende Objekttype zu **Teilsystemen** zusammenfassen. Diese werden **Vorgangsketten** genannt, weil bei vielen Objekttypen nach dem Anlegen eines Objekts durch die verantwortliche Stelle die übrigen Daten von anderen Stellen ergänzt werden, die für die Attribute verantwortlich sind. Das Objekt ist erst vollständig, wenn dieser Vorgang beendet ist. Vorgangsketten sind also Änderungsoperationen datenverantwortlicher Stellen auf sachlogisch zusammengehörenden Objekttypen. Eine differenzierte Behandlung von **Grundobjekttypen** (bekannt als "Stammdaten") und **Vorgangsobjekttypen** (bekannt als "Bewegungsdaten") kann hier nicht geleistet werden (vgl. Spitta 89, Engineering). Dort ist auch die praktische Handhabung von Datenmodellen dargestellt.

Die in dem zitierten Buch genauer dargelegte Methode zerfällt in einen groben Teil und einen detaillierten:

1. **Grobentwurf**: Finden von Objekttypen nach dem Entity-Relationship-Modell (**graphisch**) und nach dem Begriffskalkül. Er-

stellen eines Aufgabenmodells (Tätigkeiten von Stellen) nach der Methodik von Kosiol (62, Organisation).

2. **Feinentwurf**: Ergänzen der Attribute nach dem Relationenmodell mit Normalisierung (**verbal**). Notieren der erzeugenden Operationen (= Elementaraufgaben).

Man beachte, daß man bei der Bildung von Objekttypen immer wichtige Attribute wenigstens implizit mitbetrachtet, da Objekttype erst durch ihre Attribute definiert sind (Näheres in den zitierten Büchern von Wedekind und Ortner).

Ziel der Vorgehensweise ist eine Systembildung und -abgrenzung mit 1.) wenig Aufwand und 2.) so präzise wie nötig. Im Bild:

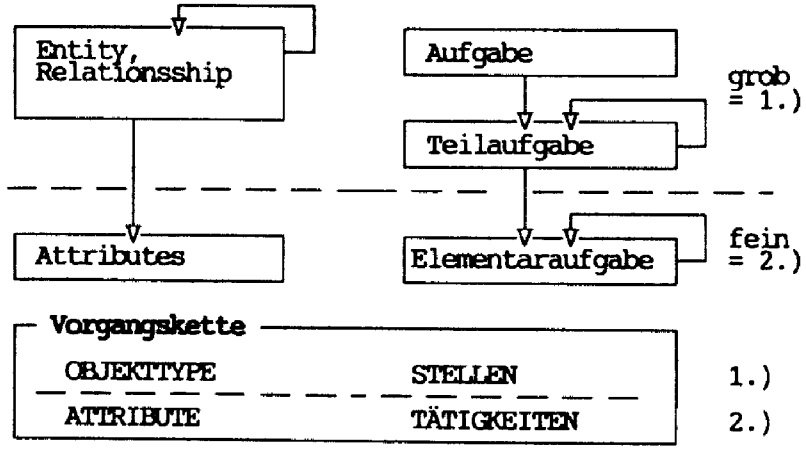

Abb.6: Schema der Systembildung mit Vorgangsketten

Diese Methode zur Konstituierung von Softwaresystemen bricht mit einem verbreiteten Paradigma des Software Engineering: Es wird nicht top down entworfen. Der Autor bezweifelt entschieden, daß es möglich ist, Datenmodelle top down zu entwerfen, wie dies auch in neuerer Zeit noch behauptet wird (vgl. Vetter 88, Strategie, S.239ff.; aber auch: Jackson 83, JSD). Viemehr findet man höher in einer Hierarchie angesiedelte Objekttype eher bottom up. Sie werden durch den Konstruktionsschritt **Generalisierung** oder auch **Inklusion** gefunden (vgl. hierzu Ortner 85, Begriffskalkül, S.23f.).

Zur Erläuterung sei ein Teil des Beispiels aus Abb.3b als Relationenmodell notiert:

BUCHBESTAND (Wirtschaftsjahr#, Anzahl-Titel, Bestands-Wert)
FACHGEBIET (Wirtschaftsjahr#, Gebiets#, Gebietsname, Anzahl-Titel, Bestands-Wert)
BUCH (Inventar#, Gebiets#, Autor, Titel, Verlag, ..., Preis)

Der Leser beachte, daß in der Hierarchie des Abb.3b die wichtigste Relation fehlt, weil man sie bei einem top-down-Entwurf nicht findet:
AUSLEIHE (Inventar#, Benutzer#, Ausleih-Datum, Rückgabe-Datum, ...)

Die in der Hierarchie höher gelegenen Relationen enthalten nur abgeleitete Daten. Sie

vererben nichts auf die Elementarrelation BUCH, im Gegenteil: sie erben (= bottom up) primäre Daten in verdichteter Form. Die hierzu erforderlichen Operationen (in Smalltalk **Methoden**) gehören zum hierarchisch höher gelegenen Objekt und nicht zum tiefer gelegenen.

7. Zusammenfassung

Der Beitrag hatte drei Ziele:

1. die Systembildung als wichtigste konstitutive Entscheidung des Software Engineering hervorzuheben und methodisch abzusichern,
2. die Durchgängigkeit objektorientierter Techniken zu untermauern und ihre Anwendung in Ansätzen zu zeigen,
3. die Notwendigkeit einer einheitlichen Sicht von Software und Data Engineering zu unterstreichen und damit deutlich zu machen, daß ein Paradigmenwechsel im Software Engineering überfällig ist.

Zur Verdeutlichung werden die wichtigsten Aussagen wiederholt:

1. **Systeme sind zu konstruieren**, sie "fallen nicht vom Himmel". Die Systembildung ist die folgenschwerste Entwurfsentscheidung bei der Softwareentwicklung.
2. Ein **top down-Entwurf** von Softwaresystemen ist **nicht möglich**. Wird er versucht, kann dies nur intuitiv statt konstruktiv erfolgen.
3. Eine **Spezifikation** kann **nur auf Basis eines konstruierten Systems** Grundlage einer gleich strukturierten Implementierung sein.
4. **Transformationsorientierte Spezifikationsmethoden** (SA, SADT, RSL, ISAC u.ä.) sind zu **"entsorgen"**.
5. **Prototyping** (s. eig. Zitat) hat **in der Phase Systemabgrenzung** nichts zu suchen, wäre sogar **schädlich** (dies ist bei der Spezifikation völlig anders, jedoch hier nicht Thema!).

Für die Unterstützung und wichtige Hinweise sei Ernst Denert, München, und Peter Löhr, Berlin, gedankt. Zum Abschluß jedoch: Das Buch **Software Engineering und Prototyping** ist auf Anregung von Reinhold Franck geschrieben worden.

Literatur

Balzer 78, Informality: Balzer,R., Goldman,N., Wile,D.: Informality in Program Specifications, in: IEEE Trans. on SE 4(1978) No.2, pp.94-103.

Balzer 79, Specification: Balzer,R., Goldman,N.: Principles of Good Software Specification and Their Implications for Specification Language, in: IEEE Proc. Conf. on Specification of Reliable Software, pp.58-67. Cambridge/Mass. 1979

Balzert 81, Methoden: Balzert,H.: Methoden, Sprachen und Werkzeuge zur Definition, Dokumentation und Analyse von Anforderungen an Software-Produkte. In: Informatik-Spektrum 4 (1981) H.3, S.145-163 und H.4, S.246-260.

Balzert 82, Entwicklung: Balzert,H.: Die Entwicklung von Software-Systemen. Reihe Informatik Bd. 34. BI, Mannheim - Wien - Zürich 1982.

Chen 76, Entity-Relationship: Chen,P.P.-S.: The Entity-Relationship Model - Toward a Unified View of Data, in: ACM Trans. on Database Systems 1(1976) No.1, pp.9-36.

Coad 90, OOA: Coad, P., Yourdon,E.: Object-Oriented Analysis. Yourdon Press/Prentice Hall, Englewood Cliffs N.J. 1990.

Codd 70, Relational Model: Codd,E.F.: A Relational Model of Data for Large Shared Data Banks, in: CACM 13 (1970) No.6, pp.377-387.

Codd 79, Extending RM: Codd,E.F.: Extending the Database Relational Model to Capture More Meaning, in: ACM Trans. on Database Systems 4(1979) No.4, pp.397-434.

Denert 91, Engineering: Denert,E.: Software-Engineering. Springer, Berlin - Heidelberg - New York et.al. 1991.

Ehrig 81, Stücklisten: Ehrig,H., Fey,W., Kreowski,H.-J.: Algebraische Spezifikation eines Stücklistensystems - eine Fallstudie, in: Floyd 81, Spezifikation, S.75-90.

Floyd 81, Spezifikation: Floyd,C., Kopetz,H. (Hrsg.): Software Engineering - Entwurf und Spezifikation. Teubner, Stuttgart 1981.

Floyd 83, Paradigmenwechsel: Floyd,C.: Grundzüge eines Paradigmenwechsels in der Softwaretechnik. Kolloquium "Information, Organisations- und Informationstechnologie", 13.-15.12.83. Humboldt-Universität Berlin/Ost 1983.

Floyd 84, Methoden: Floyd,C.: Eine Untersuchung von Softwareentwicklungsmethoden, in: Morgenbrod 84, Programmierumgebungen, S.248-274.

Guttag 77, Data Types: Guttag,J.: Abstract Data Types and the Development of Data Structures, in: CACM 20(1977) No.6, pp.396-404.

Hesse 81, Methoden: Hesse,W.: Methoden und Werkzeuge zur Software-Entwicklung - Ein Marsch durch die Technologie-Landschaft; in: Informatik-Spektrum 4(1981) H.4, S.229-245.

Hesse 84, Begriffssystem: Hesse,W., Keutgen,V., Luft,A.L., Rombach,H.D.: Ein Begriffssystem für die Softwaretechnik - Vorschlag zur Terminologie, in: Informatik-Spektrum 7(1984) H.4, S.200-213.

IBM 76, VT: IBM (Hrsg.): Verfahrenstechnik, div. Broschüren: IBM-Form SR 12-1675-0, SR 12-1657-0, F 12-008 (11/76), GL 20-1851, GO 12-1049-0, IBM Corporation 1976 ff.

Jackson 83, JSD: Jackson,M.A.: System Development. Prentice Hall, Hempstead 1983.

Kosiol 62, Organisation: Kosiol,E.: Organisation der Unternehmung. Gabler, Wiesbaden 1962.

Kreowski 81, Spezifikation: Kreowski,H.-J.: Algebraische Spezifikation von Softwaresystemen, in: Floyd 81, Spezifikation, S.46-74.

Liskow 74, ADT: Liskow,B.H.; Zilles,S.N.: Programming with Abstract Data Types, in: SIGPLAN Notices 9(1974), No. 4, pp.50-59.

Lundeberg 80, ISAC: Lundeberg,M.: ISAC - Specification of Information Systems in Order to Support the Activities of Programme and Organizing Committees for IFIP Working Conferences, Specialist Report TRITA-IBADB-4102, IFIP, Stockholm 1980.

Meyer 90, SW-Entwicklung: Meyer,B.: Objektorientierte Softwareentwicklung. Hanser, München - Wien und Prentice Hall, London 1990.

Morgenbrod 84, Programmierumgebungen: Morgenbrod,H., Sammer,W. (Hrsg.): Programmierumgebungen und Compiler. Teubner, Stuttgart 1984.

Orr 77, Design: Orr,K.T.: Structured Systems Development. Yourdon Inc., New York 1977.

Ortner 83, Konstruktionssprache: Ortner,E.: Aspekte einer Konstruktionssprache für den Datenbankentwurf. Toeche-Mittler, Darmstadt 1983.

Ortner 85, Begriffskalkül: Ortner,E.: Semantische Modellierung - Datenbankentwurf auf der Ebene der Benutzer, in: Informatik-Spektrum 8(1985) H.1, S.20-28.

Parnas 72, Criteria: Parnas,D.C.: On the Criteria to be Used in Decomposing Systems into Modules, in: CACM 15(1972) No.12, pp.1053-1058.

Parnas 84, Modular Structure: Parnas,D.L.: The Modular Structure of Complex Systems, in: 7th ICSE, pp.408-417. Orlando/Florida 1984.

Schelle 83, Software-Entwicklung: Schelle,H., Holzberger,P. (Hrsg.): Psychologische Aspekte der Software-Entwicklung. Oldenbourg, München 1983.

Schnupp 83, Softwaretechnologie: Schnupp,P.: Softwaretechnologie für den kommerziellen Anwender - Bringen die 80er Jahre einen Paradigmenwechsel? in: Schelle 83, Software-Entwicklung, S.156-171.

Schulz 90, Entwurf: Schulz,A.: Software-Entwurf, 2.Aufl. Oldenbourg, München - Wien 1990.

Smith 77, Abstractions: Smith,J.M., Smith,D.C.P.: Database Abstractions: Aggregation and Generalization, in: ACM Trans. on Database Systems, 2(1977) No.2, pp.105-133.

Spitta 89, Engineering: Spitta,Th.: Software Engineering und Prototyping. Springer, Berlin - Heidelberg - New York - Tokyo 1989.

Vetter 88, Strategie: Vetter,M.: Strategie der Anwendungssoftware-Entwicklung. Teubner, Stuttgart 1988.

Wasserman 79, Engineering View: Wasserman,A.I.: A Software Engineering View of Data Base Management, in: Weber 79, DB-Management, pp.41-63.

Weber 79, DB-Management: Weber,H., Wasserman,A.I. (eds.): Issues in Database Management. North-Holland, Amsterdam - New York - Oxford 1979.

Wedekind 81, DB-Systeme I: Wedekind,H.: Datenbanksysteme I, 2.Aufl. Bibliographisches Institut, Mannheim - Wien - Zürich 1981.

Yourdon 76, SD: Yourdon,E., Constantine,L.L.: Structured Design. Prentice Hall, Englewood Cliffs/ N.J. 1976.