SYSTEMPROGRAMMIERUNG AUS DER SICHT DER UNIVERSITAET

N. Wirth
Eidgenössische Technische Hochschule, Zürich, Switzerland

ABSTRAKT

Die Systemprogrammierung liefert einen wesentlichen Beitrag zur praktischen Ausbildung der Informatiker. Inwiefern soll und kann Systemprogrammierung an Universitäten in sinnvoller Weise gelehrt werden? Der Autor gelangt zur Ansicht, dass der Erziehung zu klarem, konstruktivem Denken Priorität eingeräumt werden soll vor der Vermittlung von technischem Detailwissen, und dass gerade in dieser Beziehung die Informatik eine bedeutende Rolle spielen kann.

Es entspricht dem Auftrag der Universität, dass sich Wesen und Inhalt ihres Lehrangebotes den Entwicklungen der Umwelt anpassen. So entstanden in der jüngsten Vergangenheit - veranlasst durch das Aufkommen der Computertechnik - Abteilungen und Lehrgänge in Computer-Wissenschaften. Lebhafte Diskussionen wurden geführt über Wesen, Ziel und Existenzberechtigung dieser neuen Studienrichtung [3,4,5,10]. Kann eine Maschine eine neue akademische Disziplin rechtfertigen? Diese Frage wurde mancherorts erfolgreich verdrängt durch das Ersetzen des Wortes "Computer" aus der Bezeichung des Faches, "denn eben, wo Begriffe fehlen, da stellt ein Wort zur rechten Zeit sich ein".

Mir will scheinen, dass sich das Wesen eines Fachgebietes am besten aus den Anforderungen ableiten lässt, welche seine zukünftige Tätigkeit an den Studien-Absolventen stellt. In diesem Sin-

ne kristallisierte sich mehr und mehr das Konstruieren von Programmen aller Art und von stets zunehmender Komplexität als Kern der Informatik heraus. Die Tätigkeit des praktizierenden Informatikers ist demnach vor allem konstruktiver Natur; er ist ein Computer-Ingenieur, wobei wir diesen Begriff nicht länger auf den Bereich der Hardware beschränken dürfen. Besondere Aufmerksamkeit hat in dieser Beziehung die Systemprogrammierung erregt [9].

Unter Systemprogrammierung verstehen wir die Konzeption und Entwicklung von grossen Computersystemen wie Sprachübersetzer, Systemkontrollprogrammen und anwendungsorientierter, komplexer Programme. Die Erstellung solcher Produkte besteht aus mehreren Phasen. Die üblicherweise mit Programmieren bezeichnete Tätigkeit ist darin lediglich als einzelne Komponente enthalten, während für das umfassendere Gesamtgebiet auch die Bezeichnung "Software Engineering" verwendet wird [6].

Entwicklungen in der Industrie sind üblicherweise projektorientiert. Ein Projekt durchläuft bis zu seiner Vollendung folgende Phasen (siehe auch [8]):

1. Die Durchführbarkeitsstudie (feasibility study). In dieser Phase wird vorerst eine detailierte Aufnahme des Ist-Zustandes vorgenommen. Darauf basiert die Kosten-Nutzenanalyse, welche die Grenzen der einsetzbaren Mittel und der erwarteten Leistungsverbesserung durch das zu entwerfende System festlegt. Ebenfalls in diese Phase gehört eine Studie der Realisierungsmöglichkeiten: Kann das Problem am besten durch Kauf oder Miete von bestehenden Systemen oder Systemkomponenten gelöst werden, oder durch eine Eigenentwicklung, oder indem eine Entwicklung in Auftrag gegeben wird. Das Resultat dieser Phase ist ein Pflichtenheft für den nächsten Schritt:

2. Die Projektanalyse. Sie besteht aus einer Vertiefung in die Projektdefinition, während der die Aufgabe in Teilaufgaben zerlegt wird. Schon hier finden wir also eine Vorbereitung der Modularität des Endproduktes. Aus der Projektanalyse resultiert

ein Pflichtenheft, welches im Detail die Anforderungen an das projektierte System spezifiziert. Ferner legt es die Form und den Umfang der herzustellenden Dokumentation fest. Dabei handelt es sich nicht nur um eine Dokumentation des Systemprogrammes zuhanden des Systemprogrammierers, sondern auch um notwendige Unterlagen für den Gebrauch des Produktes, für die Schulung und für die Werbung. Das Pflichtenheft postuliert auch die Art und den Umfang von Abnahmetests, die am Schluss über Erfolg oder Misserfolg des Projektes entscheiden. Ein wichtiges und kritisches Resultat der Projektanalyse ist der Zeitplan für die Durchführung der Entwicklungsarbeiten. Aus dem Zeitplan ist der Personalaufwand ersichtlich: zu welchen Zeiten sind welche Personalbestände erforderlich. Zeit- und Personalplan ergeben wiederum eine genauere Abschätzung der anfallenden Kosten, die sich zwangsläufig im Rahmen der im Grobkonzept festgehaltenen Mittel halten müssen.

3. Die Realisierung (Implementation). In die Phase der Realisierung gehört die Planung der technischen Einzelheiten, die eigentliche Programmierung und die Codierung in einer verfügbaren Programmiersprache. Meistens ergibt sich aus technischen Erkenntnissen während der Programmierung eine Revision der Detail-Spezifikation des Systems, und leider oft auch eine solche des Zeit- und gar des Personalplans. Grobe Fehlschätzungen sind in dieser Hinsicht geradezu sprichwörtlich geworden [1], und sind Zeugen für mangelnde langjährige Erfahrung und Ausbildung. Die Realisierungsphase enthält auch die Erstellung der bereits erwähnten Dokumentation, ja sogar die Schulung von Personal teils für die Implementierung selbst, vor allem aber für die spätere Verwendung und die Abnahmetests des Systems.

4. Anwendung und Wartung. Diese Phase gehört nicht mehr im strikten Sinn zur Entwicklung eines Systems. In der Praxis jedoch ist der Uebergang zu dieser Phase oft nur schwer festzuhalten. In ihr wird das System im praktischen Einsatz einer Bewährungsprobe enterzogen. Blatante Fehler im System müssen ausgemerzt werden. Erfahrungsgemäss treten sie derart häufig auf,

dass für diese Tätigkeit der Euphemismus "Wartung" (maintenance) die wahrheitsgetreue Bezeichnung "Korrektur" völlig verdrängt hat. Doch handelt es sich in dieser Phase nicht nur um Korrekturen von Programmierfehlern, sondern recht häufig auch von Konzeptionsfehlern und um Anpassungen des Systems an sich verändernde und erweiternde Erfordernisse.

Aus dieser Skizzierung eines Software-Engineering Projektes wird dessen Aehnlichkeit mit Projekten in den meisten anderen Zweigen des Ingenieurwesens offensichtlich. Was ein Software-Engineering Projekt von denjenigen anderer Fachzweige wesentlich unterscheidet, ist nur der Inhalt der Realisierungsphase. Es ist daher natürlich, wenn sich der Unterricht in Informatik auf dieses technische Kapitel relativ stark konzentriert.

Skeptiker werden hier einwenden, dass sich der Informatik-Unterricht an Universitäten bisher überhaupt nicht auf die Bedürfnisse der Systemprogrammierung eingestellt habe, und dass im Bereich der Programmierkurse mit allzu grosser Vorliebe kleine Spielzeugprogramme gebastelt und künstliche Problemchen gelöst werden, die in der technischen Realität keinen Platz fänden. Ich möchte aber doch zu bedenken geben, dass das Ueben an einfachen, kleinen Programmen zum Erlernen der Grundkonzepte und zur Einführung in den Umgang mit Computern in der kurzen zur Verfügung stehenden Zeit die einzige gangbare Möglichkeit darstellt. Der Vorwurf der Skeptiker ist aber dennoch relevant. Er ermahnt die Dozenten daran, dass die gestellten Uebungsaufgaben mit Sorgfalt gewählte Abstraktionen von realen Problemen sein müssen. Es ist wesentlich, dass die Informatik dadurch den engen Kontakt mit der Realität beibehält, und dass sie sich nicht wie die moderne Mathematik davon gängzlich loslöst und sich darauf konzentriert, ihre eigenen künstlichen Probleme zu postulieren und Abstraktionen und Formalismen um ihrer selbst Willen zu kreieren. Wir Dozenten werden aber auch aufgefordert, stets darauf hinzuweisen, dass die sorgfältig gewählten Aufgaben als Abstraktionen zu verstehen sind, und dass eine Abstraktion die verschiedensten realen Gestalten annimmt. Aber schwierig ist es, die Studenten ein Beispiel als Beispiel erkennen zu lassen!

In meinem Einführungskurs in das Programmieren erwähne ich gerne das Beispiel der Berechnung einer Tabelle von Primzahlen zur Demonstration der Gedankengänge bei der Problemlösung und des schrittweisen Aufbaus von Programmen. Aber bei allzuvielen Zuhörern ist mit dem besten Willen kein Interesse zu erwecken, da für sie das Problem der Primzahlen in Form von Tabellensammlungen bereits seit langem gelöst ist!

Ohne Zweifel ist für den angehenden Software-Ingenieur eine Folge von Aufbaukursen mit Betonung der praktischen Programmiertechnik unerlässlich. In diesen Kursen soll Fachwissen über verschiedene Kapitel der Informatik vermittelt werden, über Datenstrukturierung und -Representation, über Such- und Sortierprobleme, über syntaktische Strukturen und deren Analyse, über Compileraufbau und letztlich sogar über ausgewählte Kapitel der Betriebssysteme. Ohne bewusste Pflege der praktischen Uebungen sind diese Kurse jedoch beinahe wertlos. Etwas pointiert könnte man sagen, dass die theoretischen Themen lediglich als Gewürz dienen, um die Praktika zu stimulieren. Naur geht sogar weiter und erklärt: Der [Vermittlung von] Erfahrung gebührt mehr Gewicht als dem Wissen [7].

Ich habe die Erfahrung gemacht, dass den Studenten das im Vorlesungsteil solcher Kurse vermittelte Fachwissen meistens keine Schwierigkeiten bereitet; je mathematischer seine Form, desto leichter wird es "registriert". Hingegen spotten die eingereichten Programme der Uebungsaufgaben oft jeder Kritik! Welche Greuel an verknorzten Konstruktionen und falsch angewendeten Rezepten werden da zu Programmen zusammengeflickt und als "Lösungen" angeboten! Es ist nach solchen Einsichtnahmen jeweils offensichtlich, wo die Lücken in unserem Ausbildungsangebot am weitesten klaffen; doch bietet sich leider keine Allerweltslösung an, um sie zu stopfen. Sicher am wertvollsten wäre eine persönliche Betreuung eines jeden Studenten, welche in der ausführlichen Ueberprüfung eines jeden erstellten Programmes und dessen Besprechung und Korrektur kulminiert. Dass dies selbst mit unbeschränkten Mitteln undurchführbar wäre, ergibt sich aus dem akuten Mangel an Lehrpersonal mit eben diesem angestrebten

Reichtum an Programmiererfahrung. Daher bin ich dazu übergegangen - vorerst versuchsweise in einem Kurs über Grundlagen der Compiler - komplette Programme selber nach bestem Wissen und Gewissen zu schreiben und als Vorbilder zu verteilen. Diese Programme sind dann zu ergänzen, z.B. durch Anweisungen, die eine geschicktere Art der Fehlerbehandlung darstellen, oder durch Abschnitte, die der Sprache neu hinzugefügte Satzkonstruktionen verarbeiten. Anderseits können die Vorlagen modifiziert werden, z.B. indem ein Compiler, der Code für Computer A erzeugt, so abgeändert wird, dass Code für einen Computer B entsteht. Der Vorteil der Verwendung solcher Vorlagen liegt nicht nur der Reduktion von aufwendiger Codierung, sondern besteht vor allem darin, dass der Student sich an ein Vorbild punkto Struktur und Programmierstil halten kann, und somit vor Irrwegen bewahrt bleibt, aus denen er aus Zeitmangel nicht mehr herausfindet, selbst wenn er sich der begangenen Fehler bewusst geworden ist. Dem Dozenten bleibt die Hoffnung, dass sich der vorbildliche Programmierstil des Musters mit der Zeit durch Osmosis vererbe (siehe auch [11]).

Ich betone hier besonders den Programmierstil, die saubere, zweckmässige und klare Gliederung des Programmtextes. An ihm offenbart sich die Klarheit der Gedanken, die das Entstehen des Programmes begleiteten. Er bestimmt seine Uebersichtlichkeit und Nützlichkeit. Er ist die Quintessenz dessen, was sich nicht in Vorlesungen und Büchern dozieren lässt, wo man vergeblich versucht, numerische Massstäbe anzulegen, dessen An- und Abwesenheit der erfahrene Programmierer jedoch ohne Zögern feststellt.

Als unerlässlich taxiere ich für jeden angehenden Systemprogrammierer, sogar für jeden Informatikstudenten, die Ausführung einer selbständigen Arbeit, wo - ohne Vorlagen - möglichst alle Phasen eines Software-Engineering Projektes durchexerziert werden. Dabei soll nach Möglichkeit auch in Gruppen gearbeitet werden. Erst dadurch wird den Teilnehmern die Wichtigkeit von sauberen, klaren Spezifikationen ersichtlich. Die Idee der Modularität und des "interface" bleibt nicht nur abstrakte Eigenschaft und Problematik des Programmes, sondern zeichnet sich in

der Aufteilung der Arbeit und der zwischenmenschlichen Verständigung konkret ab.

Ich halte solche Software-Projekte nicht nur für den Studenten als fördernd, sondern auch für ein Informatik-Institut als stimulierend und notwendig, um mit den Schwierigkeiten praktischer Realisierungsarbeiten in Kontakt zu bleiben. Ferner bieten solche Eigenprodukte ideale Möglichkeiten zum Experimentieren mit neuen Ideen (die dadurch oft schon verworfen werden, bevor sie publiziert sind!). Der Eigenbau vermittelt genügend Vertrautheit mit der internen Struktur und Funktionsweise, um die Scheu vor Abänderungs- und Erweiterungsarbeiten zu nehmen. Damit werden Experimente möglich, vor denen man sonst zurückschrecken würde.

Allerdings erfordert das Gebot der professionellen Ehrlichkeit, dass diese Produkte nicht nur als Versuchsobjekte konzipiert sind. So ist es zum Beispiel fast Mode geworden, an Universitäten Compiler zu bauen; allzu oft werden dann aber wichtige Aspekte fast völlig ignoriert, nachdem der wirklich erforderliche Arbeitsaufwand real erkannt worden ist. Behandlung von Fehlern im Eingabetext, Probleme der Erzeugung von gutem Code für reelle Computer, Integration in bestehende Betriebssysteme sind Problemkreise, die nur allzu gern beiseite gelassen werden. Dadurch aber versagt man sich den Einblick in zentrale Aspekte der Compilerkonstruktion und erhält leicht den Hang, den nötigen Aufwand für die Erstellung kompletter, eben praktisch brauchbarer Systeme zu unterschätzen.Der Entschluss, ein praktisches Softwaresystem an einem Universitätsinstitut zu bauen, darf daher nicht leichtfertig gefasst werden. Wie in der Industrie, sollte ihm eine Projektanalyse vorangehen. Ich weiss aus eigener Erfahrung, dass der Bau eines brauchbaren, qualitativ hochstehenden Compilers leicht den 5-10 fachen Arbeitsaufwand eines entsprechenden sogenannten "Studienobjektes" erfordert. Er übersteigt bald einmal den Rahmen von Universitätsinstituten; Projekte, die sich über mehrere Jahre erstrecken und grössere Arbeitsgruppen erfordern, finden in der Industrie ihre berechtigte Stätte, da sie ohne den Hintergrund konkreter oekonomischer Realitäten

doch allzu leicht zu unfruchtbaren Monstren ausarten.

Wir haben bislang stillschweigend angenommen, dass die Systemprogrammierung ein echtes Anliegen der Universitätsausbildung sei. Aber ist dies ein Axiom? Wenn wir auf die politischen Postulate der jüngsten Vergangenheit hören, dann sicherlich nicht. Sie werfen den Universitäten vor, sich zu Dienern der Industrie erniedrigt zu haben und die Ausbildung rein zweckorientiert den Wünschen der Industrie anzupassen. Bekanntlich besteht in der Industrie nicht nur ein Bedarf, sondern sogar ein Mangel an gut ausgebildeten Systemprogrammierern. Haben wir es also hier mit einem Musterbeispiel des kritisierten Phänomens zu tun?

Nun dürfen wir einerseits festhalten, dass eine technische Hochschule oder technische Universität ohne Ausrichtung auf die reellen Gegebenheiten in der Industrie ohne Zweifel eine zweckentfremdete Institution wäre. Anderseits darf die Industrie auch nicht erwarten, dass die Absolventen bereits eine Fachausbildung erhalten haben, die ihren Einsatz ohne weitere Detail-Ausbildung gestattet. In der Tat liegt den Arbeitgebern heute vielmehr daran, dass an der Universität eine Schulung zum analytischen, zum konstruktiven, zum kritischen Denken vermittelt wird, als dass bestimmtes Fachwissen verteilt wird. Bereits werden in der Computerbranche Ingenieure mit der Fähigkeit, Aufgaben richtig einzuschätzen und anzupacken, den Informatikern vorgezogen, selbst wenn diese reich an theoretischen Kenntnissen sind und den festen Glauben haben, ihre oberste Aufgabe sei der Entwurf einer neuen Programmiersprache und die Konzeption eines Compilers.

Diese Gesichtspunkte geben uns vielleicht doch Anlass, bei einer Standortbestimmung unser Blickfeld etwas weiter zu spannen. Sie deuten dahin, dass es wichtiger ist, breiten Schichten solide Grundlagen zu vermitteln, als viele Software-Spezialisten auszubilden. Man wird dabei unweigerlich die Begriffe Bildung und Ausbildung einander gegenüberstellen. Und gerade wenn man es als Hauptaufgabe der Schule auffasst, Bildung, die Fähigkeit des selbständigen und kritischen Denkens zu vermitteln, dann erkennt

man auch die wahre Aufgabe und Chance der Informatik und speziell des Programmierens. Sie liegt nicht so sehr in der Technik der Computerverwendung, sondern in der Denkschulung. Hierbei finden wir den Computer weniger in der Rolle des Studienobjektes, sondern des Hilfsmittels, oder zumindest in einer Kombination dieser Rollen. Wie in keiner andern konstruktiven Sparte sind wir hier gezwungen, Vorgänge zu beschreiben, die trotz - oder wegen - ihrer riesigen Komplexität bis ins letzte Detail überdacht sein müssen. Wir sind angehalten, uns exakt auszudrücken und müssen einsehen, dass jede Unklarheit, jedes Verdrängen der Bemühung um klare Lösungen, jede Disziplinlosigkeit zum Scheitern verurteilt. Wir müssen uns zwingen, von "gescheiten" Lösungen abzusehen, wenn wir sie nicht voll überblicken [2]. Der Computer erscheint in der Rolle der intellektuellen Herausforderung.

Eine schwierige aber gleichzeitig zentrale Aufgabe ist es, den Unterschied zwischen guten und schlechten, zwischen rasch zusammengestellten und reiflich durchdachten Lösungen aufzuzeigen und den Programmierer zu strenger Selbstkritik zu erziehen. Aber auch hierin hat unsere Sparte eine einzigartige Chance. Hierin erkennen wir die zentrale Rolle der neuen Ideen des "strukturierten Programmierens" und der"analytischen Programmverifikation". In diesem Sinn sind sie fruchtbar, selbst wenn sie vorab an kleinen Schulbeispielen zur Perfektion ausgearbeitet worden sind und an komplexen Systemprogrammen zum Teil noch scheitern. Wenn der Programmierer dazu erzogen wird, seine Produkte nicht nur als tote, zweckgebundene Computeranweisungen aufzufassen, sondern als kleine "Kunstwerke", die durch ihre Eleganz das Gefühl der Befriedigung vermitteln, dann hat er etwas mitbekommen, das nicht nur seiner Karriere dient. Es ist genau dieses Gewürz der professionellen Integrität und der Liebe zum Handwerk, das wir in der kommerziellen Software so oft vermissen.

Wir anerkennen also, dass im Vordergrund der Ausbidlung zum Systemprogrammierer Grundkonzepte und Konstruktionsdisziplin stehen sollen und nicht so sehr technisches Detailwissen. Wir

bringen damit sogar dem Slogan "Schulung zum Denken vor Ausbildung mit Fachwissen" gegenüber einiges Verständnis auf. Umso befremdender muss uns die fast kritiklose, weitverbreitete Uebernahme von Programmiersprachen, Compilern, Terminologien und Denkschemen für Lehrzwecke anmuten. Es ist nachgerade bekannt, dass viele dieser weltweit verbreiteten und oft firmenspezifischen Sprachen und Terminologien überholt, unsystematisch und oft unzweckmässig sind. Dennoch fehlt entweder Mut oder Können in akademischen Kreisen, vom üblichen abzuweichen und neue Wege aufzuzeigen. Selbst dort, wo alte Werkzeuge neuen Erkenntnissen diametral zuwiderlaufen, wird krampfhaft versucht, den Konflikt zu verdrängen, anstatt ihn zu lösen. Man denke zum Beispiel an die durchaus ernsthaft vorgetragenen Anregungen über "Strukturiertes Programmieren mit Fortran oder Basic". An diesen Zuständen offenbart sich das Diktat der Industrie am bedenklichsten. Es ist ein Diktat, dem die Industrie selbst auch unterliegt und das ihr letztlich nicht zum Nutzen gereichen wird. Es ist ein Diktat, das aus der rasanten Entwicklung der Computeranwendungen entstand, aus der rein oekonomischen Unerlässlichkeit gewisser Standards, Normen und Nomenklaturen, und der Unmöglichkeit, in der zur Verfügung stehenden kurzen Zeit Ordnung und Uebersicht in die Vielfalt der anfallenden Probleme zu bringen.

Systemprogrammierung aus der Sicht der Universität: Erziehung zu klarem, systematischem, konstruktivem Denken, Orientierung an konkreten Problemen der Praxis, Konzentration auf das Wesentliche, ohne zwangsläufige Angleichung an Methoden und Werkzeuge, die diesen Prinzipien zuwiderlaufen.

Literaturverzeichnis

1 F.P. Brooks, jr., "The Mythical Man Month", Prentice-Hall, 1974.

2 E.W. Dijkstra, "The humble programmer", Comm. ACM, 15, 859-866 (Oct. 1972).

3 G.E. Forsythe, "A University's educational program in computer science", Tech. Rep. CS 39, Stanford University, May 1966.

4 —— "What to do until the computer scientist comes", Amer. Math. Monthly, 75, 454-462 (May 1968).

5 R.W. Hamming, "One man's view of Computer Science", J. ACM, 16, 3-12 (Jan. 1969).

6 P. Naur, B. Randell, Ed., "Software Engineering", Nato Science Committee Report, Jan. 1969.

7 P. Naur, "Datalogi Zero - a freshman course of computer science", Data 5/74, 49-51.

8 E. Schieferdrucker, "Planung von Systemprogrammen", in Systemprogrammierung, Oldenbourg, München 1972.

9 G. Seegmüller, "Systems programming as an emerging discipline", Information Processing 74, 2, 419-426 (North-Holland).

10 N. Wirth, "Die Computer-Wissenschaften und die heutige Verwendung der Computer", Universitas, 24, 371-384 (April 69)

11 —— "Program development by stepwise refinement", Comm. ACM, 14, 221-227 (April 1971).