Grounded Theory und Systemanalyse in der Informatik

Engelmeier, Gregor

Veröffentlichungsversion / Published Version
Sammelwerksbeitrag / collection article

Empfohlene Zitierung / Suggested Citation:
Engelmeier, G. (1994). Grounded Theory und Systemanalyse in der Informatik. In A. Boehm, A. Mengel, & T. Muhr (Hrsg.), *Texte verstehen : Konzepte, Methoden, Werkzeuge* (S. 141-158). Konstanz: UVK Univ.-Verl. Konstanz. https://nbn-resolving.org/urn:nbn:de:0168-ssoar-14469

Grounded Theory und Systemanalyse in der Informatik

Gregor Engelmeier
Technische Universität Berlin

1 Einleitung

In der Informatik herrscht - wie in den meisten Ingenieurwissenschaften - die "Produktionssicht" der Softwareentwicklung vor. Dabei wird das Hauptaugenmerk auf die formalen Aspekte der Produktion von Software gelegt und die empirischen Anteile dieses Prozesses werden in der Regel vernachlässigt. Ziel dieses Artikels ist es, die Rolle textinterpretativer Tätigkeiten in der Softwareentwicklung zu reflektieren. Der Schwerpunkt liegt dabei auf einer Untersuchung der Rolle die der Grounded Theory Ansatz (Glaser & Strauss, 1967) hierbei in der Systemanalyse spielen könnte.

Der Prozeß des Designs von Computerprogrammen scheint auf den ersten Blick recht verschieden von den Theoriebildungsprozessen in den Sozial- und Geisteswissenschaften zu sein, von denen in den anderen Artikeln dieses Bandes die Rede ist.

In den letzten Jahren hat sich jedoch mehr und mehr die Erkenntnis durchgesetzt, daß Softwaredesign im Kern ein kooperativer Theoriebildungsprozeß über einen von späteren Produkt (Programm) zu modellierenden Ausschnitt der Wirklichkeit ist, und daß Verfahren aus den Gesellschaftswissenschaften wertvolle Beiträge zur methodischen Kontrolle dieses Prozesses leisten können. (Knuth 1984, Curtis 1988, Winograd 1988). Ein der Problemstellung adäquates Programm ist hiernach ein Modell der untersuchten Realität.

Die zentrale These dieses Artikels ist, daß die Methoden des systematischen Textverstehens, wie sie im ATLAS-Projekt entwickelt wurden und wie sie sich in Softwarewerkzeugen wie ATLAS/ti niederschlugen, eine wesentliche Unterstützung für den Softwareentwicklungsprozeß darstellen können. Sie ermöglichen es, den sonst unstrukturiert und intuitiv ablaufenden Theoriebildungsprozeß der Softwareentwicklung transparent zu machen und zwingen den Entwickler zur Reflektion über diesen Prozeß.

Speziell die frühen Phasen der Softwareentwicklung (die "Requirements Analysis") stellen sich häufig als ein Prozeß der "Taxonomierung" des Anwendungsgebietes des angestrebten Programmsystems dar. Dabei werden zunächst nur vage erfaßte Begriffe und Anforderungen wiederholt präzisiert und verfeinert. In diesem Prozeß der Exploration des Gegenstandsbereiches wird eine vorläufige Theorie des

Gegenstandsbereiches des Softwareentwicklungsprojektes entwickelt, die während des gesamten Entwicklungsprozesses zunehmend verfeinert und formalisiert wird.

Diesem Artikel liegt eine Sicht des Softwareentwicklungsprozesses als wiederholter Textinterpretation zugrunde. Der Gegenstandsbereich (die Domain) eines Entwicklungsprojektes in der Softwaretechnik tritt dem Entwickler in der Regel textvermittelt in Form von Dokumenten, Interviewnotizen, Diagrammen und ähnlichem entgegen. Die Aufgabe der Softwareentwicklung ist es, diese Texte in einer Art und Weise zu interpretieren, daß als Endprodukt eine automatisch interpretierbare, formale Beschreibung (das "Programm") erzeugt werden kann.

2 Textinterpretation in Software Engineering und Systemanalyse

Systemanalyse wird hier als eine dem Software Design vorgelagerte Tätigkeit im Rahmen der Softwareentwicklung gesehen: "The first stage in establishing a system specification is to formulate a model of the 'real-world' entities which are to be represented in the system" (Sommerville 1989, S. 65).

Es wird gezeigt, daß die vorgestellten "Softwarelifecycle"-Modelle der Softwareent wicklung textinterpretative Tätigkeiten in großem Umfang implizieren, ohne für diese Tätigkeiten selbst methodische Unterstützung anzubieten. Diese These stimmt mit den Ergebnissen einer Feldstudie von 17 größeren Softwareprojekten in den USA überein, die 1990 veröffentlicht wurde. Curtis schreibt dort:

> "Our interviews indicated that developing large software systems must be treated, at last in part, as a learning, negotiation and communication process. These processes are poorly described in software process models that focus instead on how a software product evolves trough a series of artifacts such as requirements, functional specifications, code and so on."(Curtis 1990, S.21).

Die Tätigkeiten des Lernens, Verhandelns und Kommunizierens haben aber weitgehend die Charakteristika textinterpretativer Tätigkeiten.

2.1 Interpretation in Software-Lifecycle-Modellen

Das klassische "Softwarelifecycle"-Modell ist das in Abb. 1 dargestellte sogenannte "Wasserfallmodell" der Softwareentwicklung. Es wurde vermutlich zuerst in Royce

(1970) vorgeschlagen[1] und gliedert den Softwareentwicklungsprozeß in streng aufeinander folgende Phasen.

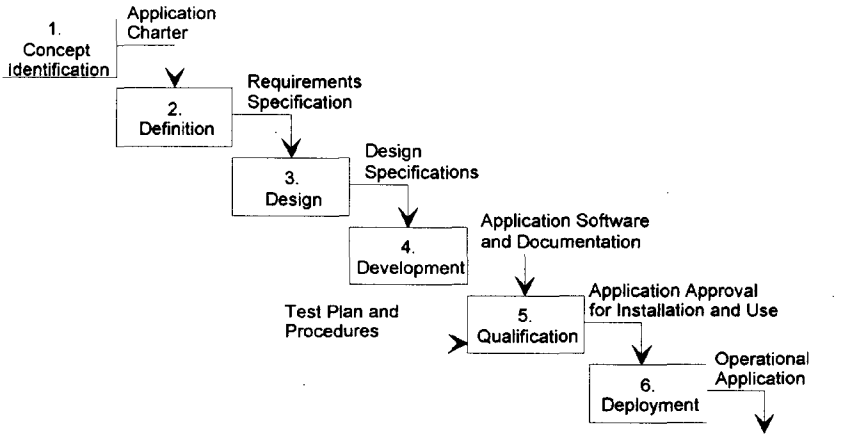

Abb. 1: Classical Software Development Cycle (nach ([MARTIN88], S. 12).

Die erste Phase - die Anforderungsdefinition[2] (concept identification) - etabliert die Anforderungen an das zu entwickelnde Produkt durch Machbarkeits-, Organisations- und Planungsstudien sowie nicht zuletzt durch die Überzeugung der wesentlichen Entscheidungsträger des Einsatzbereiches (Management) von der Zweckmäßigkeit des Projektes. Das Ergebnis dieses Prozesses - die Application Charter (Anforderungsdefinition, häufig auch das Pflichtenheft als juristisches Dokument) wird als Input der Definition-Phase gesehen[3]. Diese transformiert die informellen Anforderungen der Anforderungsdefinition in formelle funktionale Anforderungen, die im eigentlichen Design (Softwareentwurf) in Designspezifikationen umgesetzt werden. In dieser Phase wird die Struktur von Hard- und Software des Systems festgelegt und in formellen Revisionssitzungen diskutiert und validiert.

Die nächste Phase - das Development (Programmentwicklung) - produziert nun die "sichtbaren" Produkte des Entwicklungsprozesses, also Programme, Datenbanken, Manuale etc. Dabei wird die Designspezifikation nicht mehr verändert. Die so produzierten Produktkomponenten werden in der Qualifikations-

[1] siehe Sommerville 1989, S. 7.

[2] Die deutschen Bezeichnungen sind Floyd (1989-1) entnommen und entsprechen Martins Phasen ungefähr.

[3] Floyd spricht zusammenfassend von den Prozessen 2 und 3 als "Systemgestaltung".

phase (Funktions- und Leistungsüberprüfung) mit den in Phase 2 spezifizierten Anforderungen an die Software verglichen und systematischen Tests unterworfen.

Schließlich wird in der Deployment-Phase das Produkt in den Einsatzbereich eingebettet und die Benutzer in der Handhabung des Systems geschult.

Bezeichnenderweise läßt Abb. 1 eine Phase, die häufig besondere Probleme aufwirft - die Programmwartung (siehe die Diskussion von Naurs "Theory Building View" im folgenden Kapitel dieser Arbeit) - aus. In dieser Phase müssen Programmsysteme oft über viele Jahre an sich ändernde technische und organisatorische Rahmenbedingungen angepaßt werden.

In diesem Modell des Softwarelifecycles basiert, außer der Concept Identification, jede Phase auf von der vorhergehenden Phase produzierten Dokumenten, die als nachträglich nicht mehr veränderbar angesehen werden. Diese Dokumente werden in der nächsten Phase interpretiert und die so gewonnene Interpretation wird wiederum in ein neues Dokument überführt. Dabei wird die Art und Weise der Interpretation von der Antizipation des Zwecks der nächsten Phase bestimmt. In diesem Modell ist die Interaktion mit den zukünftigen Benutzern bzw. dem vom Programm zu modellierenden Realitätsausschnitt auf die ersten beiden Phasen beschränkt. Es wurde in diesem Zusammenhang sogar diskutiert, ob die Interaktion der in den Phasen drei bis fünf eingesetzten Entwickler mit den Benutzern nicht kontraproduktiv sei.

Grundsätzliche Kritik an Phasenmodellen der Softwareentwicklung (in den 70er und 80er Jahren wurden diverse Modifikationen des klassischen Wasserfallmodells vorgeschlagen) wurde - aufbauend auf Naurs Analyse der Softwareentwicklung als Theoriebildung[4] - in den 80er Jahren geübt. Das Aufkommen objektorientierter Programmiertechniken ließ Modellierungs- und Wiederverwendungstechniken eine größere Bedeutung zukommen. Wiederverwendung impliziert aber Programm- und Codewartung über lange Zeiträume, so daß Naurs Kritik immer größere Relevanz zuwuchs.

Die Phasenmodelle stellten die Produktionsschritte bei der Softwareentwicklung in den Vordergrund und vernachlässigten dabei den für die Softwareentwicklung notwendigen Erkenntnisfortschritt der beteiligten Entwickler. Floyd et al. schreiben dazu:

> "Die Grenzen der Produktionssicht ergeben sich insbesondere aus den mit dem Phasenmodell verbundenen Ansprüchen. Dieses verknüpft [...] die Gliederung des Herstellungsprozesses in Teilaufgaben in Form von Phasen mit dem Anspruch einer entsprechenden Gliederung der kooperativen Erkenntnisprozesse der Beteiligten. Diese Aspekte sind zu trennen.
>
> [...] Wo Phasenmodelle buchstäblich angewendet werden, sind Entwickler und Benutzer gezwungen, um das Modell herumzuarbeiten. Wird die Kommunikation zwischen Entwicklern und Benutzern, wie im Modell vorgesehen, auf Anfangsphasen beschränkt, so ergeben sich häufig erhebliche sogenannte Akzeptanzprobleme bei der Auslieferung des Systems, weil es sich als nicht relevant erweist.

[4] Siehe Abschnitt 2.2 dieses Artikels.

Phasenmodelle liefern statische Beschreibungen der Softwareentwicklung vor oder nach der jeweiligen Projektsituation, sie reichen nicht aus zum Verständnis der dynamisch stattfindenden Prozesse und als Grundlage ihrer Koordination in der spezifischen Projektsituation." (Floyd 1989-2, S. 21).

Im weiteren schlägt Floyd dann ein an der Design-Sicht der Softwareentwicklung orientiertes Projektmodell vor, das sich in folgenden Punkten von den Phasenmodellen unterscheidet:

- von einer vordefinierten Vorgehensweise zu einem Projektmodell als Rahmen, der in der konkreten Situation mit der jeweils relevanten Strategie auszufüllen ist,
- von einer linearen zu einer zyklischen Herangehensweise, bei der von vornherein die Softwareentwicklung in Versionen gesehen wird, und Herstellung und Einsatz verschränkt sind,
- von einer nicht einlösbaren TOP-DOWN-Strukturierung des Entwicklungsprozesses (im Gegensatz zur Strukturierung des Produktes) zu einer schrittweisen Vorgehensweise, die neben Verfeinerung und Formalisierung auch Revisionen aufgrund weitergehender Erkenntnisse oder geänderter Anforderungen zuläßt,
- von der Verringerung der Kommunikation und frühzeitigen Arbeitsteilung aufgrund definierender Dokumente zur Betonung von Kooperation, Kommunikation und der Entwicklung eines gemeinsamen "Geists",
- von der einmaligen Herstellung von Dokumenten und Programmen zu ihrer versionsorientierten Entwicklung und inkrementellen Fortschreibung,
- von der idealerweise ausschließlichen Verständigung anhand von Dokumenten zur gemeinsamen Erfahrung anhand von Vorversionen, die ausgewertet werden können. (Floyd 1989-2, S.22f).

Die methodische Umsetzung dieser Merkmale findet sich im Projektmodell von STEPS[5].

[5] Software Technik für Evolutionäre Partizipative Systementwicklung.

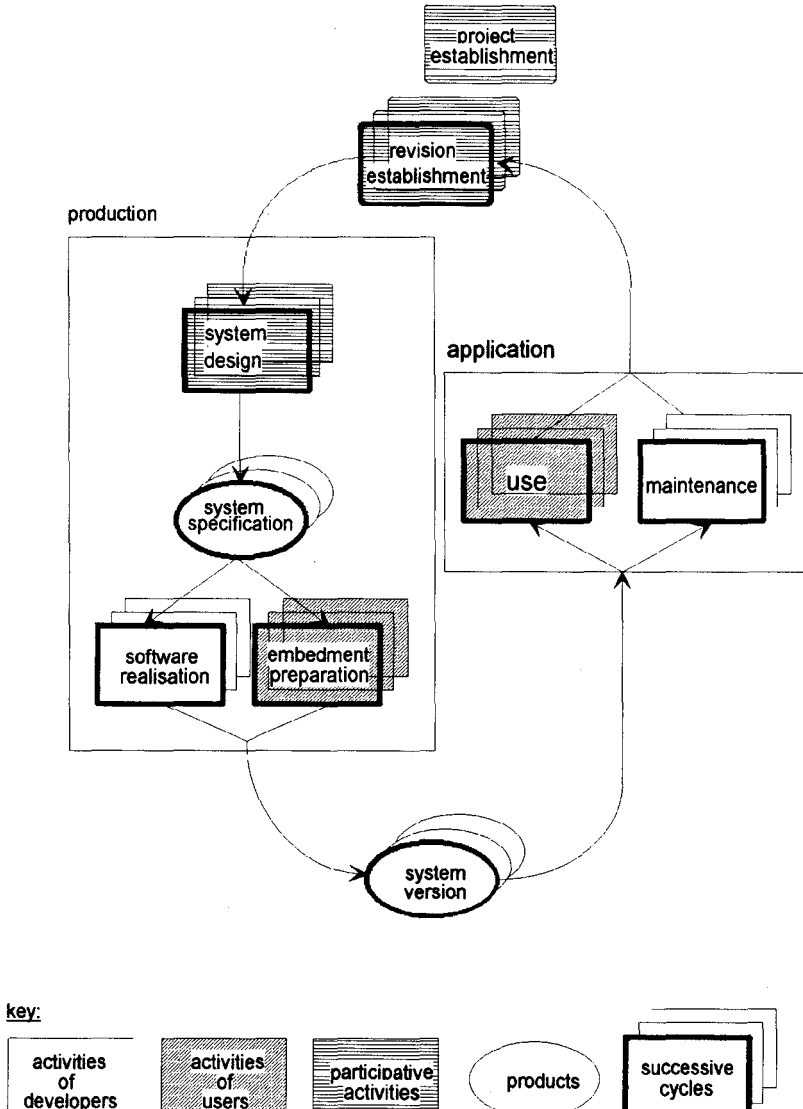

Abb. 2: Projektmodell Steps (nach [FLOYD89-1], S. 57).

Das Modell wird von Floyd wie folgt charakterisiert:

"In view of the wide variety of development situations, a project model reflecting generalized knowledge can provide only a framework for describing the domains of discourse within which software development takes place [...]. In STEPS we distinguish domains of discourse pertaining to software development as follows:

Requirements are anchored in the world of user work processes.

System functions and architecture arise as a result of design and draw on modelling concepts from the real world.

Programs serve to control the computer and thus from the connection to the technical world of realization means."(Floyd 1989-1, S. 59).

Allerdings legt auch Floyd keine Methoden dar, die die geforderte Verankerung der Anforderungen und damit des gesamten Entwicklungsprozesses in den unmittelbar der Benutzerinteraktion entstammenden Dokumenten sicherstellt.

Das weiter unten vorgestellte Modell der gegenstandsbezogenen Softwareentwicklung könnte das Projektmodell STEPS in bezug auf diese Verankerung der Softwareentwicklung in den Erhebungsdaten unterstützen und Konkretisieren.

2.2 Naurs "Theory Building View" der Softwareentwicklung

Als einen Ansatz zur Lösung der oben dargestellten Problematik formulierte Mitte der 80er Jahre Peter Naur in einer Reihe von Aufsätzen über die Grundlagen der Programmierung eine Gegenposition zur Produktionssicht der Softwareentwicklung (siehe Naur, 1992). Er sieht dabei das Hauptproblem der Softwareentwicklung weniger in der Produktion von immer neuen Dokumenten und ihrer Verwaltung. Vielmehr ist er der Ansicht, das Softwareentwicklung eine schrittweise Exploration des mit dem späteren Programmsystem zu unterstützenden Ausschnitts der Wirklichkeit durch den Softwareentwickler darstellt. Das Ziel des Softwareentwicklers ist dabei, sich eine Theorie des Gegenstandsbereich zu bilden, die ihn befähigt, auf Anforderung bestimmte Dokumente erstellen zu können. Dabei ist dieses Wissen orginärer "Besitz" des Softwareentwicklers und kann als Ganzes nicht dokumentiert werden. Die vom Entwickler erstellten Dokumente geben immer nur die operationalisierten Aspekte dieses intuitiv erworbenen Wissens wieder. Aufgrund dieser Überlegungen äußert sich Naur sehr kritisch gegenüber Softwareentwicklungsmethoden, wie sie weiter oben kurz dargestellt wurden. Eine Methode der Softwareentwicklung definiert Naur als ein System von Regeln, welches dem Entwickler vorschreibt, welche Tätigkeiten er in welcher Reihenfolge ausführen und welche Notationen er für die Dokumentation dieser Tätigkeiten benutzen soll, um zu einem qualitativ befriedigenden Softwaresystem zu kommen. Doch aus dem oben gesagten folgt für Naur: "... that on the Theory Building View, for the primary activity of the programming there can be no right method." (Naur 1985-1, S.46).

Allerdings räumt er ein, daß eine Methode im Sinne eines lockeren Leitfadens für verschiedene Vorgänge bei der Softwareentwicklung durchaus nützlich sein könne

(Naur 1985-2). Dabei müsse man allerdings die verschiedenen Methoden danach beurteilen, inwiefern durch sie der Theoriebildungsprozeß unterstützt werde, und wie sie dazu beitrügen, die Fehlerhaftigkeit der Intuition zu begrenzen. Hier sieht Naur auch die Funktion von formalen Methoden und Spezifikationssprachen. Sie ermöglichen es dem Softwareentwickler, in einer anderen Notationsform über seine Theorie des Gegenstandsbereiches nachzudenken. Allerdings weist er auch hier darauf hin, daß diese Methoden an dem Maßstab der intuitiven Erfaßbarkeit gemessen werden müßten.

Einer bestimmten zeitlichen Abfolge von Entwicklungsschritten, wie sie in den meisten Softwareentwicklungsmethoden vorgeschlagen werde, mißt Naur jedoch keine hilfreiche Wirkung bei, da sich intuitive Erkenntnis wohl schwerlich in einem sequentiell festlegbaren Prozeß bilde.

Naur zieht in Naur (1985-1) den Schluß, daß die Sicht des Softwareentwicklers als Teil eines industriellen Produktionsprozesses von verschiedenartigen Dokumenten falsch sei. Der Softwareentwickler sei vielmehr eine Art Forscher und wie dieser zugleich Manager und Entwickler von Programmsystemen und damit von Theorien. Die Kenntnis und Beherrschung verschiedener Notationen und Repräsentationen von beispielhaften Datenprozessen gehört aber auch für Naur unzweifelhaft zum Handwerkszeug des Softwareentwicklers.

Naurs Beitrag ist ein Indiz für einen grundlegenden Wandel in der Sicht der Softwareentwicklung, der sich in den letzten Jahren in vielen Veröffentlichungen - speziell zur objektorientierten Softwareentwicklung - abzeichnet (siehe Shlaer-Mellor 1988, Booch 1991, Sommerville 1989). Es wird nun nicht mehr ausschließlich in Begriffen wie Anforderungsanalyse, Entwurf, Spezifikation und Programmierung gedacht. Diese Begriffe, die tendenziell stets als Analyse und schrittweise Formalisierung der analytisch festgestellten Anforderungen begriffen wurden, werden zwar als grobes Begriffsraster beibehalten, treten jedoch gegenüber Begriffen wie dem des Prototypings mit seiner Betonung explorativen und experimentellen Vorgehens zurück. Dabei wird als primärer Vorgang der Softwareentwicklung kooperativ-diskursive Konstruktion von "Realität" als gemeinsames Verstehen von Wirklichkeit begriffen.

Diese Veränderung der Sicht von Softwareentwicklung führte zunächst zu einer gewissen Verunsicherung der Softwareentwickler bezüglich der Grundlagen ihrer Weltsicht. Plötzlich bestand die Welt nicht mehr aus festen Definitionen, Spezifikationen und Funktionen - von denen sie ja erfahren hatten, daß sie gar nicht so fest sind, wie sie eigentlich gedacht waren - sondern aus unscharfen Begriffen wie Problem-Domäne, Key-Concepts und Objekten. Dies führte in einigen Fällen zu einem tiefgehenden Nachdenken über die Natur von Werkzeugen, Wirklichkeit und Geist, wie es sich etwa in Budde & Züllighoven (1990) ausdrückt. Allerdings scheint - wie Erfahrungen im Seminar "Probleme und Konzepte des objektorientierten Systementwurfs" im WS90/91 an der TU-Berlin (Budde & Züllighofen) zeigte - der dort gemachte Rückgriff auf die in Heideggers "Sein und Zeit" ausgedrückte

Weltsicht und die daraus entliehene Begrifflichkeit für die Softwareentwicklung - wie etwa "DV-Zeug", "Zuhandenheit" und "Zeugzusammenhang" schwerlich traditionell ausgebildeten Informatikern mit ihren stark operational ausgerichteten Ansichten als Grundlage von Softwareentwicklung vermittelbar zu sein.

Eine wichtige Erkenntnis, die Naur formuliert, ist, daß die Realisierung der Theoriebildung nachgeordnet ist und als Modellierung zu einer Theorie zu sehen ist. Floyd formuliert einen erweiterten Designbegriff für die Softwareentwicklung, indem sie den Designprozeß als "kreativen Vorgang, in dem das Problem erschlossen, eine zugehörige Lösung erarbeitet und in menschliche Sinnzusammenhänge eingepaßt wird" sieht (Floyd 1989-1, S. 2, zitiert nach Reisin (1992)).

Naurs radikaler Zweifel an der Dokumentierbarkeit von Designprozessen kann jedoch so wohl nicht aufrechterhalten werden. Theoriebildung zielt in allen Wissenschaftszweigen letztlich auf die intersubjektive Vermittlung und Überlieferung von Wissen ab. Auch ist in großen Softwareprojekten die Vermittlung der Theorie über den Gegenstandsbereich wesentlicher Aspekt der Projektkoordination. Die Theorie muß also in validierbarer Form vorliegen. Als ein Instrumentarium, das es ermöglicht, Theorien nicht nur zu "überliefern", sondern auch die Theoriebildungsprozesse selbst transparent und kommunizierbar zu machen, werden hier die qualitativen Methoden der Sozialwissenschaften gesehen.

3 Gegenstandsbezogene Softwareentwicklung

Im Folgenden wird untersucht, wie eine Synthese von Methoden der qualitativen Analyse und der traditionellen Softwareentwicklung aussehen könnte und inwiefern eine solche Synthese hilfreich sein könnte, den Theoriebildungsprozeß methodisch zu unterstützen und den Einfluß der intuitiven Erkenntnis - dieses "...act of insight, althoug of extremely fallible insight" (Peirce 1903, S. 242) - zu kontrollieren.

Als Ausgangspunkt soll das Datenflußmodel der Softwareentwicklung, so wie es in der Produktionssicht erscheint, unter Berücksichtigung der Theoriekomponenten dargestellt werden.

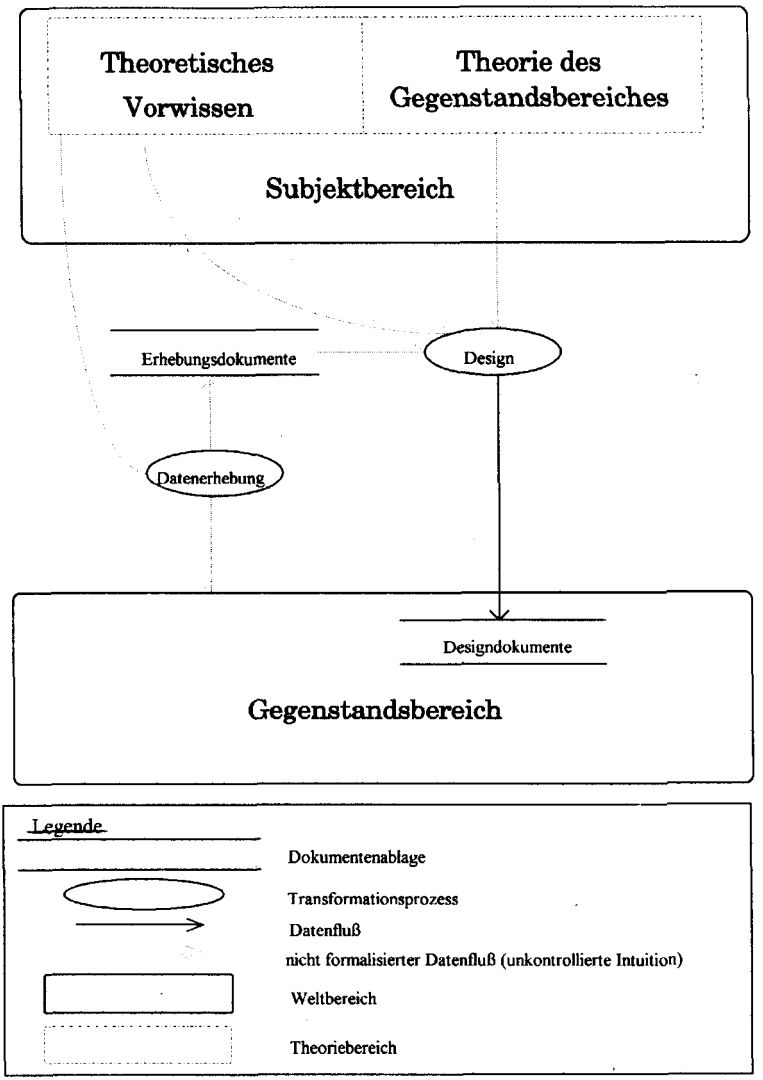

Abb. 3: Datenflußmodel der Softwareentwicklung aus Produktionssicht.

In der hier benutzten Notation werden die Vorgänge nur grob skizziert. Beispielsweise wird die Interaktion zwischen Personen nicht berücksichtigt. Stattdessen konzentriert sich dieses Modell auf die Wechselwirkung zwischen

theoretischem Wissen über den Gegenstandsbereich und dem Prozeß der Softwareentwicklung.

Die Datenerhebung als Ausgangspunkt des Entwicklungsprozesses[6] erfolgt in der Regel nicht in methodisch kontrollierter Interaktion mit den Experten des Gegenstandsbereiches (meistens den zukünftigen Benutzern des zu entwickelnden Programmsystems), sondern in "freier Kommunikation". Zwar gibt es Ansätze, diesen Vorgang methodisch zu fassen, sie bleiben aber meist rudimentär. Das Erstellen einer Ist-Beschreibung des Gegenstandsbereiches wird als ein Vorgang gesehen, der zwar erheblichen "Fingerspitzengefühls" bedarf, aber nicht eigentlich methodisch unterstützt werden kann. Entsprechend werden Softwareentwickler in ihrer Ausbildung auch selten in Methoden der Datenerhebung geschult. Ebenso fließt das theoretische Vorverständnis des Analytikers in erheblichem Maße ein. Die empirischen Sozialwissenschaften haben dagegen ein umfangreiches Instrumentarium entwickelt, um die intuitiven Einflüsse in der Datenerhebung zu kontrollieren.

Das Ergebnis der Datenerhebung ist eine Sammlung von sowohl qualitativen Daten wie Interview-Notizen, Texten aus dem Gegenstandsbereich (beispielsweise Normen) als auch von quantitativen wie etwa Ergebnissen von Zählungen, Abschätzungen, Messungen, die zu einer Zustandsbeschreibung zusammengefaßt werden. Die einzelnen Aussagen dieses Dokumentes lassen sich jedoch häufig nicht auf die erhobenen Daten zurückführen. Die an dieser Stelle getroffenen Entscheidungen über die Relevanz von Erhebungsdaten werden nur selten dokumentiert. Eine nachträgliche kritische Bewertung dieser Entscheidungen ist daher nur schwer möglich.

Die Zustandsbeschreibung des Gegenstandsbereiches enthält als einziges Dokument im Entwicklungsprozeß den gesamten Stand der Theorie über den Gegenstandsbereich, wie er zur Zeit ihrer Erstellung vorliegt. Sie wird aber in der Folge nicht durch Erkenntnisse, die während des Designprozesses gewonnen werden, aktualisiert.

Diese Zustandsbeschreibung ist das konstitutive Dokument für den gesamten Designprozeß[7]. Während des Designs erarbeiten sich die Entwickler, wie von Naur beschrieben, eine Theorie des Problemfeldes und einer Lösung, der entsprechend sie das Softwareprodukt gestalten. Bei traditionellem Vorgehen - etwa nach dem schon notorischen Wasserfallmodell der Softwareentwicklung - entsteht zwar eine

---

[6] Die Datenerhebung wird hier deshalb als Ausgangspunkt der Softwareentwicklung angenommen, weil sich der beispielsweise in STEPS modellierte Prozeß der Projektetablierung der Erfahrung nach dem Zugriff der Entwickler entzieht.

[7] Als "Design" werden an dieser Stelle, abweichend von den in Abb. 1 vorgestellten Modellen der Softwareentwicklung, nicht nur die ersten Phasen der Entwicklung verstanden, sondern die gesamte Gestaltung des Softwareprodukts bis hin zur Implementierung und Wartung.

beachtliche Anzahl von Dokumenten in den verschiedensten Notationen, sie spiegeln aber stets nur einen Teilaspekt der Theorie der Entwickler, der stark durch die Antizipation des nächsten Entwicklungsschrittes geprägt wird. Es ist für einen Außenstehenden sehr schwierig - wenn nicht, wie Naur meint, unmöglich - sich aus diesen Dokumenten die zugrundeliegende Theorie des Gegenstandsbereiches zu erarbeiten.

Die entstehenden Designdokumente (Spezifikationen, Definitionen, Testdaten etc.) werden zudem nur in den zyklischen Projektmodellen (z.B. STEPS) wiederum als Teil des Gegenstandsbereiches und damit als neuer Gegenstand der Datenerhebung gesehen. Der Produktionsprozeß endet aus der Produktionssicht der Softwareentwicklung mit der Implementierung und Inbetriebnahme des Produktes.

Zusammenfassend läßt sich sagen, daß die zentrale Tätigkeit der Softwareentwicklung nach Naur - die Theoriebildung - in der Produktionssicht der Softwareentwicklung nicht berücksichtigt wird. Der Einfluß der Intuition auf die Theoriebildung und damit auf das gesamte Design wird nicht methodisch kontrolliert. Die nachträgliche Bewertung eines Projektes als Ganzes - einschließlich der konkreten Identifikation von Schwächen - wird dadurch außerordentlich erschwert, wenn nicht sogar unmöglich.

Das Ziel des hier dargestellten Rahmens der Softwareentwicklung ist es, einen geschlossenen Dokumentationszusammenhang zwischen Datenerhebung und Design zu schaffen, in dem intuitive Einsicht stets methodisch kontrolliert und dokumentiert wird. Dies soll dazu führen, daß alle Schritte der Softwareentwicklung nachträglich verifizierbar werden und Fehlentscheidungen oder Mißverständnisse aufgedeckt und korrigiert werden können. Als Mittel dazu werden einerseits Methoden der systematischen Textinterpretation (Grounded Theory) eingeführt, andererseits wird ein zentrales Dokument - die "Theoriebeschreibung" - eingeführt, in dem alle Entwurfsentscheidungen begründet sein sollen und das selbst wiederum in den Erhebungs-Daten gegründet ist.

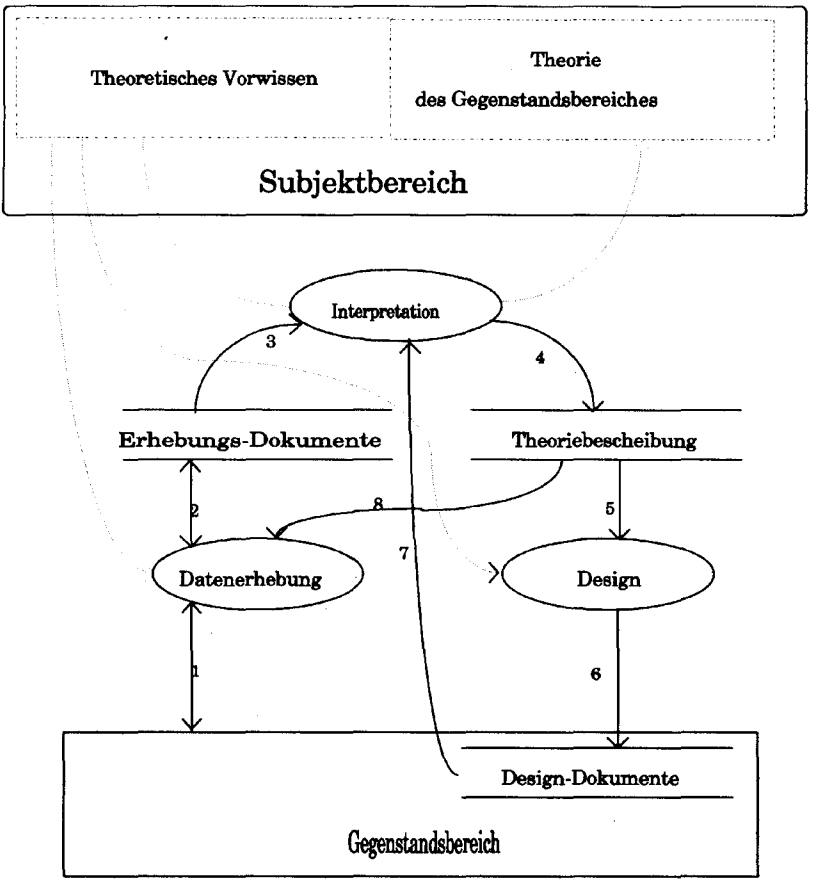

Abb. 4: Datenflußmodel der gegenstandsbezogenen Softwareentwicklung.

Wie Abb. 4) zeigt, wird durch dieses Vorgehen ein Kreislauf etabliert, in dem intuitive Einsicht zwar eine wesentliche Rolle spielt, jedoch methodisch kontrolliert und dokumentiert wird.

Zu den dargestellten Vorgängen im einzelnen:

1) Interaktion mit dem Gegenstandsbereich

   Aufgrund des hier einfließenden Alltagswissens ("common sense") und der kommunikativem Komponenten des Vorganges (beispielsweise in Interviews mit Experten des Gegenstandsbereiches) spielen intuitive Einflüsse hier eine besondere Rolle. Sie zu kontrollieren und fruchtbar zu machen ist die Aufgabe

von Erhebungsmethoden, wie sie aus den empirischen Sozialwissenschaften (Friedrichs 1984, S.192ff), der Systemanalyse (Athey 1982, S.138ff) sowie der Wissenselizitation für Expertensysteme (Muhr 1988, S. 47ff) bekannt sind.

2) Dokumentation der Erhebungsergebnisse

Als erster Schritt zur Formalisierung der in 1) gewonnenen Ergebnisse dient die Fixierung der Daten in Erhebungsdokumenten. Dabei können die verschiedensten Notationen eingesetzt werden. Die Wahl der Notation hängt vom vorliegenden Datenmaterial und der beabsichtigten Genauigkeit der Erhebungsdokumente ab. So hat eine nach dem Schema der gegenstandsbezogenen Softwareentwicklung durchgeführte Studie (Engelmeier 1992) ergeben, daß die durch humanwissenschaftliche Verschriftungsverfahren für Interviews gewährleistete Genauigkeit der Transkripte für die Softwareentwicklung zumeist in keinem Verhältnis zur beabsichtigten Verwendung derselben steht. Die Entwicklung spezifischer Verfahren zur Transkription gesprochenen Datenmaterials für die Softwareentwicklung steht jedoch noch aus. In einigen Fällen, etwa bei der Verwendung von schriftlichen Quellen aus dem Gegenstandsbereich, besteht die Datenerhebung nur aus der Suche und Auswahl dieser Dokumente. Stets muß jedoch gefordert werden, daß die Gründe für die Quellenauswahl schriftlich niedergelegt werden. Andernfalls wäre die Verifizierbarkeit nicht mehr gewährleistet.

3) Interpretation der Erhebungsdokumente

Die Erhebungsdokumente bilden nun die Basis für die Textinterpretation nach der Grounded Theory (Strauss 1987). Dabei handelt es sich um ein Verfahren, das den intuitiven Charakter textinterpretativer Vorgänge durch methodische Leitung nachvollziehbar machen will. Sowohl theoretisches Vorwissen, als auch die Antizipation des Interpretationszweckes fließen mit ein.

4) Formulierung der Theoriebeschreibung

Das Ergebnis dieses systematischen Interpretationsvorganges ist einerseits eine Theorie des Gegenstandsbereiches, die "in den Köpfen" (Naur) der Softwareentwickler existiert, andererseits jedoch ein Dokument, das die gewonnene Interpretation widerspiegelt. Dieses Dokument ist von komplexer Struktur und kann am besten als eine hypertextartige Struktur gedacht werden, die zahlreiche Verweise auf Erhebungsdaten und evtl. in früheren Interpretationsdurchgängen erstellte Designdokumente enthält.

5) Design

Die Theoriebeschreibung bildet nun die Grundlage (im Sinne von "Begründung für") für die Erstellung aller Designdokumente. Dabei kann das vorgeschlagene Datenflußmodel als Rahmen für alle Softwaredesignmethoden dienen. Es ist sicherzustellen, daß alle während des Designs erstellten Artefakte in der Theoriebeschreibung begründet werden. Intuitives Erkennen der Anwendbarkeit theoretischer Konzepte ist hier nach wie vor essentiell. Diese intuitiven

Einsichten werden jedoch einerseits in den Designdokumenten und andererseits in der Theoriebeschreibung dokumentiert.

6) Modellbildung

Die Designdokumente werden in dieser Sicht der Softwareentwicklung als Modell (Interpretation) zu der in der Theoriebeschreibung niedergelegten Theorie des Gegenstandsbreiches gesehen. Jedes Designelement muß in der Theoriebeschreibung seine Begründung finden. Dabei ist die Theoriebeschreibung nicht als Spezifikation zu sehen. Sie hat argumentativen und explikativen Charakter. Spezifikationen sind schon Teil der Designdokumente und als solche in der Theoriebeschreibung begründet.

7) Evaluierung

Die Designdokumente werden als neuer Teil des Gegenstandsbereiches gesehen, in den sie eingebracht werden. Als solche sind sie auch Grundlage für neue Theoriebildung. Beispielsweise ist bei der Entwicklung von Prototypen der Prototyp zugleich Ergebnis von Interpretation und Grundlage von Datenerhebung. Hier ähnelt dieser Ansatz den in STEPS vertretenen Konzepten der evolutionären Softwareentwicklung (siehe Floyd 1989-1).

8) Theoriegeleitete Datenerhebung

Bei der Formulierung der gegenstandsbezogenen Theorie kann es vorkommen, daß zu bestimmten Teilbereichen des Gegenstandsbereiches zusätzliche Daten benötigt werden. Es ist dann jederzeit möglich, zur Datenerhebung zurückzukehren und auf Grund der bisher gebildeten Theorie neue Daten zu sammeln, um diese Lücken zu schießen.[8] Eine unter allen Umständen einzuhaltende zeitliche Reihenfolge der hier dargestellten Tätigkeiten kann es, wie Naur schon ausführte, nicht geben. Es wird aber in der Regel so sein, daß die gegenstandsbereichsnahen Tätigkeiten zum Beginn der Entwicklungsarbeit eine größere Rolle spielen, während die theoriebildenden und modellbildenden Tätigkeiten zum Ende des Entwicklungsvorganges dominieren. Die hier angegebenen Datenflüsse sind daher eher im Sinne von Kausalzusammenhängen, denn im Sinne von zeitlicher Reihenfolge zu verstehen.

Abschließend sollen nun noch einmal die Kennzeichen einer gegenstandsbezogenen Softwareentwicklung zusammengefaßt werden:

- Gegenstandsbezogene Softwareentwicklung erkennt intuitive Einsicht als grundlegenden Vorgang in der Softwareentwicklung an. Sie ist jedoch bemüht, Intuition - bzw. deren Ergebnisse - methodischer Kontrolle zu unterwerfen.

[8] Glaser und Strauss sprechen in diesem Zusammenhang von "Theoretical Sampling" (siehe Kap. 5.2.3).

- Gegenstandsbezogene Softwareentwicklung versucht für alle während der Entwicklung entstehenden Artefakte den Bezug zum Gegenstandsbereich aufrechtzuerhalten. Dies geschieht durch explizite Bezugnahme auf die Erhebungsdaten oder auf in ihnen begründete Dokumente.
- Gegenstandsbezogene Softwareentwicklung betrachtet die Theoriebildung als zentrale Aufgabe des Softwareentwicklers. Sie legt diese Theorie in einem zentralen Dokument - der Theoriebeschreibung - nieder.
- Dieses Dokument bildet den Begründungszusammenhang für den gesamten Designprozeß. Es unterstützt die Kommunikation der Entwickler untereinander und mit den Benutzern. Es ist in den Daten des Gegenstandsbereiches gegründet. Der Bezug zu diesen Daten ist explizit.
- Die Theoriebeschreibung ermöglicht durch ihre netzartige Struktur und ihre zentrale Stellung im Entwicklungsprozeß die Exploration der gegenstandsbezogenen Theorie. Sie ermöglicht ebenfalls die Exploration der Designdokumente aus ihrem Begründungszusammenhang heraus, und erleichtert damit die Einarbeitung in komplexe Softwaresysteme.
- Die gegenstandsbezogene Softwareentwicklung ermöglicht die Verifikation des gesamten Softwareentwicklungsprozesses. Die Einführung neuer Qualitätsmaßstäbe ist möglich.

Aufgrund des komplexen Charakters der Theoriebeschreibung und der unter Umständen großen Menge von zu verwaltenden Erhebungs- und Designdokumenten, scheint eine Werkzeugunterstützung des Entwicklungsprozesses, wie er hier dargestellt wurde, unumgänglich. Eine Vorstudie, in der ein Werkzeug zur Unterstützung qualitativer Methoden in den Gesellschaftswissenschaften (ATLAS/ti) für den gegenstandsbezogenen Softwareentwurf eingesetzt wurde, hat in dieser Beziehung ermutigende Resultate gebracht (Engelmeier 1992). Eine Einbeziehung einer solchen "Qualitativen Komponente" in eine umfassende Softwareentwicklungsumgebung scheint, nach den Ergebnissen dieser Studie zu urteilen, wünschenswert.

Literatur

Athey, Thomas H. (1982): *Systematic systems approach*; Prentice-Hall International; (previously published under the title *Structured systems approach).*

Booch, G. (1983): *Softwareengineering with ADA*; The Benjamin L. Cummings Publishing Company, 1st ed.

Booch, G. (1991): *Object Oriented Design*; The Benjamin L. Cummings Publishing Company.

Budde, Reinhard & Züllighoven, Heinz (1990): *Softwarewerkzeuge in einer Programmierwerkstatt*; GMD-Bericht 182 bei R. Oldenburg Verlag.

Corbin, Juliet & Strauss, Anselm (1990): *Grounded Theory Research - Procedures, Canons, and Evaluative Criteria*; in: Qualitative Sociology; Vol. 13, No. 1, Human Science Press, New York.

Curtis, Bill & Iscoe, Neil (1990): *Modeling the software design process*; in: Zude, P. & Hocking, D. (Hrsg.) 1990.

Engelmeier, Gregor (1992): *Qualitative Analyse kooperativer Tätigkeiten bei der Textinterpretation*; TU-Berlin; (erschienen als Bericht 92-1 des IFP "ATLAS" der TU-Berlin).

Floyd, C. (1984): *Eine Untersuchung von Software-Entwicklungs-Methoden*; in: Programmierumgebungen und Compiler S. 248-274, ed. H. Morgenbrod und W. Sammer; Teubner.

Floyd, C., Schmidt, G. Reisin, F.-M. (1989-1): *STEPS to software development with users*; in [ESEC89], Springer Verlag.

Floyd, C. et al. (1989-2): *Projekte zur Softwareentwicklung*; Arbeitsunterlage zur Vorlesung "Einführung in das Software Engineering" im WS 89/90 an der TU-Berlin.

Friedrichs, J. (1984): Methoden empirischer Sozialforschung; Westdeutscher Verlag.

Glaser, B. & Strauss, A.L. (1967): *The Discovery of Grounded Theory*; Aldine Publishing Co., Chicago.

Kleining, Gerhard (1982): *Umriß zu einer Methodologie Qualitativer Sozialforschung*; in: Kölner Zeitschrift für Soziologie und soziale Psychologie; Vol. 59.

Knuth, Donald E. (1984): *Literate Programming*; in: The Computer Journal, Vol.27, No. 2.

Konrad, E. (1982): *Modelle für Information Retrieval Systeme*; in: Kleincomputer in Information und Dokumentation; K.G. Saur Verlag.

Muhr, Thomas (1988): *Methoden der Wissensakquisition unter besonderer Berücksichtigung der Wissenselizitation*; Diplomarbeit an der TU-Berlin, FB Informatik.

Naur, Peter (1985-1): *Programming as Theory Building*; in: Microprocessing and Microprogramming Vol. 15, S. 253-261; (zitiert nach Naur (1992)).

Naur, Peter (1985-2): *Intuition in Software Development*; in: Formal Methods and Software Development; vol 2, S. 60-79 ed. H. Ehrig, C. Floyd, M. Nivat; Lecture Notes in Computer Science 186, Springer (zitiert nach Naur (1992)).

Naur, Peter (1992): *Computing: A Human Activity*; ACM Press.

Peirce, Charles Sanders (1903): *Lectures on Pragmatism*; zitiert nach Felix Meiner Verlag, Hamburg 1973.

Reisin, Fanny-Michaela (1992): *Kooperative Gestaltung in partizipativen Softwareprojekten*; Verlag Peter Lang.

Shlaer, S. & Mellor, S.J. (1988): *Object Oriented Systems Analysis - Modelling the World in Data*; Yourdon Press.

Sommerville, Ian (1989): *Software Engineering*; Addison-Wesley.

Strauss, Anselm (1984): *Qualitative Analysis in Social Research - Grounded Theory Methodology (Part Two)*; FB Erziehungs und Sozialwissenschaften der Fernuniversität Hagen.

Strauss, Anselm (1987): *Qualitative Analysis for Social Scientists*; Cambride Univ. Press.

Whitehead, A.N. (1942): *Adventures of ideas*; Penguin, first published in 1933.

Zude, P. & Hocking, D. (Hrsg.) (1990): *Empirical Foundations of Information and Software Science V*; Plenum Press.