Software Qualität:
Nur eine Frage des Software Engineering?

Heinrich C. Mayr

Institut für Informatik
Universität Klagenfurt
A-9022 Klagenfurt

e-mail: mayr@ifi.uni-klu.ac.at

Einleitung

Seit Software entwickelt wird, bereitet sie Qualitätsprobleme; das Schlagwort von der 'Softwarekrise', geprägt in der Mitte der 60er Jahre, ist heute noch längst nicht überholt. '*Pfusch am Riesenairport - Das Software-Debakel am neuen Airport in Denver ist kein Einzelfall: Überall mangelhafte Programme*', so etwa die Überschrift und Headline eines zentral plazierten Artikels in der Ausgabe der KLZ vom 25.11.94, einer der größten österreichischen Tageszeitungen. Ein Artikel, der folgendermaßen endet: '.. *Aber das (das Denver-Problem, Anm. des Autors) ist kein Einzelfall, eher die Regel; oft werden schludrige Erst-Erfassungen als Betaversionen den Kunden hingeworfen. Das Resümee, das man weltweit langsam zieht: Programmieren* ***muß eine Ingenieurdisziplin werden****, die Qualität steigen*'. Und das nach mittlerweile 27 Jahren 'Software Engineering' [Ba93], trotz ISO 9000ff und CMM (Capability Maturity Model [Hu89, Pa93])!

Nun könnte man einwenden, daß im Bereich des Software Engineering, das ja den Anspruch des 'ingenieurmäßigen' Vorgehens erhebt, sehr wohl erhebliche Fortschritte gemacht wurden: Heutige Softwaresysteme sind leistungsfähiger (z.B. hinsichtlich Funktionalität und Durchsatz), benutzerfreundlicher (z.B. hinsichtlich Oberflächengestaltung, Anwenderunterstützung und Toleranz gegenüber Bedienungsfehlern) und meist auch stärker auf den jeweiligen Anwendungsbereich zugeschnitten und damit anwendungsgerechter.

Vom Redakteur einer Tageszeitung, von der breiten Öffentlichkeit und auch von den reinen Software-Anwendern kann man jedoch keine feinsinnige Abwägung aller Fakten erwarten, vielmehr muß man deren Sicht auf unser Metier, und da ist wohl die Informatik insgesamt betroffen, ernst nehmen und ensprechende Konsequenzen ziehen.

Das vorliegende Papier will einen Beitrag hierzu liefern. Anhand von drei Thesen zu Ursachen

für den derzeitigen Stand im Bereich der Softwareentwicklung werden Ansatzpunkte aufgezeigt, die zu einer Qualitätsförderung beitragen könnten.

Einige Thesen

Es gibt sicherlich eine Vielzahl von Gründen unterschiedlichster Natur für die geschilderten Probleme. Ich werde nur einige, m.E. aber sehr wesentliche davon aufgreifen. Da diese eher Grundsatzfragen betreffen, fachpolitisch orientiert sind und damit nicht wirklich beweisbar sind, formuliere ich sie als Thesen, jeweils begleitet von einigen Erläuterungen. Diese Thesen werden sicher kontrovers gesehen und Widerspruch hervorrufen. Das ist beabsichtigt und gewünscht: Die Diskussion könnte das Problembewußtsein schärfen, auf dem aufbauend sich möglicherweise Auswege aus der Misere finden lassen.

These 1[1]***:*** *Da die in der Informatik Tätigen keinen 'Stand' bilden, fehlen ihnen ein entsprechendes Standes- und Selbstbewußtsein, geeignete Standesregeln und dadurch nicht zuletzt auch ein hinreichendes Qualitätsbewußtsein.*

In kaum einer anderen Disziplin ist man derart bereit, alles Machbare auch tatsächlich zu machen, wie in der Informatik: Wo immer Software vorgestellt (z.B. auf Messen, in Präsentationen, in Kundengesprächen) und auf ihre Einsetzbarkeit/Anpaßbarkeit für einen bestimmten Anwender inspiziert wird, sind Worte wie 'jederzeit', 'kein Problem', 'auf Knopfdruck' bis zum Überdruß zu hören. Es gilt (oder es wird zumindest so getan als ob es gelte) das uneingeschränkte Primat des Anwenders/Kunden: *'Software hat sich auf den Anwender, nicht der Anwender auf die Software einzustellen*'. Natürlich ist diese Devise grundsätzlich richtig, andererseits sollten wir als selbst- und verantwortungsbewußte Informatiker vom Machbaren nur das fachlich und sachlich richtige sowie ethisch vertretbare wirklich tun und insbesondere den Anwender dabei unterstützen, seine Anforderungen in diesem Sinne zu stellen und nicht beliebiges grundsätzlich Machbares zu fordern. Das heißt, die Rolle eines/r dem Fach verantwortlichen Beraters/in muß viel stärker wahrgenommen werden, vergleichbar etwa der der Ärztin, des Steuerberaters, der Rechtsanwältin und auch des klassischen Ingenieurs. Ein verantwortungsbewußter Arzt verschreibt einer Patientin beispielsweise kein unnötiges oder möglicherweise sogar schädliches Medikament, nur weil sie das wünscht. Und eine Automobilingenieurin wird auch nicht einen Fernsehapparat in das Armaturenbrett eines Autos einbauen, nur weil ein Käufer dies möchte.

Natürlich bestätigen auch bei diesen Beispielen Ausnahmen die Regel. Solange es aber keine breit akzeptierten Spielregeln und Qualitätsstandards gibt, solange man zu allem bereit ist bzw. sein zu müssen glaubt, solange wird auch kein hinreichendes Qualitätsbewußtsein entstehen. Die Art und Weise, in der etwa die Zertifizierung nach ISO 9000 mancherorts betrieben und diskutiert wird, und auch die dabei häufig genannte Motivation bestätigen mich in dieser Ansicht.

) Thesen 1 und 2 wurden von mir letztes Jahr im Rahmen einer Strategiediskussion innerhalb des Ausschusses Forschung und Technologie der Gesellschaft für Informatik (GI) eingebracht; einige Beiträge aus der dazu geführten Diskussion sind im folgenden berücksichtigt.

Man könnte nun fragen, ob 'Stände' überhaupt noch in unsere Zeit passen, ob Standesethik und Standesvertretung nicht eher etwas gestriges, abzuschaffendes sei. Ich bin weit davon entfernt, Überholtes restaurieren zu wollen, andererseits sollte man aber versuchen, positive Aspekte wahrzunehmen und in irgendeiner Form zu übernehmen.

Häufig wird argumentiert, der Markt lasse Selbstbeschränkungen im obigen Sinne gar nicht zu und insbesondere verhinderten die am Markt durchsetzbaren Preise eine konsequente Qualitätsorientierung. Es ist richtig, daß der Markt insgesamt stark verunsichert ist: Jedermann zugängliche Massensoftware für PC's hat ein Preis-/Leistungsverhältnis, das für individuelle und branchenspezifische Systeme nicht erreicht werden kann aber entsprechende Erwartungen weckt. Viele Anbieter versuchen nun (insbesondere als Marktneulinge, um Fuß zu fassen, meist im Wege der Selbstausbeutung), diesen Erwartungen mit Niedrigpreisen entgegenzukommen. Sie zeigen damit das 'irgendwie Machbare' und erzeugen einen entsprechenden Druck. Danach verschwinden sie häufig wieder. Der Anwender, zurückgeblieben mit einer nicht weiter gewarteten Software, sucht sich notgedrungen einen neuen Anbieter, will aber natürlich für dieselbe Leistung nicht mehr als beim ersten Mal investieren. Entscheidet er sich doch dafür, muß er häufig feststellen, daß er sich zwar eine möglicherweise stabilere Betreuung nicht aber bessere Qualität eingehandelt hat. Dies erzeugt neuen Druck, den man m.E. langfristig nur durch konsequente und konzertierte (im o.g. Sinne 'ständische') Qualitätsorientierung lindern kann.

Eine ähnliche Sicht vertritt auch N. Wirth in [Wi95], indem er unter der Überschrift 'causes for fat software' Softwarekomplexität und daraus resultierende Qualitätsprobleme als negative Folgen mangelnder Selbsbeschränkung kritisiert: '*A primary cause of complexity is that software vendors uncritically adopt almost every feature that users want. Any incompatibility with the original system concept is either ignored or passes unrecognized, which renders the design more complicated and its use more cumbersome. [...] quantity becomes more important than quality...*'. Seine Schlußfolgerung, die in ähnlicher Weise auch von anderen Autoren vertreten wird, ist, Software 'schlank' zu halten, d.h. ihre Komplexität zu reduzieren, indem man ihren Umfang auf ihre Funktionalität abstimmt ('*software girth has surpassed its functionality*') diziplinier Methoden anwendet und zu den 'essentials' zurückkehrt.

These 2: *Das Software-Ingenieurwesen (Software-Engineering) ist - bei allen erreichten Verbesserungen - noch keine wirkliche Ingenieurdisziplin.*

Von Software Engineering wird heute zwar auf breitester Ebene im Zusammenhang mit der Herstellung von Software geredet. Eine wirklich ingenieurmäßige Softwareentwicklung wird jedoch nicht flächendeckend praktiziert. Das liegt zum einen daran, daß die meisten Softwarehersteller (eigenständige Unternehmen oder DV/IV-Abteilungen) eher kleine bis sehr kleine Einheiten sind, in denen der Entwicklungsprozess noch nicht definiert, ausgereift oder gar optimiert ist [Hu89,Pa93]. Dadurch können Software Engineering Methoden gar nicht umfassend und nutzbringend geplant und eingesetzt werden.

Andererseits scheinen die von der Wissenschaft angebotenen Methoden für die Praxis nicht überzeugend bzw. bekannt genug zu sein: Sie würden sonst sicherlich konsequenter verwendet werden, als dies heute der Fall ist, insbesondere von ausgebildeten Informatikern. Ingenieurmethoden sollen ja dem Verwender helfen, seine Aufgaben systematisch, zielorientiert, qualitätsorientiert und

vollständig zu lösen - und zwar in jedem Fall rationeller, als wenn sie nicht eingesetzt würden. Diesen Reifegrad haben die Methoden des Software Engineering zumindest in ihrer Gesamtheit noch nicht. Häufig steht ihrer Verwendung auch ein gemessen am erwarteten oder versprochenen Nutzen als zu hoch empfundener Einführungsaufwand entgegen. Dazu kommt, daß in der Vergangenheit eine schier unübersehbare Menge von Konzepten, Modellen, Methoden und zugehörigen Werkzeugen entwickelt, publiziert, angepriesen und schließlich wieder weggeworfen bzw. dem Vergessen überantwortet wurden. Viele davon unterscheiden oder unterschieden sich nur in Nuancen voneinander, Intergration fand nur beschränkt statt, passable Schnittstellen zwischen Werkzeugen gibt es erst in letzter Zeit. Fast immer wurde jedoch der Anspruch erhoben, der entscheidende Schritt in Richtung optimaler Unterstützung des Entwicklungsprozesses sei jetzt getan.

Einer der Gründe für diese Erscheinung (oder vielleicht auch eine Folge davon?) ist der Umstand, daß die Kriterien der Bewertung von wissenschaftlicher Leistung in der Informatik stark akademisch und weniger ingenieurwissenschaftlich geprägt sind und deshalb im Zweifel vom Produkt wegführen. Im Vordergrund steht als Erfolgskriterium (zumindest im deutschsprachigen Raum) die internationale Veröffentlichung (z.B. über eine Methode), nicht aber deren erfolgreiche Umsetzung und Bewährung in der Praxis, ein Produkt oder dessen Qualität. Viele der im Forschungsbereich geförderten sog. Entwicklungsprojekte führen daher meist nicht einmal zu Prototypen sondern höchstens zu Vorstufen davon. Viele derjenigen, die Softwaremethoden lehren oder weiterentwickeln, haben keine oder kaum Erfahrung in der Entwicklung und Betreuung von marktreifer Software. Um den Anforderungen einer Ingenieurdisziplin gerecht zu werden, ist daher die positivere Bewertung von praxisorientierter Arbeit, von Patenten, von Technologietransfer u.ä. geboten.

Schließlich kann man sich auch fragen, ob es überhaupt richtig ist, Softwareentwicklung als Ingenieurdisziplin zu sehen: Ingenieursdenken ist anschaulich, Informatikdenken abstrakt. Kurz umrissen besteht die Aufgabe des Softwareentwicklers darin, ***von Konkretem zu abstrahieren und daraus Formales (ein Softwaresystem) zu konstruieren***. Der Begriff Software ist dabei nicht auf die reine Programmebene beschränkt, vielmehr werden hier die Ergebnisse jeder beliebigen Phase eines (einem bestimmten Vorgehensmodell folgenden) Entwicklungsprozesses als Software angesehen. Informatik ist damit möglicherweise eine technische Disziplin neuen Typs, die eigene Methoden entwickeln, bzw. aus verschiedenen anderen Disziplinen übernehmen muß. Aus einem anderen Blickwinkel ist dies auch Gegenstand von These 3.

These 3: *Der Ansatz, Softwareentwicklung als reine Ingenieurdisziplin zu sehen, deckt höchstens deren konstruktiven Aspekt ab: Softwareentwicklung ist zu einem beträchtlichen Teil aber auch eine Produktionsdisziplin. Eine Nutzung von Methoden aus diesem Bereich kann zu Verbesserungen des Entwicklungsprozesses und seiner Ergebnisse führen.*

Diese These berührt das Selbstverständnis des/der Informatikers/in und ist daher nach meiner Erfahrung diejenige, zu der am stärksten mit subjektiven Glaubensgrundsätzen anstatt mit Sachargumenten diskutiert wird, obwohl im Ansatz vergleichbare Ideen seit Jahren diskutiert werden (z.B. [Ma87, We89, Ho92]). Dabei liegt es mir fern, die Tätigkeit eines Systemingenieurs oder Programmierers abzuwerten. Nimmt man aber einen unvoreingenommenen Standpunkt ein, dann

kann man durchaus weite Bereiche der Aktivitäten von Softwareherstellern als Produktionstätigkeiten ansehen. Dies gilt insbesondere für solche Softwarehersteller, die nicht Individualsoftware sondern vertikale (z.B. Branchensoftware) oder horizontale (z.B. für den Bereich Rechnungswesen) Software erstellen und pflegen. In diesen Fällen, in denen man übrigens ja schon seit langer Zeit von 'Softwareprodukten' spricht, ohne sich der Implukation dieses Begriffes wirklich bewußt zu sein, bestehen 'Projekte' meist in der Entwicklung oder Modifikation eines oder einer beschränkten Zahl von Modulen, jeder mit einer klar definierbaren bzw. definierten Funktionalität. D.h. diese Projekte könnten auf der Basis standardisierter Arbeitsablaufdefinitionen ('Arbeitsgangbeschreibungen' im Sinne der Produktionsplanung und -steuerung, siehe z.B. [Sch93]) und Arbeitspläne durchgeführt werden. Herkömmliche PPS-Systeme könnten dann zur Planung und Steuerung in solchen Umfeldern eingesetzt werden. Über Erfahrungen hierzu wird in einer anderen Arbeit berichtet [MU95].

Der Entwurf und die Konstruktion dieser Arbeitspläne und Arbeitsgänge wäre dagegen eine Aufgabe des Software Engineering (vergleichbar den ersten drei Schritten des rechten Y-Zweiges in Scheers Modell, siehe Abb. 1). Die Lebensdauer solcher Projekte liegt zwischen einigen Stunden und einigen Wochen. Nach meinen eigenen Erfahrungen liegen in einem mittleren Softwarehaus gleichzeitig mehrere Dutzend bis zu einigen Hundert solcher 'Projekte' vor und müssen dementsprechend eingeplant und koordiniert werden. Dabei kommt es häufig zu 'Splittungen' und 'Raffungen', wie sie für den Produktionsbereich typisch sind. Klarerweise sind die heute verbreiteten Softwarewerkzeuge für das Projektmanagement für die Administration und Planung derartiger Projekt(mengen) nur bedingt geeignet, da sie ja dem Paradigma der Ingenieurdisziplin Software Engineering folgen und damit auf wenige, aber umfangreiche Projekte abzielen.

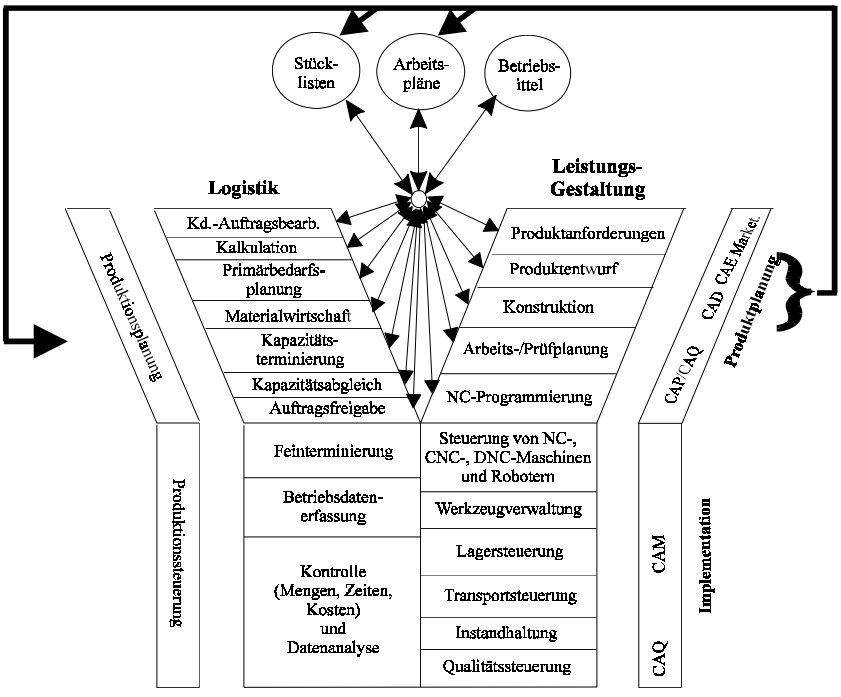

Abb. 1: Y-Modell der CIM-Integration [Sch93], vom Software Engineering zur Software Produktion

Bis zu einen gewissen Grad läßt sich das Produktionsparadigma aber auch auf die Erstellung von Individualsoftware bzw. neuer Softwaresysteme anwenden: Zur Beherrschung der Komplexität derartiger Projekte ist es heute üblich, einerseits hierarchische und Baustein-orientierte Architekturkonzepte (z.B. [De79, DH80, HSE90]) zu verfolgen und andererseits nach vordefinierten Prozessmodellen vorzugehen (z.B. [Bo88, BD93]). Darüberhinaus wird unter dem Schlagwort 'Software Reuse' zunehmend intensiv die Wiederverwendung der Ergebnisse früherer Entwicklungsprozesse diskutiert und praktiziert (z.B. [GK94, HM95]). Dabei beschränkt sich die Wieder-verwendung nicht nur auf die Produkte der Realisierungsebene, d.h. Programmbausteine, Klassen u.ä., sondern sie erstreckt sich auch auf die

höheren Ebenen des Entwicklungsprozesses (z.B. [EW93], [KM94]). Bei entsprechender Definition der Arbeitsgänge zur Bausteinproduktion, zur Baustein- bzw. Entwurfselementwiederverwendung könnte auch hier wieder PPS-ähnlich, also logistisch geplant und gesteuert werden. Voraussetzung hierfür ist eine Aufwandsabschätzungsmethode, die es erlaubt, die für die Produktion bzw. Anpassung/ Einpassung eines Bausteins benötigte Zeit, Mittel und Werkzeuge möglichst genau zu bestimmen, da diese Parameter für die Einlastungsverfahren (Scheduling) von PPS-Systemen benötigt werden. Wir schlagen hierfür eine aus der Function Point Analyse abgeleitete Methode vor [MU95].

Ausblick

Vieles in diesem Papier ist nicht grundsätzlich neu, in dieser Form kombiniert wurde es meines Wissens aber noch nicht behandelt. Es sollte damit die Aufmerksamkeit auf einige Aspekte der Softwareentwicklung gelenkt werden, bei denen sich durch etwas Umdenken bzw. durch einen Paradigmenwechsel Lösungsansätze für bestehende Probleme der Softwareentwicklung ergeben könnten. Die getroffene Auswahl dieser Aspekte ist mit Sicherheit nicht vollständig, es besteht weiterer Diskussions- und Untersuchungsbedarf. Dies gilt auch für die Umsetzung der hier vorgeschlagenen Ansätze.

Für die kritische Durchsicht und einige interessante Anregungen danke ich meinem Mitarbeiter, Herrn Dr. Roland Kaschek.

Literatur

[Ba93] Bauer, F.L.: Software Engineering - wie es begann. Informatik Spektrum, Vol. 16., Nr. 5, 1993, S. 259-260.

[BD93] Bröhl, A.-P.; Dröschel, W. (eds.): Das V-Modell. R. Oldenbourg-Verlag, 1993.

[Bo88] Boehm, B.W.: A Spiral Model of Software Development and Enhancement. IEEE Computer, Vol. 21, Nr. 5, Mai 1988, S. 61-72.

[De79] Denert, E.: Software-Modularisierung. Informatik-Spektrum Vol. 2, 1979, S. 204-218.

[DH80] Denert, E.; Hesse, W.: Projektmodell und Projektbibliothek: Grundlagen zuverlässiger Software-Entwicklung und Dokumentation. Informatik Spektrum Vol. 3, 1980, S. 215-218.

[Dr89] Dreger, J.B.: Function Point Analysis. Prentice Hall, 1989.

[EW93] Eder, J.; Welzer, T.: Meta Data Model for Database Design. In (Marik, V.; Wagner, R. eds.): Proc. DEXA 93, 4th Conf. on Database and Expert Systems Applications, Springer Verlag, 1993, pp. 677-680.

[GK94] Gall, H.; Klösch, R.: Reuse Engineering: Software Construction from Reusable Components. Proc. 16th IEEE Annual Int. Computer Software & Applications Conf., Sept. 1992, S. 79-86.

[Ho92] Hochmüller, E.: AUGUSTA - Eine reuse-orientierte Software-Entwicklungsumgebung zur Erstellung von Ada-Applikationen. Dissertation, Univ. Klagenfurt, 1993.

[HM95] Hochmüller, E.; Mittermeir, R.T.: Improving the Software Development Process by Planned Software Reuse. In (J. Györkös et al. eds.): Proc. ReTIS'95, 4th Int. Conf. on Re-Technologies for Information Systems. Oldenbourg Verlag, 1995 (to appear).

[HSE90] Henderson-Sellers, B.; Edwards J.M.: The Object-Oriented Systems Life Cycle. ACM Communications, Vol. 33, Nr. 9, Sept. 1993, S. 142-159.

[Hu90] Humphrey, W.S.: Managing the Software Process. Addison-Wesley Publ. Co., 1st Reprint, 1990.

[IFPUG94] Int. Function Point Users Group: Counting Practices Manual, Rel. 4.0, 1994.

[Kn69] Knuth, D.E.: The Art of Computer Programming. Vol. 1 and 2, Addison-Wesley Publ. Co., 1969.

[KM94] Kop, Ch.; Mayr, H.C.: Reusing Domain Knowledge in Requirement Analysis. In (Györkös, J. et al. eds.): Proc. RE'94, 3rd Conf. on Re-Engineering of Information Systems, Universitätsverlag Ljubjana, 1993, S. 133-147.

[Ma87] Matsumoto, Y.: A Software Factory: An Overall Approach to Software Production. In (Freeman, P. ed.): IEEE Tutorial 'Software Reusability', 1987, S. 155-178.

[MU95] Mayr, H.C.; Urich, Ch.: A CIM Based Approach to Software Project Management. Submitted for publication.

[Pa93] Paulk, M.C. et al.: Capability Maturity Model for Software, Version 1.1.

[Sch93] Scheer, A.-W.: Wirtschaftsinformatik - Informationssysteme im Produktionsbetrieb. Springer Verlag, 1993.

[We89] Weber, H.: From CASE to Software Factories. Datamation, April 1989, S. 34-36.

[Wi95] Wirth, N.: A Plea for Lean Software. IEEE Computer, Vol. 28, Nr. 2, Feb. 95, S. 64-68.