Informatik-Spektrum 20: 5–12 (1997)

Hauptbeitrag

Ein tätigkeitstheoretischer Ansatz zur Entwicklung von brauchbarer Software

Christian Dahme, Arne Raeithel †

Zusammenfassung Wie kommt man zu einer relativ leicht kommunizierbaren, transparent nachvollziehbaren Transformation von den Wünschen und der realen Situation zu brauchbarer Software? Dazu wird ein tätigkeitstheoretischer Ansatz vorgestellt. In diesem wird Software-Entwicklung als Teil eines Transformationsprozesses verstanden, der menschliche Tätigkeiten in eine maschinelle Form überführt. Grundlage für diesen Ansatz ist die Tätigkeitstheorie, aus der auch der in Software transformierbare Anteil ableitbar ist. Er ermöglicht darüber hinaus u. a. eine relativ einfache, überschaubare Transformation in objektorientierte Software, günstigen Ausgangs- bzw. Randbedingungen für eine evolutionäre Software-Entwicklung, Mittel für eine Kooperation (u. a. eine verständliche Sprache), Mißverständnisse zwischen den Projektbeteiligten gezielt anzugehen. Insgesamt liefert dieser Ansatz einen Rahmen sowie methodisches und konzeptionelles Rüstzeug und Hilfsmittel für die kooperative und evolutionäre Software-Entwicklung.

Schlüsselwörter Software-Entwicklung, Tätigkeitstheorie, Handlung, Operationen, Modell, reproduzierbares Wissen, mitteilbares Wissen, Verfahrenswissen, Allgemeinwissen, evolutionäre Software-Entwicklung, Kooperation

Summary How does one achieve a relatively easily communicable, transparent, and easily understood transformation of a set of wishes and a real situation into useful software? Here we present an activity-theoretical approach. In this, software development is understood as part of a transformation process which converts human activity into a machine-based form. The basis for this approach is activity theory, from which one can also derive the part which is transformable into software. This theory also enables, among other things, a relatively straightforward transformation into object-oriented software, favorable initial and boundary conditions for an evolutionary software development, tools for cooperation (inluding an understandable language), and one can directly address misunderstandings between those involved in the project. Overall, this approach yields both a framework and a methodological and conceptual tool and support for cooperative and evolutionary software development.

Key words Software-development, Activity theory, Action, Operations, Model, reproducible knowledge, communicable knowledge, procedure knowledge, general knowledge, evolutionary software-development, Cooperation

Computing Reviews Classification D.1,D.2, H.1, I.6.5

1. Einführung

Viele Software-Projekte werden durch falsche Annahmen von vornherein gefährdet. So ist es meistens falsch, anzunehmen, daß

- der Anwender bei Projektbeginn genau weiß, was er will,
- der Anwender das, wovon er weiß, daß er es will, vollständig mitteilen kann,
- der Entwickler ausreichend verstanden hat, was der Anwender mitteilen konnte,
- das kommunizierte Wissen ausreicht, um die vom Anwender gewollten Funktionen produzieren zu können,
- der Entwickler eine wissenschaftlich begründete Methode besitzt, um das verstandene in Software zu übertragen,
- der Anwender versteht, was der Entwickler außer den vorgelegten Beispielen noch leisten könnte,
- der Anwender also wüßte, welche Software möglich wäre, wenn der Entwickler besser über seine Bedürfnisse unterrichtet wäre.

Eine recht naive und gefährliche Annahme versteckt sich außerdem in der Zuversicht, daß Entwickler und Anwender es schon rechtzeitig merken werden, wenn Mißverständnisse auftreten, und diese durch Nachfragen leicht beseitigen können. Vielmehr ist es die Regel, daß Verständnislücken auf beiden Seiten viel zu spät und manchmal gar nicht aufgeklärt werden, so daß für den Anwender Unnützes mit viel Mühe implementiert wird, und zugleich zentrale, aber bislang implizit gebliebene Anforderungen gerade noch nicht erfüllt sind.

Das Risiko der Software-Entwicklung, eingeschlossen deren Management, erwächst nicht allein aus den Tücken der Kommunikation, sondern auch aus Mängeln der bislang vorhandenen Methodik. Verfahren zur Softwarespezifikation weisen oft eine reichlich große Beliebigkeit auf und sind selten ausreichend methodisch kontrolliert. Was zum Beispiel heißt „abstrahieren“? Wovon wird hier abgesehen und was wird als

Christian Dahme[1], Arne Raeithel[2]

[1] Humboldt-Universität zu Berlin, Mathematisch-naturwissenschaftliche Fakultät II, Institut für Informatik, Lindenstrasse 54 a, D-10099 Berlin, e-mail: dahme@informatik.hu-berlin.de

[2] Arne Raeithel ist während der Arbeit an diesem Artikel am 1.12.1996 verstorben. Er gehört zu denjenigen, die maßgeblich die Tätigkeitstheorie in die Informatik getragen haben.

die brauchbare, implementierbare Form aus der Beschreibung herausgezogen? Wie kommt man von den Wünschen des künftigen Anwenders zur formalen Beschreibung als Voraussetzung für die Software?

Wir wollen hier einen Ansatz beschreiben, der solche Probleme zu erkennen und methodisch zu mildern ermöglicht. Dabei werden wir zunächst eine allgemeine Methodik für die Ableitung von Möglichkeiten der Softwareumsetzung aus einer Analyse von menschlichen Tätigkeiten vorstellen (bis zum Abschnitt 5). Anschließend werden wir, so vorbereitet, diesen Ansatz in die kooperative und evolutionäre Software-Entwicklung einordnen.

Allgemein kann man Software als Transformation von Anteilen realer bzw. möglicher menschlicher Tätigkeit in eine äußere, maschinelle Form, als *Vergegenständlichung* von menschlichen Fähigkeiten, verstehen. Als künftiges Mittel (Medium, Werkzeug oder Automat) soll Software die Tätigkeiten der Anwender erleichtern und unterstützen oder die Effektivität dieser Tätigkeiten erhöhen, aber auch bestimmte neuartige Tätigkeiten überhaupt erst ermöglichen.

Aus dieser Sicht ergeben sich insbesondere für die frühen Phasen der Software-Entwicklung folgende grundlegende Fragen:

- Was läßt sich von einer menschlichen Tätigkeit in Software übertragen? Mit anderen Worten: Welcher Anteil der menschlichen Tätigkeit läßt sich objektivieren bzw. hat die Potenz, in Form einer Turingmaschine realisiert werden zu können?
- Wie kommt man zu diesem Anteil?
- Wie geht man mit dem nicht formalisierten oder dem nicht formalisierbaren Anteil um, der für die adäquate Nutzung der Software stets notwendig ist? (Interpretation, subjektive Wertung, Kontext)

Hier setzt der tätigkeitstheoretische Ansatz an. Nach diesem Ansatz ist der Anteil einer Tätigkeit, der sich potentiell in Software übertragen läßt, durch folgendes charakterisiert:

1. Er bezieht sich auf Anteile einer inneren, orientierenden Tätigkeit (Abschnitt 3) oder läßt sich in solche transformieren.
2. Diese Anteile liegen als Operationen (Abschnitt 4) vor oder lassen sich von Handlungen in Operationen überführen (Operationalisierbarkeit).
3. Es liegt Wissen vor, mit dem sich diese Operationen (vollständig) reproduzieren lassen – reproduzierbares Wissen (Abschnitt 5).
4. Dieses Wissen ist mitteilbar (Abschnitt 5).
5. Es kann in öffentliches Wissen überführt werden (Abschnitt 5).

In den Abschnitten 3, 4 und 5 werden wir dies schrittweise erläutern. Dabei entsteht eine sehr allgemeine Strategie, um den in Software übertragbaren Anteil feststellen zu können. Im Abschnitt 6 werden wir abschließend erläutern, wie dieser Ansatz mit den Problemen von Kooperation und Kommunikation umgeht.

Ein tätigkeitstheoretischer Ansatz bietet also:

a) eine Sprache, die sowohl Anwender als auch Entwickler der Software verstehen können; dies erleichtert erheblich die Herausbildung einer Projektsprache,
b) eine Strategie, um den potentiell in Software übertragbaren Anteil einer menschlichen Tätigkeit bestimmen zu können,
c) die Feststellung, Berücksichtigung bzw. Integration von nicht formalisierten bzw. nicht formalisierbaren Anteilen,
d) eine relativ einfache, überschaubare Transformation von einem tätigkeitstheoretischen beschriebenem Modell in objektorientierte Software,
e) einen Zugang zur Bildung gegenständlicher Modelle als innere, orientierende Tätigkeit und als Voraussetzung für die Software-Entwicklung,
f) ein Konzept der Zusammenarbeit als Koordination, Kooperation und Ko-Konstruktion [17] in einer Praxisgemeinschaft, in das sich die existierenden Moderations- und Gruppenführungsmethoden sehr gut einfügen lassen, und damit insgesamt
g) die nötigen Mittel, um Mißverständnisse zwischen den Projektbeteiligten (künftigen Anwendern, den Software-Entwicklern, deren Management sowie Fachwissenschaftlern bzw. Systemanalytikern) gezielt anzugehen und in kurzen Zyklen immer wieder auszuräumen, um so die Diskrepanzen zwischen dem „Gewünschten“ und dem „Möglichen“ möglichst gering zu halten.

2. Zusammenhang zwischen tätigkeitstheoretischem Ansatz und objektorientierter Programmierung

Ein mit Hilfe des tätigkeitstheoretischen Ansatzes erstelltes Modell kann relativ leicht in objektorientiert programmierte Software transformiert werden (Abb. 1).

Unter Eigenbewegung eines Gegenstandes oder Mittels verstehen wir, daß diese Bewegung unabhängig von der Tätigkeit eines Menschen erfolgt, jedoch von einer solchen ausgelöst sein kann.

Abb. 1 Orientierung zur Transformation in objektorientierte Software (aus [2])

Element im tätigkeitstheoretischen Ansatz	Element in der objektorientierten Programmierung
Gegenstand der Tätigkeit	Objekt
Operationen	Methoden
Mittel der Tätigkeit	Objekt
Aktivieren bzw. Auslösen der Operationen (u. a.) • als Ausdruck des zielorientierten Handelns • als Resultat von Operationen	Botschaften
Eigenbewegung des Gegenstands bzw. des Mittels	Methoden

3. Grundbegriffe und Gestaltungsziele des tätigkeitstheoretischen Ansatzes

Wir können unsere Strategie der Software-Entwicklung hier nur in den wesentlichen Umrissen darlegen (ausführlich s. Dahme [2, 3]; Raeithel [12, 13, 15], s. aber auch [11] und [6]). Nach der Erläuterung der Grundbegriffe präsentieren wir eine Hierarchie von daraus abgeleiteten Gestaltungszielen. In den Abschnitten 4 bis 6 wird dann zu zeigen sein, wie die Erreichung solcher Ziele methodisch angesteuert werden kann.
Grundlegend für den Ansatz sind die aus der Tätigkeitstheorie [9] bekannten drei Analyseebenen einer Tätigkeit:

- Tätigkeit und Motiv
- Handlung und Ziel
- Operationen und Bedingungen

sowie die Unterteilung von Tätigkeiten in innere und äußere. Wir wollen dies nun stichwortartig erläutern: Eine *Tätigkeit*

- wird durch ein *Motiv*, dem ein Bedürfnis zugrunde liegt, ausgelöst,
- ist auf einen *Gegenstand* gerichtet, der zur Bedürfnisbefriedigung geeignet ist, und
- ist mit der Befriedigung dieses Bedürfnisses beendet.

Dabei bezieht sich das Motiv einer Tätigkeit auf die Frage: „*Warum tut ein Mensch das?*"

Zur Veranschaulichung möge u. a. das folgende Beispiel dienen: Im Rahmen eines Projekts (s. [2]) bestand die Aufgabe darin, den Arbeitsplatz eines Dispatchers in einem Kliniksanatorium zu analysieren und Vorschläge für eine mögliche Computerunterstützung zu machen. Dabei bestand die Tätigkeit der Dispatcherin darin, den Patienten, die vom Arzt bestimmte Therapien verordnet bekommen hatten, medizinisch möglichst wirksam die vorhandenen Therapiemöglichkeiten zuzuordnen.

Um ausgehend von einem Motiv (z. B. die Dispatcherin möchte gerne zufriedene Patienten) zu einer Bedürfnisbefriedigung zu kommen, muß die Person eine Situation erreichen, die dieses ermöglicht. In unserem Beispiel besteht eine solche Situation darin, ausgehend von den vorhandenen Ressourcen für jeden Patienten eine medizinisch möglichst wirksame zeitliche Zuordnung der verordneten Therapien zu vorhandenen Therapiemöglichkeiten zu finden.

Wird diese Situation bewußt zielgerichtet angestrebt, so wird dieser Prozeß *Handlung* genannt. Als Synonym für Handlung spricht man auch von *Aufgabe*. Die Lösung der Aufgabe besteht dann in der „Realisierung der angestrebten Situation". Um diese angestrebte Situation zu erreichen, können auch mehrere Handlungen notwendig sein (etwa im Sinne von Zwischenlösungen oder einer schrittweisen Realisierung der Tätigkeit). So kann die Tätigkeit der Dispatcherin durch folgende Handlungen realisiert werden: „Alle Kurbücher, in denen die vom Arzt verordneten Therapien enthalten sind, einsammeln", „Alle Therapien der Patienten in die entsprechenden Therapiebögen eintragen", „Die Patienten medizinisch möglichst wirksam zuordnen", „Die Therapiezeitpläne der einzelnen Patienten in die Kurbücher eintragen".

Bei einer Handlung ist zwischen dem *Ziel – was* man erreichen will – und dem *Verfahren – wie* man dieses Ziel erreichen will – zu unterscheiden. Die Verfahren, die eine Handlung konkret realisieren, werden als *Operationen* bezeichnet. Dabei sind die Operationen vor allem von den gegebenen Bedingungen, unter denen sie stattfinden, abhängig. – So kann die Handlung „Alle Kurbücher einsammeln" z. B. durch folgende Operationen realisiert werden: „von jedem Patienten das Kurbuch abholen", „die Kurbücher dezentral oder zentral einsammeln lassen" oder zu ermöglichen, „die entsprechenden Daten über ein Datennetz abrufen zu können".

Für eine Handlung, die automatisch abläuft, ist ein bewußt angestrebtes Ziel nicht mehr notwendig. Sie wird damit zur Operation. Die Möglichkeit, daß Handlungen zu Operationen werden können, nennt man auch *Operationalisierbarkeit* einer Handlung (Beispiele hierzu s. [3]).

Da Operationen nur von den Bedingungen abhängen, unter denen sie stattfinden können, haben sie die Potenz zur *Automatisierung*, d. h., sie können gegebenenfalls völlig aus einer Tätigkeit herausgenommen und einem Automaten übertragen werden. Damit haben wir eine wesentliche Bedingung, um Teile einer Tätigkeit in Software übertragen zu können.

Andererseits können die Mittel einer Tätigkeit – Medien, Werkzeuge oder Automaten – deren Durchführung erleichtern oder überhaupt erst ermöglichen: das Medium der Sprache die alltägliche Verständigung; die Werkzeuge Papier und Stift, Taschenrechner oder Tabellenkalkulation das Berechnen von Zahlen; der Automat Mailserver das Verteilen und Versenden von elektronischer Post.

Solche Mittel können u. a. durch Ausgliedern von Teilen einer Tätigkeit entstehen, indem diese ausgegliederten Operationen in einem Werkzeug vergegenständlicht werden, oder indem man einen Automaten entwickelt, welcher diese Operationen im Prinzip selbständig realisiert, z. B. ein Musterprogramm für das Weben eines Stoffes, oder indem man kommunikative Operationen (Antworten, Benachrichtigen, Nachlesen, Recherchieren, usw.) zu einem neuen Medium bündelt (wie bei Mosaic oder Netscape).

Software als Mittel zur Unterstützung menschlicher Tätigkeit entsteht in der Regel durch eine Ausgliederung von Operationen aus (realer oder denkbarer) menschlicher Tätigkeit, die dann durch den Computer mittels Software übernommen werden.

Abbildung 2 zeigt, welche Bedeutung die drei Analyseebenen einer Tätigkeit für die Software-Entwicklung haben.

Abb. 2 Bedeutung der drei Analyseebenen einer Tätigkeit für die Softwareentwicklung

Analyseebene	Leitfrage	Beispiele für softwareentwicklungs-relevante Gestaltungsziele
Tätigkeit	Warum (Motiv)	Benutzer- und Fehlerfreundlichkeit Einbettung in die Anwendertätigkeit
Handlung	Was (Ziel)	Dialoggestaltung, Handhabbarkeit, visuelle Rückmeldungen, Hilfen
Operationen	Wie (Verfahren)	direkt in ein Programm transformierbare effiziente und revidierbare Objekte, Metoden und Botschaften bzw. Algorithmen und Datenstrukturen

3.1. Für die Software-Entwicklung relevante Gestaltungsziele

Bei der Benutzerfreundlichkeit sind zwei Aspekte zu unterscheiden [2]: 1. Was ist zweckmäßig für den Anwender? (führt zur Software-Ergonomie). 2. Was motiviert? (im Sinn der Anregung einer positiven Haltung des Anwenders zu seinem Arbeitsmittel).

Mit dem Kriterium der Fehlerfreundlichkeit meinen wir ein weiteres anstrebbares und globales Gestaltungsziel. Hierbei geht es um die Berücksichtigung von möglichen aus der Sicht der Entwickler *designfremden* Motiven oder Zielen, die ein Anwender bei der Nutzung der Software und quer zu den Absichten der Entwickler haben kann. Häufig sind letztere von den Entwicklern nie bedacht worden, was zu *Fehlern* bei der Nutzung der Software führen kann.

Die bekanntesten Beispiele für Fehlerfreundlichkeit bei graphischen Benutzeroberflächen wie Windows 95 oder MacOS sind die allgemeine Rücknahmefunktion (UnDo) und die Papierkorb-Metapher des Löschens mit dem dreiphasigen Verfahren: Wegpacken, Zeit haben zum Besinnen, Löschen oder Rückholen.

Jedes einzelne Anwendungsprogramm muß darüber hinaus die bereichsspezifischen Fehler (z. B. was man beim Zeichnen von Grafiken falsch machen kann) berücksichtigen. Daneben sollten auch die „bereichsverwechselnden" Fehler des Benutzers (er möchte z. B. einen Bitmap-Text nachträglich editieren) angemessen beantwortet werden. Was aber ist der angemessene Umgang mit Benutzerfehlern?

Wie Wehner und Mitarbeiter [16] detailliert zeigen, stecken in den Handlungsfehlern die fruchtbarsten Quellen für individuelles Lernen und auch für verallgemeinerbare Innovationen. Software sollte Fehler in diesem Sinn zulassen, ohne daß sie gleich katastrophale Auswirkungen haben. Oder in einer Metapher gesagt: Mit einem guten Schnitzmesser muß man sich auch schneiden können, um daraus seine Handhabung zu lernen, ohne daß gleich die ganze Hand ab ist.

Katastrophale Fehler sind also (vor allem durch ausführliches Beta-Testen) so weit wie irgend möglich auszuschließen. Daneben sind Mittel und Wege bereitzustellen, wie die Anwender nach ihren „produktiven Fehlern" die Arbeit immer noch fortführen und sich dabei bessere Umgangsweisen mit den vorhandenen Mitteln aneignen können. Hier sind die Dialoggestaltung, die visuellen Rückmeldungen jeder Benutzeraktion und eine möglichst kontextsensitive Hilfefunktion überaus wichtig. Solche vorausschauend implementierten Möglichkeiten gehören zum Verläßlichkeitskontext der jeweiligen Software und gestatten es den Anwendern, einen persönlichen Stil ihrer Arbeit auszubilden. Beim Umgang mit dem Betriebssystem wäre ein solcher Stil etwa: Hochstarten, seine Sachen suchen, das heute zu bearbeitende Dokument öffnen, versenden und drucken, auf Rückfragen antworten, und so fort.

Benutzer- und Fehlerfreundlichkeit sind zwei wesentliche Merkmale einer guten *Einbettung* der Software in die existierende Tätigkeit der Anwender. Zwingt dagegen das neue Programmsystem zur Ausbildung von ganz neuen und fremden Handlungsstilen, dann wird es gegenüber den Motiven der Anwender sperrig und wird kaum eine Akzeptanzchance erhalten.

Dennoch ist es möglich, erfolgreiche Software zu gestalten, die ganz neue Motive und damit neue Tätigkeiten ermöglicht. Das klassische Beispiel hierfür ist die Erfindung des aktiven Rechenblattes durch die Programmierer von VisiCalc auf dem Apple II: Sie haben vollkommen vertraute Operationen auf neuartige Weise kombiniert: ausführbare Anweisungen schreiben, jedoch nicht in ein Programm, sondern in parallel sichtbare Zellen einer Tabelle, und so die Planungstätigkeit von kleinen Selbständigen revolutioniert, die seitdem wie Planer und Controller von großen Firmen viele Varianten ihrer Geschäftspläne durchspielen können.

4. Innere, orientierende Tätigkeit und gegenständliches Modell

In der Tätigkeitstheorie unterscheidet man zwischen der äußeren, praktischen Tätigkeit, die man sieht und überwiegend mit den Händen und Füßen durchführt, und der inneren Tätigkeit, die überwiegend in unserem Kopf abläuft, auch geistige oder psychische Tätigkeit genannt.

Eine andere Formulierung dieser Unterscheidung ist die von *orientierenden* und *realisierenden* Tätigkeiten. Damit wird die Differenz der jeweiligen Resultate hervorgehoben: Während die Produkte der realisierenden Tätigkeiten als solche veräußert oder verbraucht werden, dienen die Ergebnisse der orientierenden Tätigkeiten vor allem dazu, die anderen Tätigkeiten zu regulieren, sie auf den sozialen und gegenständlichen Kontext auszurichten und ihre innere Struktur in Richtung einer besseren, leichter vermittelbaren Organisation zu ordnen.

Für die Entwicklung und Nutzung von Software sind die inneren, orientierenden Tätigkeiten von besonderem Interesse. Mittel und Gegenstand einer inneren, orientierenden Tätigkeit sind Bilder, Symbole und Sprache. Solche Tätigkeiten finden auf einer symbolischen Ebene bzw. in einer symbolischen Welt statt. Diese Welt ist jedoch nicht allein eine „private Welt" rein geistiger Vorstellungen und Vorgänge, sondern auch eine gemeinsame, kulturelle und soziale Einbettung kooperativ tätiger Personen. Das „Innen" ist somit auch ein sozialer Raum, in dem sich die Beteiligten aneinander und miteinander orientieren können.

So ist z. B. „etwas berechnen" eine innere Tätigkeit. Sie findet folglich auf der symbolischen Ebene statt. Software, die dieses Berechnen realisiert, ist die Vergegenständlichung von Teilen innerer Tätigkeit. Software gehört folglich zur symbolischen Welt und mit ihr können die Anwender Objekte auf der symbolischen Ebene bearbeiten. In diesem Sinne werden die Computer oft allgemein als symbolverarbeitende Maschinen[1] bezeichnet.

4.1. Virtualität und symbolische Modelle

Objekte der symbolischen Welt sind insbesondere gegenständliche Modelle [3], d. h. Repräsentanten realer Objekte auf symbo-

[1] Man sieht dabei von den Komplikationen ab, die sich bei rechnergestützten Prozeßsteuerungen durch die angekoppelten Sensoren und Motoren ergeben, denn dort müßte man von Signalverarbeitung und (Steuer-) Signalerzeugung sprechen. Wir lassen diese Differenzierung im folgenden unberücksichtigt.

lischer Ebene, so z. B. das Modell eines bestimmten physikalischen oder biologischen Objektes, das Modell einer Fabrik oder einer medizinischen Einrichtung oder ein Modell, das menschliches Handeln in einer bestimmten Situation abbildet.

Dabei ist diese symbolische Welt einerseits eine nur den jeweiligen Modellierern so erscheinende, konstruierte, überwiegend in ihrer Vorstellung existierende Welt. Andererseits handelt es sich nicht um eine bloße Illusion, da die Modelle durch die Tätigkeiten der Modellierer wirksam verändert werden und ihrerseits auf diese zurückwirken, zum Beispiel in der Form suggestiver Anregungen, die am Modell abgelesen werden können, obwohl sie vorher nicht bewußt hineingeschrieben wurden.

Symbolische Modelle sind damit *virtuell* (von lat. *virtus*, Männerkraft, Tugend, Tauglichkeit): (1) der Möglichkeit oder der Kraft nach vorhanden – und dennoch (2) nur in der Erscheinung und im Auftreten vorhanden.

Symbolische Modelle können nämlich für ihre Verwender in allen praktischen Belangen den jeweiligen Bereich der realen Welt vertreten (Modelle im Sinne von Stellvertretern, s. [3]), haben objektive Wirkungen auf die mit ihnen arbeitenden Personen und sind dennoch dem Namen nach, einer strikten traditionellen Definition folgend, nicht „echt", sondern „bloß konstruiert". *Sie wirken, als ob sie echt wären*.

Der Sinn aller virtuellen Welten besteht darin, uns in den anderen realen Welten zu orientieren, sie verstehen, prognostizieren und gestalten zu können. Geeignete Software, zweckmäßig eingesetzt, kann hier einen guten Beitrag leisten, man denke etwa an entscheidungsunterstützende Systeme oder Simulationen von physikalisch-chemisch-geologischen Prozessen wie Wetter, Ökosystemen, Verbrennungsprozessen und so fort.

Virtualität ist also aus dieser Sicht nicht etwa ein absoluter Gegensatz zur Realität, sondern sie ist eine *innere, orientierende Form von Wirklichkeit*, die in besonderer Weise mit den handelnden Personen verbunden ist und nur schwer unabhängig von ihrem Standpunkt erfaßt werden kann.

4.2. Software-Entwicklung und Modellbildung als Transformationsprozeß

Ein möglicher Weg, um zu einer solchen virtuellen Welt zu kommen, ist die Bildung oder die Aneignung von Modellen und Theorien im entsprechenden Gegenstands- bzw. Objektbereich. Dabei kann Modellbildung als ein Transformationsprozeß (Abb. 3) verstanden werden, um ein Objekt von der realen in die virtuelle Welt zu überführen. Dadurch entsteht ein Modell dieses Objektes. Solche Modelle sind uns u. a. nicht nur aus der Physik, Biologie und Ökonomie bekannt, sondern unser tägliches Handeln wird bewußt oder unbewußt durch solche Modelle orientiert.

Da Software die Vergegenständlichung innerer, orientierender Tätigkeit ist, setzt die Entwicklung von Software in der Regel die Bildung eines Modells dieser Tätigkeit voraus. Statt „Modell" sagt man auch, daß man eine „Vorstellung" davon braucht, was man in der und mit der Software abbilden will.

Dabei versucht der Software-Entwickler, nicht immer bewußt ein Modell zu bilden. So kann er z. B. durch Probieren, Skizzieren, „Basteln" auf indirekte Weise – im Hintergrund – zu einem Modell kommen, ohne daß er sich dieses Modells

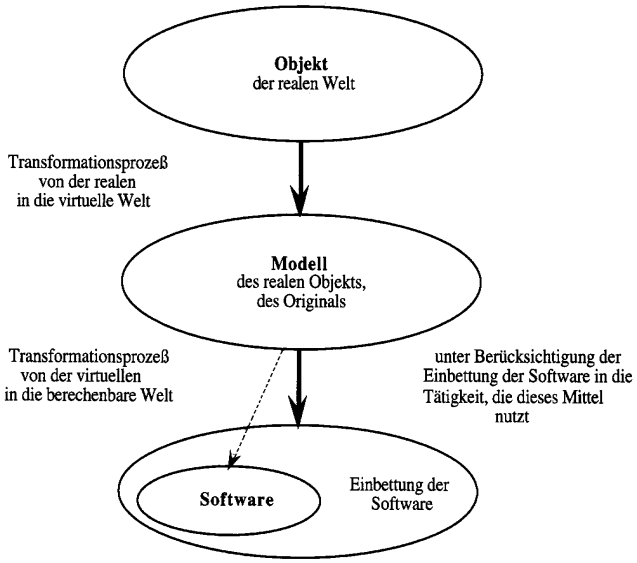

Abb. 3 Modellbildung und Softwareentwicklung als Transformationsprozeß (aus[2])

bewußt werden muß. Darüber hinaus gibt es zwei Hauptwege, um direkt zu einem solchen Modell zu kommen: durch *phänomenologische* Modellbildung oder durch *theorieorientierte* Modellbildung (s. ausführlich: [3]). Modellbildung kann auch eine relativ eigenständige Tätigkeit darstellen, die dann vor der Software-Entwicklung liegt und eine eigene Qualifikation erfordert (z. B. systemanalytische Spezialisierung oder Ausbildung als „Wissensingenieur").

Um Software zu produzieren, ist noch ein zweiter Transformationsprozeß notwendig, und zwar von dieser symbolischen, virtuellen Welt der Modelle in die berechenbare Welt des Computers mit den durch den Computer und durch die Software selbst gegebenen Grenzen. Dabei sind neben der Frage, was sich von einer Tätigkeit in Software abbilden läßt, noch formale Bedingungen, wie z. B. Berechenbarkeit, und solche, die durch die Programmiersprache und die Rechnerkonfiguration gegeben sind, zu berücksichtigen.

Andererseits ist bei einer solchen Transformation neben der implementationstechnischen Ebene (die sich primär auf die Operationen bezieht) auch die Oberflächen- und Interaktionsebene, die mit der Funktionsebene harmonisieren muß, zu berücksichtigen. Dazu werden die virtuellen Objekte der Designphase am besten so umgebaut, daß sie für den Anwender als leicht verständliche und wirksame Software-Objekte ihre orientierende, virtuelle Kraft entfalten können. Im Vertrauen auf die Verläßlichkeit der angestoßenen Operationen, der angezeigten Materialzustände und Aufgabenerfüllungsgrade können die Anwender ihre jeweilig nächsten Teilziele in Ruhe bestimmen und ansteuern. Neben einer entsprechenden Gestaltung der Arbeitsfenster (z. B. Werkzeugteil, Materialsicht, Orientierungspanel) und der Hilfefunktionen ist wieder die Einbettung der Software in die existierende Anwendertätigkeit von äußerster Wichtigkeit (s. Abschnitt 3). Die Anwender begrüßen in der Regel die Übernahme ihrer spezifischen Sprache, ihrer üblichen Arbeitsschritte in Dialoge und Menüs. Dies erfordert die Nachbildung ihrer gewohnten Mittel und Materialien als Software-Objekte.

5. Strategie der Transformation: Vom impliziten zum öffentlichen Wissen

Wissen als Resultat innerer Tätigkeiten kann in verschiedenen Formen vorliegen. Wissen, das in einer bestimmten Phase nur der Person zugänglich ist, die dieses Wissen besitzt, wird *implizites* Wissen genannt. Dieser Ausdruck ist mehrdeutig. Wir schlagen vor, zwei für den Entwurf von Software entscheidend wichtige Bestandteile des impliziten Wissens zu unterscheiden. Weitere Elemente, wie die emotionale Färbung, die Flexibilität in neuen Situationen, die partielle Unmöglichkeit der Mitteilung des impliziten Wissens, klammern wir hier also aus.

Für die Transformation in Modelle und Software kommen in Frage:
(1) *individuell reproduzierbares Wissen* (Können) und
(2) *mitteilbares* individuelles Wissen (Privatwissen).

Wenn man in der Lage ist, von einer vorgegebenen Ausgangssituation zu einer vordefinierten, angestrebten Situation zu kommen, verfügt man über individuell reproduzierbares Wissen, auch kurz *Können* genannt. Es bezieht sich darauf, wie man das ins Auge gefaßte Ziel einer Handlung zuverlässig erreicht. Es ermöglicht die Rekonstruktion bzw. die Wiederholung dieser Handlung und man weiß, was man tun muß, um in die Nähe der angestrebten Situation zu kommen. Mit anderen Worten, man verfügt wenigstens über ein gewohntes, flexibel einsetzbares Verfahren. Eine bewährte und selbstregulativ wirksame, intuitiv einsetzbare Methode ist dagegen schon eine höhere Stufe des Könnens, die typische Form, die das Expertenwissen annimmt (s. auch [18]). Im für die Softwareherstellung günstigsten Fall kennt man sogar einen Algorithmus, um zum angestrebten Resultat zu kommen.

Wenn wir unser individuelles Wissen anderen Menschen zur Verfügung stellen können, handelt es sich um *mitteilbares* Wissen, wobei folgende Arten der Mitteilung zu unterscheiden sind:

- Direkte Kommunikation, die insbesondere das Medium der Sprache, aber auch graphische Zeichen, sowie Gestik, Mimik und Dramatik, nutzt. Solche Medien müssen für die Beteiligten möglichst selbstverständlich und gleichzeitig unauffällig bleiben, da sonst die Direktheit und virtuelle Unvermitteltheit schlagartig abbrechen kann.
- Indirekte Kommunikation, die über einen Mittler erfolgt.

Ein solcher Mittler kann sein:
(a) ein Buch, eine Notiz, ein Brief, eine interaktive CD und dergleichen,
(b) eine Ton- oder Bildaufzeichnung einer direkten Kommunikation,
(c) der Überbringer einer mündlichen oder mimischen Botschaft,
(d) ein Software-Objekt, in dem der programmierbare Teil von ehemals individuellem Wissen und Können bereits vergegenständlicht wurde, zusammen mit der Dokumentation der Entwurfsprinzipien (Metaphern, vorausgesetzte Einbettung, usw.).

Mitteilbares Wissen, das nicht mehr an die Person gebunden ist, die es hervorgebracht hat, weil es in einer mitgeteilten, öffentlichen, kopierbaren und für das angesprochene Publikum verständlichen Form vorliegt, nennen wir *öffentliches* Wissen. Es ist nötig, hierbei mehrere Grade von Öffentlichkeit zu unterscheiden, um auch die übliche Geheimhaltung neuer Prinzipien (z. B. vor der Patentierung) unter diesen Begriff fassen zu können. In einem solchen Fall wäre dieses ja immerhin noch *gruppenöffentliches* Wissen.

Analog zum impliziten Wissen unterscheiden wir:
(3) öffentliches aneignungsfähiges Wissen, das wieder in individuelles *Können* (individuell reproduzierbar) verwandelt werden kann,
Diese Form des Wissens wollen wir *Verfahrenswissen* nennen.
(4) öffentliches aneignungsfähiges Wissen, das nicht auf die Verfahren beschränkt ist, sondern eine allgemeine orientierende Funktion hat, z. B. als gegenständliches Modell, technische Klassifikation oder Einzelfallbericht.
Diese Form des Wissens wollen wir als *Allgemeinwissen* bezeichnen. In den Kognitionswissenschaften wird es auch deklaratives Wissen genannt.

Nur öffentliches Wissen, das sowohl ein reproduzierbares Verfahren als auch das Modell des Gegenstandes, auf das sich das Verfahren bezieht, betrifft, und das bezogen auf die jeweilige Expertengemeinschaft allgemein verständlich ist, kann transparent und nachvollziehbar in Software übertragen werden.

Gerade für unscharfe Probleme bzw. Situationsbeschreibungen ist es jedoch typisch, daß hier noch überwiegend implizites Wissen vorliegt. Im Extremfall ist es noch nicht einmal mitteilbar. Eine Aufgabe der *Systemanalyse* besteht folglich darin, implizites Wissen in öffentliches und reproduzierbares zu überführen, wobei sie sich auf anderes Wissen (z. B. im Sinne von Analogie) stützen kann.

Wissen, das in mathematisierter Form öffentlich vorliegt, ermöglicht, erleichtert und unterstützt die Modellbildung sowie die Transformation der Modelle in Software. Dabei hängt der Grad der Mathematisierung eines Gegenstandsbereiches (als Anwendungsgebiet der Software) mit der Entwicklung der Wissenschaftsdisziplin (insbesondere seiner Theorienbildung und Methodenausprägung), zu der dieser Gegenstandsbereich gehört, zusammen [3].

Das andere Extrem besteht in individuellem Wissen, das noch nicht einmal mitteilbar ist, weil die Person selbst keine klare Sprache hat, um darüber zu sprechen, oder weil diese Sprache allzu eigenartig ist. Hier sind die Psychologen und andere Moderatoren gefordert und können auch eine Anzahl von Methoden der Wissensakquisition anbieten [15].

6. Kooperation und Evolution als organisatorischer Rahmen der Entwicklung von komplexen Softwaresystemen

Wir wollen nun der Frage nachgehen, welchen Beitrag Kooperation und Evolution für diesen Ansatz leisten können und umgekehrt dieser Ansatz für ersteres leisten kann. Kooperation und Evolution haben unter anderem in bezug auf die in Abschnitt 5

beschriebene Transformation eine „katalytische“ Funktion, um zu brauchbarer Software zu kommen.

Unter Evolution wird im allgemeinen eine allmählich fortschreitende Entwicklung verstanden. So ist z. B. ein Prozeß immer dann evolutionär, wenn dieser etwas Neues hervorbringt, dieses Neue auf schon Bekanntem aufbaut und sich das Neue im Vorhandenen bewähren muß. In der Natur ist die Entstehung des Neuen zufällig. Hier liegt eine gewisse Gefahr des evolutionären Herangehens, denn bei der Software-Entwicklung sollte das Design des zu entwickelnden Software-Objektes nicht dem Zufall überlassen werden. Trotzdem kennzeichnet diese Beschreibung etwas Typisches für evolutionäre Software-Entwicklung.

Andererseits kann man Evolution auch als Reaktion auf sich ändernde Bedingungen verstehen, d. h. als Anpassung (Adaptation) an diese geänderten Bedingungen. Hiermit haben wir einen Aspekt der Evolution innerhalb der Softwaretechnik. Er wird auch als Softwareevolution vgl. [4, 7, 20] bezeichnet (vgl. auch evolutionäres Prototyping [2, 5]).

Evolution im Sinne von Anpassung kann aber auch als ein bewußt beeinflußter Prozeß bei der Modellbildung und Konstruktion von Software verstanden werden. Hierbei geht es um eine Anpassung u. a. zwischen

- Zielvorstellung und Realität
- Gewünschtem und Möglichem
- Vorhandenem (Altem) und Gewünschtem (Neuem).

In diesem Fall kann auf die Ausgangs- und Randbedingungen der Evolution bewußt Einfluß genommen werden, indem u. a. das Anzustrebende sowie das Mögliche möglichst genau bestimmt wird. Dadurch kann der Zufall erheblich eingeschränkt (in manchen Fällen sogar aufgehoben) werden. Aus dieser Sicht ist „bewußtes“ evolutionäres Herangehen dadurch gekennzeichnet, günstige Ausgangs- bzw. Randbedingungen für den evolutionären Prozeß zu schaffen, die maßgeblich die „Richtung“ der Evolution beeinflussen. Eine Gefahr kann hier darin bestehen, eine „falsche“ Richtung – meistens schon ganz am Anfang – eingeschlagen zu haben (z. B. durch ungeeignete Methoden, nicht ausreichend methodisch kontrolliertes Vorgehen, zu schwach strukturierte (unterschiedlich interpretierbare) Situationsbeschreibung – zu wenig öffentliches Wissen).

Mit dem tätigkeitstheoretischen Ansatz hat man ein Mittel in der Hand, um zu solchen günstigen Ausgangs- bzw. Randbedingungen zu kommen.

Innerhalb dieses Ansatzes findet die Evolution als Lern- bzw. Selbstorganisationsprozeß statt, innerhalb dessen sich u. a. ein immer besseres Verständnis vom Gewünschten und dessen Realisierungsmöglichkeiten herausbildet. Mittel hierfür sind insbesondere die drei Analyseebenen (Abschnitt 3), Software-Entwicklung als Transformationsprozeß (Abschnitt 2 und 4) sowie die Überführung von implizitem Wissen in öffentliches.

Wie wir in Abschnitt 1 ausgeführt haben, sind an der Software-Entwicklung in der Regel mehrere beteiligt (vgl. g in Abschnitt 1), die sich relativ autonom bzw. mehr oder weniger kooperativ verhalten können (vgl. [3]).

Ob, inwieweit und mit wem einer eine Kooperation eingeht, hängt von der jeweiligen Situation ab. Darüber hinaus hängt eine Kooperation von der Kooperationsfähigkeit der beteiligten Personen ab. Allgemein kann man sagen: Ein Mensch wird in der Regel immer dann Kooperationsbeziehungen eingehen, wenn er Tätigkeiten durchführen möchte, die er allein nicht, nicht auf dem gewünschten Niveau oder nicht mit der nötigen Effektivität durchführen kann (dabei sind unterschiedliche Grade der „Notwendigkeit“ einer Kooperation sowie verschiedene Niveaustufen der Kooperation zu unterscheiden [3]).

Der Sinn einer Kooperation besteht in der Regel darin, Synergieeffekte nutzen zu wollen, die durch die Kooperation entstehen. Im Rahmen der (evolutionären) Software-Entwicklung können sich solche Synergieeffekte u. a. dadurch ergeben, daß

- die Beteiligten durch ihre verschiedenen „Sichten“ das zu lösende Problem vollständiger und allseitiger erfassen, konkretisieren und formulieren können (Vermeidung von nicht wahrgenommenen (nicht erfaßten) Wissensdefiziten),
- Wissen und Methoden der an der Kooperation Beteiligten (auch aus unterschiedlichen Disziplinen), die zur Lösung des Problems beitragen, integriert werden können,
- durch geeignete Kommunikation ermöglicht bzw. gefördert wird, Anteile von impliziten Wissen in öffentliches Wissen zu transformieren (z. B. bei der Systemanalyse, Modellbildung, Softwaregestaltung u. dgl.),
- durch entsprechende Selbstorganisation global nicht beherrschte Situationen, die jedoch lokal beherrschbar sind, bewältigt werden können.

Diese Prozesse finden in der Regel als Selbstorganisationsprozeß statt [3, 13, 14]. Der Prozeß selbst kann nicht zielgerichtet gesteuert werden, sondern nur seine Bedingungen. Er kann jedoch bewußt initiiert werden, falls die Randbedingungen für einen solchen Prozeß gegeben sind (s. z.B. [8]).

Gleichzeitig sollte eine solches Vorgehen jedoch folgendes ermöglichen:

- Mißverständnisse, wie sie in Abschnitt 1 angedeutet wurden, den an der Software-Entwicklung Beteiligten bewußt zu machen, ihnen zu ermöglichen, diese zu erkennen und gezielt angehen zu können,
- die zum Teil unterschiedlichen Sichten der Beteiligten (u. a. bezüglich Problem, Gegenstand und Ziel) auf die zu untersuchende Situation verstehen zu können.

Dafür liefert der hier beschriebene tätigkeitstheoretische Ansatz verschiedene Mittel (siehe a bis f in Abschnitt 1).

Dr. habil. Christian Dahme (links) ist Diplom-Mathematiker. Von 1970-1978 arbeitete er als Software-Entwickler an der Deutschen Bauakademie. Seit 1978 ist er an der Humboldt-Universität. Dort promovierte er 1983 in Informationsverarbeitung und habilitierte 1988 in Systemanalyse. Seine Forschungsschwerpunkte sind kooperative und evolutionäre Software-Entwicklung, Systemanalyse, Modellierung, sozialwissenschaftliche und psychologische Grundlagen der Informatik.
PD Dr. Arne Raeithel (rechts) war Diplom-Psychologe. Er promovierte 1982 an der FU Berlin und habilitierte 1991 an der Universität Hamburg.

Sein Forschungsspektrum innerhalb der Psychologie reichte von klinischen über arbeitspsychologische, kognitionswissenschaftliche, allgemeinpsychologische, semiotische und anthropologische bis hin zu methodischen Fragen sowohl im Bereich der Grundlagen- als auch der angewandten Forschung. Seit 1984 beschäftigte er sich mit verschiedenen psychologischen Fragen der Informatik, u.a. mit einer tätigkeitstheoretischen Begründung von Analyse-, Entwurfs- und Evaluationsmethoden, CSCW, Mensch-Maschine-Interaktion. Er entwickelte eine Erhebungs- und Analysesoftware für Repertory Grids zur Marktreife.

Literatur

1. Dahme, Christian: Selbstorganisation und Tätigkeitstheorie, in: Selbstorganisation – Jahrbuch für Komplexität in den Natur-, Sozial und Geisteswissenschaften, hrsg. von U. Niedersen, Bd. 1 „Selbstorganisation und Determination"; Dunker & Humblot, Berlin 1990, S. 149 ff
2. Dahme, Christian: Software-Entwicklung mit HyperCard – Benutzerfreundliche Interfacegestaltung; Verlag Addison Wesley 1995
3. Dahme, Christian: Systemanalyse menschlichen Handelns, Grundlagen und Ansätze zur Modellbildung, Westdeutscher Verlag, Opladen/Wiesbaden 1996
4. Floyd, Christiane; Krabbel, Anita; Ratuski, Sabine; Wetzel, Ingrid: Zur Evolution der evolutionären Systementwicklung – Erfahrungen aus einem Krankenhausprojekt; in diesem Heft
5. Floyd, Christiane: A Systematic Look at Prototyping. In: R. Budde, K. Kuhlenkamp, L. Mathiassen, H. Züllighoven (Hrsg.): Approaches to Prototyping. Proceedings of the Working Conference on Prototyping, Springer Verlag, Berlin, Heidelberg, 1984, S. 1–18
6. Gryczan, Guido: Prozeßmuster zur Unterstützung kooperativer Tätigkeit. Deutscher Universitätsverlag, Wiesbaden, 1996
7. Hesse, Wolfgang: Wie evolutionär sind die objektorientierten Analysemethoden? – Ein kritischer Vergleich, in diesem Heft
8. Kuhnt, Beate: Systemische Beratung in kooperativen Softwareprojekten, in diesem Heft
9. Leontjew, Alexei N.: Tätigkeit, Wissen, Persönlichkeit, Klett, Stuttgart 1977
10. Pomberger, Gustav; Weinreich, Rainer: Qualitative und quantitative Aspekte prototypingorientierter Software-Entwicklung – Ein Erfahrungsbericht, in diesem Heft
11. Nardi, Bonnie (Ed.): Context and Consciousness. Activity Theory and Human-Computer-Interaction, MIT Press, Boston 1996
12. Raeithel, Arne: Zur Ethnographie der kooperativen Arbeit. In Horst Oberquelle (Hrsg.). Kooperative Arbeit und Computerunterstützung. Stand und Perspektiven. S. 99–111, Verlag für Angewandte Psychologie, Stuttgart 1991
13. Raeithel, Arne: Ein kulturhistorischer Blick auf rechnergestützte Arbeit. In Wolfgang Coy und andere (Hrsg.). Sichtweisen der Informatik, S. 125–140, Vieweg, Braunschweig 1992
14. Raeithel, Arne: On the ethnography of cooperative work. In Yrjö Engeström and David Middleton (Eds.). Communication and Cognition at Work. Cambridge University Press (in press)
15. Raeithel, Arne, Velichkovsky, Boris M.: Joint Attention and Co-Construction. New ways to foster user-designer collaboration. In Bonnie Nardi (Ed.). Context and Consciousness. Activity Theory and Human-Computer-Interaction. pp. 199–233. MIT Press, Boston 1996
16. Wehner, Theo (Hrsg.): Sicherheit als Fehlerfreundlichkeit. Arbeits- und sozialpsychologische Befunde für eine kritische Technikbewertung. Westdeutscher Verlag, Opladen 1992
17. Wehner, Theo; Raeithel, Arne; Clases, Christoph; Endres, Egon: Von der Mühe und den Wegen der Zusammenarbeit. Ein arbeitspsychologisches Kooperationsmodell. In Egon Endres und Theo Wehner (Hrsg.). Zwischenbetriebliche Kooperation. S. 39–58, Psychologie-Verlags-Union, Weinheim: im Druck
18. Bromme, Rainer: Der Lehrer als Experte. Asanger, Heidelberg 1991
19. Budde, R., K.-H. Kautz, K. Kuhlenkamp, H. Züllighoven: Prototyping. Springer-Verlag, Berlin Heidelberg 1992
20. Lehman, M. M.: Programs, life cycles, and laws of software evolution; Proceedings of the IEEE, Vol. 68, No. 9, pp. 1060–1076 (1980)
21. Gryczan, G.; H. Züllinghoven: Objektorientierte Systementwicklung . Leitbild und Entwicklungsdokumente; Informatik-Spektrum, Heft 5 (1992) 15, S. 264–272

Eingegangen am 05.08.1996, in überarbeiteter Form am 06.01.1997