VOM ATELIER ZUR WERKSTATT: ANSÄTZE ZUR SYSTEMATISCHEN SOFTWARE-ENTWICKLUNG

Prof. Dr. rer. nat. Jochen Ludewig
Institut für Informatik der Universität Stuttgart

Zusammenfassung

Software Engineering bedeutet den Versuch, die "Art of Computer Programming" durch ein systematisches Vorgehen nach dem Vorbild anderer Ingenieur-Disziplinen zu ersetzen. Der Artikel skizziert die - vor allem durch die kurze Geschichte und die spezifischen Eigenschaften der Software begründeten - Schwierigkeiten und zeigt, wo und wie Verbesserungen möglich sind. Am Schluß wird eine "Software-Kultur" als eigentliches Ziel beschrieben.

Hinweis: Dieser Beitrag beruht zum Teil auf früheren Arbeiten des Verfassers, insbesondere [1].

1 Atelier und Werkstatt

Ein *Atelier* bezeichnet in diesem Beitrag den Arbeitsplatz eines Künstlers. Dem wird die *Werkstatt* als eine industrielle Umgebung gegenübergestellt. Die beiden Begriffe dienen also als Metaphern, nicht als konkrete Beispiele. Allerdings sind vergleichbare Begriffe, beispielsweise die "Software Factory", in der Vergangenheit durchaus mit der Intention einer weitergehenden Analogie geprägt worden. Dieser Aspekt kommt im Abschnitt 4 noch kurz zur Sprache.

1.1 Unsere Schwierigkeiten mit der Bearbeitung von Software-Systemen

Software - wir verstehen darunter **die Menge aller dauerhaft gespeicherten Informationen, die ein Programm bilden oder zu seiner Herstellung, Änderung oder Verwendung dienen** - unterscheidet sich in vieler Hinsicht von anderen industriellen Produkten, vor allem durch

- den immateriellen Charakter
- die nicht-kontinuierlichen Eigenschaften
- die ungeheure Komplexität
- die leichte Replizierbarkeit (und damit das Fehlen einer eigentlichen Fertigung)
- die kurze, sehr rasche Entwicklung, die in vierzig Jahren aus einer völlig neuen Erfindung eine Schlüsseltechnologie hat werden lassen.

Daher ist es nicht erstaunlich, daß Software auch besondere Schwierigkeiten macht. Worin diese im einzelnen bestehen, ist durchaus umstritten, aber die Auswirkungen sind allgemein bekannt: In vielen Software-Projekten werden die zuvor geschätzten **Kosten** und **Termine** erheblich überschritten, und das Resultat entspricht in **Funktion** und **Qualität** nicht den Erwartungen.

1.2 Software-Krise und Software Engineering

Gegen Ende der sechziger Jahre zeigten sich die oben genannten Probleme erstmals in einem Maße, daß man dafür ein neues Wort ("Software-Krise") prägte und auf Abhilfe sann. Teilnehmer eines internationalen Workshops in Garmisch sahen 1968 den Ausweg in der Orientierung an den Ingenieur-Wissenschaften, und sie brachten das Wort *Software Engineering* in die Diskussion

Software Engineering war damit zunächst nichts als eine Idee, ein neuer Denkansatz: Der Programmierer sollte sich nicht mehr als Künstler verstehen, sondern als ein Ingenieur, der nach wohldefinierten Verfahren arbeitet und dessen Ergebnisse nach objektiven Maßstäben beurteilt werden können.

Im Hinblick auf die Intention, die hinter dem neuen Wort steht, ist es interessant, die Produkte aus Werkstatt und Atelier miteinander zu vergleichen. Dabei ist zu beachten, daß der Vergleich überhaupt nur bei Einzelstücken sinnvoll ist (z.B. im Anlagenbau), weil, wie oben festgestellt wurde, in der Software der Aspekt der Serienfertigung fehlt. Die folgende Tabelle stellt einige Charakteristika gegenüber:

	Werkstatt (technisches Produkt)	**Atelier (Kunstwerk)**
geistige Voraussetzung	das erforderliche technische Know-How ist vorhanden und verfügbar	Inspiration des Künstlers
Termine	in der Regel mit genügender Genauigkeit planbar	wegen Abhängigkeit von Inspiration nicht planbar
Preis	an den Kosten orientiert, damit kalkulierbar	nur durch den Marktwert, nicht durch die Herstellungskosten bestimmt
Standards	entspricht technischen Regeln und Normen	durchbricht Regeln, soweit solche überhaupt vorhanden sind
Bewertung	nach objektiven Kriterien prüfbar	nur subjektive Bewertung möglich
Urheber	der Urheber bleibt meist anonym, keine dauerhafte Bindung zum Produkt	Künstler betrachtet das Kunstwerk als Teil seiner selbst, bleibt verbunden

Das Bild, das die Software heute in der Regel bietet, entspricht leider viel mehr dem des Kunstwerks als dem des technischen Produkts:

- Termine und Kosten werden in der Planung ohne rationale Grundlage geschätzt und in der Realisierung vielfach weit überschritten;
- Regeln sind nicht existent oder nicht bekannt oder werden ignoriert, und das gleiche gilt verstärkt für Normen;
- durch Programmtest läßt sich nur eine sehr schwache Aussage über die Funktionalität des Programms erzielen, alle anderen Bewertungen (z.B. der Portabilität oder der Robustheit) sind subjektiv und daher nicht allgemein anerkannt;
- der Programmierer baut seine Persönlichkeit in das Programm ein, so daß er Kritik daran kaum akzeptieren kann; das Programm wird auf diese Weise für andere Menschen unverständlich und ist nicht mehr wartbar, wenn der Urheber nicht mehr zur Verfügung steht.

Wie man sieht, ist der Anspruch des Software Engineering bis heute nicht eingelöst (siehe z.B. [2]). Doch die oben genannten Merkmale technischer Produkte spiegeln sich bereits in den Aktivitäten dieses jungen Gebietes: Es gibt seit Mitte der siebziger Jahre eine wachsende Zahl veröffentlichter Arbeiten über Software-Management, -Kostenschätzung, -Spezifikation, -Entwurf, -Metriken, -Validierung und formale Verifikation. Seit Beginn der achtziger Jahre sind empirische Daten (Einsatzerfahrungen und vergleichende Statistiken) hinzugekommen. Schließlich entstand der Begriff des "ego-less-programming" [3], wodurch ein Programmierstil bezeichnet wird, bei dem das Produkt nicht mehr die Handschrift seines Verfassers trägt, sondern vorgegebenen Normen entspricht und damit auch anderen Programmierern leicht verständlich ist.

2 Der technische Ansatz

Entsprechend der Bezeichnung "Software Engineering" ist es naheliegend, die Probleme durch technische Maßnahmen zu bekämpfen. Dabei orientiert man sich seit etwa 1970 an den Entwicklungsphasen, die ein Software-System durchläuft, also am sogenannten **Software Life Cycle**:

Systemanalyse

Spezifikation der Anforderungen

Grobentwurf und Modulspezifikation

Feinentwurf

Codierung und Modultest

Integration und Test

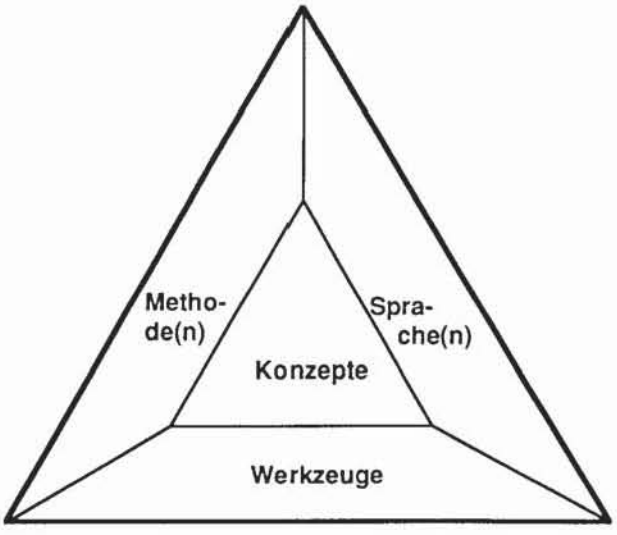
Betrieb mit Korrektur, Anpassung und Erweiterung ("Wartung")

Diese Gliederung wurde erstmals 1970 von Royce [4] publiziert. Ihr großer Nutzen bestand nicht so sehr in der speziellen Gliederung als vielmehr in der Strukturierung überhaupt, die die Aufmerksamkeit auf *alle* Phasen lenkte; zuvor hatten sich die Informatiker im wesentlichen auf die Codierung konzentriert.

2.1 Möglichkeiten technischer Verbesserungen

Um die möglichen Ansatzpunkte sichtbar zu machen, ist eine weitere, zum Life Cycle orthogonale Gliederung in **Methoden**, **Sprachen** und **Werkzeuge** zweckmäßig. Methoden sind Mengen von Regeln, die uns beim Vorgehen leiten. Sprachen sind durch Syntax und Semantik gekennzeichnete Notationen. Werkzeuge schließlich sind Hilfsmittel, die uns bei der Arbeit unterstützen, in der Praxis häufig die Arbeit erst ermöglichen. Beispiele auf der Ebene der Codierung wären die Methode der prozeduralen Formulierung eines Programms, die Sprache Pascal und das Werkzeug Pascal-Compiler.

Die folgende Graphik (aus [5]) soll den engen Zusammenhang zwischen diesen drei Begriffen anschaulich machen. Ihnen gemeinsam ist ein abstrakter Kern, die zugrundeliegenden semantischen Konzepte.



Um technische Verbesserungen zu erzielen, können also - nicht unabhängig voneinander, sondern in enger Abstimmung - Methoden entwickelt, Sprachen definiert und Werkzeuge bereitgestellt werden.

2.2 Methoden

Noch heute sind viele Programmierer im Grunde davon überzeugt, daß Programmieren eigentlich einfach sei und keiner Methoden bedürfe (d.h. daß sie intuitiv über alle wesentlichen Methoden verfügten). Tatsächlich wurden viele Methoden in die Sprachen "eingebaut", so daß wir sie heute ganz unbewußt verwenden, beispielsweise die schrittweise Verfeinerung (als Prozedur-Konzept), der Verzicht auf strukturensprengende Sprunganweisungen (als Verzicht auf das GOTO in modernen Sprachen), die Definition der verwendeten Daten (als Deklarationszwang). In der Art und Weise, wie bereitwillig oder widerspenstig Programmierer mit den entsprechenden Konzepten umgehen, erkennt man, wieweit sie die Methoden verstanden haben.

Andere Methoden sind noch nicht Allgemeingut und werden daher in zu geringem Maße beachtet, vor allem das Prinzip, jede Information nur in einem möglichst engen Rahmen verfügbar zu machen und die Zugriffe strikt zu kontrollieren ("information hiding" [6]). Solche modernen Ansätze sind daher wichtige Schulungsthemen.

Auf den Ebenen der Spezifikation und des Entwurfs wurden seit etwa 1975 verschiedene Methoden entwickelt, die die ersten Schritte auf dem Weg zum Software-Produkt weniger abenteuerlich machen. Ein besonders populärer Ansatz ist "Structured Analysis", ein anderer "Jackson Structured Design". Ihre große Beliebtheit besagt nicht, daß sie perfekt funktionieren, sondern vor allem, daß die Anwender ein starkes, bis heute kaum befriedigtes Bedürfnis nach methodischer Führung haben.

Für den Software-Test wurden ebenfalls Methoden entwickelt, die ein systematisches Vorgehen fördern (z.B. durch geeignete Testdatenauswahl oder durch Verfahren zur Bewertung des Test-Erfolgs). Das gleiche gilt auch für die quantitative Beurteilung der Software ("Metriken"). Hier ist allerdings bis heute kein Einverständnis gewachsen, das den allgemeinen Austausch von Software-Daten sinnvoll machte.

Zahlreiche Methoden sind ganz oder teilweise dem organisatorischen Bereich zuzuordnen; dort werden auch die Vorgehensmodelle behandelt (siehe 3.2 und 3.3).

2.3 Programmiersprachen

Entsprechend der großen Aufmerksamkeit, die die Codierung seit Beginn der Informatik hatte, wurde das Feld der Programmiersprachen seit langem mit großem Aufwand und entsprechendem Erfolg bearbeitet (vgl. [7]). Auf die Assemblersprachen folgten um 1960 die ersten "höheren" Sprachen (FORTRAN, COBOL, ALGOL60). in den beiden folgenden Jahrzehnten entstanden zahlreiche weitere Sprachen; einerseits verfolgte man das Ziel, in einer Sprache alle bekannten Möglichkeiten bereitzustellen (PL/I, ALGOL68, Ada), andererseits gab es einen Trend zu Sprachen, die - unter verschiedenen Aspekten - einfach waren, so das extrem einfach erlernbare BASIC, das konzeptionell bestechend einfache Pascal und das maschinennahe C. Im letzten Jahrzehnt brachte das Prinzip "Information Hiding" Impulse, die zu MODULA-2 und Ada führten und letztlich in den Trend zur objektorientierten Programmierung (Smalltalk, C++) mündeten. Parallel zu den erwähnten Standardsprachen entstanden Spezialsprachen für spezielle Anwendungen (z.B. CHILL) oder auf der Grundlage neuer Konzepte (LISP, PROLOG); letztere sind erst in jüngster Zeit durch die erhöhte Leistung der Rechner praktisch brauchbar geworden.

Betrachtet man die Rezeption der Programmiersprachen, so kommt man nicht an der Feststellung vorbei, daß die Praxis vielfach auf dem Stand der frühesten Sprachen stehengeblieben ist, allenfalls steigt man auf neuere Varianten um, die gewisse Verbesserungen, aber keine konzeptionellen Durchbrüche bedeuten (FORTRAN 77, COBOL 85). Vielfach wird noch heute in Assembler gearbeitet.

Für diese Stagnation gibt es nur teilweise rationale Gründe. In vielen Fällen dürften die Trägheit von Einzelpersonen und Organisationen und die Angst vor Neuerungen, begründet durch die im Durchschnitt unzulängliche Ausbildung der Programmierer und ihrer Vorgesetzten, die wahren Motive sein. Moderne Sprachen sind keine Allheilmittel, und es trifft zu, daß man in allen Sprachen gut oder schlecht programmieren kann, doch fördert (und erfordert) jede Sprache einen ihr spezifischen Denkansatz, der die Software-Qualität wesentlich beeinflußt.

Die Darstellung der Programme durch Text, also durch Zeichenketten, wurde schon sehr früh als unzulänglich empfunden, und man entwickelte graphische Notationen, die die Lesbarkeit verbessern sollten. Diese blieben zunächst auf der Ebene des Codes (Flußdiagramme, Nassi-Shneiderman-Diagramme) und konnten auch wegen ihrer umständlichen Bearbeitung keinen durchschlagenden Erfolg haben. Eine wesentliche Änderung kam durch zwei Einflüsse: Zum einen wurden - vor allem seit 1980 - Werkzeuge realisierbar, die die Bearbeitung graphischer Darstellungen wesentlich erleichterten, zum anderen entstanden Notationen auch für Spezifikationen und Entwürfe (z.B SADT, Data Flow Diagrams, Jackson Diagramme, Entity-Relationship-Diagramme) Es ist anzunehmen, daß wir in dieser Hinsicht erst am Anfang der Entwicklung stehen

2.4 Werkzeuge

Nachdem die Werkzeuge für die Codierphase (Compiler, Linker, Laufzeitsysteme, mittelbar auch Editoren, Datei-Systeme) bereits einen hohen Entwicklungsstand erreicht hatten, begann man in den Anfangsjahren des Software Engineerings, auch Werkzeuge für andere Phasen zu entwickeln. Nachdem die Spezifikationsphase als Schwachpunkt der Entwicklung erkannt war, standen Spezifikationssysteme um 1980 herum im Mittelpunkt des Interesses [8]. Obwohl auf diesem Gebiet in der Nachfolge des Pioniersystems PSL/PSA [9] beachtliche Ergebnisse erzielt wurden, kam es nicht zu dem erwarteten Durchbruch bei den Anwendern. Dies hatte vermutlich verschiedene Gründe:

- Die Anwender waren methodisch nicht ausreichend vorbereitet.
- Die Werkzeuge waren methodisch nicht konsolidiert.
- Die Qualität der - dabei sehr teuren - Werkzeuge war unbefriedigend
- Die Bedienschnittstelle war primitiv.
- Der Übergang zur Codierung war nicht oder unzureichend unterstützt.

Folgerichtig ging in den achtziger Jahren der Trend zu integrierten (also im ganzen Life Cycle einsetzbaren) Werkzeugen, die als "Software Engineering Environments" oder ähnlich bezeichnet wurden. Nachdem nun auch auf relativ billigen Maschinen graphische Schnittstellen möglich geworden sind, ist es unter dem neuen Schlagwort "CASE" [10] zu einem eigentlichen Werkzeug-Boom gekommen. Dabei wird leicht übersehen (oder in der Werbung bewußt überspielt), daß die konzeptionellen Probleme weiterhin überwiegend ungelöst sind.

Generell läßt sich aber feststellen, daß heute eine Reihe interessanter Werkzeuge auf dem Markt sind, die bei entsprechend sorgfältiger Einführung beträchtlichen Nutzen vor allem für die Produkt- und die Projekt-Qualität haben können. Allerdings müssen dazu beim Management einige wichtige Voraussetzungen erfüllt sein:

- Die Einführung eines Werkzeugs muß als langfristige Investition betrachtet werden; kurzfristig, d.h. für einige Monate, *sinkt* in der Regel die Produktivität.
- Die Schulung erfordert sehr viel Aufwand.
- Werkzeuge können keine Defizite auf anderen Gebieten kompensieren (Führung, Ausbildung, Planung, Kontrolle).

3 Der organisatorische Ansatz

Es stimmt zwar, daß die Software viele technische Probleme mit sich bringt, doch wird sie darüberhinaus oft zum Sündenbock für verschiedene andere Unzulänglichkeiten gemacht, die vor allem im organisatorischen Bereich liegen. Offenbar entstehen besondere Schwierigkeiten an der Schnittstelle zwischen Organisation und Technik, also beispielsweise, wenn eine Software-Entwicklung geplant wird. Dieses Problem wird durch zwei Effekte verstärkt:

- Viele der Software-Entwickler betreiben ihren Beruf wie ein Hobby, dem sie sich mit Leib und Seele widmen, dabei aber die organisatorische, kaufmännische und juristische Umgebung weitgehend ausblenden. Diese Leute sind oft sehr sympathisch, aber schwer zu führen.
- Anders als im Maschinenbau und in der Elektrotechnik haben viele der Führungskräfte in der Informatik kein ausreichendes Grundwissen, um von den Fachleuten anerkannt zu werden und kompetente Entscheidungen zu treffen.

Der Konflikt bekommt also von beiden Seiten Brennstoff.

3.1 Möglichkeiten organisatorischer Verbesserungen

Organisatorische Verbesserungen zielen entsprechend auf eine verbesserte Führung und engere Kontrolle, die entgegen einem gängigen Vorurteil zum Nutzen *aller* Beteiligten ist. Natürlich sind dies nur zwei Seiten derselben Münze: Wo keine Führung war, ist keine Planung vorhanden, und es ist nicht möglich, die Resultate zu bewerten. Das gilt auf technischer wie auf organisatorischer Ebene. Software-Management und Software-Qualitätssicherung bedingen sich also gegenseitig.

3.2 Software-Management

Es ist zwar banal, aber in der Praxis oft ignoriert, daß Software-Projekte zunächst einmal *Projekte* sind und entsprechend vorbereitet sein müssen. Wenn Leute oder Betriebsmittel fehlen, wenn die Fluktuation allzu hoch ist, wenn Kompetenzen ungeklärt sind oder die Anforderungen laufend geändert werden, scheitern nicht nur Software-Projekte.

Ein zentraler Punkt (und in vielen Fällen der entscheidende Schwachpunkt) ist die **Planung**. Metzger [11] schätzt, daß hier bei der Hälfte der gescheiterten Software-Projekte die Ursache liegt. Planung ist auf diesem Felde besonders schwierig, weil

- viele Projekte völlig neu sind und daher ein Analogieschluß nicht möglich ist,
- in weit geringerem Maße als bei Ingenieuren auf fertige oder käufliche Komponenten, die exakt kalkulierbar sind, zurückgegriffen wird,

- das Produkt letztlich aus einem weitgehend homogenen Material besteht (Sprache), das für den Entwicklungsgang kaum Strukturen impliziert.

Heute versucht man, systematisch vorzugehen, indem man die Entwicklung in Phasen, Aktivitäten oder ähnlich gliedert, die Schnittstellen dazwischen möglichst präzise definiert und damit eine laufende Fortschrittskontrolle ermöglicht. Die verschiedenen **Vorgehensmodelle** [12] weisen mehr oder minder große Unterschiede auf, besonders bezüglich der Frage, ob das Produkt in einer einzigen großen Entwicklung entsteht ("Life Cycle-Ansatz"), durch eine Probe-Entwicklung vorweggenommen ("Prototyping") oder schrittweise erweitert und modifiziert wird ("evolutionäre Entwicklung"). Je nach der konkreten Situation sind die Vor- und Nachteile zu bewerten, doch gilt in jedem Fall, daß man ein bestimmtes Modell wählen und danach arbeiten sollte. Die nicht getroffene Entscheidung ist ebenso schlecht wie die nicht durchgesetzte.

3.3 Software-Qualitätssicherung

Das Wort "Software-Qualitätssicherung" wird in wenigstens drei verschiedenen Bedeutungen (und allen Mischungen daraus) gebraucht:

- Vor allem in kleineren Organisationen versteht man darunter alle Aktivitäten, die der Qualität zugute kommen, also systematisches Vorgehen, Prüfungen aller Art usw.
- In größeren Organisationen, vor allem solchen mit einer gewissen QS-Tradition in anderen Bereichen, ist die Software-QS eine bestimmte Person oder Stelle, die sich speziell mit den Fragen der Qualitätssicherung befaßt. Im Idealfall wirkt diese Person rein beobachtend, d.h. sie protokolliert quasi den Projektverlauf, die Durchführung der einzelnen Maßnahmen (führt sie aber in der Regel nicht selbst durch), die auftretenden Probleme usw. Diese Informationen dienen der Projektleitung und dem Management als Erfolgsmesser und Entscheidungsgrundlagen
- Gelegentlich wird QS auch als Synonym für Test oder Prüfung gebraucht.

Der beste derzeit verfügbare Begriffsstandard [13], das IEEE Standard Glossary, definiert "Software-Qualitätssicherung" (SQS) wie folgt:

> ***quality assurance***. A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical *requirements*

Folgt man dieser Definition, so bietet sich eine Gliederung in **konstruktive**, **analytische** und **organisatorische** Maßnahmen an. Wenn dieser Abschnitt nur auf die analytische SQS eingeht, so soll dies keine Verengung des Begriffs bedeuten, denn die beiden anderen Aspekte sind durch die Abschnitte 2 und 3.2 bereits abgedeckt.

Das Vertrauen in Software kann durch Ausprobieren, Analysieren oder Beweisen untermauert werden. Der erste Ansatz führt zum **Test**, der zweite zum **Review**, der dritte zur **Verifikation**.

Aufgrund der nicht-kontinuierlichen Natur der Software ist es selbst bei sehr einfachen Programmen nicht möglich, durch Testen zu Korrektheitsaussagen zu kommen. Immerhin kann uns ein Test (wenn wir Glück haben) zeigen, daß die Software in bestimmten Situationen nicht korrekt arbeitet. Die Wahrscheinlichkeit, solche Defekte zu entdecken, kann durch systematische Testdatenauswahl erhöht werden; da aber die einleitende Bemerkung über Korrektheitsaussagen in jedem Falle gilt, sollte man den Ausdruck "Austesten" lieber vermeiden, denn er suggeriert eine Vollständigkeit, die es nicht gibt ("Antesten" wäre zutreffend).

Reviews sind nach allen Erfahrungen außerordentlich zweckmäßig. Sie erzeugen zwar hohen Aufwand, sind aber statistisch relativ wirksamer als Tests und lassen sich auf allen Ebenen, also nicht nur für den Programmcode, sondern auch für Spezifikationen, Entwürfe, Handbücher usw. durchführen. Schließlich wirkt ein Review immer auch als Schulungsmaßnahme für alle Beteiligten (siehe unten). Die geringe Verbreitung dieser Technik ist ein Zeichen von Ignoranz (oder ein Hinweis auf die weit verbreitete Haltung "lieber heute bequem als morgen gesund").

Formale Verifikation [14] galt lange Zeit als die Wunderwaffe der Zukunft. Es scheint, daß wir auf eine handhabbare Form dieser Technik "ad calendas graecas" warten müssen. Offenbar wurden zwei Probleme unterschätzt:

- Formale Spezifikation, die Voraussetzung der Verifikation, ist außerordentlich schwierig, selbst für Fachleute. Das gleiche gilt für die eigentliche Verifikation. Ausgewachsene Applikationen wurden noch nirgends verifiziert.
- Formale Spezifikationen sind oft nicht leichter lesbar als Programme, sondern schwerer. Daher gibt es keinen Grund, Ihnen größeres Vertrauen entgegenzubringen.

Wir müssen daher befürchten, daß die Verifikation auch in Zukunft nur in bestimmten Teilbereichen eine Rolle spielen wird, also bei relativ kleinen Bibliotheksfunktionen und bei Software, die überschaubar ist und in extrem sensiblen Bereichen, z.B. Kernkraftwerken, eingesetzt wird.

Mit der SQS verfolgen wir primär das in der Definition genannte Ziel, das Vertrauen in die Korrektheit der Software zu stärken. Ein sekundäres, in der Wirkung oft ebensowichtiges Ziel ist die mittel- und langfristige Verbesserung des Entwicklungsprozesses. In dieser Hinsicht ist der Test von den oben genannten am wenigsten wirksam.

3.4 Datensammlung

Wer eine neue, reiche Goldmine entdeckt hat, kann mit geringem Aufwand und Know-How sehr erfolgreich schürfen. Nach einiger Zeit sind jedoch die größten Nuggets eingesammelt, und der weitere Abbau erfordert verfeinerte Techniken.

Die Software-Technik durchläuft eine ähnliche Entwicklung; wir haben inzwischen fast alles geschafft, was sich in "brute force"-Ansätzen machen läßt. Nun ist es an der Zeit, die Verfahren zu verfeinern. Dieser Schritt ist in allen Wissenschaften durch den Übergang von qualitativen zu quantitativen Aussagen gekennzeichnet. In der Software-Technik fehlen uns dazu aber heute die Daten. Um also einen Schritt voranzukommen, müssen wir in großem Maßstab Daten erheben. Das gilt auf der Ebene einzelner Firmen wie für die Zunft im ganzen.

Dabei geht es - wenigstens zunächst - keineswegs um irgendwelche exotischen, schwer zugänglichen Kennzahlen. Vielmehr wäre es heute schon hochinteressant, verläßliche Angaben über Programm- und Dokumentenlängen, Fehlerzahlen, Aufwand zur Entwicklung und Wartung usw. zu haben [15] An vielen Stellen der Industrie wurden die notwendigen, organisatorisch sehr aufwendigen Schritte eingeleitet (z.B. [16], [17]). Das Buch von Cont, Dunsmore, Shen [18] gibt einen sehr guten Überblick der einschlägigen Metriken.

In der Abteilung Software Engineering der Universität Stuttgart haben wir ein Projekt begonnen, in dem solche quantitativen Zusammenhänge systematisch untersucht und zum Aufbau eines Projekt-Simulators herangezogen werden [12].

4 Elemente einer Software-Kultur

In den traditionsreichen Ingenieurdisziplinen wie Maschinenbau und Elektrotechnik sind im Laufe der Entwicklung Normen entstanden, die heute nicht mehr in Frage gestellt, sondern ohne Diskussion beachtet werden. Typische Beispiele dafür sind Notationen (technische Zeichnungen), fundamentale Sicherheitsregeln (man tritt nicht unter eine schwebende Last, man trennt Geräte vor dem Öffnen vom Netz) und Konstruktionsregeln (Sicherheitsfaktoren, Sollbruchstellen, Erdung metallischer Gehäuse). Vor allem durch die handwerklichen Traditionen der Schlosser, Elektriker usw. ist eine allgemeine Übereinkunft gewachsen, was als gute Qualität anzusehen ist. Alle diese Aspekte bilden zusammen das, was man als Industrie-Kultur bezeichnen kann.

In der kurzen Zeit seit Beginn der Informatik konnte sich noch keine entsprechende Software-Kultur entwickeln. Daher fehlt es uns an einheitlichen und weit verbreiteten Notationen, anerkannten Konstruktionsregeln und stillschweigend beachteten Qualitätsstandards. Immerhin sind in den letzten Jahren Bücher erschienen (z.B. [19], [20]), die den heutigen Stand des Software Engineerings in befriedigender Weise zusammenfassen. (Sie können natürlich nicht besser sein als dieser Stand.) Auch gibt es erfolgreiche Versuche, die positiven Aspekte der Werkstatt auf die Software-Entwicklung zu übertragen [21].

Es ist nicht möglich, ein *Gebiet* wie ein Bauteil künstlich zu altern. Wir können nur versuchen, Beiträge zur Software-Kultur zu leisten und Standards zu *setzen*, wo sie noch nicht gewachsen sind. In diesem Prozeß kommt der Software-Qualitätssicherung eine Schlüsselrolle zu.

Eine Voraussetzung für die Emanzipation unserer Disziplin ist allerdings die Überwindung des noch weit verbreiteten Wunderglaubens: Wer darauf wartet, daß ein neuer Ansatz (z.B. Artificial Intelligence), eine Programmiersprache oder ein Werkzeug den Durchbruch bringen werden, drückt sich in Wahrheit davor, die mühsame Arbeit der kleinen Verbesserungen auf sich zu nehmen [22].

5 Quellen- und Literaturverzeichnis

[1] Ludewig, J. : Software Engineering: Computer-Programme als technische Produkte. **TECHNISCHE RUNDSCHAU**, Heft 7, 1987, 50-57.

[2] Zelkowitz, M.V., R.T. Yeh, R.G. Hamlet, J.D. Gannon, V.R. Basili: Software Engineering practices in the US and Japan. **IEEE COMPUTER**, June 1984, 57-66.

[3] Weinberg, G.M.: **The Psychology of Computer Programming**. Van Nostrand Reinhold, New York, 1971.

[4] Royce, W.W.: Managing the development of large software systems. **IEEE WESCON**, August 1970, pp.1-9. Nachgedruckt in **Proc. of the 9th ICSE** (IEEE), pp.328-338.

[5] Frühauf, K., J. Ludewig, H. Sandmayr: **Software-Projektmanagement und -Qualitätssicherung**. vdf, Zürich, und Teubner, Stuttgart, 1988.

[6] Parnas, D.L.: On the criteria to be used in decomposing systems into modules. **Communications of the ACM, 15** (1972), 1053-1058.

[7] Ludewig, J.: **Sprachen für die Programmierung**. BI-Hochschultaschenbuch Nr. 622, Bibliographisches Institut, Mannheim, 1985.

[8] Ludewig, J.: Languages, methods, and tools for software specification. in J. Zalewski, W. Ehrenberger: **Hardware and Software for Real Time Process Control**. North Holland, Amsterdam etc., 1989, pp.225-256.

[9] Teichroew, D., E.A. Hershey III: PSL/PSA: a computer aided technique for structured documentation and analysis of information processing systems. **IEEE Trans. Software Eng., SE-3** (1977), 41-48.

[10] Ludewig, J.: CASE - eine kritische Übersicht. in H.J. Scheibl: **Software-Entwicklungssysteme und -werkzeuge**. Tagung der Technischen Akademie Esslingen, September 1989, 1.4-1 bis 1.4-10.

[11] Metzger, P.W.: **Managing a programming project**. 2nd ed., Prentice Hall, Englewood Cliffs, New Jersey, 1981.

[12] Ludewig, J.: Modelle der Software-Entwicklung: Abbilder oder Vorbilder? Antrittsvorlesung an der Universität Stuttgart, 28.6.1989. Abdruck in **Softwaretechnik-Trends**, Oktober 1989.

[13] Standard glossary of software engineering terminology. **IEEE Std 729-1983**. Zusammen mit zehn weiteren Normen enthalten im Buch **Software Engineering Standards**. IEEE, New York.

[14] Gries, D.: **The science of programming**. Springer-Verlag, New York usw., 4th print, 1987.

[15] Boehm, B.W. (1973): Software and its impact: a quantitative assessment. **DATAMATION, 19**, 5, 48-59.

[16] Grady, R.B., D.L. Caswell: **Software Metrics: Establishing a company-wide program**. Prentice Hall, 1987.

[17] Möller, K.H.: Steigerung der Softwarequalität durch Zielvereinbarungen und Restfehlerprognosen. in K. Tomica: **Software-Qualitätssicherung**. SAQ, Postfach 2613, 3001 Bern, 1987, pp. 131-143.

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

[19] Fairley, R.E.: **Software Engineering Concepts**. Mac Graw Hill, New York usw., 1985

[20] Sommerville, I.: **Software Engineering**. Addison-Wesley, Wokingham, England, usw., 2nd edition, 1985

[21] Matsumoto, Y., et al.: SWB System: a software factory. in Hünke: **Software Engineering environments**. North Holland Publishing Company, Amsterdam, New York, Oxford, 1981, pp.305-317.

[22] Brooks, F.P., Jr.: No silver bullet - essence and accidents of software engineering. **IEEE COMPUTER 20** (1987), 4, 10-19.