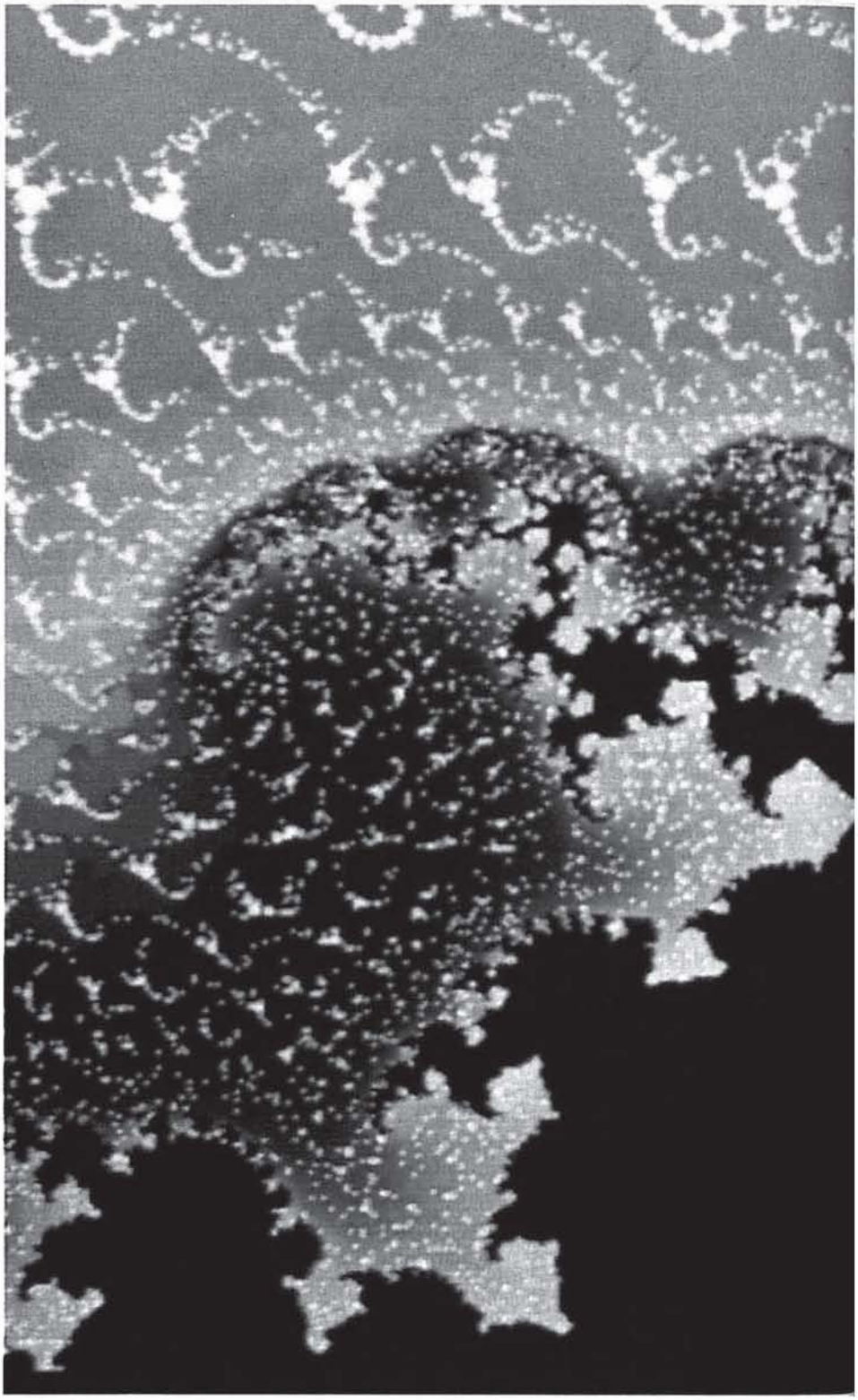
Voraussetzungen für Fortschritte

Von Jochen Ludewig

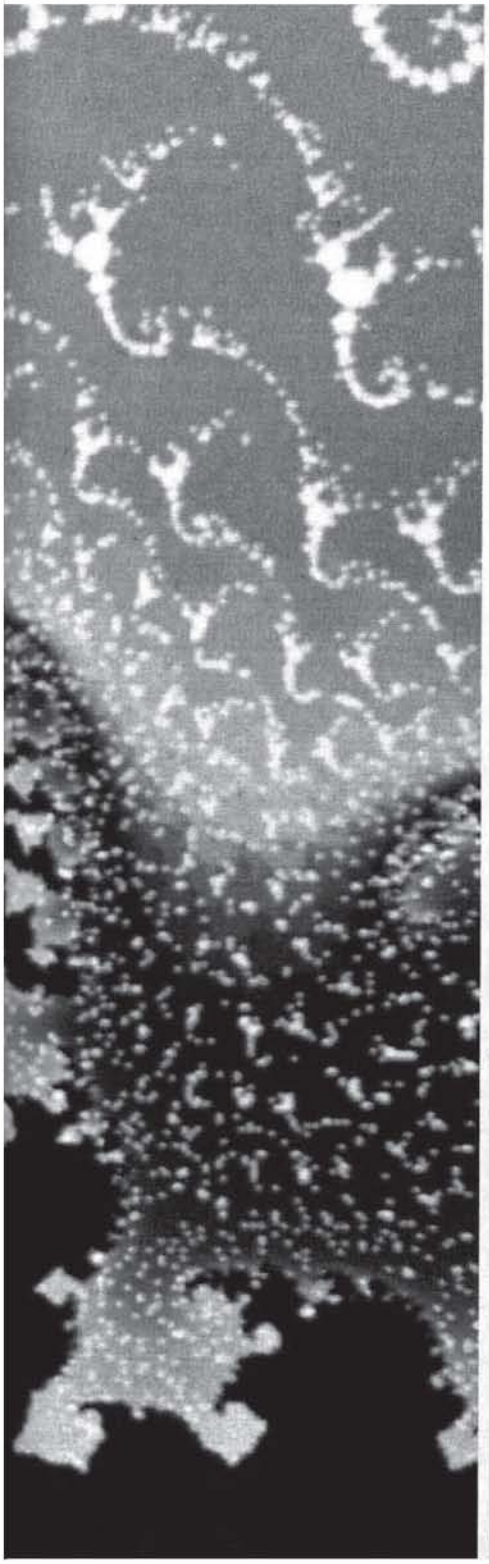
Nach einer Klärung der Begriffe wird diskutiert, wieweit wir heute die verschiedenen Tätigkeiten im Software-Engineering beherrschen. Die neuen Paradigmen (objektorientierte und logische Programmierung) werden bewertet. Der Beitrag schließt mit einigen Thesen zu den Voraussetzungen weiterer Verbesserungen.

Even though no technological breakthrough promises to give the sort of magical results with which we are so familiar in the hardware area, there is both an abundance of good work going on now, and the promise of steady, if unspectacular progress.

F. P. Brooks (1987)



Prof. Dr. Jochen Ludewig, Institut für Informatik der Universität Stuttgart, Azenbergstraße 12, D-7000 Stuttgart 1.



Das Bild zeigt die Dynamik der von rationalen Funktionen in C erzeugten interaktiven Prozesse. (Foto: Dr. Paolo Venzi, Bellinzona/© Art-Matrix, P.O. Box 880, Ithaca, NY 14851).

Solange im Software-Engineering die babylonische Sprachverwirrung anhält, muß jede Ausführung mit einer Begriffsklärung beginnen.

Worum geht es?

Software umfaßt nach IEEE-Standard 729–1983 *Programme, Abläufe, Regeln und alle zugehörige Dokumentation, die mit dem Betrieb eines Rechnersystems zu tun hat,* also beispielsweise Planungsunterlagen, Testdaten, Quellcode, Review-Verfahren und Benutzungsanleitungen.
Software-Engineering ist ein Kunstwort, das 1968 in die Diskussion über die Software-Crisis geworfen wurde; es bezeichnet das *systematische Vorgehen bei der Entwicklung, Anwendung, Wartung und Außerbetriebnahme von Software.*
Methoden, Sprachen (Notationen) und Werkzeuge sind die Komponenten der Systeme, mit deren Hilfe die Entwickler ihre Aufgaben zu fassen und zu lösen suchen.
Der **Software-Life-Cycle** ist die übliche Gliederung und Differenzierung der Tätigkeiten im Software-Engineering. Bild 1 zeigt ihn in der Form des Kostenmodells, also des strengen Phasenplans (FRÜHAUF et al., 1991).
Diese Gliederung hat sich entgegen einiger Kritik sehr gut bewährt; viele Einwände werden hinfällig, wenn man beachtet, daß der Software-Life-Cycle nicht *ein* bestimmtes Vorgehen impliziert, sondern zunächst nur die (stets notwendige) logische Sequenz der Überlegungen anzeigt. Es ist Sache des speziellen Prozeßmodells, ob und wieweit diese Schritte sequentiell, parallel oder verschränkt stattfinden.
Schließlich ist festzustellen, welche **Ziele** wir mit dem Software-Engineering verfolgen. Es geht um folgende Kenngrößen (mit teilweise gleitenden Übergängen):

Produktivität

Qualität mit den Aspekten

- Gebrauchsqualität (Qualität aus der Sicht derer, die *mit* der Software arbeiten)
- Wartungsqualität (Qualität aus der Sicht derer, die *an* der Software arbeiten)
- Prozeßqualität (Qualität des Projekts, zum Beispiel Einhaltung der Termin- und Kostenvorgaben, Transparenz)

Investitionsschutz (Weiter- und Wiederverwendbarkeit, damit auch Kompatibilität und Standardisierung)

Firmenkultur und Attraktivität der Arbeitsplätze

Alle Verbesserungen im Sinne dieser Ziele werden nachfolgend als **Fortschritte** bezeichnet.

Was bewirkt die Leistungssteigerung der Hardware?

Zu Beginn der fünfziger Jahre war die Elektronik der neuen Computer das eigentliche Wunder – und das einzige Problem; die Idee, *komplexe* Aufgaben mit diesen Maschinen zu lösen, war noch kaum gedacht. Doch mit dem Essen kam der Appetit, und die Grenzen der Hardware wichen zurück, so daß die Schwierigkeiten, Software zu konstruieren, unvermeidlich in den Vordergrund rückten. Bild 2 zeigt, wie sich die sehr unterschiedlichen Entwicklungsgeschwindigkeiten auf das Kostenverhältnis von Hard- und Software ausgewirkt haben.
Bild 3 macht deutlich, welche Aspekte der Informatik in welchem Maße durch den technischen Fortschritt der Hardware beeinflußt werden. Wie man sieht, ist die Wirkung um so schwächer, je gewichtiger unsere menschliche Rolle ist. Darum kann die Methodik als das Sorgenkind des Software-Engineering eingestuft werden, während die großflächigen Bildschirme mit vielfarbigen Fenstern und Menüs aller Art uns atemberaubende, vor wenigen Jahren noch unvorstellbare Schnittstellenqualität bieten.
Offenbar hätte auch ein stärkerer Fortschrittsballon auf die Methodik kaum Wirkung. Um sie zu verbessern, muß sich die Intelligenz aus eigener Kraft auf ein höheres Niveau (vielleicht das Plateau rechts in Bild 3) begeben.

Was können wir heute und was nicht?

Welche der anfallenden Aufgaben können wir sehr gut, leidlich gut, eher schlecht oder gar nicht lösen? Die Abgrenzung ist unvermeidlich unsicher und subjektiv; sie hilft uns aber zu erkennen, in welche Richtung sich das Software-Engineering entwickeln kann.

Was wir sehr gut können

Von allen Softwarearbeiten können wir zweifellos am besten *codieren* – Grund genug für viele, Codieren mit Softwareentwicklung zu verwechseln.

Was wir – vielleicht – hinbekommen

Wartung (Debugging, Modifikation, Portierung)

In der Praxis gelingt es uns in aller Regel, einen reproduzierbar auftretenden

Fehler zu enttarnen und zu beheben; ebenso schaffen wir die Anpassung der alten Programme an neue Aufgaben oder ihre Übertragung auf andere Rechner.
Wir sind dabei nicht so effizient, wie wir wünschen, das heißt, unser Aufwand erscheint uns als unangemessen hoch. Außerdem hinterlassen unsere Änderungen Spuren in der Software, die wie fortschreitende Korrosion langsam zur Zerstörung führen: Daten- und Ablaufstrukturen werden korrumpiert, weitere Fehler in das Programm getragen, die Konsistenz der Dokumente geht verloren. Die Effektivität ist also vordergründig.

Anforderungsdefinition und Entwurf

Für diese Tätigkeiten sind in den vergangenen 15 Jahren zahlreiche Methoden, Notationen und Werkzeuge entstanden, zum Beispiel Diagramme zur Darstellung der Datenflüsse, Entity-Relationship-Darstellungen und Data Dictionaries zur Beschreibung der Datenstrukturen, endliche Automaten und Petri-Netze zur Klärung zeitlicher Abhängigkeiten, das Entwurfsverfahren von JACKSON und die Kriterien von YOURDON und CONSTANTINE zur Festlegung geeigneter Strukturen.
Aber keines der Verfahren erlaubt eine schematische Anwendung. Lehrbücher machen oft Vorschläge, die nie ausprobiert wurden, und Werkzeuge sind oft von überraschend niedriger Qualität. Wir *beherrschen* diese Tätigkeiten also nicht, aber wir *kommen durch*, und wir wissen vor allem eine Menge darüber, was *falsch* ist.

Test und Integration

Hat man sich die Hoffnung, mit dem Test die Korrektheit beweisen zu können, einmal aus dem Kopf geschlagen, so findet man eine Reihe praktikabler Verfahren, die ein systematisches Vorgehen erlauben. Das gleiche gilt für die Integration, vor allem wenn man in Hochsprachen codiert, die (durch ihre Übersetzer und Binder) viele triviale Fehler an den Schnittstellen ausschließen.
Für beides, Test und Integration, gibt es einfache Werkzeuge, die die administrativen Arbeiten übernehmen; der intellektuelle Teil, zum Beispiel die Auswahl der Testdaten, bleibt natürlich Sache der Entwickler.

Softwaremanagement und Kostenschätzung

Auch über die Durchführung der Projekte hat sich im Laufe der letzten 20 Jahre einiges an Wissen angesammelt,

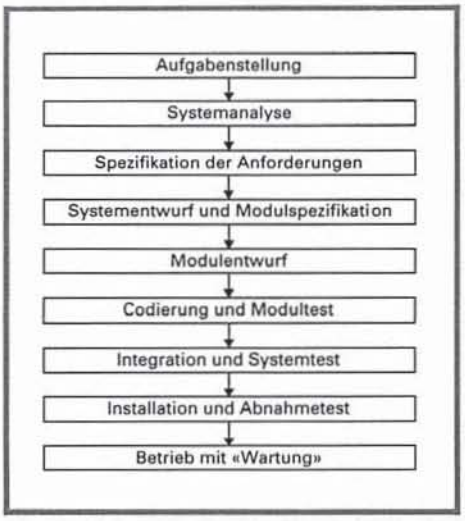


Bild 1. Das Life-Cycle-Modell.

dessen Beachtung zwar den Erfolg nicht garantiert, aber doch wesentlich wahrscheinlicher macht (FRÜHAUF et al., 1991); freilich ist dies teilweise das alte Know-how des Projektmanagements, das einfach für die Software erst entdeckt werden mußte.
Mit Verfahren wie COCOMO (BOEHM, 1981) sind wir heute auch in der Lage, die Kosten eines Projekts befriedigend genau abzuschätzen, soweit wir uns mit dem Projekt nicht in Neuland begeben.

Was wir nicht können

Qualitätsplanung

Eine Festlegung der angestrebten Qualitäten und eine konsequente Realisierung dieser Anforderungen sind heute kaum möglich. Wir haben dazu weder die Kriterien und Notationen (Metriken) noch die Verfahren, die eine bestimmte Qualität herbeiführen.

Systematische Softwareentwicklung

Alle in der heutigen Literatur propagierten Verfahren haben Patchwork-Charakter. Sie tragen die Erkenntnisse vieler Leute zusammen wie einen dicken Band von Reiseberichten, aber das ergibt noch keine Landkarte. Eine vom Start bis zum Ziel systematische – und das heißt vor allem auch sicher planbare – Entwicklung liegt außerhalb unserer Fähigkeiten.

Configuration Management

Die Verwaltung umfangreicher Softwaresysteme aus vielen Komponenten, die jeweils noch in verschiedenen Versionen und Varianten vorliegen, gelingt uns heute höchstens ansatzweise. Werkzeuge sind nützlich, weisen aber erhebliche Schwächen in der einen oder anderen Richtung auf (zum Beispiel Versionenmanagement). Damit ist die Rekonstruktion einer bestimmten alten Konfiguration ebenso schwierig wie die Untersuchung, mit welchen vorhandenen Komponenten ein neuer Baustein verträglich ist.

Wiederverwendbare Software

Es steht außer Frage, daß Wiederverwendung (TRASZ, 1988) die Goldader des Software-Engineering ist. Bis heute gelingt uns das Schürfen nicht recht, weil wir aus den verschiedensten Gründen nicht imstande sind, Komponenten zu entwickeln, deren Wiederverwendung attraktiv und rentabel ist. Im günstigsten Falle werden vorhandene Module als Steinbrüche für Neuentwicklungen verwendet.
Mit der objektorientierten Programmierung wird die Hoffnung verbunden, wiederverwendbare Bausteine zu erhalten.

Was wir gar nicht können

Exakte und anschauliche Definition der Semantik und Pragmatik

Bis heute haben wir nur bei sehr kleinen, den «Mickey-Mouse-Beispielen» die Möglichkeit, die Semantik einer Programmkomponente exakt und anschaulich zu definieren. Die exakten Verfahren, zum Beispiel die algebraische Spezifikation, sind schwer verständlich, die anschaulichen (zum Beispiel verschiedene grafische Formen) sind nicht exakt. Dieses aus der Sicht der Praxis *völlig* ungelöste Problem trägt wesentlich zur Schwierigkeit bei, mit der Wiederverwendung voranzukommen.

Zuverlässige Wartung vernetzter Dokumente

Ist die Wartung einer einzelnen Komponente schon eine anspruchsvolle Aufgabe, so ist die Wartung *vernetzter* Dokumente heute ganz offensichtlich in der Praxis nicht zu schaffen. Hier liegt letztlich der Grund, daß alte Software identisch ist mit altem Code: Alle anderen Dokumente sind im Laufe der Zeit unvermeidlich obsolet geworden.

Automatische Beurteilung der Qualität

Der Traum von der automatischen Qualitätsbeurteilung, der in den späten siebziger Jahren zu wilden Phantasien geführt hat (HALSTEAD, 1977), ist bis heute nicht wahr geworden. Wir sind gerade eben in der Lage, einfachste Metriken, vor allem die Zeilenzahl und den Aufwand, sauber zu erfassen (GRADY, CASWELL, 1987). Mehr ist auch nicht in Sicht.

Die Suche nach dem Optimum

Immer wieder muß im Software-Engineering daran erinnert werden, daß es – abgesehen von ethischen Fragen, die hier aber kein Gegengewicht darstellen – letztlich um ein Kostenminimum geht. Darum ist in jedem Falle zu prüfen, wo dieses liegt; wo man zuviel oder zuwenig Aufwand treibt, wird das Optimum verfehlt.

Wo wir aber von unseren Gefühlen, unseren emotionalen Bedürfnissen gesteuert werden, optimieren wir anders: Wir machen bevorzugt das, was wir gut können und was uns daher auch Spaß macht; was wir nicht können, meiden wir.

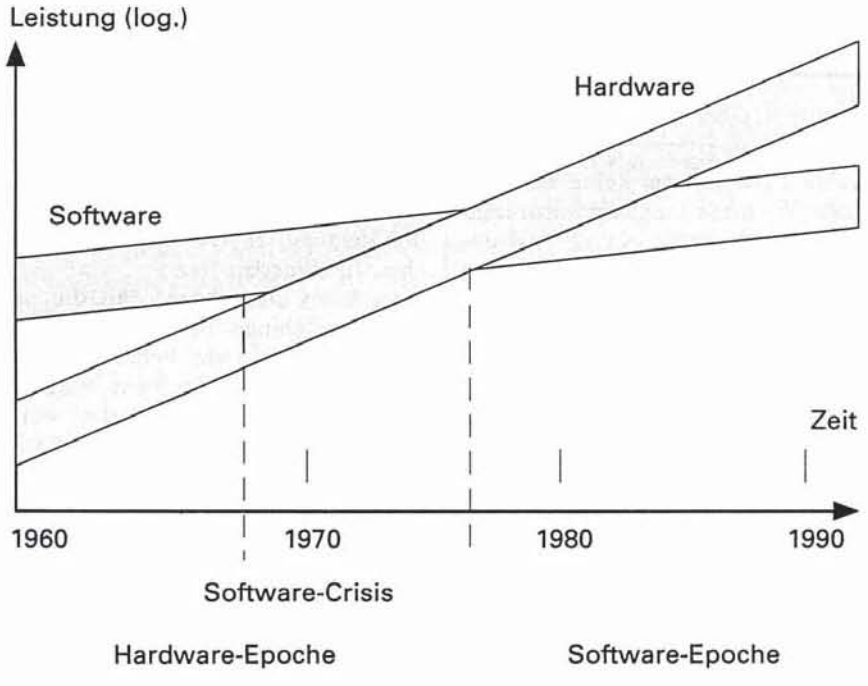
Einem rational ausgerichteten Software-Engineering stellen sich daher vor allem zwei Aufgaben:

- Es ist festzustellen, wo die Optima liegen und welcher Aufwand in welcher Richtung zu ändern ist.
- Die erforderlichen Tätigkeiten müssen gelehrt und geübt werden, damit sie nicht mehr abschrecken. Durch positive Anreize muß ihre Attraktivität speziell erhöht werden, denn daß ein Konzept *technisch* sinnvoll ist, bewegt erfahrungsgemäß nur sehr wenige Menschen dazu, sich auch daran zu halten.

Softwarekosten sind überwiegend Wartungskosten. Innerhalb der Entwicklung (also ohne Wartung) ist der Kostenanteil des Tests und der Integration meist offensichtlich zu hoch. Darum kann man schon nach der Klärung des Ist-Zustands oben feststellen, daß sich eine Erhöhung der Anstrengungen vor allem in den *frühen Phasen* direkt lohnt. Wer also seine Mitarbeiter systematisch *schult* und *motiviert*, so daß sie die modernen Methoden und Programmiersprachen anwenden können und auch anwenden, und ihnen moderne Werkzeuge zur Verfügung stellt, die diese Verhaltensänderung unterstützen, der bewegt sich mit großer Wahrscheinlichkeit auf das Optimum zu. Das gleiche gilt für Anstrengungen in Richtung *Modellierung*, *Planung* und *Qualitätssicherung*. Wer dagegen großen Aufwand treibt, um bessere Editoren und schnellere Compiler zu entwickeln, der ist längst am Optimum vorbei.

Zu einer differenzierteren Bewertung einzelner Ansätze sind aber Daten notwendig, Daten, die uns heute kaum zur Verfügung stehen. Das Thema der *Metriken* erweist sich damit als Angelpunkt des Software-Engineering. Solide Zahlen, nach einheitlichen Regeln erhoben und über längere Zeit gesammelt, werden uns in die Lage versetzen, Methoden, Sprachen und Werkzeuge rational zu vergleichen.

Bild 2. Entwicklung in der Hard- und der Software.



Reengineering mit erheblichem Automatisierungsgrad

Die Trümmerwüsten alter Software schreien eigentlich nach Werkzeugen, die die Restauration und die Ersetzung unterstützen. Bislang ist aber wenig Konkretes verfügbar, so daß die «Entsorgung» (LUDEWIG, 1990) nach wie vor überwiegend in Handarbeit geleistet werden muß.

Zusammenfassung der Bestandsaufnahme

Wir haben in den vier Kategorien («beherrscht», «möglich», «nicht zu schaffen» und «heute ganz unmöglich») eine Reihe von Tätigkeiten oder Tätigkeitsgruppen eingeordnet. Dabei überdecken die später genannten offensichtlich mehrere der früher genannten.

Wir kommen also mit den Einzelaspekten leidlich gut zurecht, am allerbesten natürlich mit der Codierung. Wir schaffen es aber bislang nicht, den *gesamten* Prozeß, von der Konzeption bis zur Wartung, als *vernetzten* Vorgang, der sich in *vernetzten* Dokumenten spiegelt, in den Griff zu bekommen.

Die Bedeutung der neuen Paradigmen: objektorientierte und logische Programmierung

Zwei neue Ansätze zur Konzeption und Darstellung von Programmen konkurrieren seit Beginn der achtziger Jahre mit der traditionellen *imperativen* Programmierung: die *objektorientierte* und die *logische* Programmierung, zunächst repräsentiert durch die Sprachen Smalltalk-80 und PROLOG.

Objektorientierte Programmierung

Die objektorientierte Programmierung (PETERSON, 1987; WEGNER, 1987) wurde zwar konzeptionell schon vor langer Zeit vorbereitet (1967 mit SIMULA, 1972 durch PARNAS mit dem Begriff des «information hiding»), erreichte größere Popularität aber erst mit der Programmiersprache *Smalltalk-80*.

Sie hatte ihre ersten praktischen Auswirkungen an der Benutzerschnittstelle, und wer an eine objektorientierte Bedienoberfläche gewöhnt ist, empfindet alles andere als veraltet. Durch die *Hypertext-Systeme* wird daraus eine umfassende Benutzungs- und Navigationsphilosophie. Aber die objektorientierte Betrachtungsweise geht weit darüber hinaus, sie schafft durch die Verlagerung der Prozeduren *in* die Daten oder Da-

tentypen, im Jargon: der *Methoden* in die *Objekte* oder *Klassen*, einen neuen Programmierstil, der sowohl für eine evolutionäre Entwicklung als auch für die Wiederverwendung ausgezeichnete Voraussetzungen schafft:

- Jedes *Objekt*, also in traditioneller Terminologie jedes Datum, ist – vor allem durch seinen Typ, die *Klasse* – mit einem «Bewußtsein» ausgestattet, dank dessen es selbst «weiß», welcher Art es ist und wie es auf Botschaften anderer Objekte reagieren sollte.
- Klassen, die Typen der Objekte, sind durch die «Vererbung» hierarchisch verbunden, so daß eine Klasse in angepaßter Form wiederverwendet werden kann, ohne selbst verändert zu werden. Die Anpassung kann dabei sowohl additiv als auch reduzierend oder modifizierend wirken.

Praktisch kann diese Frucht bis heute aber nur von den Spezialisten geerntet werden, denn die Klassenhierarchie erzeugt in der Software eine Struktur, die durch die Relationen «Unterklasse von» und «Schickt Botschaft an» geprägt ist. Diese Struktur ist – besonders bei Mehrfachvererbung – offenbar schwerer zu überblicken als die traditionelle mit den Relationen «Teil von» und «ruft auf». Darum ist es bis heute nicht gelungen, wiederverwendbare Einheiten zu schaffen, die leicht integrierbar und nützlich, trotzdem aber unabhängig und klar abgegrenzt sind von ihren Anwendungen.

Logische Programmierung

Ein anderes neues «Paradigma» ist die logische Programmierung, die vor allem durch PROLOG Verbreitung fand. Die logische Programmierung schafft der Informatik neue Möglichkeiten, indem sie Wissen nichtnumerischer Art, zum Beispiel über bestimmte logische Verknüpfungen, handhabbar macht und so beispielsweise den Aufbau sogenannter *Expertensysteme* gestattet. Die Unterscheidung zwischen *Fakten*, *Regeln* und *Inferenzmechanismus* erlaubt den Aufbau von *Wissensbasen*, in denen Kenntnisse gesammelt und nutzbar gemacht werden können.

Einsatzgebiete der objektorientierten und der logischen Programmierung

Beiden Ansätzen gemeinsam ist das Merkmal, daß sie bisher vorwiegend dort eingesetzt wurden, wo die Vorteile offensichtlich und die Nachteile (hinsichtlich Effizienz und Qualität) tolerierbar sind. In Zukunft ist zu erwarten, daß die objektorientierte Programmierung, die gegenüber der imperativen eine echte Erweiterung, keine Alternative darstellt, diese langsam verdrängt. Die logische Programmierung wird dagegen neue Gebiete erschließen, die mit konventionellen Mitteln kaum in Programme zu fassen waren.

Was ist zu tun? Voraussetzungen für Fortschritte

Die Zunahme von Schlag- und Reizwörtern erleichtert den Praktikern die Arbeit nicht, im Gegenteil. Die um Fortschritte bemühten Praktiker sind vor allem verwirrt, und den Anbietern von Case-Tools (CHIKOFSKY, 1988) bläst der Wind, den sie mit ihrer Werbung entfacht haben, ins Gesicht. Ganz ähnlich geht es den Herolden der Künstlichen Intelligenz.

Man kann also sagen, daß die neuen Waschmaschinen bei weitem nicht so weiß waschen, wie behauptet wurde. Aber ein anderes Problem wiegt noch schwerer: Dort, wo sie geliefert wurden, fehlen in der Regel Wasser- und Stromanschluß, soll sagen: Weder die alten noch die neuen Ansätze können wirksam werden, wo das Unternehmen nicht aufnahmebereit ist. Denn ebensowenig wie man ein Entwicklungsland auf den Stand Mitteleuropas bringen kann, indem man einige Bücher und Operationssäle liefert, läßt sich ein akzeptabler Stand des Software-Engineering

Fünf Jahre ABB-Informatikschule

Die ABB-Informatikschule ist fünf Jahre alt. Am 1. Oktober 1985, mit dem Stellenantritt ihres damaligen und heutigen Leiters, nahm das Projekt einer BBC-Informatikschule konkrete Gestalt an. Ein Team von BBC-Fachleuten hatte bis dahin in dreivierteljähriger Projektarbeit das Konzept der Schule ausgearbeitet. Sozusagen als Pflichtenheft lag es auf dem Pult des frischgebackenen Schulleiters, verbunden mit der Auflage, noch im gleichen Jahr die ersten Kurse durchzuführen.

Am 2. Dezember war es dann soweit. Mit neun Teilnehmern wurde der erste, 20tägige Kurs «Systematisches Programmieren 1» gestartet. Schon kurze Zeit später öffnete sich die Schule auch für Mitarbeiter externer Firmen. Heute kommen 50 % unserer Kursteilnehmer von Firmen, die nicht zur ABB-Gruppe gehören.

Seit der Gründung wurden rund 20 000 Teilnehmertage erteilt. 40 Personen, darunter zwei Damen, besuchten den Lehrgang «Sofware Engineer ABB» und erhielten das begehrte Zertifikat. Die 5. Auflage dieses erfolgreichen Programms mit 14 Teilnehmern wurde vor ein paar Wochen gestartet.

Die Schule ist in den fünf Jahren seit der Gründung ihrer ursprünglichen Zielsetzung, eine fachlich und didaktisch anspruchsvolle Ausbildung für professionelle Softwareentwickler anzubieten, treu geblieben. Es ist auch heute noch unser primäres Anliegen, Fachleuten in Industrie und Wirtschaft, die hauptberuflich in Softwareprojekten tätig sind, das für ihre Arbeit benötigte Wissen und die erforderlichen Fähigkeiten zu vermitteln.

Die Hilfsmittel und die Werkzeuge, welche die Schule heute einsetzt, haben sich jedoch gewandelt. Mit einem Dutzend PC/XTs hatten wir vor fünf Jahren den Unterricht im Fach «Systematisches Programmieren» begonnen, was damals in jeder Beziehung «state of the art» war. Heute stehen unseren Kursteilnehmern über 30 PS/2, die in einem Tokenring-LAN untereinander und mit dem ABB-Großcomputer, einer IBM 3090 40J, verbunden sind, eine Mikro-VAX mit 16 Arbeitsplätzen, drei DEC-Workstations und seit kurzem ein IBM-RISC-System 6000 mit acht Arbeitsplätzen zur Verfügung.

Auch die Softwareprodukte für den Unterricht haben sich dem Fortschritt der Technik angepaßt. Besaßen die Studenten 1985 mit Turbo Pascal für die damalige Zeit schon eine hochmoderne Programmierumgebung, so stehen ihnen heute mit Sybase, dem technologisch führenden Datenbanksystem, mit teamwork, dem erfogreichsten CASE-Produkt auf dem Markt, und mit Glockenspiel C++, als objektorientierter Programmiersprache, die modernsten Werkzeuge zur Verfügung.

Anläßlich des Kolloquiums zum fünfjährigen Bestehen der ABB-Informatikschule sprachen Prof. Dr. J. Ludwig von der Universität Stuttgart zum Thema «Software-Engineering in der Praxis der neunziger Jahre» (siehe vorliegenden Beitrag), Dr. M. Glinz von der ABB-Informatikschule über «Objektorientierte Spezifikation», Prof. Dr. D. Tsichritzis von der Universität Genf über «Object-Oriented Software Development» und Prof. Dr. C. A. Zehnder von der ETH Zürich über «Informationssysteme und ihr Einsatz in den neunziger Jahren».

durch singuläre Maßnahmen oder den Einkauf von Werkzeugen erreichen; vielmehr ist eine höhere *Softwarekultur* notwendig (LUDEWIG, 1987).

Jede Firma, jede Behörde, die Fortschritte im Software-Engineering erreichen will, muß zunächst ihre eigene Situation verstehen und formulieren, kurz: zu *modellieren* (LUDEWIG, 1989). Darum ist die Aufstellung von *Datenmodellen* (zur Beschreibung der Information, mit der konkret gearbeitet wird) und *Prozeßmodellen* (zur Beschreibung der Abläufe bei dieser Arbeit, vgl. AGRESTI, 1986; THAYER, 1988) eine Vorbedingung aller anderen Maßnahmen. Aus den Modellen folgt dann, welche Schritte erforderlich sind.

Es hat keinen Sinn, sich einen schnellen Wagen zu kaufen, wenn man nicht weiß, wohin man will.

Bei allen Veränderungen muß erkennbar sein, ob der angestrebte Erfolg tatsächlich erzielt wurde. Dazu müssen Maßstäbe und Kriterien definiert sein, kurz: *Metriken* (CONTE, DUNSMORE, SHEN, 1986). Bis heute werden Metriken vielfach als bürokratische Beckmesserei mißverstanden. Dabei geht es um nichts anderes als eine rationale Basis der Erfolgskontrolle. Wer dahinter Haarspalterei vermutet, sei auf die ganz und gar praxisbezogenen Hinweise von BOEHM (1987) verwiesen.

Wer den schnellsten Weg bestimmen will, muß eine Uhr besitzen.

Das genaue Gegenteil einer Wunderwaffe sind Normen (VOGES, 1987): Niemand erwartet von ihnen sensationelle Fortschritte, aber ohne sie ist kein *Engineering* vorstellbar. Normen sind ein sehr zuverlässiges Anzeichen für die Konsolidierung einer technischen Disziplin.

Wer nicht ganz allein auf den Straßen ist, sollte sich mit den andern über Rechts- oder Linksverkehr und andere wichtige Fragen geeinigt haben.

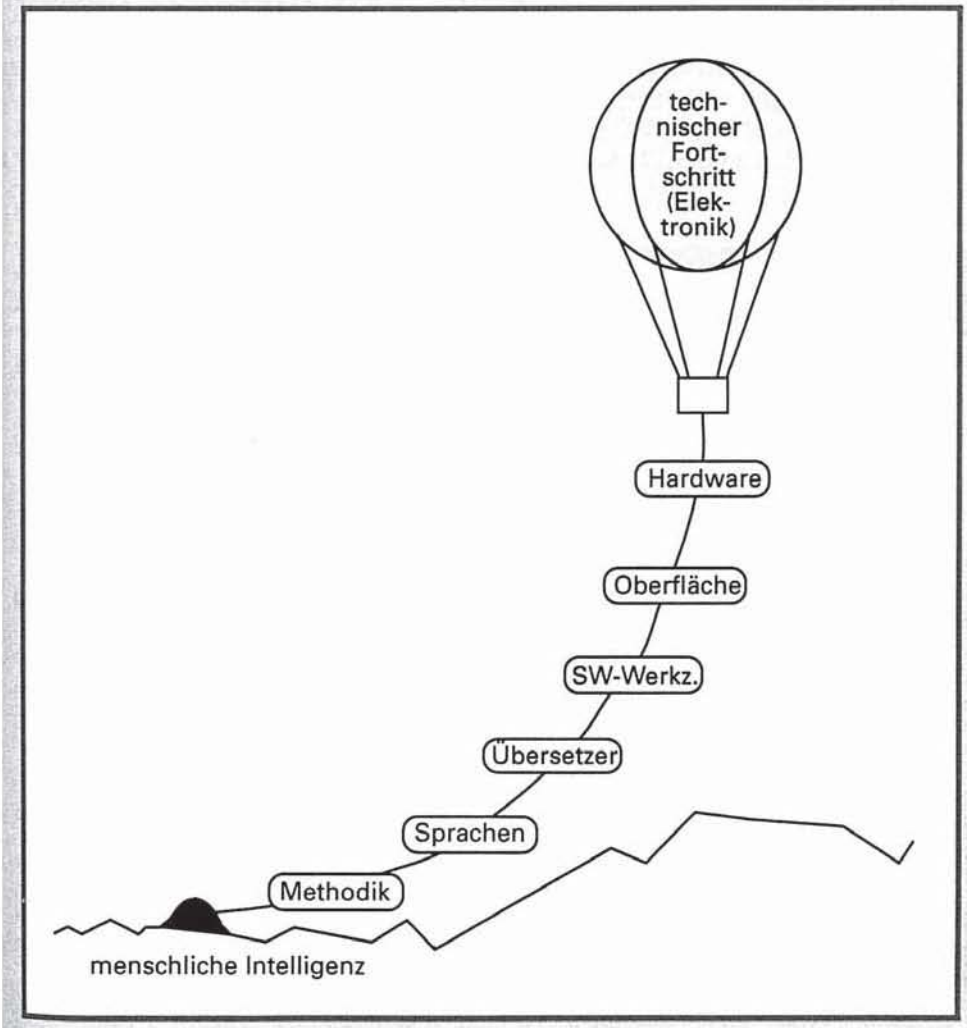
Erste Voraussetzung für planmäßige Veränderungen *jeder* Art ist eine zweckmäßige *Organisation* mit klaren Kompetenzen, in der auch das Qualitätswesen seinen angemessenen Platz hat (FRÜHAUF et al., 1991).

Um mit mehreren Personen ein Fahrzeug, zum Beispiel ein Schiff, zu steuern, muß die Rollenverteilung geklärt sein.

Wenn man daran denkt, daß im politischen Bereich «Kultur» gleichbedeutend ist mit «Schulwesen», so folgt direkt, daß die *Schulung* das wesentliche Mittel darstellt, um auf diesem Wege voranzukommen.

Wie uns die Fluggesellschaften demonstrieren, ist der wichtigste Beitrag zur Verkehrssicherheit eine solide Ausbildung und fortgesetztes Training. 13 TR

Bild 3. Was kann der technische Fortschritt bewegen?



Literatur

Agresti W. W.: New Paradigms for Software Development, IEEE Tutorial, Order No. FJ707, 1986.

Boehm B. W.: Software Engineering Economics, Prentice Hall, Englewood Cliffs, N. J., 1981.

Boehm B. W.: Industrial software metrics top 10 list, IEEE Software, 84–85, September 1987.

Brooks F. P., jr.: No silver bullet – essence and accidents of software engineering, IEEE COMPUTER 20, 4, 10–19, 1987.

Chikofsky E. J.: Computer-Aided Software Engineering (CASE), IEEE Computer Society, Order No. FX1917, 1988.

Conte S. D., Dunsmore H. E., Shen V. Y.: Software Engineering Metrics and Models, The Benjamin/Cummings Publishing Company, Menlo Park, CA, 1986.

Frühauf K., Ludewig J., Sandmayr H.: Software-Projektmanagement und -Qualitätssicherung, Teubner, Stuttgart, und vdf, Zürich. 2. Auflage, 1991.

Grady R. B., Caswell D. L.: Software Metrics: Establishing a companywide program, Prentice Hall, Englewood Cliffs, N. J., 1987.

Halstead M. H.: Elements of Software Science, Elsevier, New York, 1977.

IEEE: Standard glossary of software engineering terminology, IEEE Std. 729–1983, 1983.

Ludewig J.: Software und Qualitätssicherung – Versuch einer Annäherung. In: Tomica K. (Hrsg): Software-Qualiatätssicherung 1987. Schweiz. Arbeitsgem. für Qualitätssicherung, Bern, pp. 7–25, 1987.

Ludewig J.: Modelle der Softwareentwicklung – Abbilder oder Vorbilder? GI Software-Technik-Trends

Ludewig J.: Softwareentsorgung (Sitzung auf der GI-Jahrestagung 1990, mit Beiträgen von R. Thurner, H. Münzenberger, W. Kalmbach, U. M. Osann, U. Kleinau, J. Willems). In: Reuter (Hrsg.), GI 20. Jahrestagung Band I, Springer, Berlin usw., pp. 192–232, 1990.

Peterson G. E.: Object Oriented Computing, IEEE-Tutorial, New York, 1987.

Thayer R. H.: Software engineering project management, IEEE Tutorial, Order No. 751, 1988.

Tracz W.: Software Reuse – Emerging Technology, IEEE Tutorial, Order No. FJ846, 1988.

Voges U.: Normungsaktivitäten auf dem Gebiet des Software-Engineering, Automatisierungstechn. Rundschau 29, 4, 183–187, 1987.

Wegner P.: Dimensions of object-based language design, OOPSLA '87 Proceedings, afcet, Paris, 168–182, 1987.

Quelle: Referat, gehalten anläßlich des Kolloquiums der ABB Informatikschule «Informatik-Praxis in den neunziger Jahren».