Gedanken zu Computern und Menschen

Dirk Siefkes:
Informatik als kulturelle Entwicklung
Sommersemester 1999

Christian Siefkes
Jamal Abu Hasan

Technische Universität Berlin
Sekr. FR 6–2
Franklinstr. 28/29, 10587 Berlin

error@cs.tu-berlin.de
http://tal.cs.tu-berlin.de/error/

28. Juli 1999
Version vom 6. August 1999

Inhaltsverzeichnis

Vorab

Aus der Ankündigung der Veranstaltung *Informatik als kulturelle Entwicklung*[1]:

> In Studium und Beruf orientieren wir uns nicht nur an wissenschaftlich-technischen Anforderungen, sondern ebenso (weniger bewußt) an sozialen und kulturellen Gegebenheiten und Zielen. Wie wirken sich diese Orientierungen auf die Entwicklung der Disziplin, ihre Forschung, Lehre und Anwendungen aus? Um die Frage konkret untersuchen zu können, arbeiten wir sorgfältig mit einigen wichtigen Texten der Informatikgeschichte: von Neumann, Backus, Dijkstra, Naur, Bauer u.a. Wir wollen sehen, wie die Texte von kulturellen, insbesondere wissenschaftlichen Orientierungen der Autoren geprägt sind. [...]
>
> Zu jedem Thema lesen die Teilnehmer eine Arbeit und evtl. Hintergrundliteratur, erstellen eine kommentierte Inhaltsangabe und schreiben ein kurzes Positionspapier. So entsteht während des Semesters eine Ausarbeitung der LV.

Dies sind unsere Beiträge dazu.

1 Was sehe, höre, fühle, denke ich beim Programmieren?

Sehen: nicht viel, außer dem Wesentlichen (Code, Fehlermeldungen ...).
Hören: was hat denn dieses Piepsen schon wieder zu bedeuten?
Fühlen: viel Freude oder völligen Frust.
Denken:
»Jetzt weiß ich, wie es geht« beim Codieren,
»Diesmal muß es aber klappen!« beim Compilieren,
»Warum macht er nicht einmal, was ich will?« beim Testen.

2 Metaphern[2]

Von Neumann beschreibt sein »Gerät« mit Ausdrücken, die für Menschen bzw. Lebenwesen überhaupt verwendet werden (organs, memory ...). In der Besprechung des Textes haben einige angemerkt, daß er nicht anders gekonnt hätte, als bekannte Begriffe aus anderen Bereichen zur Beschreibung dieses neuen Bereichs

[1] `http://tal.cs.tu-berlin.de/lv/ss99/basis_ig.html`
[2] Literatur: von Neumann, John (1987): First Draft of a Report on the EDVAC. In: *Papers of John von Neumann on Computing and Computer Theory*, Hg. Williams Aspray und Arthur W. Burks, Cambridge.

zu verwenden, und deshalb auf die menschliche Sphäre zurückgreifen mußte. Dem ersten Teil der Aussage stimme ich zu – natürlich hätte es die Verständlichkeit des Textes gesenkt, wenn er mit Neologismen wie »Informatik« und kryptischen Abkürzungen wie »RAM« und »PC« um sich geworfen hätte, und wo hätte er diese Begriffe überhaupt hernehmen sollen? Nicht aber dem zweiten: Statt seine Geräte implizit als Lebewesen zu charakterisieren, hätte er auch Begriffe der vorhandenen technischen Terminologie bzw. aus Industrie, Verkehr oder Landwirtschaft etc. übertragen können – wie das andere ja auch gemacht haben. Während von Neumann von »memory« (Gedächtnis) spricht, nennt Charles Babbage im 19. Jahrhundert die vergleichbare Komponente seiner *Analytical Engine* »store« (Lager, Speicher).

3 Maschine Mensch?[3]

Angeregt durch John von Neumann werden in den USA Parallelen zwischen Menschen/Lebewesen und Computern gezogen. Anders in Deutschland, wo von »Speicher« statt von »memory« gesprochen wird und Begriffe wie »Elektronengehirn« von der Fachwelt nie ernstgenommen werden und schließlich verschwinden.

Hier werden stattdessen Parallelen zu den Rechenbüros herausgearbeitet, wo zahlreiche Menschen (meist Frauen, in den USA damals »computer« genannt) mit mechanischen Rechenmaschinen nach Anweisung Berechnungen durchführen. Peter Eulenhöfer zeigt, wie die Computerfrauen mit dieser Interpretation abgewertet werden; sie werden für ersetzbar erklärt und bestenfalls für Hilfsarbeiten eingesetzt. Um so höher wird die Rolle der (männlichen) Mathematiker bewertet, die die Rechenbüros überwachen und die Anweisungen geben; aus ihnen werden die Informatiker (heute noch oder wieder zu 90 % männlich), die Computer bauen und programmieren. Ihre Aufgaben gelten als höchst anspruchsvoll und keineswegs automatisierbar.

Schauen wir wiederum in die USA: Hier wird letztlich der ganze Mensch dem Computer gleichgesetzt, also für automatisierbar erklärt. Auch wenn dies von Neumann selbst vielleicht noch nicht so gemeint hat, legt sein Vergleich der Elemente der Zentraleinheit eines Computers mit denen des menschlichen Gehirns eine solche Interpretation nahe. In mehr oder weniger ausgeprägter Form werden derartige Vorstellungen denn auch von vielen aufgegriffen, am extremsten von Verfechtern der »Künstlichen Intelligenz« wie Marvin Minsky und Hans Moravec, der den Computer für den evolutionären Nachfolger des Menschen hält. Von Neumann, der »Schurke«, dürfte entscheidend dazu beigetragen haben, daß diese Sichtweise sich entwickeln konnte.

[3] Literatur: Eulenhöfer, Peter (1998a): Der Informatiker als »deus ex mathematica«. In: *Sozialgeschichte der Informatik. Kulturelle Praktiken und Orientierungen*, Hg. Dirk Siefkes; Peter Eulenhöfer; Heike Stach und Klaus Städtler, Wiesbaden: Deutscher Universitäts-Verlag, S. 257–273.

Wie sind die Auswirkungen *dieser* Gleichsetzung? Bedeutet auch dies eine Abwertung des Gleichgesetzten, hier also des Menschen überhaupt?

Er scheint, daß sich viele Menschen nicht abgewertet fühlen, sondern dieses Menschenbild als ganz selbstverständlich empfinden (Zitat eines Theorie-Tutors: »Ich bin eine probabilistische Turingmaschine«). Offen bleibt dabei allerdings: Wer sind die Informatiker, die Kontrolleure des Menschen? Wer schreibt die Programme?

Computer tun, was von ihnen erwartet wird, und denken nicht nach, ob es auch anders ginge. Sehr viele Menschen verhalten sich ebenso. Hat das damit zu tun, daß sie tatsächlich wie Computer sind? Oder daß sie dieses Selbstbild akzeptiert haben und sich entsprechend verhalten? Ich hoffe, letzteres.

4 Verständnis[4]

»Programmers rightly regarded their work as a complex, creative art that required human inventiveness« [S. 25]. Backus und sein Team haben diese Vorstellung zum Teil zerstört. Ihr Fortran-Compiler und seine Nachfolger erzeugen fast ebenso guten Code wie die besten Programmierer[5].

Der Computer kann nicht viel. Er kann nicht denken und nicht fühlen. Er kann nicht kreativ sein. Er kann keine Probleme lösen, außer man sagt ihm genau, wie es geht. Er hat kein Verständnis. Wie gut der Computer mit einem Problem fertig wird, sagt uns etwas über die Natur dieses Problems.

Schachspielen wurde früher als eine der herausragendsten menschlichen Fähigkeiten betrachtet. Schachspieler galten als Genies. Diese Einschätzung war offensichtlich falsch. Computer können heute ebensogut Schach spielen wie die besten Menschen. Dabei sind sie keineswegs genial, sondern ausgesprochen beschränkt.

Sie scheitern etwa kläglich, wenn es um Dinge geht, die schon Kindern keine Probleme bereiten: die Bedeutung eines Satzes verstehen etwa. Es sieht nicht so aus, als ob sich dies ändern wird. Anders als beim Schachspielen werden hier Qualitäten benötigt, die genuin menschlich sind.

Beim »Codieren« in Maschinensprache sind solche Qualitäten unwichtig, sonst hätten die Compiler ihren Siegeszug nicht angetreten. Ein anderes Bild ergibt sich, wenn es um das Formalisieren eines Problems und seine Umsetzung in eine »höhere« Programmiersprache geht. Vom »Aussterben der Programmierer«, das einst prophezeit wurde, kann keine Rede sein. Ob Formalisieren eine Kunst ist, sei dahingestellt. Verständnis erfordert es allemal.

[4] Literatur: Backus, John (1981): The History of Fortran. In: *History of Programming Languages*, Hg. Richard L. Wexelblat, New York et al.: Academic Press, S. 25–45.

[5] Im Singular verwenden wir meist die weibliche Form, im Plural die kurze, die wir als geschlechtsneutral verstehen.

5 Sozialisation[6]

»Die dringlichste Aufgabe der Reform des Informatikstudiums ist die Abschaffung des Grundstudiums« [S. 3].

Aber das geht doch nicht! Wer sich Diplom-Informatikerin nennen will, deren Studium muß sich an der Rahmenordnung orientieren. Und die schreibt nicht nur die Existenz, sondern auch recht genau Form und Inhalte des Grundstudiums vor. Wer keinen Titel hat, verdient weniger Geld. Und ohne Zweifel wird es für den Bachelortitel bald eine ähnliche Rahmenordnung geben.

Das Grundstudium existiert nicht ohne Grund. Daß »nichts dabei herauskommt«, stimmt nur für den Stoff, der offiziell vermittelt werden soll. Dafür stellt es »die Wurzeln der Sozialisation des ›studierten Informatikers‹« dar, wie der Autor unserer künftigen Studienordnung genau weiß. Das Grundstudium hat Konsequenzen. Wer es durchlaufen hat, hat bestimmte Vorstellungen über die Informatik, ihre Grundlagen und Vorgehensweisen erworben. Diese sind meist so verinnerlicht, daß sie kaum zu hinterfragen sind. Damit ist in aller Regel sichergestellt, daß die künftigen Informatiker nicht allzuviel anders machen als ihre Vorgänger.

Warum sollte man in der Informatik überhaupt etwas anders machen? Die Informatik ist doch eine glückliche Disziplin: sie profitiert – man sieht es am Jahr-2000-Problem – von ihren eigenen Fehlern. Unglücklich sind nur die Anwender, die mit der Informationstechnik leben müssen, die ihnen die Informatiker vorsetzen. Aber sie haben keine Wahl. Bekanntlich leben wir in der Informationsgesellschaft: ohne Computer geht nichts mehr. Sie hätten mehr Wahl, wäre die Informatik vielfältiger. Das würde die Bedeutung und den Einfluß aller, die Informatik auf die herkömmliche Weise betreiben, verringern. Sie wären nicht mehr unentbehrlich.

Das Grundstudium sichert ihre Art der Informatik vor Alternativen, und damit ihre Macht. Das ist sein Zweck; darum wird es nicht abgeschafft werden. Zu viele Leute profitieren davon.

6 Offene Standards[7]

Peter Naur schreibt: »A motive that was directly associated with the ACM ALGOL comittee [...], was to establish ›a single universal computer language‹« [S. 114]. Und Alan Perlis betont, daß in den USA insbesondere die Benutzergruppen Interesse an einem einheitlichen Standard hatten [S. 76]. Kein Wunder, waren

[6] Literatur: Siefkes, Dirk (1999): Hybridobjekte als Gegenstände der Informatik. In: *Die Rolle von Schemata in der Informatik als kultureller Entwicklung*, Berlin: TU Berlin, S. 5–14.

[7] Literatur: Perlis, Alan J. (1981): The American Side of the Development of Algol. In: *History of Programming Languages*, Hg. Richard L. Wexelblat, New York et al.: Academic Press, S. 75–91.
Naur, Peter (1981): The European Side of the Last Phase of the Development of Algol 60. In: *History of Programming Languages*, Hg. Richard L. Wexelblat, New York et al.: Academic Press, S. 92–139.

sie doch durch die proprietären Sprachen fest an die einmal gewählte Plattform gebunden. Bei einem Wechsel der Hardware war in der Regel auch die selbstgeschriebene Software hinfällig – ein teurer Spaß. Eine maschinenunabhängige Sprache, die nur einfaches Neucompilieren bzw. geringfügige Anpassungen erforderte, hätte das Umsteigen sehr erleichtert.

Dies war natürlich nicht im Interesse der Hersteller, die von einer möglichst festen Bindung ihrer Kunden an ihre Produkte profitierten. Daher verwundert es nicht, daß es auch heute »offene Standards« schwer haben. Ein Beispiel ist die Programmiersprache Java, bei deren Konzeption Maschinenunabhängigkeit erklärtes Ziel war. Microsoft versucht, dies aufzuweichen und die Benutzer mit »nützlichen Erweiterungen«, die natürlich nur unter Windows laufen, an die eigene Plattform zu binden. Ähnliches gilt für HTML und JavaScript, wo sich Netscape und Microsoft ein Wettrennen mit inkompatiblen Erweiterungen geliefert haben.

Für Hersteller, die keine große Bedeutung auf dem Markt haben, ist die Einhaltung offener Standards im eigenen Interesse. So sah Sun mit der Etablierung von Java die Chance, seine exotisch gewordene Plattform zu profilieren, und IBM warf sich voll auf Java, in der Hoffnung, damit sein kaum noch verwendetes Betriebssystem OS/2 wiederbeleben zu können. Sobald bzw. solange die eigene Bedeutung ein gewisses Maß überschreitet, steigern die offenen Standards dagegen eher die Chancen der Konkurrenz als die Konkurrenzfähigkeit und sind entsprechend unbeliebt. So bekannte sich Netscape erst zu offenen Standards, als es unter dem Angriff Microsofts in die Knie zu gehen drohte – vorher hatte es selbst HTML mit proprietären Erweiterungen überladen.

Die Interessen der Hersteller sind mit denen der Benutzer also nur zeitweilig identisch. In letzter Zeit allerdings beginnen die Benutzer verstärkt, die Sache selbst in die Hand zu nehmen. Bei freier Software, auch »OpenSource« genannt, gibt es keinen Hersteller, der die alleinige Kontrolle über ein Programm hat. Oft wird ein Programm von Menschen geschrieben, die in aller Welt verteilt sind. Die Autoren sind zugleich Benutzer ihrer Software. Doch auch wenn eine Firma beteiligt ist, bleibt die »Offenheit« der Software dadurch gesichert, daß der Quellcode allgemein zugänglich gemacht werden muß und von allen modifiziert werden darf. Daher kann keine Firma die Software mit proprietärer Technologie »verschmutzen«, bei der andere außen vor bleiben.

Natürlich ist OpenSource zur Zeit ein Hype; die dauerhafte Bewährung muß abgewartet werden. Ob Linux, GNU und Co. die Dominanz der Marktgiganten (einst IBM, heute Microsoft) brechen können, ist unklar. Aber man kann hoffen ...

7 Diskussionen[8]

Nachdem Peter Naur handstreichartig die Rolle eines »Papstes« bei der Festschreibung von Algol 60 ergriffen hatte [Bauers Kommentar, S. 129], wurde der Entscheidungsprozeß bezüglich des Aufbaus der Sprache stark reglementiert. Unzufrieden mit den »unangemessenen« mündlichen Diskussionen [S. 98] setzte Naur durch, daß zu jedem Thema alle Komitteemitglieder individuell Änderungsvorschläge erarbeiteten und schriftlich vorlegten. Über diese Änderungsvorschläge wurde dann abgestimmt. Die Diskussion wurde auf die Beseitigung von Unklarheiten und Fehlern beschränkt.

Hat Naur dem Algol-Projekt mit seiner Vorgehensweise einen Gefallen getan? Sicher konnten so in kurzer Zeit mehr Entscheidungen getroffen werden, so daß die Sprache schneller zu ihrer endgültigen Form kam. Möglicherweise jedoch um den Preis, daß die so gefundenen Lösungen weniger überzeugend und ausgereift waren, als wenn man sich mehr Zeit zur Diskussion gelassen hätte. Schließlich entspringen gute Ideen oft nicht einem einsam grübelnden Geist, sondern dem Hin- und Herspringen der Gedanken in einem Gespräch, einer Diskussion. Schwächen und Ansätze zum Weitermachen in einer Idee werden eher von anderen entdeckt, genauso wie Fehler in einem Text der Autorin selbst am wenigsten auffallen. Spontane Reaktionen einer Zuhörerin decken Schwachstellen, Unverständliches und Nicht-Einleuchtendes sofort auf. Dieses unmittelbare Feedback fehlt bei einer schriftlichen Darstellung. Oft können auch andere da weitermachen, wo eine nicht mehr weiter weiß. Daher ist der unmittelbare Austausch für die Entwicklung von Gedanken so wertvoll.

Kein Wunder, daß Naurs Vorgehensweise zu Problemen führte: so wurde über Prozeduren, ein zentrales Sprachkonzept, erst in der allerletzten Stunde in einer Kampfabstimmung entschieden [Bauer, S. 130]. Verschiedene andere Vorschläge zu diesem Thema waren in früheren Abstimmungen abgelehnt worden, doch letztlich brauchte man eine Lösung. Daß der angenommene Vorschlag zweideutig war, fiel zwar schnell auf, doch da war das Treffen schon vorbei. Nachdem sich Naur mit einigen Komitteemitgliedern ausgetauscht hatte, entschied er sich schließlich für eine Lösung, über die allerdings nicht abgestimmt wurde [S. 111f].

Besser wäre es gewesen, diese und andere Fragen jeweils so lange zu diskutieren, bis sich eine allgemein oder mehrheitlich akzeptable Lösung abgezeichnet hätte. Dann hätten immer noch einzelne Komitteemitglieder beauftragt werden können, die so gemeinsam gefundene Lösung schriftlich festzuhalten und der Gruppe später zur abschließenden Diskussion vorzulegen, um Naur Arbeit abzunehmen und eine konsistente Arbeitsgrundlage sicherzustellen. Damit wäre das parallele Arbeiten an Vorschlägen, die gegeneinander konkurrierten oder sich eigentlich gut ergänzt hätten, vermieden worden. Der freie Gedankenaustausch wäre nicht unterdrückt worden.

[8] Literatur: Naur, Peter (1981): The European Side of the Last Phase of the Development of Algol 60. In: *History of Programming Languages*, Hg. Richard L. Wexelblat, New York et al.: Academic Press, S. 92–139.

8 Begriffe[9]

Wieweit sind die üblichen Begriffe angemessen für das, was bei uns *Informatik* und im englischen Sprachraum *computer science* heißt?

An dem Begriff *computer science* kritisiert Zemanek [S. 157], er sei einerseits zu weit, da er auch die technische Seite des Computerbaus umfaßt, andererseits zu eng, da völlig von der Maschine Computer abhängig. Relevant ist die Programmierung, die Software, unabhängig von der Hardware, der Rechenmaschine, wie auch Bauer feststellt [S. 336].

Die Alternative *computing science* konnte sich nie durchsetzen. Sie würde die Anbindung an das Gerät vermeiden und die Betonung darauf legen, wo man in der Informatik meist landet: beim Berechnen. Laut Zemanek wird die Rolle der Berechnung damit überbewertet [S. 157]. Die Berechnung ist zwar als *Mittel* der Informatik essentiell, sie macht aber nicht unbedingt deren *Wesen* aus. Schließlich will man als Informatikerin nicht mit abstrakten mathematischen Ideen arbeiten, sondern Probleme und ihre Lösungen beschreiben, in eine für die Maschine faßbare Form bringen. Daher ist der Begriff *Informatik* angemessen, wenn man ihn mit Zemanek anhand seines lateinischen Ursprungs versteht: »Im Lateinischen bedeutet *informare* Form geben oder Beschreiben« [S. 157].

Oft wird der Begriff *Informatik* allerdings so verstanden, daß es dabei um die Verarbeitung von *Informationen* geht (so beispielsweise die Selbstdarstellung des Fachs an den Unis Hamburg, Frankfurt und Saarbrücken). Das ist unglücklich, da Computer nur mit *Daten* umgehen können. Zu Informationen werden diese erst durch menschliche Interpretation. Daher sind Behauptungen der Art »Die gesamte Informationsmenge hat sich in den letzten 50 Jahren um den-und-den Faktor gesteigert« unsinnig, da sich Informationen nicht objektiv messen lassen. Ein Film in Fernsehqualität enthält etwa die 100.000fache Menge an Daten (Bytes) wie ein Buch; beim Informationsgehalt dürfte es dagegen häufig zweifelhaft sein, ob überhaupt von einer *Steigerung* gesprochen werden kann. Es geht um die Verarbeitung von Daten, nicht von Informationen. Daher hat Peter Naur den Begriff *datalogi* vorgeschlagen, der wohl auch im dänischen Sprachraum eine gewisse Bedeutung erlangt hat.

Aber natürlich ist längst entschieden, welcher Begriff allgemein verwendet wird, ganz unabhängig von seiner Angemessenheit: Im deutschen Sprachraum werden wir mit dem Begriff *Informatik* leben müssen, ob wir wollen oder nicht. Daher tun wir gut daran, uns nicht durch den Anklang an *Information* verwirren zu lassen, sondern mit Zemanek an seine Wurzeln und an die »Beschreibung [...] als Zentralproblem der Informatik« [S. 160] zu denken.

[9] Literatur: Bauer, Friedrich L. (1974): Was heißt und was ist Informatik? *IBM Nachrichten*, 24(223):333–337.
Zemanek, Heinz (1971): Was ist Informatik? *Elektronische Rechenanlagen*, 13(4):157–161.

9 Vom Problem zum Programm[10]

Ein Programm, das macht, was es machen soll, entsteht in zwei Schritten: Zuerst wird aufgrund des Problems und der gewünschten Lösung die formale Spezifikation erstellt, die festlegt, zu welchen Ergebnissen das Programm je nach Situation kommen soll. Offen bleibt dabei, *wie* es dazu kommt: Dies wird im zweiten Schritt festlegt, der eigentlichen Programmierung. Dabei muß gezeigt werden, daß das *Wie* dem *Was* entspricht, daß das Programm die Anforderungen der Spezifikation erfüllt. Da sowohl Programm wie auch Spezifikation mathematische Ausdrücke, große Formeln [S. 1401] sind, kann und muß dies mathematisch bewiesen werden. Softwaretests reichen dazu nicht aus, da sie nie alle Fälle abdecken können und daher nur das Vorhandensein, nicht aber die Abwesenheit von Fehlern zeigen können (genauso wenig würde sich eine Mathematikerin anhand von Beispielen von der Wahrheit eines Theorems überzeugen lassen – lediglich die Falschheit kann so gezeigt werden).

Die Konsequenzen, die Dijkstra aus dieser Sichtweise zieht (Informatik-Erstsemester sollen in einer Sprache programmieren, die nicht implementiert ist, so daß sie von Anfang an beweisen müssen und nicht testen können), sind zwar radikaler als bei anderen – die Position selbst ist jedoch weit verbreitet.

Auch wenn man diese Position mit ihrer Trennung in zwei Schritte, die hintereinander ausgeführt werden, akzeptiert, bleibt die Frage: Warum wird der erste Schritt so vernachlässigt? Auf den Schritt vor der Spezifikation geht Dijkstra in seinem Einsteigerkurs nicht ein. In unserem Grundstudium ist es ebenso: Fast immer werden kleine, unzusammenhängende Programmieraufgaben verlangt, bei denen die formalisierte oder leicht formalisierbare Spezifikation vorgegeben ist. Der Übergang von Sachverhalten in der (nicht formalisierbaren) Welt zu Spezifikationen kommt in den Pflichtveranstaltungen nie, in Praktika nur in Ausnahmefällen ins Bild. Die logische Konsequenz dieser Trennung wäre jedoch, beiden Schritten gleiche Aufmerksamkeit zu bieten, etwa gleich viele Veranstaltungen zum Schritt vor und zum Schritt nach der Formalisierung anzubieten.

Allerdings ist zweifelhaft, ob diese zeitliche Trennung überhaupt sinnvoll ist, wie M.H. van Emden anmerkt [S. 1408]. Halten wir daran fest, stehen wir am Ende vielleicht mit einem Programm da, das zwar die Spezifikation erfüllt, aber nicht das macht, was wir eigentlich wollen. Schließlich können nicht nur im Programm, sondern auch schon in der Spezifikation Fehler sein. Und diese Fehler kann kein Beweis und keine Symbolmanipulation entdecken …

[10] Literatur: Dijkstra, Edsger W. (1989): On the Cruelty of Really Teaching Computing Science. *Communications of the ACM*, 32:1397–1414.

10 Theorien[11]

Peter Naurs Sichtweise auf das Programmieren ist gefährlich. Für Naur ist das wichtigste Ergebnis des Programmierens nicht die Erzeugung von Programmtext und Dokumentation, sondern die Bildung von Theorien zur Lösung des jeweiligen Problems. Eine derartige Theorie liegt jedem Programm implizit zugrunde. Man muß sie verstehen, um das Programm erfolgreich erweitern und anpassen zu können. Doch dieses Verständnis kann nicht aus Quellcode und Dokumentation allein gewonnen werden; der persönliche Austausch mit den ursprünglichen Programmierern ist dafür unabdingbar. Ein Programm, das von seinen ursprünglichen Entwicklern verlassen wurde, ist tot; es kann von einem anderen Team nicht konsistent weiterentwickelt werden.

Naurs Sicht ist gefährlich für Firmen, die damit in Abhängigkeit von ihren Programmierern geraten. Sie können nicht mehr einfach eine Programmiererin oder ein Programmierteam durch andere ersetzen, ohne um ihre bisherige Software fürchten zu müssen. Das »Software Engineering« entstand gerade mit dem Ziel, diese Abhängigkeit zu vermeiden, die Softwareentwicklung vom Individuum zu lösen. Häufig richteten sich die Texte zur Popularisierung des Software Engineerings sogar explizit an Manager mit dem Hinweis, daß die Kontrolle über die Entwicklung damit in die Hände des Managements übergehe. Naur macht diese Hoffnung zunichte.

Seine Sicht ist aber auch gefährlich für die Informatiker. Sie verleitet möglicherweise zur Selbstüberschätzung (»ohne mich geht es nicht«) und zum wilden Drauflos-Hacken (»warum soll ich mich denn um eine klare Programmstruktur oder um Dokumentation bemühen, wenn das eh nicht reicht, das Programm zu verstehen«).

Natürlich sind diese Gefahren kein Argument gegen die Plausibilität von Naurs Sichtweise. Die Erfahrung spricht für seine Theorie. Auch 30 Jahre Software Engineering und immer weiter verfeinerter Entwicklungsmethodologien haben nichts daran geändert, daß es bei großen Systemen oft einfacher ist, sie neu zu schreiben, als den alten Code von einem neuen Team modifizieren zu lassen.

Wir sollten den Tatsachen ins Auge sehen und mit Naurs Erkenntnis leben lernen.

[11] Literatur: Naur, Peter (1992): Programming as Theory Building. In: *Computing: A Human Activity*, New York: ACM Press, S. 37–49.

11 Hierarchien[12]

Das Paradigma der Objektorientierung stammt ursprünglich aus der Sprache *Simula*, die speziell für Simulationen geschrieben wurde. Es tritt an mit dem Versprechen, die Wirklichkeit besonders gut modellieren, abbilden zu können.

Das wichtigste in einem objektorientierten Programm sind, ganz klar, die *Objekte*. Wie sich ein Objekt verhält, ist in der dem Objekt zugewiesenen *Klasse* festgelegt. Klassen werden von höheren Klassen abgeleitet, deren Werte und Verhalten sie *erben* – beispielsweise sind *Mercedes* und *Ford* beide von *Auto* abgeleitet, das wiederum von *Fahrzeug* abgeleitet ist. So sind sämtliche Klassen in der *Klassenhierarchie* zusammengefaßt und direkt oder (in der Regel) indirekt alle Erben der höchsten Klasse (die in Java *Object* heißt).

Diese hierarchische Herangehensweise ist deshalb besonders angemessen zur Modellierung der Wirklichkeit, weil sie der Struktur der Wirklichkeit entspricht – so scheinen die Prediger des objektorientierten Paradigmas zu denken. Im täglichen Leben sollen wir von Objekten (Autos, Kaffeemaschinen, Bäumen . . .) umgeben sein, behaupten Gosling und McGilton [S. 33]. Diese Objekte haben einen *Zustand* (Geschwindigkeit, Richtung, Benzinverbrauch eines Autos) und *Verhaltensmöglichkeiten* (losfahren, anhalten, wenden, gegen Bäume fahren) [S. 34].

Natürlich ist das ein Irrtum. Objekte und Klassenhierarchien sind nicht immanent in der Welt vorhanden, wo sie nur entdeckt werden müssen. Sondern sie müssen so gewählt werden, daß sie für den Zweck eines Programms Sinn machen. Sie müssen *erfunden*, nicht *gefunden* werden.

So ist es meine Entscheidung, ob ich die Klassen *Auto* und *Fahrrad* beide von *Straßenfahrzeug* ableite, und das wiederum von *Fahrzeug*, das auch die Oberklasse von *Luftfahrzeug* ist, von der wiederum *Düsenjet* und *Segelflieger* abgeleitet werden. Oder ob ich *Auto* und *Düsenjet* von *Motorfahrzeug* ableite und *Fahrrad* und *Segelflieger* von *MotorlosesFahrzeug*, die beide Erben von *Fahrzeug* sind. Das liegt nicht daran, daß ich die beste, die »eigentliche« Struktur noch nicht erkannt habe. Unterschiedliche Hierarchien können sinnvoll sein, je nachdem, worum es mir in meinem Programm geht.

In einem objektorientierten Programm wird der Zustand eines Objekts durch eine endliche Anzahl von Variablen, sein Verhalten durch eine endliche Anzahl von Prodezuren festgelegt. In der Wirklichkeit sind dagegen Eigenschaften und Verhaltensmöglichkeiten etwa eines Autos prinzipiell nicht aufzählbar. Es kann immer wieder zu überraschendem, unerwartetem Verhalten kommen oder eine Eigenschaft kann plötzlich eine Rolle spielen, an die ich vorher nie gedacht hatte. Ich muß mich entscheiden, was mir (im Programmkontext) wichtig ist und wovon ich meine, absehen zu können.

[12] Literatur: Gosling, James und Henry McGilton (1996): *The Java Language Environment. A White Paper*. Mountain View, Ca: Sun. Online:
`http://java.sun.com/docs/white/langenv/`
Zugriff am 30. Juni 1999.

Objekte, Klassen und Hierarchien entspringen also meiner Theorie (in Naurs Sinn) zum Erreichen des Programmziels; sie haben keine Entsprechung in der Wirklichkeit. Die Objektorientierung ist ein sehr mächtiges Paradigma. Aber über die Welt sagt sie uns nichts.

12 Aus Geschichte lernen?[13]

Eins der Ziele des Interdisziplinären Forschungsprojekts *Sozialgeschichte der Informatik* war es, »aus den gewonnen Erkenntnissen Perspektiven für die zukünftige Gestaltung der Informationstechnik und der Wissenschaftsdisziplin Informatik abzuleiten« [S. 3].

Aber geht das überhaupt? Historiker haben da ihre Zweifel: »sie stritten ab, daß aus der Geschichte Erklärungsmuster oder gar Prognosen abgeleitet werden können« [S. 29].

Was bringt es dann, sich mit der Geschichte des eigenen Fachs zu beschäftigen, wenn sich daraus keine Verhaltensregeln herleiten lassen? Ist das nicht Zeitverschwendung? Sollte man stattdessen nicht lieber weitere Methoden und Techniken lernen, die man später im Leben – im Beruf! – unmittelbar einsetzen kann?

Sicherlich wird man bei der Beschäftigung mit der Geschichte des eigenen Fachs so einiges nebenher lernen, was man später gut gebrauchen kann. Texte kritisch lesen und selber Texte schreiben etwa – beides spielt im normalen Informatikstudium keine große Rolle. Einen eigenen Standpunkt finden, in Diskussionen vertreten und weiterentwickeln lernt man vielleicht auch.

Natürlich braucht man für all das nicht unbedingt geschichtliche Veranstaltungen. Die Beschäftigung mit der Geschichte bringt aber noch mehr: Man lernt die Methodologie eines anderen wissenschaftlichen Fachs kennen – der Geschichte eben. Die Absolutheit der Methoden des eigenen Fachs wird damit relativiert (zumal sich die im allgemeinen eher ingenieursmäßig betriebene Informatik in ihren Herangehensweisen deutlich von der Geschichtsschreibung unterscheidet).

Alle, die ein Nebenfach machen oder sich im Wahlfach mit anderen Fächern auseinandersetzen, lernen natürlich ebenfalls andere wissenschaftliche Ansätze kennen – meist allerdings völlig getrennt von der Informatik; Verbindungen werden nicht hergestellt. Die Beschäftigung mit der Geschichte des eigenen Fachs überwindet diese Trennung: man wird mit einer neuen Sichtweise auf vertraute Dinge konfrontiert.

Die geschichtliche Sichtweise kann es auch ermöglichen, angebliche Notwendigkeiten in Frage zu stellen. Eulenhöfer drückt die Hoffnung aus, daß die geschichtliche Distanz es ermöglicht, »scheinbar selbstverständliche Sprech-, Denk- und Handlungsweisen als soziale und kulturelle Konstrukte zu erkennen und zu

[13] Literatur: Eulenhöfer, Peter; Dirk Siefkes und Heike Stach (1998): Sozialgeschichte der Informatik. *FIfF-Kommunikation*, (2):3–4.
Eulenhöfer, Peter (1998b): Disziplingeschichte und die Disziplinierung der Geschichte. *FIfF-Kommunikation*, (2):29–33.

benennen« und damit einen differenzierteren Blick auf die heutige Informatik zu gewinnen [S. 32]. Doch dafür muß man sich zuerst von dem Bild eines linearen Geschichtsverlaufs verabschieden, dem zufolge die heutige Situation der (bisherige) Höhepunkt einer langen Reihe konsequenter Entwicklungen ist.

Die Beschäftigung mit der Geschichte ist kein Allheilmittel – aber eine Chance.

Literatur

Backus, John (1981): The History of Fortran. In: *History of Programming Languages*, Hg. Richard L. Wexelblat, New York et al.: Academic Press, S. 25–45.

Bauer, Friedrich L. (1974): Was heißt und was ist Informatik? *IBM Nachrichten*, 24(223):333–337.

Dijkstra, Edsger W. (1989): On the Cruelty of Really Teaching Computing Science. *Communications of the ACM*, 32:1397–1414.

Eulenhöfer, Peter (1998a): Der Informatiker als »deus ex mathematica«. In: *Sozialgeschichte der Informatik. Kulturelle Praktiken und Orientierungen*, Hg. Dirk Siefkes; Peter Eulenhöfer; Heike Stach und Klaus Städtler, Wiesbaden: Deutscher Universitäts-Verlag, S. 257–273.

Eulenhöfer, Peter (1998b): Disziplingeschichte und die Disziplinierung der Geschichte. *FIfF-Kommunikation*, (2):29–33.

Eulenhöfer, Peter; Dirk Siefkes und Heike Stach (1998): Sozialgeschichte der Informatik. *FIfF-Kommunikation*, (2):3–4.

Gosling, James und Henry McGilton (1996): *The Java Language Environment. A White Paper*. Mountain View, Ca: Sun. Online:
`http://java.sun.com/docs/white/langenv/`
Zugriff am 30. Juni 1999.

Naur, Peter (1981): The European Side of the Last Phase of the Development of Algol 60. In: *History of Programming Languages*, Hg. Richard L. Wexelblat, New York et al.: Academic Press, S. 92–139.

Naur, Peter (1992): Programming as Theory Building. In: *Computing: A Human Activity*, New York: ACM Press, S. 37–49.

von Neumann, John (1987): First Draft of a Report on the EDVAC. In: *Papers of John von Neumann on Computing and Computer Theory*, Hg. Williams Aspray und Arthur W. Burks, Cambridge.

Perlis, Alan J. (1981): The American Side of the Development of Algol. In: *History of Programming Languages*, Hg. Richard L. Wexelblat, New York et al.: Academic Press, S. 75–91.

Siefkes, Dirk (1999): Hybridobjekte als Gegenstände der Informatik. In: *Die Rolle von Schemata in der Informatik als kultureller Entwicklung*, Berlin: TU Berlin, S. 5–14.

Zemanek, Heinz (1971): Was ist Informatik? *Elektronische Rechenanlagen*, 13(4):157–161.