SAQ-Fachtagung
Software-Qualitätssicherung 1987
Hotel Zürich in Zürich vom 3. April 1987

Software und Qualitätssicherung – Versuch einer Annäherung

Jochen Ludewig
Institut für Informatik, ETH Zürich, Zürich

Zusammenfassung

Ausgehend vom traditionellen Gegensatz zwischen den Gewohnheiten der Programmierer und den Ansprüchen der Qualitätssicherung wird die Programm-Produktion der Herstellung anderer Produkte gegenübergestellt. Aus den Übereinstimmungen und Besonderheiten folgen die Wege, die es für die Software-Qualitätssicherung gibt.
Ein Blick auf die wichtigsten Lehrbücher und eine Bemerkung zu den tieferen Gründen der Software-Qualitätsprobleme schliessen den Vortrag ab.

Gliederung

How did a nice disciplin like Quality get mixed up with Computer Science ?
Überschrift 1.1 in Dunn, Ullman, 1982

1. Einleitung

Das Konzept eines Fusionsreaktors sieht heute so aus, daß in einem starken Magnetfeld, dessen Spulen bis zur Supraleitung abgekühlt sind, ein Plasma erzeugt wird. Nur wenn die unvorstellbar tiefe Temperatur der Supraleitung und die unvorstellbar hohe des Plasmas gleichzeitig eingestellt werden können, liefert die Anordnung (vielleicht) Energie.

Dieses Bild kommt mir in den Sinn, wenn ich über das Verhältnis von Qualitätssicherung und Software nachdenke.
Software-Leute halten sich vielfach für Künstler, die hohe Kreativität mit beispielhaftem Einsatz verbinden und auch dann noch ein Programm fertigbringen, wenn die Termine, die Hardware und die Vorgesetzten dies kaum zulassen. QS-Leute, die nichts von Programmen verstehen und versuchen, ihre gußeisernen Begriffe über die filigranen Kunstwerke des Verstandes zu stülpen, wirken da als pedantische Ignoranten, sie sind nutzlos, ja, im Wege.
Die QS-Leute dagegen schätzen sich als die Bewahrer der Qualität ein und damit als die wirklichen, wenn auch verkannten Schutzengel des Unternehmens, die ungeliebt ihre Pflicht tun. Software-Freaks rufen bei ihnen Kopfschütteln hervor: Das sind doch undisziplinierte Kinder, die aus ihren schlechten Erfahrungen nichts lernen und die immer so furchtbar kreativ sein müssen, weil ihre Planung höchstens bis zum Abend desselben Tages reicht.
Können wir beide Gruppen so zusammenbringen, daß daraus Synergie entsteht ?

2. Die Begriffe "Software" und "Qualität"

Software entstand als Kunstwort in Anlehnung an das Wort *Hardware* (Eisenwaren). In IEEE Std 729-1983 ist der Begriff so definiert (alle Zitate aus Normen ohne die Querverweise):

> **software**: (1) Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system.
>
> (2) programs, procedures, rules, and any associated documentation pertaining to the operation of a computer system. (ISO)

Wir halten uns an die zweite, weitere Bedeutung. Damit sind alle Informationen eingeschlossen, die zum Rechner-Programm in Beziehung stehen, auch wenn sie informal oder chiffriert sind. Einzige Bedingung ist, daß sie in permanenter Form vorliegen: Gedanken oder Gespräche, die in keiner Form aufgezeichnet sind, zählen wir nicht zur Software.

Das Wort Qualität wird - wie viele andere Wörter, z.B. Alter, Höhe - in zwei verschiedenen Färbungen verwendet: Der ursprüngliche *beschreibende* Sinn liegt in der Herkunft (*lat.* Beschaffenheit). So sprechen wir von ungenügender Qualität (geringem Alter, großer Höhe). Im Laufe der Zeit hat sich der *wertende* Gebrauch in den Vordergrund geschoben: "die Qualitäten demonstrieren" heißt nicht einfach "die Beschaffenheit zeigen", sondern impliziert, daß es sich um hohe Qualität handelt (vgl.: "Im Alter bekommt man in der Höhe schlecht Luft.").
Beide Auslegungen stehen heute nebeneinander, und es wäre weltfremd, eine davon ausschließen zu wollen. (Entsprechend wirkt "Unqualität" nur komisch, so wie "Unalter" und "Unhöhe" auch.)

DIN 55 350, Teil 11 (Begriffe der Qualitätssicherung, Grundbegriffe) hält sich an die ursprüngliche Bedeutung:

> **Qualität:** Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse beziehen.

Es folgen in der Norm fünf Anmerkungen, von denen hier zwei besonders wichtig sind:

> *Anmerkung 2:* Ein Produkt ist z.B. jede Art von Waren, Rohstoffen, aber auch der Inhalt von Konzepten und Entwürfen. Eine Tätigkeit ist z.B. jede Art von Dienstleistung, aber auch ein maschineller Arbeitsablauf wie ein Verfahren oder ein Prozeß.
>
> *Anmerkung 4:* Die Qualität wird durch die Planungsqualitäten und Ausführungsqualitäten in allen Phasen des Qualitätskreises bestimmt.

Die amerikanischen Normen (ANSI/ASQC A3-1978, IEEE Std 729-1983) sind (ähnlich der DIN-Definition von "Gebrauchstauglichkeit") stärker an der wertenden Interpretation des Wortes orientiert:

> **quality:** The totality of features and characteristics of a product or service that bears on its ability to satisfy given needs.
>
> **software quality:** (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, conform to specifications.
> (2) The degree to which software possesses a desired combination of attributes.

(3) The degree to which a customer or user perceives that software meets his or her composite expectations.

(4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer.

Wir werden das Wort "Qualität" im Sinne der deutschen Norm, also beschreibend, verwenden, auch wenn der Ausdruck "Qualitätssicherung" sich offenbar auf die wertende Bedeutung bezieht.

3. Die Entwicklung der Qualitätssicherung in der Industrie

Solange die Waren in kleinen, überschaubaren und streng hierarchisch organisierten Betrieben hergestellt wurden, war die Qualitätssicherung kein Thema. Jeder Meister **verkörperte** die Qualitätssicherung. Wo er versagte, wurde er durch die Einbindung in Zünfte und Gilden in die Pflicht genommen. Strenge Prüfungsrituale und der unmittelbare Kontakt zu den Kunden machten solches Versagen sehr selten.

Die Idee der QS entstand, als im 19. Jahrhundert die Produktion zergliedert und der Hersteller damit anonym wurde. Für Firmen, die komplexe Waren (wie z.B. Autos) in großer Stückzahl auf den Markt brachten, waren die Folgekosten durch mangelhafte Komponenten und fehlerhafte Montage nicht mehr akzeptabel. Gleichzeitig begann man auch, Bauteile mit definierten Eigenschaften einzukaufen (beispielsweise Schrauben) und mußte sicherstellen, daß diese wirklich verwendbar waren. Die QS schloß also die Lücke, die durch den Abbau persönlicher Beziehungen zwischen dem Unternehmer, seinen Arbeitern und seinen Lieferanten entstanden warihr Einsatzgebiet ist die Massenproduktion. Wo es nicht sinnvoll ist, *alle* Exemplare zu kontrollieren, werden dabei statistische Mittel angewandt (Stichproben). Diese Art der QS zielt darauf ab, nur solche Waren auszuliefern - und nach Möglichkeit: zu produzieren -, die sich von einem wohldefinierten idealen Produkt nur innerhalb gewisser Toleranzen unterscheiden. Höchstes Ziel ist die fehlerfreie Reproduktion.

Anders als die Produktion hatte die Entwicklung ihr handwerkliches Gepräge bis vor wenigen Jahren behalten. Nachdem aber dort die gleichen sozialen Veränderungen vollzogen waren wie hundert Jahre zuvor bei der Produktion, wird die QS auf diesen Bereich ausgedehnt. Die Tatsache, daß bei einer stark automatisierten Fertigung jede Konstruktionsänderung hohe Kosten verursacht - man denke an die Umstellungen in einer Fließband-Produktion - war ein zweiter wichtiger Grund.

Dieser neue Typ von QS ist nicht mehr mit der **Reproduktion**, sondern mit der **Schöpfung** eines Produkts befaßt. Infolgedessen sind die statistischen Mittel nicht zu gebrauchen. Stattdessen geht es darum, die Konsistenz verschiedener Informationen zu überprüfen, beispielsweise die Einhaltung von Regeln und Normen oder die Widerspruchsfreiheit verschiedener Dokumente. Allgemeine Qualitätsziele wie möglichst große Zeitintervalle zwischen Ausfällen (MTBF = Mean Time Between Failure) gehören ebenfalls in diese Kategorie.

4. Der Ruf nach Qualitätssicherung in der Programm-Industrie

Die Entwicklung von Computer-Programmen hat keine Tradition. Sie ist erst vor etwa 40 Jahren aufgekommen, als programmierbare Rechner entstanden. Die Programmierung erschien zunächst als eine Fortsetzung der Elektronik mit neuen Mitteln. Die Hardware setzte den Möglichkeiten enge Grenzen, und ein Programm war umso besser, je kunstvoller und trickreicher es die Technik ausnutzte.

Im selben Maße, in dem die Leistungen der Elektronik wuchsen, wichen die Grenzen der Programmierung zurück. In den sechziger Jahren waren der Speicherplatz und die Geschwindigkeit groß genug, um sehr komplexe Programme zuzulassen. Höhere Programmiersprachen (FORTRAN, ALGOL-60, COBOL, später PL/I) erlaubten es zudem, von den Befehlen eines speziellen Rechners zu abstrahieren und sich vor allem dem zu lösenden Problem zu widmen. Dabei zeigte sich sehr bald, daß sich die Erfahrungen mit kleineren, aus heutiger Sicht sehr kleinen Programmen nicht auf große Programme übertragen ließen. Je weniger die Hardware dem Programmierer Beschränkungen aufbürdete, umso stärker wurde eine andere Beschränkung wirksam, die sich **nicht** rein technisch beseitigen läßt: Die Fähigkeit eines einzelnen Menschen oder einer Gruppe, ein komplexes System wirklich zu verstehen, ist eng begrenzt.

So scheiterten vor etwa zwanzig Jahren (und bis heute) einige große Software-Projekte, die übrigen wurden in der Regel nur mit erheblichen Termin - und Kostenüberschreitungen beendet, und die Leistungen blieben hinter den Erwartungen zurück. Es entstand das Schlagwort von der **"Software Crisis"**.

Da die Programmierer nicht die ersten Menschen waren, die komplexe Systeme bauten (auch wenn sie dies vielfach bis heute glauben), war es naheliegend, sich am Vorbild älterer Ingenieur-Disziplinen zu orientieren. Mit dem Wort **"Software-Engineering"** wurde eine neue Disziplin **postuliert**, die die Parallelen nutzbar machen sollte. Eine der zentralen

Bestrebungen darin war und ist es, den Programmierer vom Bild des Künstlers zu lösen und ihn stattdessen der Gruppe der Ingenieure zuzuordnen (Ludewig, 1987). Es wuchs also, völlig unabhängig von der traditionellen QS, der Wille, die Programmentwicklung klar zu organisieren und zu planen und die Zwischen- und Endergebnisse zu prüfen.

Mit dem Anwachsen der EDV-Abteilungen (und ihrer Kosten) rückte die Programmierung schließlich notwendigerweise auch in das Blickfeld derer, die sich hauptberuflich mit QS befassen. Damit bahnt sich eine notwendig komplizierte Partnerschaft an: Während eine Seite traditionsreich und in der industriellen Massenproduktion verankert ist, stellt die andere gern ihre jugendliche Unbekümmertheit zur Schau und pocht darauf, daß Software nun einmal ganz anders als alles bisher produzierte ist.

5. Besonderheiten der Software und der Software-Qualitätssicherung

Der Unterschied zwischen Software und anderen Produkten ist weniger scharf als vielfach behauptet wird; viele Software-Eigenschaften finden sich auch in anderen Produkten: Vor allem hochintegrierte Schaltungen haben bezüglich Prüfbarkeit und fehlender Stetigkeit völlig gleiche Eigenschaften. Die in diesem Abschnitt aufgezählten Merkmale gelten also nicht exklusiv für Software, sind aber charakteristisch und gemeinsam nur bei Software zu finden.

Selbst wenn ein Programm in hohen Stückzahlen verkauft wird, spielt die Fertigungskontrolle **keine** wesentliche Rolle: Die Duplizierung von Software ist extrem einfach und - im Vergleich zu den Entwicklungskosten - billig. Damit kann der gesamte Bereich der Statistik ausgeblendet werden: Herstellung von Software ist reine Entwicklungsarbeit! Parallelen lassen sich also nur zur QS in Entwicklungsprojekten ziehen.

Erhebliche Unterschiede zu anderen Produkten ist im immateriellen Charakter der Software begründet. Eine Programm-Kopie ist vom Original nicht zu unterscheiden. Darum entstehen im Zuge einer größeren Entwicklung aus praktischen Gründen Kopien, die anschließend als Originale weiterleben: Unbemerkt ist die Konsistenz verloren gegangen. Zu einem späteren Zeitpunkt ist nur noch festzustellen, daß die Teile nicht zueinanderpassen.

Im Gegensatz zu vielen (nicht allen) anderen Produkten ist Software nicht prüfbar. Zwar läßt sich ein Programm testen, doch wird dabei selbst bei trivialen Algorithmen nur ein verschwindender Bruchteil der möglichen Fälle geprüft. Da die Funktion eines Programms nicht stetig auf Änderungen reagiert, lassen sich Ergebnisse im allgemeinen nicht inter- oder

extrapolieren. Schon eine geringe Änderung an Programm oder Daten (ein einziges Bit) kann zu einer beliebig großen (und u.U. katastrophalen) Änderung des Verhaltens führen. Reagiert das Programm also beispielsweise bei den Werten 999 und 1001 fehlerfrei, so kann es dennoch mit 1000 "abstürzen".

Software unterliegt keiner Abnutzung. Es ist daher sinnlos, Maße anzuwenden, die für die Beurteilung von Verschleißerscheinungen gedacht waren. Beispielsweise kann man zwar feststellen, wie oft ein bestimmter Programmfehler auftritt, doch läßt sich daraus kein MTBF ableiten, denn der Fehler ist ja permanent, er zeigt sich nur nicht dauernd, sondern in Abhängigkeit von den Daten (ebensogut könnte man den MTBF einer vielspurigen Autobahnbrücke definieren, bei der eine Spur über dem Fluß endet, indem man die herabstürzenden Autos zählt).

Da sich Software nicht abnutzt, gibt es auch keine Wartung im ursprünglichen Sinne. Software-Wartung ist keine Wiederherstellung des ursprünglichen Zustands, sondern stets eine (partielle) Neuschöpfung. Damit sind alle QS-Probleme der Entwicklung erneut zu lösen.
Softwarefehler kündigen sich nicht an, sondern treten sprungartig auf. Es kann darum auch keine präventive Wartung geben.

Schließlich macht auch das Selbstverständnis der Programmierer Software zu einem sehr speziellen Produkt. Man frage einen Elektroniker und einen Programmierer, ob sie nicht eine relativ simple Änderung vornehmen können, die ihr Werk erheblich verbesserte.
Der Elektroniker wird mit großer Wahrscheinlichkeit zugeben, daß die Änderung an sich einfach sei, doch müsse dazu das Gehäuse geöffnet und das Gerät teilweise zerlegt werden. Nach der Änderung sei ein neuer Abgleich fällig, auch sei die Schaltung heikel, beim Löten könne leicht etwas schief gehen. Kurz: Es lohne nicht.
Die meisten Programmierer beeilen sich in dieser Situation zu bestätigen, daß die Sache wirklich einfach sei (wenn sie das Programm selbst geschrieben haben). Daß sie dabei mindestens ebensoviele Probleme haben wie der Elektroniker, wird ihnen nur kurz klar, wenn sie nach einigen Stunden noch immer nach einem trivialen Fehler suchen, den sie bei der Änderung in das Programm getragen haben. Im Durchschnitt halten Programmierer das Programmieren für einfacher, als es tatsächlich ist (oder sich für intelligenter, als sie wirklich sind).

6. Stand der Programm-Technik

Natürlich ist es schwierig, den Stand der Programm-Technik zu fassen. Maßstab ist hier, was jedes Unternehmen mit Know-How auf dem Gebiet des Software Engineering tun oder einsetzen könnte. Zelkowitz et al. (1984) haben die Situation in der Praxis beschrieben; einen ähnlichen, wenn auch weit weniger anspruchsvollen Überblick versuchen wir zur Zeit in der Region Zürich zu gewinnen (Dünki, Galasso, 1987).

6.1 Life Cycle und Software Management

6.1.1 Life Cycle Model und Phasenplan

Jeder Herstellungsprozeß eines neuen Produkts läuft prinzipiell nach dem gleichen Schema ab: Auf die Klärung der Anforderungen *(Analysieren, Spezifizieren)* folgt die Wahl der Lösungsstruktur *(Entwerfen)*. Dann werden die Komponenten realisiert *(Codieren)*, einzeln geprüft *(Testen)* und zusammengesetzt *(Integrieren)*. Das Gesamtsystem wird geprüft *(Testen des Systems)*, am Bestimmungsort aufgestellt oder eingebaut *(Installieren)* und eingesetzt *(Betreiben)*. Dabei werden im Laufe der Zeit Korrekturen und Änderungen notwendig *(Warten)*. Die in Schrägschrift zugefügten Tätigkeiten bilden zusammen den *Software Life Cycle*. Mit der Entdeckung dieser Gliederung auch für die Programmentwicklung wurde der Grundstein des (praxisorientierten) Software Engineerings gelegt (Boehm, 1976). Im Zusammenhang sieht das so aus:

Analysieren
Spezifizieren — *Software Life Cycle*
Entwerfen
Codieren
Testen
Integrieren
Testen des Systems
Installieren
Betreiben und Warten

In dieser Variante des Life Cycle Models wurden bewußt Verben verwendet, um zu betonen, daß es sich um Tätigkeiten handelt.
Nur im Idealfall genügt ein einziger Durchlauf, um ein Programm zu entwickeln, doch kommt dieser Fall in der Praxis aus verschiedenen Gründen nicht vor (Swartout, Balzer, 1982; Boehm, 1983). Die einzelnen Tätigkeiten sind tatsächlich vermischt und müssen wiederholt werden, wenn man bei logisch nachfolgenden Aktivitäten auf Schwierigkeiten stößt und "zurückbuchstabieren" oder ergänzen muß. Darum werden die Tätigkeiten in der graphischen Darstellung meist durch Pfeile in *beiden* Richtungen verbunden.

Aus der Sicht des Managements ist ein solches Bild wenig hilfreich. Solange man jederzeit wieder von rechts unten nach links oben zurückfallen kann, nützt das Modell der Projekt-Kontrolle nicht. Darum ist aus dieser Perspektive eine vereinfachende Betrachtungsweise vorteilhaft, bei der nicht Tätigkeiten, sondern zeitlich streng geordnete *Phasen* unterschieden sind. Offensichtlich können die Tätigkeiten wegen der Notwendigkeit von Wiederholungen nicht 1 : 1 auf die Phasen abgebildet werden; so dient beispielsweise die Entwurfsphase nur überwiegend, aber nicht ausschließlich der Tätigkeit Entwerfen. Dafür ist es möglich, *Meilensteine* zu definieren, die eine Kontrolle des Projektfortschritts gestatten.
Gewöhnlich wird mit dem Meilenstein ein Dokument - vorläufig - abgeschlossen (*Baselining*, Boehm, 1983). Wird später eine Modifikation nötig (z.B. eine Änderung der Spezifikation in der Entwurfsphase), so muß diese wesentlich sorgfältiger analysiert werden als während der Spezifikationsphase, denn das Dokument war abgenommen und hat bereits als Vorgabe weiterer Dokumente gedient.

6.1.2 Software Management

Das Software-Management hat folgende Aufgaben:

- Planung
- Kontrolle
- Rekrutierung und Schulung, Bereitstellung notwendiger Betriebsmittel und Herstellung geeigneter Randbedingungen
- Entscheidung zentraler Fragen vor und in dem Projekt.

Mit dem Phasenplan wurde ein wichtiger Aspekt der Planung schon eingeführt, weitere wichtige Planungsaspekte sind (Metzger, 1973):

◊ Organisation (Aufstellung eines Organigramms)
◊ Tests
◊ Änderungen der Dokumente
◊ Ausbildung (projektbezogen)
◊ Prüfungen und Berichte
◊ Installation und Einsatz
◊ Betriebsmittel und Ergebnisse

Die Kontrolle ist eng an der Planung orientiert, sie soll hier - wie die übrigen Aufgaben des Managements - nicht näher diskutiert werden (vgl. Abschnitt 7).

6.2 Entwicklungstechnologie

Bis heute gibt es keine Verfahren, Sprachen oder Werkzeuge, die alle Tätigkeiten der Programmentwicklung durchgehend unterstützen. Produkte, die mit diesem Anspruch gepriesen werden, halten einer kritischen Prüfung

nicht stand. Für jede einzelne Tätigkeit (d.h. für die Anfertigung jedes einzelnen Dokuments) gibt es aber erfolgreiche Ansätze.

6.2.1 Spezifikation

Die Spezifikation kann in einer natürlichen oder formalen Sprache abgefaßt werden. Da das eine nicht zur notwendigen Klarheit führt, das andere bislang äußerst schwer handhabbar ist, sind Mischformen relativ populär, die einer formalen Syntax nur eine vage Semantik zuordnen (*halbformale Spezifikation*, vgl. Ludewig, Glinz, Matheis, 1985).

6.2.2 Entwurf

Für den Entwurf, also die Wahl der Lösungsstruktur, gibt es heute einige Regeln und auch Regelwerke, beispielsweise das von Jackson (Cameron, 1983). Werkzeuge für Spezifikation und Entwurf sind oft verbunden oder gar nicht unterschieden (*Software Engineering Environments*, vgl. Hünke, 1981). Der Feinentwurf ist als gesonderte Tätigkeit wenigstens dann notwendig, wenn die Codierung in einer niederen Sprache (z.B. Assembler, FORTRAN, COBOL) stattfinden muß. Pseudo-Code und Nassi-Shneiderman-Diagramme sind die typischen Sprachen für diese Arbeit; Werzeuge (NSD-Generatoren, Cross-Referencer) sind dafür verfügbar.

6.2.3 Codierung

Die Codierung wird schon länger als andere Aktivitäten durch ausgezeichnete Sprachen und Mittel (Editoren, Compiler, Laufzeitsysteme, Debugger) unterstützt. Heute sind MODULA-2 und Ada die besten universellen Sprachen, die praktisch einsetzbar sind (vgl. Ludewig, 1985).

6.2.4 Integration und Wartung

Die Integration wird durch Konfigurationswerkzeuge, Versionen- und Variantenverwaltung unterstützt.
Das Warten ist trotz dem hohen Anteil der Wartungskosten (typisch ca. 50 %) an den Gesamtkosten keine spezielle Tätigkeit, sondern eine Mischung aller Tätigkeiten, die bei der Entwicklung auftreten. Entsprechend gibt es dafür keine zusätzlichen Sprachen und Werkzeuge. Viele existierenden Systeme sind allerdings für die Modifikation weniger gut geeignet als für die Neutentwicklung, weil das "Tracing", also die Verfolgung einer Modifikation durch die Dokumente, nur mit erheblichen Einschränkungen funktioniert.

6.3 Möglichkeiten der Prüfung

Die Prüfung von Dokumenten kann grundsätzlich auf zwei Arten erfolgen, nämlich durch *Inspektion* und durch *Test*, also ohne oder mit Rechnerunterstützung. Der Test ist naturgemäß auf solche Dokumente beschränkt, die in einer formalen Sprache abgefaßt sind, das bedeutet praktisch: auf Programme. Inspektionen sind dagegen für jedes Dokument und auf jeder Stufe der Entwicklung geeignet.

6.3.1 Test

Für den Test gibt es heute einiges methodisches Wissen (beispielsweise die Prinzipien der Black- und White-Box-Tests). Dazu gehört auch die Einsicht, daß Testen keinen Korrektheitsbeweis liefert. Testtreiber und Datei-Vergleicher erleichtern die Durchführung, Werkzeuge zur Instrumentierung und Messung ermöglichen die Feststellung, wie weit die Tests "flächendeckend" waren (Messung der Test-Überdeckung).
Myers (1979) hat das Standdardwerk über das Testen geschrieben.

6.3.2 Inspektion

Das Verfahren der Inspektion wurde von allem von Fagan (1986) zu einer höchst wirksamen Methode entwickelt. Das Grundprinzip ist die planvolle Erörterung eines Dokuments, z.B. eines Entwurfs oder Programms, durch eine kleine Gruppe (etwa vier bis fünf Personen). Die Teilnehmer sind so ausgesucht, daß alle wesentlichen Aspekte vertreten sind (beim Entwurf etwa der Entwerfer, ein Codierer, ein unbeteiligter Fachmann). Ein Moderator leitet das Gespräch, die Ergebnisse werden laufend protokolliert. Ein Vorgesetzter sollte nicht beteiligt sein.
Eine etwas andere Form der Inspektion ist unter der Bezeichnung "Walkthrough" bekannt.

7. Möglichkeiten der Software-Qualitätssicherung

Aus den Techniken des Software Managements, der Bearbeitung und Prüfung lassen sich die Möglichkeiten der Software-Qualitätssicherung (SQS) ableiten.

7.1 Software-Qualitätssicherung durch Planung und Organisation

Die Einrichtung einer QS-Organisation, die die Geschäftsleitung in Sachen der Qualitätssicherung vertritt, ist auch im Bereich der Software eine wichtige Voraussetung systematischer QS-Arbeit. In jedem einzelnen Projekt ist die Planung diejenige Aktivität, die auf den Erfolg den stärksten Einfluß hat (Metzger, 1973). Damit ist sie ein Schlüsselfeld der SQS.
Dieser Aspekt wird von K. Bär behandelt (in diesem Tagungsband).

7.2 Software-Qualitätssicherung durch konstruktive Maßnahmen

Bei einer Serienproduktion kann die Qualität im Notfall noch *nach* der Herstellung durch Sortierung beeinflußt werden; Produktivität und Qualität sind damit innerhalb eines gewissen Rahmens substituierbar. Bei der Konstruktion kann die Qualität dagegen keinesfalls nachträglich hinzugefügt werden, sie muß während der ganzen Entstehung hineingewirkt werden wie

der sprichwörtliche rote Faden in die Seile in der britischen Kriegsmarine. Eine scharfe Trennung zwischen Software Engineering und SQS ist nicht möglich, beides sind nur zwei Ansichten derselben Sache. Pointiert könnte man definieren: **SQS ist Software Engineering, wenn man es wirklich tut und kontrolliert und dokumentiert.**
K. Frühauf behandelt die Möglichkeiten der konstruktiven SQS (in diesem Tagungsband).

7.3 Software-Qualitätssicherung durch analytische Maßnahmen

Obwohl sich hohe Qualität nicht durch eine Prüfung erzeugen läßt, ist diese doch aus drei Gründen unverzichtbar:

- Die Wirksamkeit der Organisation, der Planung und der konstruktiven Maßnahmen muß überwacht werden.
- Extrem schlechte Qualität muß, falls sie einmal entsteht, erkannt werden, damit rechtzeitig ein Ausweg gesucht wird.
- Fremdsoftware muß vor der Verwendung geprüft werden.

Die darum notwendige SQS durch analytische Maßnahmen wird von E. Menet behandelt (in diesem Tagungsband).

8. Stand der Literatur

In diesem Abschnitt ist keine umfassende Würdigung der Literatur möglich oder beabsichtigt. Das Ziel ist vielmehr, auf einige wichtige Lehrbücher und Artikel hinzuweisen, die einen leichten Zugang zu dem Gebiet "Software-Engineering und -Qualitätssicherung" gestatten.

Die Literatur im Software Engineering ist noch immer spärlich, wenn man nach Lehrbüchern sucht. Vor allen anderen ist hier das Buch von **Fairley** (1985) zu nennen. **Boehm** (1980) befaßt sich entgegen dem Titel nicht nur mit Kostenschätzung (das bekannte COCOMO-Modell), sondern auch mit Software Engineering im allgemeinen.

Einen sehr guten Ersatz für Lehrbücher bilden aber die zahlreichen **IEEE Tutorials** (siehe IEEE Computer Society, 1987), großformatige Paperbacks, in denen meist die relevanten Artikel zu einem bestimmten Thema gesammelt sind, beispielsweise Programm-Entwurf oder Kostenschätzung. In einem Gebiet, das sich zu schnell entwickelt, um den Autoren Zeit zum Erwerb und zur Zusammenstellung der eigenen und fremden Arbeitsergebnisse zu lassen, bewährt sich diese Art von Literatur ausgezeichnet. Speziell zu erwähnen ist der Band von **Chow** (1985), der eine Zusammenstellung der wichtigsten Veröffentlichungen (1975 bis 1982) zum Thema SQS enthält.

Der Artikel von **Boehm** (1983) gibt einen ausgezeichneten Überblick der Software-Management-Prinzipien.

Conte, Dunsmore und **Shen** (1986) befassen sich ausführlich mit allen Metriken des Software Engineering, auch mit Kosten- und Aufwandsschätzung. Qualitätsmaße werden speziell von **Höcker et al.** (1984) analysiert. **Arthur** (1985) bildet die Qualitäten auf Metriken ab und gibt für einzelne Programmiersprachen (ALC, COBOL, PL/I) spezielle Metriken an.

Die Validation von Software ist im Tagungsband von **Hausen** (1984) unter verschiedenen Gesichtspunkten diskutiert (Inspektion, Test, Symbolische Ausführung, Verifikation). **Beizer** (1984) geht mit seinem dicken Buch vor allem das Thema "Test" an, befaßt sich aber auch mit der SQS.

Willmer entwickelt in ihrer Dissertation ein Tätigkeiten- und ein Produktmodell, mit dem sich die Qualität systematisch beurteilen läßt.

Zum Thema SQS allgemein war das ausgezeichnete Buch von **Dunn** und **Ullman** (1982) nach seinem Erscheinen jahrelang konkurrenzlos. **Cho** (1980) bietet zwar ebenfalls einen Überblick, doch erscheint die Auswahl des Inhalts (Programmiersprachen, QS-Ansätze) nicht sehr modern.

Asam, Drenkard und **Maier** (1986) haben ein praxisorientiertes Buch (aus der SIEMENS-Welt) geschrieben, das die Qualitäten definiert und zu ihrer Beurteilung anleitet. **Sneed** (1983) hat ein kleineres Buch speziell für dkommerzielle Anwendungen geschrieben.

Besondere Hervorhebung verdient schließlich das neue Buch der **DGQ-NTG** (1986). In handlicher Form steht damit erstmals eine "SWQS-Fibel" zur Verfügung. Dabei ist es sicher auch ein Vorteil, daß unter den Autoren beide Seiten, Software und QS, kompetent vertreten waren.

In den letzten Jahren sind auch verschiedene Normen entstanden, die für dieses Thema Bedeutung haben (CSA (1982), IEEE Standards 730-1981, 729-1983, 828-1983). **Meekel** und **Troy** (1984) haben die Standards verglichen.

9. Software-Qualitätsbewußtsein

Wie oben angedeutet wurde, bietet die QS der Software noch eine Reihe schwieriger technischer Aufgaben. Ich bin aber davon überzeugt, daß der Kern des Problems tiefer liegt. Ich gehe dabei von der Vorstellung aus, daß wir uns weit weniger als bewußt von rationalen Gesichtspunkten leiten lassen, daß also hier auch die Software-Psychologie (Shneiderman, 1980) eine wichtige Rolle spielt.

Das Kapitel "Qualitätsmotivierung: Voraussetzungen" im Büchlein von Masing (1973), S. 53 ff., wurde sicher nicht mit Blick auf die Software geschrieben; trotzdem liest es sich wie eine knappe Aufzählung unserer offenen Probleme im Bereich des Software Engineerings. Nachfolgend sind die sechs Punkte zitiert und kommentiert.

1. Zielqualitäten müssen klar und eindeutig formuliert sein.

Die wichtigste Zielqualität unserer Programme ist die Korrektheit. Wir können die Funktionalität bis heute nur mit größter Mühe präzise angeben; in der Praxis spielen die betreffenden formalen Verfahren absolut keine Rollen. Damit ist die Frage, ob ein Programm korrekt arbeitet, in vielen Fällen nicht eindeutig zu beantworten.
Noch wesentlich ungünstiger ist die Situtation bei den vielen anderen Eigenschaften, die für Gebrauch und Wartung der Software zentrale Bedeutung haben, beispielsweise Robustheit oder Lesbarkeit. Versuche, für solche Qualitäten Maße zu definieren, haben bis heute keinen Erfolg erzielt (d.h. keines der vorgeschlagenen Maße ist unumstritten oder allgemein anerkannt).

2. Jeder Mitarbeiter muß in der Lage sein, die gestellten Anforderungen zu erfüllen, d.h. die Zielqualität zu erreichen.

Der Ausbildungsstand in der Informatik ist schlecht. Das gilt sowohl für die "Maurer", also die Programmierer, als auch für die "Poliere", also die Software-Manager. Darüberhinaus sind viele Programmierer gar nicht bereit (und durch ihre Vorgesetzen auch nicht daran gewöhnt), sich Zielvorgaben unterzuordnen.

3. Jeder Mitarbeiter muß in der Lage sein, schlechte von guter Arbeit unterscheiden zu können.

Die Frage, was gute, was schlechte Software ist, läßt sich bis heute nur subjektiv entscheiden. Jeder weiß es, aber kaum zwei Personen stimmen überein. Bücher und Veröffentlichungen mit Programmbeispielen geben

hier ein deutliches Zeugnis. Sind sich aber selbst die Fachleute nicht darüber einig, welche Merkmale gute Software auszeichnen, so ist ein Mitarbeiter natürlich in der Regel damit völlig überfordert.

4. **Jeder Mitarbeiter muß wissen, was er zu tun hat, um schlechte Arbeit zu verhindern.**

   Aufgrund des ungenügenden Ausbildungsstandes weiß dies der Mitarbeiter wahrscheinlich nicht; wo doch, so wird er kaum versuchen, dieses Wissen zu gebrauchen. Denn für ihn persönlich lohnt sich Software-Qualität eben nicht, er hat beispielsweise nur Nachteile, wenn er so dokumentiert, wie es jeder predigt, aber fast niemand praktiziert, denn er wird wesentlich mehr Zeit brauchen, und durch die gute Dokumentation wird gerade sein Modul nachher als besonders einfach erscheinen.

5. **Jeder Mitarbeiter muß wissen, was er zu tun hat, wenn er schlechte Arbeit nicht verhindern konnte.**

   Was soll ein Programmierer angesichts massiver Mängel tun ? Soll er sich durch Mahnungen bei seinen Kollegen unbeliebt machen ? Oder soll er einen Vorgesetzten ansprechen, der keinen anderen Wunsch hat, als den nächsten anstehenden Termin einzuhalten, wie die Software auch aussehen möge ? Der Einzelne hat in einer Umgebung, die die Qualität nur durch Lippenbekenntnisse preist, aber in Wirklichkeit ganz andere Prioritäten setzt, keine Chance.

6. **Jeder Mitarbeiter muß die Konsequenzen schlechter Arbeit für den Betrieb genau kennen.**

   Welche Konsequenz hat schlechte Qualität für den Betrieb ? Auch der Kunde hat ja keine klare Vorstellung davon, was er eigentlich genau hätte bekommen sollen. So treten Mängel nur ausnahmsweise durch spektakuläre Systemabstürze oder Fehlfunktionen zu Tage, im Normalfall hat schlechte Software nur zur Folge, daß ein Wartungsprogrammierer Jahre später ein paar Tage länger flucht, bis eine Änderung durchgeführt ist, oder daß ein Benutzer das System nie wirklich ausnutzen kann, weil er die komplizierte Bedienung nicht begreift.
   Schlechte Qualität wird also höchstens ein vages Gefühl der Abneigung hervorrufen. Nur in den wenigsten Fällen läßt sich der Mangel bis zum Programmierer zurückverfolgen.

Welche Schlüsse können wir aus dieser Konfrontation traditioneller Qualitätssicherung und traditionsloser Software-Produktion ziehen ?

Während in der Produktion materieller Gegenstände (Musikinstrumente, Möbel, Kathedralen, Bücher) ursprünglich eine ausreichende Qualitätssicherung gegeben war, ohne daß diese spezieller Betrachtung bedurfte hätte, gab es bei den Programmen nie etwas ähnliches. Maßnahmen zur Software-QS bilden also nichts Verlorenes nach, sondern sind ganz und gar künstlich, solange sie sich nicht auf ein allgemeines Qualitätsbewußtsein stützen könen. Hier liegt nach meiner Meinung der Ansatzpunkt für grundsätzliche Verbesserungen.

- Zunächst muß geklärt werden, was "gute Software" bedeutet. Betrachtet man verschiedene Lehrbücher der Informatik, so wird deutlich, daß die Meinungen weit auseinander gehen.

- Als nächstes wäre dann gute, beispielhafte Software zu schaffen und zu verbreiten, damit jeder sehen und erleben kann, wie so etwas aussieht.

- Schließlich sollte jeder Programmierer eine Lehrstelle gehabt haben, in der er in die Entwicklung guter Software eingebunden ist. Auf diese Weise hätte er Gelegenheit, den beruflichen Stolz zu entwickeln, an dem wir gute Schreiner, Krankenschwestern oder Buchhalter erkennen.
  Masing (1973), S. 54, sagt hierzu: » Nur wenn durch lange Zusammenarbeit betriebsintern Einigkeit darüber besteht, was eine "fest angezogene Schraube" oder ein "einwandfreier Schweißpunkt" ist, dürfen derartige Anweisungen ergehen. « Da wir in der Software die präzise Definition in absehbarer Zeit nicht erreichen werden, ist die Einigkeit unsere einzige Chance !

- Auch für die Rolle des Meisters wäre eine Entsprechung zu schaffen: Solange es Vorgesetzte in der Programmierung als unter ihrer Würde betrachten, die Ergebnisse ihrer Mitarbeiter genauestens zu prüfen und zu verwerfen oder zu loben, ist eine wesentliche Änderung der Situation nicht zu erwarten. Heute sind es wohl nicht nur der Termindruck und die falsche Einschätzung der Programmierung, die die Manager daran hindern, sondern vielfach auch der Mangel an fachlicher Kompetenz. Der Weg zur Software-QS führt also über eine bessere Qualifikation der Manager.
  Masing (1973), S. 54: » Die Betriebsleitung muß, um von allen Mitarbeitern Qualitätsarbeit fordern zu können, bestimmte Voraussetzungen schaffen. Sind sie nicht vorhanden, bleibt Qualitätsbewußtsein mehr oder minder purer Zufall. «

Bis dieser Konsens erreicht ist, sind alle Maßnahmen zur Software-Qualitätssicherung nichts als eine - bitter notwendige - erste Hilfe zur Linderung der Symptome.

10. Literatur

Arthur, L.J. (1985): **Measuring Programmer Productivity and Software Quality.** John Wiley & Sons, New York usw.

Asam, R.N. Drenkard, H.-H. Maier (1986): **Qualitätsprüfung von Software- Produkten.** Siemens AG, Berlin und München.

Beizer, B. (1984): **Software Testing and Quality Assurance.** Van Nostrand Reinhold Co., New York etc.

Bergland, G.D., R.D. Gordon (ed.) (1981): Tutorial: **Software Design Strategies.** 2nd ed., IEEE Catalog No. EHO 184-2.

Boehm, B.W. (1976): Software Engineering. **IEEE Transactions on Computers, C-25,** 1226-1241.

Boehm, B. W. (1980): **Software Engineering Economics.** Prentice Hall, Englewood Cliffs, N.J.

Boehm, B.W. (1983): Seven basic principles of Software Engineering. **Journal of Systems and Software, 3,** 3-24.

Cameron, J.R. (1983): **JSP & JSD: The Jackson Approach.** IEEE Tutorial, IEEE Comp. Soc. Press, Order No. 516.

Cho, C.-K. (1980): **An introduction to software quality control.** John Wiley & Sons, New York etc.

Chow, T.S. (1985): **Software quality assurance.** IEEE Tutorial, IEEE Comp. Soc. Order No. 569

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

CSA (1982): Standard for software quality assurance program, part I. CSA Preliminary Standard Q396.1 [vgl. IEEE (1981)]

DGQ-NTG (1981): **Software-Qualitätssicherung.** Arbeitsgruppe 143 der Deutschen Gesellschaft für Qualität e.V. und der Nachrichtentechnischen Gesellschaft im VDE, DGQ-NTG-Schrift Nr. 12-51, Beuth, Berlin, und VDE-Verlag, Offenbach.

Dünki, P., C. Galasso (1987): Untersuchung über den Stand des Software Engineerings im Großraum Zürich. Diplomarbeit in der Abt. Informatik der ETH Zürich, März 1987.

Dunn, R., R. Ullman (1982): **Quality Assurance for Computer Software.** McGraw-Hill Book Company, New York usw.

Fagan, M.E. (1986): Advances in Software Inspection. **IEEE Trans. Softw. Eng., SE-12,** 744-751.

Fairley, R. (1985): **Software Engineering Concepts.** McGraw-Hill Book Company, New York usw.

Hausen, H.-L. (ed.) (1984): **Software-Validation.** North Holland, Amsterdam usw.

Höcker, H., W.D. Itzfeldt, M. Schmidt, M. Timm (1984): **Comparative Descriptions of Software Quality Measures.** GMD-Studien Nr. 81, GMD, Postfach 1240, D-5205 St. Augustin 1

Hünke, H. (ed.) (1981): **Software Engineering Environments.** Proc. of the Symposium held at Lahnstein, June 16 - 20, 1980. North Holland Publishing Company, Amsterdam, New York, Oxford.

IEEE (1981): Standard for Software Quality Assurance Plans.
IEEE Std 730-1981 (Revision of ANSI/IEEE Std 730).

IEEE (1983): Standard glossary of software engineering terminology.
IEEE Std 729-1983.

IEEE (1983): **Proc. of the 2nd Workshop on Software Engineering Standards.** (SESAW-II). San Francisco, California, May 1983.

IEEE (1983): Standard for software configuration management plans.
IEEE Std 828-1983.

IEEE Computer Society (1987): Recent & Upcoming Selections from the COMPUTER SOCIETY PRESS. Verfügbar von: Computer Soc. of the IEEE, Av. de la Tanche, 2, B-1160 Bruxelles, Belgien. vgl. Cameron, 1983; Bergland, Gordon, 1981; Chow, 1985; Miller, Howden, 1981; Putnam, 1980

Ludewig, J. (1985): **Sprachen für die Programmierung - eine Übersicht.**
BI-Hochschultaschenbuch Nr. 622, Bibliographisches Institut Mannheim, 194 S.

Ludewig, J., M. Glinz, H. Matheis (1985): Software-Spezifikation durch halbformale, anschauliche Modelle. pp. 193-204 in Hansen (Hrsg.): **GI/OCG/ÖGI-Jahrestagung 1985**, IFB 108, Springer-Verlag, Berlin.

Ludewig, J. (1987): Software Engineering: Computer-Programme als technische Produkte.
TECHNISCHE RUNDSCHAU, Heft 7, 1987, 50-57.

Masing, W. (1973): **Qualitätslehre.**
Deutsche Gesellschaft für Qualität e.V., Berlin und Frankfurt a.M., 2. Auflage.

Meekel, J., R. Troy (1984): Comparative study of standards for software quality assurance plan. **3rd Software Engineering Standards Application Workshop**, (SESAW-III), San Francisco, California, October 1984, 60-67.

Metzger, (1973): **Managing a Programming Project.** Prentice Hall, Inc., New Jersey.

Miller, E., W.E. Howden (ed.) (1981): **Tutorial: Software testing & validation techniques.**
2nd ed., IEEE Catalog No. EHO 180-0.

Myers, G.J. (1979): **The art of software testing.** John Wiley & Sons, New York etc.

Putnam, L.H. (ed.) (1980): **Software Cost Estimating and Life Cycle Control.** IEEE Catalog No.EHO 165-1

Shneiderman, B. (1980): **Software Psychology: Human Factors in Computer and Information Systems.** Winthrop Publishers, Inc., Cambridge, Massachusetts.

Sneed, H. (1983): **Software-Qualitätsicherung für kommerzielle Anwendungssysteme.**
Verlagsgesellschaft R. Müller, Köln.

Swartout, W., R. Balzer (1982): On the inevitable intertwining of specification and implementation.
Commun. ACM, 25, 7, 438-440.

Willmer, H. (1985): **Systematische Software Qualitätssicherung anhand von Qualitäts- und Produktmodellen.** Informatik-FB 97, Springer-Verlag, Berlin usw.

Zelkowitz, M.V., R.T. Yeh, R.G. Hamlet, J.D. Gannon, V.R. Basili (1984):
Software Engineering Practices in the US and Japan.
IEEE COMPUTER, June 1984, 57-66.