Im Zentrum des Aufsatzes steht die grundsätzliche Diskussion der Unterschiede und der Übereinstimmungen zwischen Software und anderen technischen Produkten. Jochen Ludewig zieht daraus praktische Schlüsse für das systematische Softwareengineering und leitet Rahmenbedingungen und Ziele für die Lehre, insbesondere an der ETH Zürich, ab. Der Aufsatz baut auf der am 13. Mai 1986 an der ETHZ gehaltenen Antrittsvorlesung auf.

Softwareengineering

Computerprogramme als technische Produkte

Von Jochen Ludewig

Software ist eine Wortschöpfung, die dem gewachsenen Begriff «Hardware» (ursprünglich: «Eisenwaren») gegenübergestellt wurde. Der neue Begriff war notwendig, weil die Rechner (Computer) nicht nur aus Elektronik und Mechanik (der Hardware) bestehen, sondern auch – zu einem wertmäßig ständig wachsenden Teil – aus Anweisungen und Daten (der Software). Wir fassen den Begriff heute weiter und definieren Software als **die Menge aller dauerhaft gespeicherten Informationen, die ein Programm bilden oder zu seiner Herstellung, Änderung oder Verwendung dienen,** also beispielsweise Planungsunterlagen, Testdaten und Benutzungsanleitungen. Anders als ein *Programm* muß Software nicht logisch abgeschlossen sein: Auch drei Zeilen Code oder das Lebenswerk eines Programmierers sind Software.

Während im Französischen mit «Logiciel» eine schöne Übertragung gelungen ist, fehlt uns bis heute ein deutsches Wort. (Vielleicht kommt «Logation» als Mischung aus «Logik» und «Information» in Frage.)

Engineering ist ebenfalls nicht genau ins Deutsche übersetzbar, am treffendsten ist das Wort «Technik». Im Kontext dieses Aufsatzes ist dabei an traditionsreiche Ingenieurdisziplinen wie Maschinenkonstruktion, Elektrotechnik oder Bauingenieurwesen gedacht.

Prof. Dr. Jochen Ludewig, Institut für Informatik, ETH Zürich.

Softwareengineering

Bis in die frühen sechziger Jahre bestand Programmieren vor allem im Versuch, die aus heutiger Sicht sehr schwachen Leistungen der Hardware, also der verfügbaren Computer, möglichst vollständig zu nutzen. Rechenzeit und Speicherplatz waren sehr viel teurer als *Brainware*, also die Arbeit der Programmierer. Die Hardware gab der Software die Grenzen vor.

Durch die eindrucksvollen Fortschritte der Elektronik wichen diese Grenzen immer weiter zurück, und man wagte sich an die Entwicklung sehr großer Programme. Dabei traten erstmals die Schwierigkeiten auf, die uns bis heute vertraut sind [1, 2]: Termin- und Kostenschätzungen erwiesen sich oft als zu optimistisch, und in einigen spektakulären Fällen mußten Projekte ganz eingestellt werden. (In der Schweiz war IFS, das Integrierte Fernmeldesystem, ein später Repräsentant solcher gescheiterten Vorhaben.) War die Software mit Müh und Not fertiggestellt, so entsprach sie vielfach nicht den Anforderungen. Das Wort von der *Softwarecrisis* entstand.

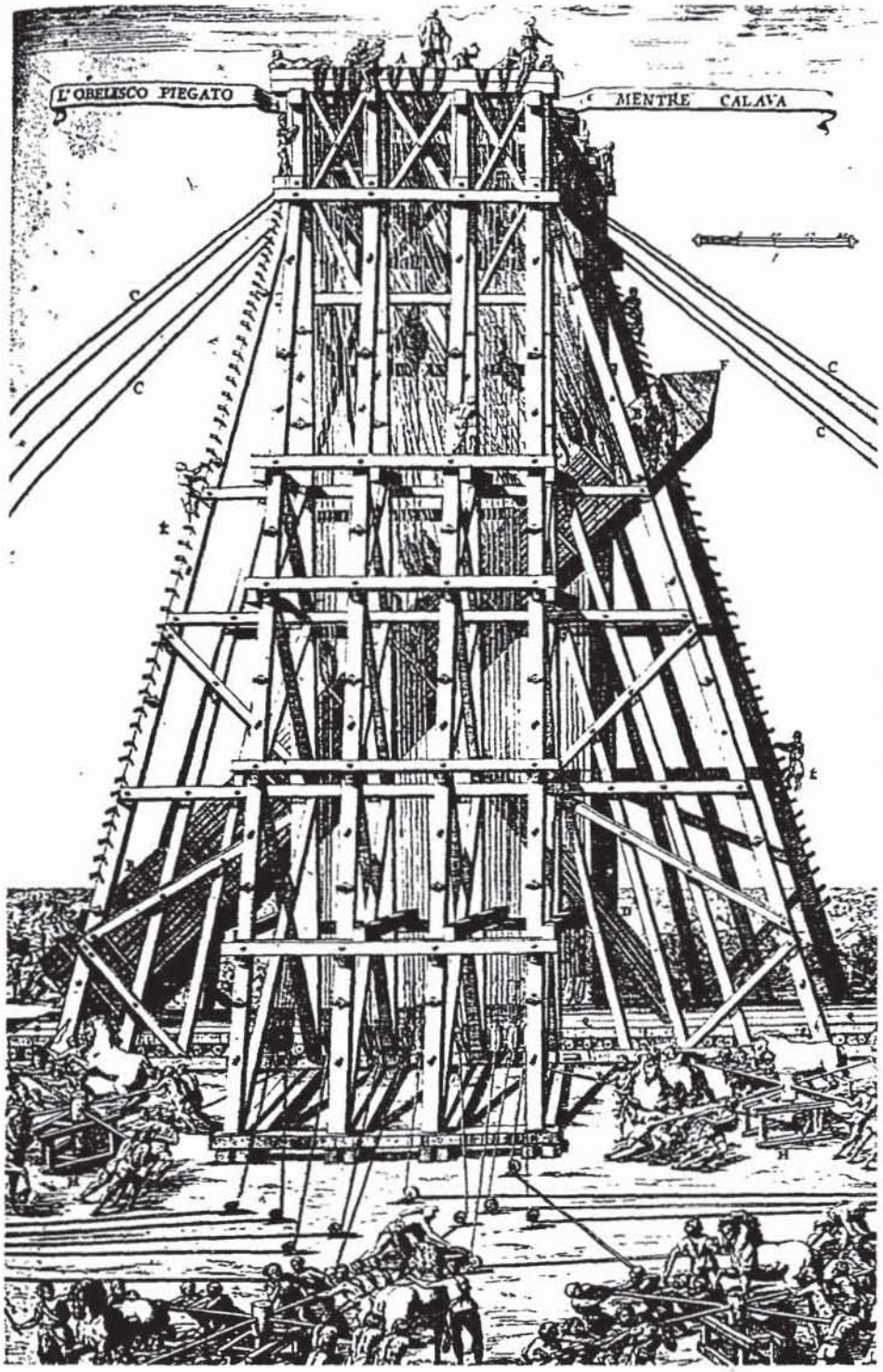
Teilnehmer eines internationalen Workshops in Garmisch, 1968, sahen den Ausweg in der Orientierung an den Ingenieurwissenschaften, und sie brachten das Wort *Softwareengineering* in die Diskussion. Softwareengineering war damit zunächst nichts als eine Idee, ein neuer Denkansatz: Der Programmierer sollte sich nicht mehr als Künstler verstehen, sondern als ein Ingenieur, der nach wohldefinierten Verfahren arbeitet und dessen Ergebnisse nach objektiven Maßstäben beurteilt werden können.

Kann Software als technisches Produkt betrachtet werden?

Im Hinblick auf die Intention, die hinter dem neuen Wort steht, ist es interessant, das technische Produkt mit dem Kunstwerk zu vergleichen. Dabei soll

Tabelle 1. Technisches Produkt kontra Kunstwerk.

	technisches Produkt	Kunstwerk
Termine	in der Regel mit genügender Genauigkeit planbar	Inspiration ist Voraussetzung, daher nicht planbar
Preis	an den Kosten orientiert, damit kalkulierbar	nur durch den Marktwert bestimmt, nicht kalkulierbar
Standards	entspricht technischen Regeln und Normen	durchbricht Regeln, soweit solche überhaupt bestehen
Bewertung	nach objektiven Kriterien prüfbar	nur subjektive Bewertung möglich
Urheber	der Urheber bleibt meist anonym, keine dauerhafte Bindung zum Produkt	der Künstler betrachtet das Kunstwerk als Teil seiner selbst, bleibt damit verbunden



Aufrichten des Obelisken im Vatikan 1586 (Domenico Fontana: «Del transporto del obelisco vaticano». Roma, 1743).
Große Projekte lassen sich nur planmäßig durchführen, wenn ihr Ablauf gut verstanden ist und eine Organisation besteht, die für die zu erwartenden Probleme gerüstet ist. Im Bereich des Bauwesens ist dies wenigstens seit der Renaissance der Fall. Dagegen erscheinen große Softwareprojekte bis heute als Abenteuer.

der Begriff des technischen Produkts weit gefaßt werden, so daß auch handwerkliche Arbeiten eingeschlossen sind. Tabelle 1 stellt einige Charakteristika gegenüber.

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 (etwa 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 daher nicht mehr wartbar, wenn der Urheber nicht mehr zur Verfügung steht.

Wie man sieht, ist der Anspruch des Softwareengineering bis heute nicht eingelöst [3]. Doch die oben genannten Merkmale technischer Produkte spiegeln sich bereits in den Aktivitäten dieses jungen Gebiets: Es gibt seit Mitte der siebziger Jahre eine wachsende Zahl veröffentlichter Arbeiten über Softwaremanagement, -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» [4], 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.

Produktionsablauf und Software-Life-Cycle

Wenn wir versuchen, die Methoden der anderen Techniken auf die Software zu übertragen, so müssen wir zunächst klären, wieweit Software überhaupt mit anderen technischen Produkten vergleichbar ist.

Prinzipiell gibt es zwei Arten von technischen Produkten, nämlich solche, die im Auftrag eines Kunden angefertigt werden (Beispiele: Maßanzug, Kraftwerk), und solche, die in großer Stückzahl auf den Markt gelangen (Konfektionskleidung, Fernsehgerät). Dazwischen gibt es alle möglichen Mischformen (Auto mit Ausstattung nach Kundenwunsch). Bei der Software ist die Situation genau gleich; das Spektrum reicht von der vieltausendfach verkauften Diskette mit einem Spiel für den Homecomputer über parametrisierbare Programmpakete bis zu einem sehr speziellen Programm, das für einen einzigen Forschungssatelliten geschaffen wird.

Die Herstellung eines Einzelstücks durchläuft charakteristische Phasen:

- Die Anforderungen werden festgestellt (Maßnehmen beim Anzug, Feststellung der geforderten Leistung und der Rahmenbedingungen beim Kraftwerk).
- Das Produkt wird entworfen (Festlegung des Schnitts, Konzeption des Kraftwerks).
- Der Entwurf wird bis ins Detail verfeinert (Wahl der Knöpfe und des Futters, Bauplanung).
- Das Produkt wird realisiert (nähen, bauen).
- Das Produkt wird vom Kunden geprüft und übernommen (Anprobe, Abnahmeprüfung).
- Es werden Wartungs- und Anpassungsarbeiten nötig (Knopf annähen und Bundweite ändern, Überholung der Turbine und Umstellung auf anderen Brennstoff)

Wird die Entwicklung von vielen Personen durchgeführt, so sind begleitende Projektleitung und Qualitätssicherung unerläßlich.
Bei Serienprodukten fallen Wartung und Anpassung ganz weg oder werden (außer bei großen Systemen) nicht vom Hersteller durchgeführt. Dafür wird der Produkt*typ* von Serie zu Serie verbessert und veränderten Anforderungen des Marktes angepaßt.
Die Beobachtung, daß auch Software die genannten Entwicklungsphasen durchläuft (oder wenigstens durchlaufen sollte), führte zur «Entdeckung» des *Software Life Cycles* oder *Phasenmodells*. Eine typische Fassung ist die folgende:

Systemanalyse
Spezifikation der Anforderungen
Grobentwurf und Modulspezifikation
Feinentwurf
Codierung und Modultest
Integration und Test
Betrieb mit Korrektur, Anpassung und Erweiterung («Wartung»)

Als das Life Cycle Model zu Beginn der siebziger Jahre aufkam [5], war sein Vorteil vor allem der, daß gewisse Defizite offensichtlich wurden: War das Gebiet der Codierung, also der Realisierung, recht genau durchforscht und durch Werkzeuge (Editoren, Compiler) wirkungsvoll unterstützt, so lagen die früheren Phasen als Forschungsgebiete wie als Einsatzphasen für Methoden und Werkzeuge völlig brach. Das gleiche galt am «rechten Rand», beim Test und besonders bei der Korrektur und Anpassung, der sogenannten Wartung.
So entfaltete sich gegen Mitte der siebziger Jahre eine beträchtliche Aktivität: Es entstanden die ersten Software-Spezifikationssysteme, Entwurfsmethoden wurden propagiert, Konzepte und Sprachen für die Beschreibung der Softwarearchitektur kamen auf, und für den Test wurden Hilfsmittel realisiert [6]. Ein breites, wenn auch noch keineswegs befriedigendes Angebot auf dem Markt der Softwarewerkzeuge ist heute das augenfällige Ergebnis.
Die Anwendung des klassischen Phasenplans zeigte, daß die naive Gliederung der Arbeiten in sequentielle Arbeitspakete unrealistisch ist. Daher wurden Verfeinerungen vorgenommen, vor allem durch die Einführung von Schleifen, die die Rückkehr in eine bereits abgeschlossene Phase erlauben. Auch dieses neue Modell ist inzwischen umstritten, neuere Methoden, beispielsweise das «rapid prototyping», lassen sich so nicht fassen. Trotzdem muß die Einführung des Phasenplans als bisher größter Erfolg des Softwareenginnering betrachtet werden.

Besonderheiten des Produkts Software

Software unterscheidet sich von andern technischen Produkten vor allem

- durch die Eigenschaften des verwendeten Materials

Wer sollte Informatik studieren und wer nicht?

Da ich seit Beginn des Wintersemesters 1985/86 die Einführungsvorlesung Informatik I/II für Informatiker gebe, möchte ich mit einer sehr persönlichen Bemerkung zur Frage schließen, wer zu einem Studium der Informatik ermuntert werden sollte und wer besser nicht.

Voraussetzungen

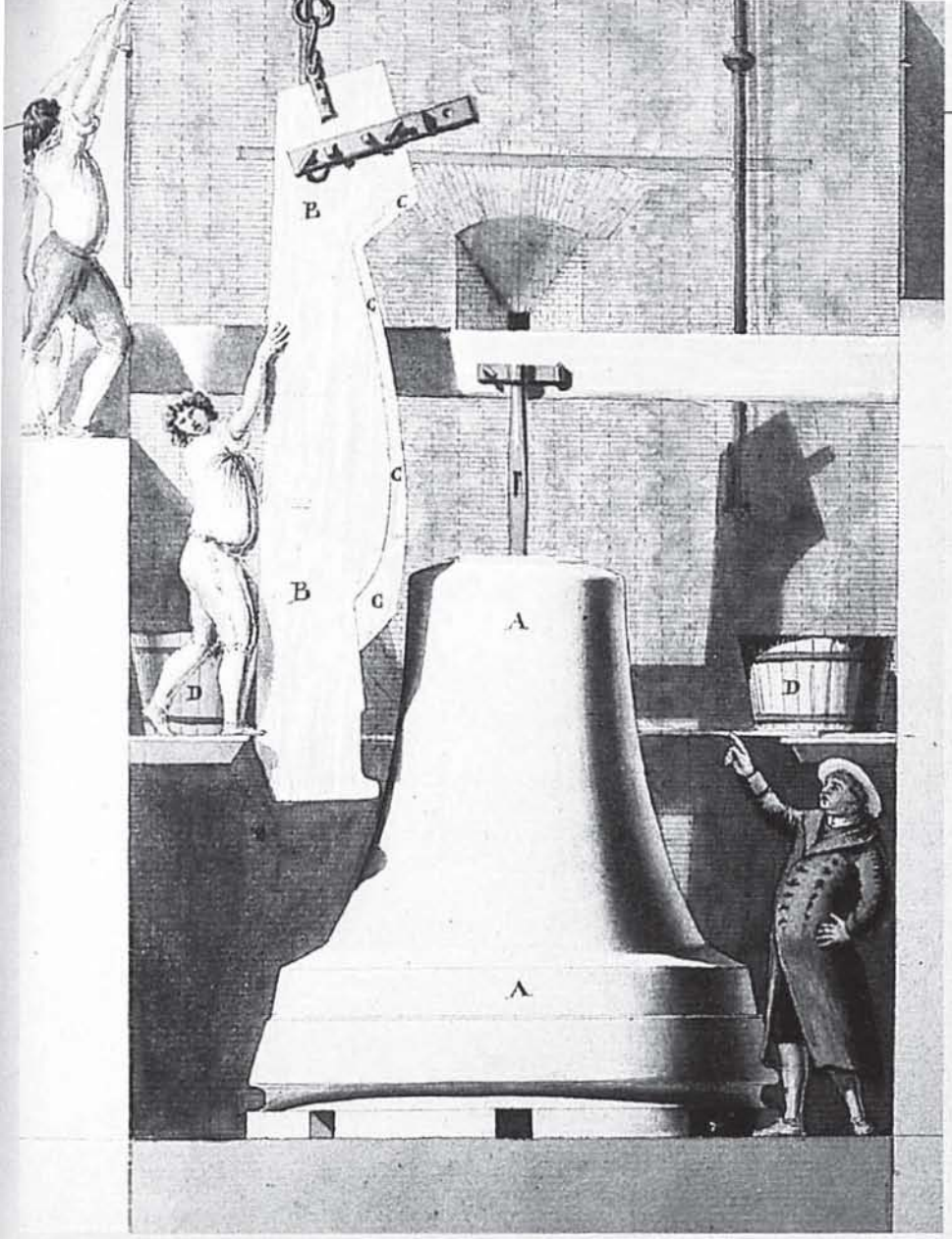
Die Voraussetzungen sind bei der Informatik die gleichen wie bei andern technischen Fachrichtungen: Schnelle Auffassungsgabe, gutes Gedächtnis, Spaß am Lösen schwieriger Probleme und eine glückliche Mischung aus Kreativität und Disziplin sind zweifellos wichtig. Zwei Eigenschaften spielen in der Informatik eine besonders wichtige Rolle, nämlich das solide mathematisch-naturwissenschaftliche Grundwissen und die Fähigkeit zur Kommunikation. Wie sollte jemand, der Schwierigkeiten hat, sich einem (mitdenkenden) Menschen mitzuteilen, dazu bei einer (nicht mitdenkenden) Maschine in der Lage sein?
Erfahrung am Computer, wie sie die meisten Schüler heute schon vor Eintritt ins Hochschulstudium sammeln können, ist vorteilhaft, aber keineswegs Bedingung. Wer die oben genannten Voraussetzungen erfüllt, wird keine Mühe haben, das Programmieren im Studium zu erlernen. Im Gegenteil: Besser keine *Vor*bildung als eine *Ver*bildung, beispielsweise durch Programmierung in der Sprache Basic, die leider in vielen Schulen gebraucht wird. Mir ist es dann weitaus lieber, wenn die Studienanfänger ein solides Fundament in Mathematik, Deutsch und Englisch mitbringen.

Studienabbrecher

Jahr für Jahr scheitert rund die Hälfte der Informatikstudenten an den Prüfungen des Vordiploms. Es ist zu vermuten, daß darunter viele sind, die das Informatikstudium falsch eingeschätzt haben, nämlich als Fortsetzung der Programmbastelei («Hack now, fix later») mit den Weihen der Wissenschaft. Vielleicht sind sie auch Opfer einer idiotischen Werbung geworden (Tenor: «Programmieren ist kinderleicht»). Diesen Kandidaten möchte ich dringend zu einem anderen Fach raten. Ein Hochschulstudium in Informatik ist definitiv keine Weiterbildungsveranstaltung für «Hacker», und es besteht in der Abteilung Informatik keine Bereitschaft, irgendwelche Tendenzen in diese Richtung zu tolerieren.

Frauen in der Informatik

Das Fach Informatik bietet sehr gute berufliche Perspektiven, und dies gerade auch den Frauen. Es ist daher kaum zu verstehen, daß es 1985 unter den 200 frischgebackenen Informatikstudenten an der ETH ganze 5 weibliche gab. Man wird außerhalb der Schweiz kaum eine Hochschule finden, bei der der Prozentsatz nicht wenigstens zweistellig ist (vor dem Komma!).
Rationale Gründe für diese Situation sind mir nicht erkennbar; gerade Frauen, die sich nicht ausschließlich auf den Beruf konzentrieren wollen, haben in der Informatik ausgezeichnete Möglichkeiten, Familie und Beruf zu verbinden, denn es gibt wohl keinen akademischen Beruf, der ähnliche Flexibilität erlaubt (Teilzeitarbeit, freie Arbeitszeit, Arbeit zu Hause).



Qualitätssicherung: Meister und Gehilfen in der Glockengießerei (Giuseppe Valadies: «Disegni e spiegazione della fonderia». 1786).
Die Sicherung der Qualität war viele Jahrhunderte hindurch die wichtigste Aufgabe des Handwerksmeisters. Erst im 19. Jahrhundert wurde durch die arbeitsteilige Produktionsweise eine organisierte Qualitätssicherung notwendig. Im Softwareengineering stehen wir heute vor dem gleichen Problem, freilich ohne auf eine handwerkliche Tradition zurückgreifen zu können.

- durch die Möglichkeit, ein Muster praktisch kostenlos und fehlerfrei zu reproduzieren, und
- dadurch, daß es keinerlei Verschleiß gibt (Tabelle 2)

Aus diesem Vergleich folgen einige wichtige Feststellungen:

- Software hat keinen Materialwert, sondern ist reine Arbeitsleistung. Für diese fehlt uns das Gefühl; wir haben zwar Hemmungen, ein konkretes Material (beispielsweise Holz oder Kupferdraht) zu vergeuden, wir haben diese Hemmungen aber nicht, wenn wir unangemessen viel Zeit damit verbringen, eine praktisch irrelevante Verbesserung an einem Programm durchzuführen. Dabei kostet eine einzige Programmzeile mit allen Vorarbeiten, Dokumentation usw. statistisch etwa 50 Franken (bei starken Schwankungen).
- Programme weisen keine natürliche Strukturierung durch Materialgrenzen auf (ein Auto hat beispielsweise separate Reifen, schon allein darum, weil diese auf andere Weise und aus anderem Material hergestellt sind als die Felgen). Damit besteht bei den Programmierern die Neigung, allzu große, amorphe Programme zu schaffen. Die für das Verständnis des Programms notwendige Strukturierung entsteht also nur durch die Einsicht des Programmierers.
- Eine Begrenzung der Möglichkeiten durch die Materialeigenschaften (zum Beispiel durch die Belastbarkeit von Stahl und Beton beim Brückenbau) ist für den Ingenieur zwar eventuell schmerzlich, aber akzeptabel. Setzt uns aber die geistige Leistungsfähigkeit eine – zudem unscharfe – Grenze, so sind wir kaum bereit, diese zu beachten. Darum schreiben viele Programmierer Programme, die sie selbst nicht verstehen können und die entsprechend fehlerhaft sind. Der Programmierer ist damit ständig in der Situation eines Betrunkenen, der seine Fahrunfähigkeit eingestehen muß: Wie wir wissen, ein schwerer Schritt.
  Beurteilt man die Programmiersprachen nur danach, was sich damit realisieren läßt, so sind Verbesserungen prinzipiell nicht möglich, im Gegenteil, in den «höheren» Sprachen werden die Möglichkeiten eingeschränkt. Gute Sprachen unterscheiden sich von den schlechten also nicht durch größere Möglichkeiten, sondern durch bessere Unterstützung bis hin zum nützlichen Zwang (beispielsweise zur expliziten Deklaration aller Variablen).

Die Möglichkeit der kostenlosen, fehlerfreien Reproduktion hat neben ihren offensichtlichen Vorteilen auch Schattenseiten. Diese werden sichtbar, wenn man die Serienproduktion einer Maschine (etwa eines Automobilgetriebes) mit der eines Programms vergleicht. Beim Getriebe fallen nach den beträchtlichen Entwicklungskosten noch weitaus höhere Fertigungskosten an. Darum ist es für den Hersteller rentabel, das Produkt zu verbessern; im Rahmen der Gesamtkosten spielt die Weiterentwicklung keine so große Rolle, vielleicht werden sogar die Fertigungskosten gesenkt. Bei der Software hingegen stehen der Entwicklung, die nichts als Kosten verursacht, Verkauf oder Benutzung gegenüber, was reinen Gewinn bringt, so-

Tabelle 2. Vergleich der Software mit anderen technischen Produkten.

	Werkstoff des technischen Produkts	Werkstoff der Software (Programmiersprachen)
Kosten	gering bis sehr hoch	nur Anfangsinvestition, dann null
Wesen	konkret, meist sichtbar	abstrakt, unsichtbar
Konstruktion	aus verschiedenen Werkstoffen	Codierung meist nur mit einer einzigen Sprache
Grenzen	durch Materialeigenschaften vorgegeben	durch Verständnis des Programmierers vorgegeben

lange das Produkt nicht geändert wird. Daher kann die Erprobung des Produkts die Entwicklung kaum beeinflussen. Da es zudem keine Abnutzung gibt, entsteht auch kein Zwang zur Erneuerung, so daß alte Software oft ganz unabhängig von ihrer Qualität unsterblich erscheint [7]. Ein Vergleich macht die Aussage plastisch: Der Automobilbau ist nach 100jähriger Evolution wesentlich stabiler als die erst 40jährige Informatik. Trotzdem wird noch überwiegend in Programmiersprachen aus der Zeit um 1960 codiert; Autos aus jenen Jahren gelten heute als Oldtimer.
Anderseits ist die Wiederverwendung vorhandener Programme im Softwareengineering genau das, was das Einsparen bei der Energieerzeugung ist: billiger, sicherer und weit weniger beachtet als andere Verfahren.

Stand und Rückstand des Softwareengineering

Die Mittel

Wer sich heute an Arbeiten auf dem Gebiet des Softwareengineering macht, findet dafür sehr gute Programmiersprachen, Übersetzer und Hilfsprogramme vor. So sind – mit einem Schwerpunkt an der ETH Zürich – in den letzten Jahren ausgezeichnete konventionelle Programmiersprachen und Übersetzer dafür entstanden (MODULA-2, ADA), und für spezielle Aufgaben sind die nichtkonventionellen Sprachen (LISP, PROLOG, SMALLTALK und andere) heute für fast alle gängigen Rechner erhältlich [8]. Ebenso gibt es viele Datenbanksysteme, wobei noch zwischen verschiedenen Datenbankmodellen gewählt werden kann [9]. Vielfach werden Grafikpakete angeboten, die es erlauben, zu vertretbaren Kosten eine attraktive Benutzerschnittstelle zu realisieren. Und natürlich haben wir heute relativ billige, sehr leistungsfähige Computer, vor allem Arbeitsplatzrechner mit hochauflösendem Bildschirm, für die dank der Implementierung eines Standardbetriebssystems (vor allem UNIX) schon bei der Einführung in den Markt sehr viele Programmpakete verfügbar sind.

Die Grundlagen

Als Grundlagen wurden – vor allem im Hochschulbereich – zahlreiche Methoden zur formalen Beschreibung und Entwicklung von Software entwickelt und untersucht, beispielsweise die algebraische Spezifikation [10], Petri-Netze und ihre zahlreichen Erweiterungen [11], Prozeßmodelle wie CSP [12], Transformations- und Verifikationsverfahren. Eine weitere wichtige Grundlage sind die vielen (und dennoch längst nicht ausreichenden) empirischen und experimentellen Daten über Softwarekosten und -fehler, den Ablauf von Softwareprojekten, die Produktivität der Programmierer und vieles andere [13].

Die Methodik

Auf dem Gebiet der Methodik konnte das Softwareengineering auf der *strukturierten Programmierung* (schrittweise Verfeinerung, Begrenzung der Modulgröße, Beschränkung auf «saubere» Strukturen und Verzicht auf die Sprunganweisung) aufsetzen [14]. Etwas jünger ist das heute im Softwareengineering zentrale Prinzip des «information hiding» (erstmals beschrieben von PARNAS, 1972 [15]). Dazu kommen heute ein tiefes Verständnis des Phasenmodells, erprobte Konzepte für alle Entwicklungsphasen (Spezifikation, Entwurf, Codierung, Test) und auch zahlreiche Werkzeuge (zum Beispiel Entwurfssysteme). Auch für das Softwaremanagement, also Projektplanung und -durchführung, Kostenschätzung, Dokumentation, Qualitätssicherung, gibt es heute brauchbare Anleitungen [16].

Der Rückstand des Softwareengineering

Der grundsätzliche Rückstand des Softwareengineering gegenüber den klassischen Ingenieurdisziplinen besteht darin, daß uns eine breite Standardisierung und Kanonisierung bis heute fehlt. Nicht nur die Kenngrößen, Prüfverfahren und Entwicklungsmethoden müssen genormt werden, sondern die Standardisierung fehlt bereits bei der Terminologie und bei der Notation. So kann heute ein schweizerischer Elektroniker relativ mühelos einen amerikanischen oder sogar japanischen Schaltplan lesen; der Programmierer hat schon größte Schwierigkeiten, einen Kollegen aus der eigenen Gruppe in seinen Programmentwurf einzuweihen.
Mit «Kanonisierung» sind hier die Sicherung und die Festigung des Wissens gemeint. Bislang haben Lehrbücher und Lehrpläne für das Softwareengineering gefehlt, doch zeichnet sich eine Verbesserung langsam ab (FAIRLEY, 1985; SOMMERVILLE, 1985, darin speziell der Anhang über die Ausbildung in Softwareengineering 9 [17, 18]). Dagegen bleibt es schwierig, Fachleute von Scharlatanen, die den Boom ausnutzen, zu unterscheiden, und viele halten sich schon für Experten, weil sie programmieren oder irgendwann einmal programmiert haben. Schließlich fehlt es weiterhin auf der Führungsebene fast aller sich mit Software befassenden Organisationen an Kompetenz, und der Mangel wird nicht behoben oder durch externe Hilfe kompensiert, weil er dazu erst eingestanden werden müßte.

**

Zusammenfassend läßt sich also sagen, daß es für das Softwareengineering ausgezeichnete Mittel, ausgereifte Konzepte zur formalen Beschreibung, empirische Daten und praktische, meist auch praktikable Methoden und Werkzeuge gibt. Trotzdem lassen Fachleute – soweit sie nicht im Verkauf arbeiten – ein Gefühl der Stagnation erkennen. Hierfür sehe ich die folgenden Gründe:

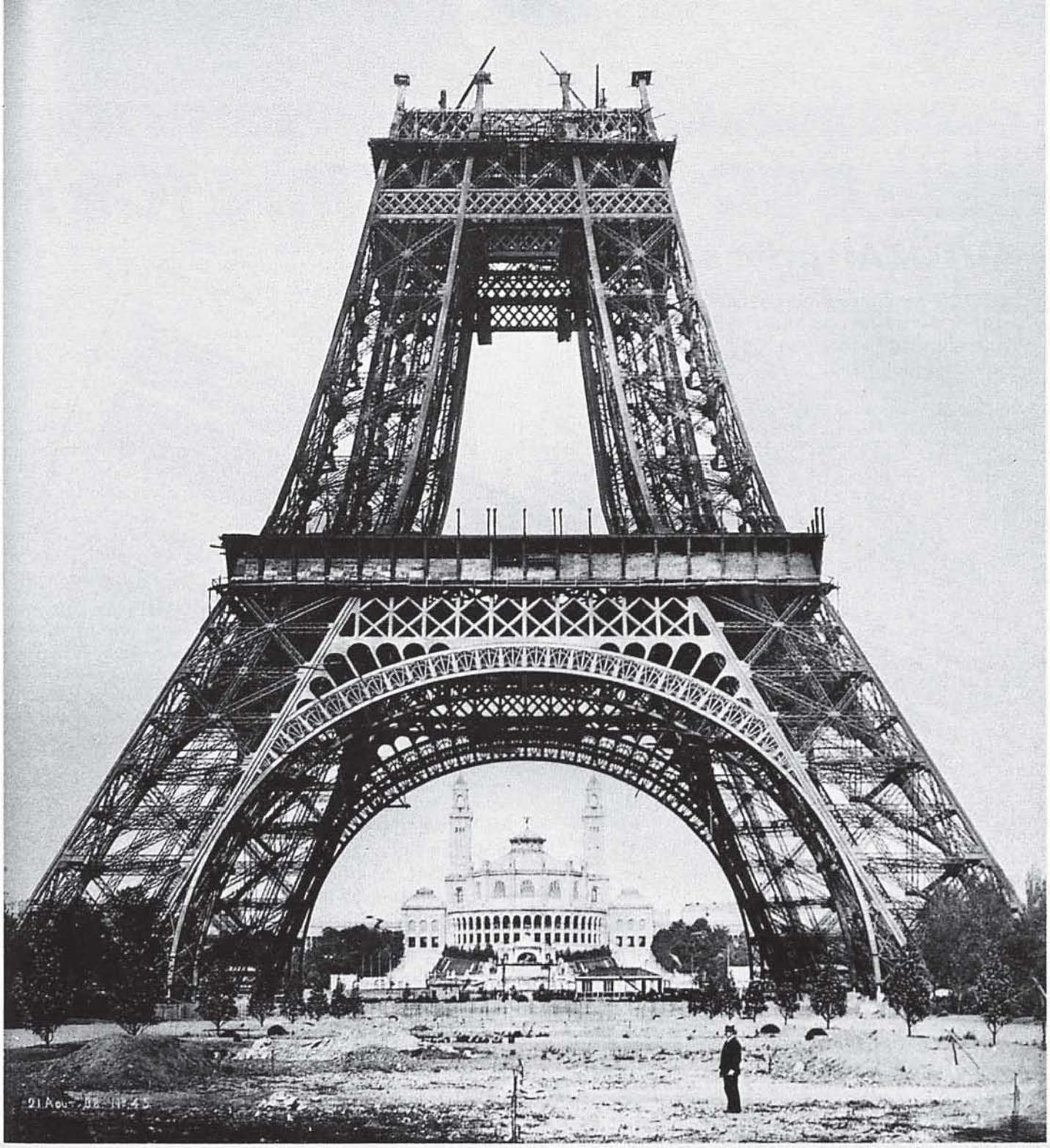
- Der Aufwand zur Realisierung von Werkzeugen ist wesentlich höher, als zunächst vermutet worden war. Dadurch sind experimentelle Arbeiten, wie sie beispielsweise auf dem Gebiet der Übersetzer noch möglich waren, praktisch ausgeschlossen. Kommt aber eine Entwicklung zum Abschluß, so muß das Ergebnis bedingungslos und über lange Zeit vermarktet werden. Selbst wenn Insider ein System für veraltet und überladen halten, kommt eine Neuimplementierung nicht in Frage.
- Ein experimenteller Vergleich zwischen verschiedenen Methoden oder Werkzeugen wäre so aufwendig, daß er nur unter extremen Voraussetzungen (beispielsweise NASA-Projekte) möglich ist. Kaum ein Anwender kennt mehr als ein System.
- Alle Werkzeuge sind nur auf bestimmten Rechnern und Betriebssystemen lauffähig, können also im allgemeinen weder kombiniert noch direkt verglichen werden.

Diese Gründe verhindern eine Konkurrenz auf dem Softwaremarkt, so daß es bisher nicht zu der erwünschten Evolution gekommen ist.

Softwareengineering in der Hochschule

Als *Technik* ist Softwareengineering primär eine Angelegenheit derer, die Informatik praktizieren, also der Industrie, der Banken usw. Doch es gibt gute Gründe, auch in der Hochschule in dieser Richtung tätig zu sein:

- In der Forschung können Arbeiten durchgeführt werden, die nicht kurzfristig zu verwertbaren Ergebnissen führen.
- Experimentelle Arbeiten oder vergleichende Untersuchungen sind nur von einer neutralen Stelle aus sinnvoll.



- Der Arbeitsmarkt verlangt nach Informatikern mit Kenntnissen in Softwareengineering [19]. Neben dem Wunsch nach gut ausgebildeten neuen Mitarbeitern steht dabei oft auch die Hoffnung, ein noch fehlendes Know-how zu erwerben; die Hochschulinformatiker sind heute ja vielfach in der Situation, schon unmittelbar nach dem Studium als Experten eingeschätzt (und bezahlt) zu werden.

Forschungsziele

Das naheliegende Ziel für ein Projekt auf dem Gebiet des Softwareengineering ist die Schaffung eines neuen XXX-Systems, wobei XXX je nach Vorkenntnissen und Interessen für Spezifikation, Test oder ähnliches steht. Die Durchführung dieses Projekts wäre unproblematisch, krankte aber nach allen Erfahrungen daran, daß es mit großer Wahrscheinlichkeit keine konkreten Folgen hätte, denn die kritische Projektgröße würde kaum erreicht, so daß kein wirklich brauchbares Werkzeug entstünde; gelänge es wider Erwarten doch, so könnte es sich am Markt mangels Vergleichsmöglichkeit nicht durchsetzen.

Darum ist es an der Zeit, die Fragen anders zu stellen. Vielleicht gibt es ja die richtige Methode, die Komponenten des richtigen Werkzeugs bereits. Wir müssen Wege finden, die vorhandenen Grenzen der Konstruktion: Foto des Eiffelturms im Bau (Gustave Eiffel: «La tour de 300 mètres». Paris, 1900).

Die Eigenschaften der Werkstoffe und die Fähigkeiten des Ingenieurs begrenzen den Raum des technisch Machbaren; die erste Grenze akzeptieren wir, wenn auch ungern, die zweite dagegen versuchen wir vielfach zu ignorieren. Vor hundert Jahren demonstrierte Gustave Eiffel, wie weit die Möglichkeiten der Eisenkonstruktion reichten. Da der Programmierer ein Material mit Idealeigenschaften verwendet, liegt seine Grenze einzig in ihm selbst. Darum ist für ihn die Gefahr besonders hoch, Konstruktionen anzufertigen, die die Grenzen überschreiten. Noch in einer anderen Hinsicht ist der Eiffelturm ein interessantes Beispiel: Von Beginn an wurde bezweifelt, ob man das Machbare auch tatsächlich machen sollte.

Ansätze zu bewerten, zu vergleichen und wo möglich zu integrieren. Konkret folgen daraus zwei Arbeitsthemen:

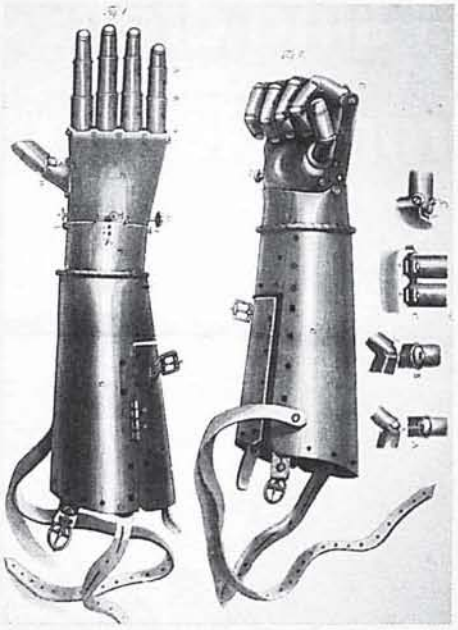
- Vergleich der vorhandenen Methoden und Werkzeuge, Integration durch eine leistungsfähige Datenhaltung, die sogenannte *Software Engineering Data Base*
- Kombination formaler und halbformaler Spezifikationssprachen

Dabei wollen wir unsere Ergebnisse nicht nur publizieren, sondern vor allem selbst anwenden und damit laufend kontrollieren. Die Kooperation mit der Praxis ist hierfür unerläßlich.
Für die Diplom- und Semesterarbeiten, die die Studenten in unserer Gruppe durchführen, lassen sich daraus die folgenden Themen ableiten:

- empirische Feldstudien
- vergleichende Arbeiten zu speziellen Sprachen, Methoden, Werkzeugen
- Implementierungen in MODULA-2, ADA und SMALLTALK auf vernetzten Arbeitsplatzrechnern
- strikte Anwendung der Softwareengineeringprinzipien

Die Lehre

In der eigentlichen Lehre sind die Grundlagen und Techniken des Softwareengineering zu vermitteln. Wie die Teilnehmerzahl einer Spezialvorlesung zeigt, entspricht dies auch den Wünschen der Studenten.
Zum Lernen reicht aber die logische Begründung nicht aus; der Student braucht auch Motivation und Training. Welche Möglichkeiten bietet hier das Studium? *Allein oder zu zweit* schreiben die Studenten *kleine Programme*, die nur *für kurze Zeit* oder *gar nicht* eingesetzt werden.
In der Praxis ist die Situation typisch umgekehrt: *Viele* Programmierer arbeiten gemeinsam an *großen* Systemen, die viele Jahre lang eingesetzt, korrigiert, geändert und erweitert werden. Aus *diesen* Randbdingungen folgt die Motivation für Softwareengineering, nicht aus denen des Studiums.
Wir können daraus zwei verschiedene Schlüsse ziehen: Wir planen die Ausbildung in Softwareengineering für einen Zeitpunkt *nach* dem Eintritt in das Berufsleben ein, oder wir versuchen, den Studenten möglichst viele der notwendigen Erfahrungen bereits im Studium zu verschaffen.
Die erste Alternative (Nachdiplomschulung) ist sehr wichtig und wird auch in der Industrie mit Nachdruck verfolgt (Beispiel: BBC-Informatikschule; Egg, 1986 [20]). Allerdings bestehen dabei selbst für große Firmen etliche Schwierigkeiten, von der Lehrerrekrutierung bis zur notwendigen Freistellung der Mitarbeiter.



Prototyping: Die eiserne Hand des Götz von Berlichingen (Christian von Mechel: «Die eiserne Hand des tapferen deutschen Ritters Götz von Berlichingen». Berlin, 1815).
Besonders wo der Kontakt zwischen dem Benutzer und dem technischen System eng ist, läßt sich eine gute Lösung nur evolutionär erreichen. Auch die berühmte eiserne Hand des Götz von Berlichingen hatte einen einfacheren Vorgänger, der nur kurz verwendet wurde. Im Softwareengineering kennt man das Vorgehen, ob dem zunächst mit geringem Aufwand ein einfaches Muster angefertigt wird, als «Prototyping».

Daher sollten die Hochschulen das eine (die Nachdiplomausbildung) fördern, ohne das andere, die Ausbildung im Curriculum, zu lassen. Wie können wir dabei Projekterfahrung vermitteln?
Sorgfältig ausgewählte Projekte, die unter dem Aspekt des Softwareengineering besonders ergiebig sind, lassen sich als Seminar durchführen. Die Arbeitsbelastung ist allerdings für alle Beteiligten sehr hoch. (In [21] sind entsprechende Erfahrungen einer Lehrveranstaltung am NTB in Buchs SG dokumentiert.)
Aufwendigere Kurse wurden an verschiedenen Hochschulen eingerichtet und mit großem Erfolg durchgeführt. Zum Beispiel arbeiten die Informatikstudenten der TU Berlin ein Jahr lang in Gruppen von etwa 15 Leuten; ein Tutor leitet das Projekt.
An der ETH Zürich bieten sich die Gruppensemesterarbeiten als Gefäß einer solchen Veranstaltung an, doch liegt auch dabei wieder der notwendige Aufwand an oder über der oberen Grenze. Wir werden daher versuchen, die Arbeiten der Studenten generell stark zu vernetzen, so daß sie insgesamt als großes Projekt wirken. Ferner ist zu prüfen, ob Voraussetzungen und ausreichendes Interesse für freiwillige Kompaktkurse in der vorlesungsfreien Zeit gegeben sind.

13 ⓜ

Literatur

1 Boehm B. W. (1973): Software and its impact: a quantitative assessment. Datamation 19, 5, 48–59.
2 Brooks F. P. (1975): The Mythical Man-Month. Addison-Wesley, Reading, Massachusetts.
3 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.
4 Weinberg G. M. (1971): The Psychology of Computer Programming. Van Nostrand Reinhold, New York.
5 Boehm B. W. (1976): Software Engineering. IEEE Transactions on Computers, C-25, 1226–1241.
6 IEEE (1975–1985): Proceedings of the International Conference on Software Engineering. Washington, 1975; San Francisco, 1976; Atlanta, 1978; München, 1979: San Diego, 1981; Tokio, 1982; Orlando, 1984; London, 1985; Monterey, 1987.
7 Schnupp P. (1978): Ist Cobol unsterblich? In Alber (Hrsg.), Programmiersprachen, 5. Fachtagung der GI, Informatik-Fachbericht 12, Springer-Verlag, Berlin usw.
8 Ludewig J. (1985): Sprachen für die Programmierung. BI-Hochschultaschenbuch Nr. 622, Bibliographisches Institut, Mannheim.
9 Zehnder C. A. (1985): Informationssysteme und Datenbanken. B. G. Teubner, Stuttgart, 3. Auflage.
10 Klaeren H. A. (1983): Algebraische Spezifikation. Springer-Verlag, Berlin usw.
11 Reisig W. (1985): Systementwurf mit Netzen. Springer-Verlag, Berlin usw.
12 Hoare C. A. R. (1985): Communicating Sequential Processes. Prentice Hall, Englewood Cliffs, N. J.
13 Basili V. R., R. W. Selby, D. H. Hutchens (1986): Experimentation in Software Engineering. IEEE Transactions on Software Engineering, SE-12, 733–743.
14 Gries D. (ed.) (1978): Programming Methodology. Springer-Verlag, Berlin usw.
15 Parnas D. L. (1972): On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15, 1053–1058.
16 Lustman F. (1985): Managing computer projects. Reston Publ. Co., Reston, Virginia.
17 Fairley R. E. (1985): Software Engineering Concepts. Mac Graw Hill, New York usw.
18 Sommerville I. (1985): Software Engineering. Addison-Wesley, Wokingham, England, usw. 2nd edition.
19 Strunz H. (1984): Anforderungen der Softwareindustrie an die Informatik. Angewandte Informatik, Jahrgang 1984, 513–515.
20 Egg K. (1986): Informationen zur BBC-Informatikschule. Unveröffentlicht (Quelle: Dr. K. Egg, BBC-PAI, 5401 Baden).
21 Ludewig J. (1985): Techniken der Programmierung im kleinen und im großen. KLR-Bericht Nr. 85-149 C, Brown-Boveri-Forschungszentrum, Baden-Dättwil.

Die Bilder sind historischen Büchern entnommen, die in der Eisenbibliothek stehen, einer Stiftung der Georg Fischer AG, Schaffhausen. Ich danke dem Bibliothekar, Herrn CLEMENS MOSER, für seine freundliche Hilfe bei der Auswahl und dem Fotografieren der Bilder. Die Bestände der Eisenbibliothek, die im Klostergut Paradies in Langwiesen TG untergebracht sind, sind für technikgeschichtliche Studien allgemein zugänglich.