artec
Forschungszentrum
Nachhaltigkeit

Software Manufaktur – Software Engineering – Software Architektur.

Konkurrierende Leitbilder in der Geschichte der Softwaretechnik

Hans Dieter Hellige

Universität Bremen

artec Forschungszentrum Nachhaltigkeit
Enrique-Schmidt-Str. 7, 28359 Bremen
www.uni-bremen.de/artec

artec-paper Nr. 221
Juni 2019
ISSN 1613-4907

Das artec Forschungszentrum Nachhaltigkeit ist ein interdisziplinäres Zentrum der Universität Bremen zur wissenschaftlichen Erforschung von Fragen der Nachhaltigkeit. Das Forschungszentrum Nachhaltigkeit gibt in seiner Schriftenreihe „artec-paper“ in loser Folge Aufsätze und Vorträge von MitarbeiterInnen sowie ausgewählte Arbeitspapiere und Berichte von Forschungsprojekten heraus.

Impressum

Herausgeber:

Universität Bremen
artec Forschungszentrum Nachhaltigkeit
Postfach 33 04 40
28334 Bremen
Tel.: 0421 218 61801
Fax: 0421 218 98 61801
URL: www.uni-bremen.de/artec

Kontakt:

Katja Hessenkämper
E-Mail: hessenkaemper@uni-bremen.de

Software Manufaktur – Software Engineering – Software Architektur
Konkurrierende Leitbilder in der Geschichte der Softwaretechnik

Hans Dieter Hellige

Software Manufaktur – Software Engineering – Software Architektur
Konkurrierende Leitbilder in der Geschichte der Softwaretechnik

Hans Dieter Hellige

Abstract

Die folgenden Ausführungen nehmen das 50. Jubiläum der berühmten NATO-Konferenzen von Garmisch und Rom zum Anlass zu einem kritischen Blick auf das meist vereinfachte Geschichtsbild der Software Community. Dem vorherrschenden linearen Erfolgsmodell wird eine erweiterte historische Perspektive gegenübergestellt und die These entwickelt, dass die lange vor 1968/69 beginnende softwaretechnische Methodenentwicklung sehr wesentlich ein diskontinuierlicher Lernprozess war, in dem frühere Erkenntnisse über die Grenzen der Engineering-Metapher und erste Ansätze zu iterativen und evolutionären Designmethoden aus dem Blick gerieten und Jahrzehnte später erst wieder neu entdeckt werden mussten. Es wird daher die bisher nur sehr verkürzt wahrgenommene Traktat-Literatur der ‚Vorläufer' des Software Engineerung analysiert und gezeigt, dass von Beginn mit diesem die Leitbilder der *Software Manufaktur* und *Software Architecture* konkurrierten.

- Das 1. Kapitel skizziert zunächst den Entstehungskontext der Garmisch-Konferenz und legt dabei Friedrich L. Bauers Intentionen und Programmatik sowie deren Folgen für die disziplinäre Verortung der Disziplin Software Engineering dar. Dann wird anhand eines Vergleiches von Rückblicken von Pionieren und Repräsentanten der Software Engineering-Community bei den 10-Jahres-Jubiläen der NATO-Konferenzen die Zunahme von Problemwahrnehmungen und die allmähliche Entzauberung des Garmisch-Mythos herausgearbeitet. Gleichwohl hielt man beharrlich an einem vagen Ingenieur- und Industrialisierungs-Leitbild fest, das, wie der Wissenschaftshistoriker Michael Mahoney mehrfach dargelegt hat, nie gründlicher wissenschaftlich fundiert wurde. Die Engineering-Metapher hatte so letztlich zwar eine wichtige Anstoßfunktion für die Institutionalisierung der Disziplin, sie reichte aber offenbar nicht aus, um die Entwicklung der Softwaretechnik als einen durchgängigen Verwissenschaftlichungs- und Industrialisierungsprozess zu begründen.

- Im 2. Kapitel werden Software-Manufakturen im militärischen und staatlichen Contractor-Sektor betrachtet, die unter dem Einfluß von Operations Research und Systems Engineering standen und den Schwerpunkt auf eine Team-orientierte Teilung der Arbeit legten. In diesen gab es bereits vor den NATO-Konferenzen von deren Teilnehmern nicht zur Kenntnis genomme Engineering-Konzepte und Prozess- und Lifecycle-Modelle für die Softwareherstellung. Doch es werden hier bereits die Grenzen von wasserfallartigen Entwicklungsprozessen erkannt und erste Modelle für ein iteratives und evolutionales Vorgehen erörtert.

- Das 3. Kapitel behandelt das Software-Architektur-Konzept im Corporate Software-Sektor, das hierarchische Organisationsprinzipien mit einem nicht-analytischen erfahrungs- und qualitätsorientierten Designkonzept verknüpfte. Dabei wird gezeigt, dass schon parallel zur Propagierung des Software-Engineering-Begriffs seit den 60iger

Jahren Software Architecture als Methodenleitbild in der Community kursierte, mit dem neben der Produkt- und Prozess-Strukturierung vor allem der Synthese-Charakter der Software-Entwicklung betont wurde.

- Das 4. Kapitel legt skizzenhaft dar, dass als Folge der Krise des Software-Engineerings das Software-Architektur-Leitbild seit den späten 80er Jahren eine Renaissance erlebte mit dem Ziel einer Vollendung bzw. einer Ablösung des Software-Engineering-Konzeptes.[1]

1 Software-Engineering: 50 Jahre auf dem Weg zur Ingenieurwissenschaft

Kaum eine andere Teildisziplin der Informatik pflegt ein derart ausgeprägtes Geschichtsbewusstsein wie das Software Engineering. Alle zehn Jahre und insbesondere beim 25. und 50. Jubiläum wurde an den ‚Gründungsakt' der Disziplin im Jahre 1968 erinnert und in ausführlichen Bestandsaufnahmen erörtert, ob und in welchem Maße die Programmatik der Gründerväter eingelöst wurde. Nicht nur die Rückblicke von Beteiligten der legendären NATO-Konferenzen von Garmisch und Rom von 1968/69, sondern auch die historischen Darstellungen der Entwicklung der Softwaretechnik sind stark auf den Garmisch-Mythos fixiert. Die Garmisch-Tagung erscheint gar als „die Geburtsstunde der modernen Softwareentwicklung und des Software Engineering"[2]. Dieses nicht ganz falsche, aber stark vereinfachende Narrativ[3] folgt meist dem in vielen Publikationen artikulierten Selbstverständnis der Organisatoren und Teilnehmer der beiden Tagungen. Den Entstehungshintergrund für diese Fokussierung bildeten das äußerst geschickte argumentative und institutionelle Vorgehen von Friedrich L. Bauer als dem Organisator der Garmisch-Konferenz und die nachfolgende effektvolle Propagierung des beschlossenen wissenschafts- und professionspolitischen Programms in dem vielzitierten Konferenzband. Dieser verknüpfte eine zugespitzte Beschreibung der „software crisis" bzw. der „software gaps" mit Angeboten einer ganzen Reihe von Rezepten für deren Lösung.[4] Durch diese sollte sich das Programming vom vorherrschenden „software tinkering (bricolage)", vom „dilettantism of a home-made 'trickology' " zu einer „scientific discipline" entwickeln und mit dieser auch die Grundlage für eine Industrialisierung der Softwareproduktion geschaffen werden.[5]

---

[1] Anregungen zu den folgenden Ausführungen bekam ich durch Gespräche mit Friedrich L. Bauer im GI-Präsidiumsarbeitskreis „Geschichte der Informatik", durch Diskussionen mit Michael Mahoney und eine von Gerard Alberts geleitete Konferenz am CWI Amsterdam im Jahre 2006. Wichtige Hinweise und wertvollen Rat verdanke ich vor allem Ulf Hashagen vom Forschungsinstitut des Deutschen Museums München.

[2] Broy 2017, Informatik als Wissenschaft, S. 201.

[3] Haigh 2010a, Crisis, S. 2.

[4] Naur, Randell 1969, Garmisch-Report, S. 16 ff., 120 ff.; Buxton, Randell 1970, Rome Report, S. 7, 13 (zitiert wird nach den in der Online-Version mitgeteilten Orginalseitenzahlen der Reports); Haigh 2010 a, Crisis, S. 9 ff.

[5] Bauer, Mitteilung an MacKenzie, zit. in: ders. A View from the Sonnenbichl, S.107; Bauer 1993, Software Engineering, S. 259; Bauer 1976, Programming, S. 223.

1.1 Friedrich L. Bauers Informatik-Programm, die NATO-Konferenzen 1968/69 und die Disziplin-Genese des Software Engineering

Aufgrund des großen Echos in der Scientific Community sah sich Bauer schon bald als eigentlicher Erfinder des "Software Engineering", das neben der von ihm gleichzeitig protegierten Fachbezeichnung "Informatik" ein weiterer Kernbegriff für die von ihm forcierte Disziplingründung werden sollte. Er hatte sich schon seit 1955 in der von ihm sogenannten "Algol-Verschwörung" für die Entwicklung eines Grundstocks an Algorithmen, eines „universal framework of concepts", einer darauf beruhenden „common language" eingesetzt und deren Zustandekommen in mehreren internationalen Treffen organisiert.[6] Mit Algol 60 „as a kind of scientific notation, an extension of mathematical language" wollte er dem Wildwuchs der Sprachkonzepte und Notationen, der sich aus der Konkurrenz der Hardware-Firmen und dem korrespondierenden ‚undisziplinierten' Künstlergebahren der Programmer Community ergeben hatte, Einhalt gebieten und zugleich saubere Strukturierungs- und Elementarisierungsprinzipien in die Softwarekonstruktion einführen. Stets abgeneigt gegen Tricks, tolle Ideen und features, zielte sein Bestreben auf „begriffliche Universalität", eine „rigorose Spezifikation der Problemstellung" und eine durch formale algebraische Programmtransformation optimierte „Programmkonstruktion".[7] Seit seiner Berufung an die TU München 1963 begründete Bauer dort „eine umfangreiche Schule der Programmiertechnik"[8] und wirkte zusammen mit Klaus Samelson als ein intensiver Agendasetter für die neue Disziplin. Im Jahre 1967 verdichteten sich diese Aktivitäten in mehreren Schritten zu einer Institutionalisierung des Faches: in der Errichtung des ersten vollständigen Studiengangs „Informatik", in der maßgeblichen Beteiligung an der Ausgestaltung des „Überregionalen Forschungsprogramms" für Datenverarbeitung, insbesondere an dem Schwerpunkt „Technologie und Systemprogrammierung", sowie durch geschicktes Lobbying beim Bundesminister für wissenschaftliche Forschung in der erfolgreichen Lancierung der Disziplinbezeichnung „Informatik".[9] Den Abschluss fanden diese ‚verschwörerischen' Aktivitäten in der von im ausgegangenen Gründungsinitiative für die „Gesellschaft für Informatik" im Juni 1969.[10]

Bauers Agenda in Sachen „Software Engineering" entwickelte sich jedoch nicht aus seinem bisherigen Forschungsprogramm heraus, sondern kam durch Anstöße von außen

---

6 Bauer 2004, Die Algol-Verschwörung; Nofre, Priestley, Albert 2014, When Technology Became Language, S. 25.

7 Bauer 1980, Mathematik und Informatik, S. 32, 35.

8 Bauer, Informatik – Geburt einer Wissenschaft, S. 25.

9 Siehe hierzu Bauer 2007, Geburt der Informatik in München, S. 27 ff.; Coy 2004, Was ist Informatik, S. 479 ff.

10 Krückeberg 2002, Die Geschichte der GI.

zustande. Bei einem Gastsemester in Stanford im Frühjahr 1967 gewann er den Eindruck, dass die USA zwar durch militärfinanzierte „koste es was es wolle-Projekte“ bei der Hardware weit überlegen waren, dass Europa aber im Softwarebereich durchaus gegenhalten konnte.[11] Zu seinen bislang überwiegend fach- und hochschulpolitischen Zielvorstellungen kamen nun wirtschaftspolitische Aspirationen hinzu: Europa sollte mit seiner „manufacturing atmosphere“ und seinen niedrigeren Löhnen den US-Vorsprung bei Computern durch Software-Aktivitäten ausgleichen, denn „if Europe cannot capture its proper share of the cake at this time, its future in the computer field would be dim“.[12] Eine Gelegenheit bot sich, als der US-Delegierte im NATO Science Committee Isidor Isaac Rabi Anfang 1967 die Errichtung einer multinationalen „Study Group Computer Science“ vorschlug, die in einem zu gründenden „International Institute of Computer Science“ den Stand und vor allem die Probleme der Disziplin erforschen sollte. Den Anlass hierfür bildeten die „software crisis“ im Militärbereich und Befürchtungen, dass der Flaschenhals Software am Ende die Großforschung strangulieren könnte. Nachdem der deutsche Vertreter im NATO Science Committee Eduard Pestel Bauer für die Mitarbeit gewonnen hatte, ergriff dieser sofort die Initiative mit dem Ziel, das geplante Institut nach Deutschland zu holen und zwar nach Möglichkeit nach Bayern. Es gelang ihm auch bei den Verhandlungen im Herbst 1967, den NATO-Auftrag an die Study Group statt auf das Gesamtgebiet der Computer Science auf den Engpass Softwareproduktion zu fokussieren und dabei den Schwerpunkt auf die Überwindung des üblichen „tinkering“ durch einen „clean fabrication process“ zu setzen, und zwar mit dem viel zitierten Ausspruch: „What we need, is software engineering“.[13] Begriff und Konzept hatte er sehr wahrscheinlich bei seinem Gastaufenhalt in den USA kennengelernt, denn dort wurden Engineering-Methoden zur Überwindung der „software crisis“ bereits seit 1960 breiter diskutiert, 1966 hatte der ACM-President den Begriff „Software Engineering“ sogar als ein offizielles Programm der Computer Science verkündet.[14] Bauers Verdienst lag also wie auch bei Disziplinbezeichnung „Informatik“ nicht in der Erfindung des Begriffs, sondern in dessen geschickter Einfädelung in den fachpolitischen Diskurs.

Nachdem Bauer Ende 1967 zum Vorsitzenden der geplanten Arbeitstagung ernannt und das vorgeschlagene Arbeitsprogramm im März 1968 vom NATO Science Committee gebilligt worden war, baute er das Software Engineering-Projekt in seine bisherige Agenda ein. Noch vor der Garmisch-Konferenz beantragte er Mittel für die Berufung von Heinz Rutishauser und Gerhard Seegmüller nach München sowie für ein 5,1 Mio-

---

11 Bauer 1970, Der *computer* in unserer Zeit, S. 6 f.; Broy 2007, Von der Ingenieurmathematik zur Informatik, S. 243.

12 Bauer 1969, Software Engineering S. 189

13 Bauer 1969, Software Engineering S. 190; ders. 1993, Software Engineering – wie es begann, S. 259; Naur, Randell 1969, Garmisch-Report, S. 13 f.

14 Siehe dazu ausführlicher unten Kapitel 3.

Projekt für die Entwicklung des modernsten Betriebssystems für den TR 440 als Vorbild für ein „Einheitsmodell für künftige Anlagen“.[15] Dabei wollte Bauer mit dem „Münchner Betriebssystem“ (BSM) an die Pionierabeiten von Seegmüller anknüpfen, der aufgrund der „alten Ingenieuridee des Zerlegens“ schon 1962/63, d.h. noch vor Dijkstras viel diskutiertem „THE-Multiprogramming System“ von 1965, eine geschichtete Architektur entwickelt hatte. Bei dem bereits Ende 1968 unter der Leitung von Gerhard Goos begonnenen Bau des BSM wurden sogar schon industrielle Prinzipien des „software engineering“ angewendet: modularer Aufbau, strukturiertes Programmieren und streng hierarchische Schichtung.[16] Darüber hinaus beantragte Bauer Mittel für die Gründung eines „Zentrums industrieller software-Fertigung“ im Voralpenland, um im Verbund mit der örtlichen Luftfahrt- und Elektronikindustrie ein Gegengewicht gegen Software-Zentren in Frankreich und in den USA zu bilden. Dies war wohl auch als Vorleistung gedacht, um die Chancen des Münchner Raums als Sitz des von der NATO geplanten „International Institute of Computer Science“ zu erhöhen, das sein eigentliches Ziel war.[17]

Mit der Garmisch-Konferenz gelang es Bauer, Begriff und Programmatik des Software Engineering in der fachwissenschaftlichen Öffentlichkeit publik zu machen. Dabei war es vor allem die zugkräftige Metapher, der die Konferenz ihre große positive Resonanz verdankte, sie hatte sich schon während der Konferenz nichtzuletzt wegen ihrer Unbestimmtheit zu einem autosuggestiven Leitbild entwickelt. Zudem harmonisierte der schnell berühmt gewordene Report die durchaus vorhandenen Auffassungsdifferenzen, die in der Nachfolgekonferenz dann auch offen hervortraten. So stieß die zentrale argumentative Verknüpfung von Krisensymptomatik und Verwissenschaftlichungsdruck auf lebhaften Widerspruch. Während Bauer dem Verdikt Edsger Dijkstras, die bisherige Software-Profession sei auf Betrug gebaut, weitgehend zustimmte, erschien Douglas McIlroy das Krisengerede als eine gewaltige Übertreibung. Er verglich die NATO-Konferenz wenig später sogar mit einer „heady group therapy session, full of breastbeating about the software crisis and the general malaise of our trade and the way we practice it.“[18] Die bereits in Garmisch präsenten Kontroversen fanden infolge des Herstellungsprozesses nur teilweise Eingang in den Tagungsband.[19] Aus Gründen der Zeitersparnis wurden nur die den Verfassern des Berichtes Randell und Naur wichtig erscheinenden Passagen der Tonbandprotokolle transkribiert, die ausgewählten Diskussionsbeiträge mit Auszügen aus schriftlichen Statements und nachträglichen

---

15 Informationen in diesem Abschnitt, die Bauers Aktivitäten in einem neuen Licht erscheinen lassen, verdanke ich Ulf Hashagen vom Forschungsinstitut des Deutschen Museums.

16 Bauer 2007, Geburt der Informatik in München, S. 44 f.

17 Siehe Anm. 15 und Goos 2008, Die Informatik in den 70er Jahren, S. 134 f.

18 McIlroy 1972, Outlook for Software Components, S. 245.

19 Mitteilung von Gerhard Goos bei der CWI-Konferenz: Pioneering Software in the 1960s, Amsterdam, November 2006.

Kommentaren von Teilnehmern kombiniert und in gestraffter Form nach den während der Konferenz beschlossenen Gliederungspunkten neu arrangiert. Bauer sprach später von einer „carfully digested compilation“, McIlroy dagegen von einem „triumph of misapplied quotation“. Da von den eingereichten 50 Working Papers nur 8 im Druck aufgenommen und die Tonbänder, wie den Teilnehmern im Interesse „offenherziger Beiträge“ zuvor zugesichert, nach dem Abschluss des Reports gelöscht wurden, lässt sich der tatsächliche Diskussionsverlauf nicht mehr rekonstruieren.[20]

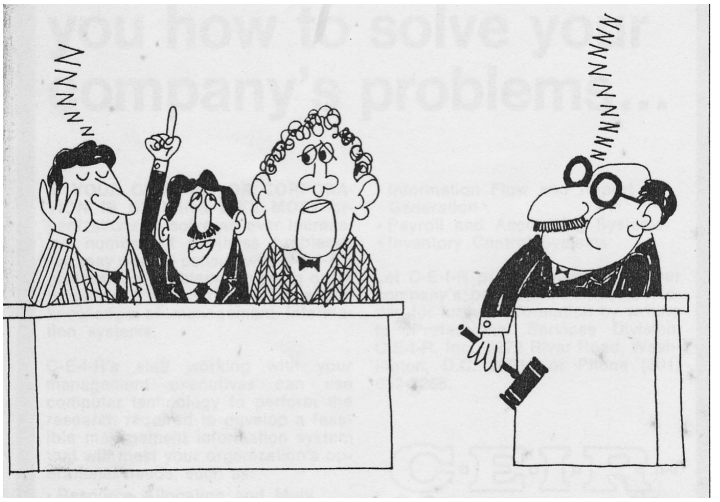
F. L. Bauer in der Garmisch-Konferenz 1968, der bei der Tagung den Spitznamen „Uncle Fritz“ erhielt. (http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/BAUER.html)

Der Software Engineering-Begriff wurde bei den NATO-Konferenzen maßgeblich von den Anhängern formalisierter, strukturierter und quantifizierender Methoden im Sinne eines „mathematical engineering“ (Edsger Dijkstra, Stanley Gill, James N. Patterson Hume, Friedrich L. Bauer) sowie streng naturwissenschaftlich basierter Ingenieurmethoden und eines ebenfalls wissenschaftlichen Industrial Engineering (Robert W. Bemer, Douglas McIlroy) besetzt, sehr oft auch zusammengefasst in der Leerformel „sound engineering principles“. Die „concerns“ des Designs wurden dabei, da schwer formalisierbar, ausgeklammert und die „requirements“ der „user“ nur zu Beginn ermittelt. Nach Dijkstra hatte der „user“, da nicht genau definierbar, „no place in computer science or software engineering“.[21] Auch der Vorschlag von Edward E. David von den Bell Labs, mit „case studies in industry“ zu beginnen, die „boundary conditions“ der

[20] Naur, Randell 1969, Garmisch-Report, S. 10 f.; Dijkstra 1968, Verslag von het bezouk, S. 0; Bauer 1969, Software Engineering, S. 190; Randell, The 1968/69 NATO Software Engineering Reports, Dagstuhl, 1996, S. 37 f.; Haigh 2010 b, Dijkstra’s Crisis, S. 7.

[21] Dijkstra-Zitat von 1979 in Boehm 2011, Future Software Engineering Opportunities S. 15.

Designprobleme zu analysieren und sie in die Theoriebildung einzubeziehen, fand kaum Beachtung.[22] Mit Blick auf ein verwissenschaftlichtes Konzept hatte Bauer in dem von ihm „handverlesenen Teilnehmerkreis“ akademische Vertreter aus Universitäten und industriellen Forschungsinstituten bevorzugt.[23] Er selber sah in diesen Jahren wie die Mehrheit der europäischen Vertreter in der zu schaffenden Disziplin Software Engineering eine „applied science“ mit dem Theorie- und Methodenvorbild der Mathematik, insbesondere der westeuropäischen Form der Ingenieur-Mathematik und mathematischen Physik. Da für Bauer die in Schichten und Modulen zu entwerfenden Softwaresysteme im Wesentlichen „mathematical structures“ darstellten, aus denen sich implizit bereits ein Vorgehensplan für die industrielle Fertigung ergebe,[24] sollte das neue Lehrgebiet Software-Entwicklung ein aussichtsreiches Ausbildungsfeld in der angewandten Mathematik bzw. in der künftigen grundlagenwissenschaftlich orientierten Informatik werden.[25] Auch der von Dijkstra ins Spiel gebrachte Titel „Mathematical Engineer“ kam aus einer ähnlichen deutsch-niederländischen Tradition, er stieß jedoch bei amerikanischen Teilnehmern auf Ablehnung, die in ihm einen Widerspruch in sich sahen und sich auch gegen eine Platzierung des neuen Lehrgebietes in mathematischen Fachbereichen oder in der Computer Science wehrten.[26]



Cartoon zu F. L. Bauers Bericht über die Garmisch-Konferenz von 1968
in der Zeitschrift Datamation (Bauer 1969, Software Engineering S. 189)

22 Buxton, Randell 1970, Rome Report, S. 13.

23 Haigh 2010 a, Crisis, S. 16 ff.

24 Naur, Randell 1969, Garmisch-Report, S. 23, 53, 74.

25 Gerhard Goos und Albert Endres, Mitteilungen bei der CWI-Konferenz, Nov. 2006; Haigh 2010a, Crisis, S. 11 f.; Haigh 2010b, Dijkstra’s Crisis, S. 13, 16 f.; Leimbach 2011, Geschichte der Softwarebranche, S. 221 ff.

26 Naur, Randell 1969, Garmisch-Report, S. 127; Dijkstra 1968, Verslag von het bezouk, S. 1; Gill 1972, The Origins S. 230, 242.

Von der Verwissenschaftlichung durch Formalisierung, Strukturierung und Modularisierung von Funktionen und Arbeitspaketen sowie von Korrektheitsprüfungen und quantifizierenden Leistungsmessungen versprach sich die ‚mathematische' Richtung eine Software-Produktion von höherer Qualität und Zuverlässigkeit. Der Entwicklungsprozess sollte nach einer vorab überprüften Spezifikation plangetrieben in „successive stages" ablaufen.[27] Vor allem aber wollte man durch die „division into manageable parts" und die hierarchische Strukturierung der zu produzierenden Software wie der entsprechenden Organisation der Produktion die Kommunikation zwischen den Gruppen einschränken und die Kontrolle über den Design- und Produktionsprozess zurückgewinnen, nach Ascher Opler durch „formal measurement by control group", nach Robert Bemer in Zukunft womöglich durch ein "machine-controlled production environment".[28] Neben der mathematisch-methodischen Disziplinierung zur Eindämmung des Wildwuchses der „Go to-Programme" sollte nun auch die ‚Fabrikdisziplin' dem Künstlerschlendrian der Softwareentwickler ein Ende bereiten.

Obwohl sich der von der NATO nicht klassifizierte Garmisch-Report in der Software-Community schnell verbreitete, zerschlugen sich die hohen Erwartungen in die angekündigte Institutsgründung. Denn bei der folgenden „very fractious conference" in Rom im Oktober 1969 trat der bereits in Garmisch erkennbare „split between academia and industry" offen hervor,[29] so dass es zu keiner Einigung über das Grundverständnis und die konstituierenden Techniken und Methoden der neuen Disziplin kam. Angesichts der tiefen Gegensätze gab es weder einen Vorschlag für eine Fortsetzung der NATO-Konferenzen noch für eine Beantragung der Institutsgründung. Dabei war es gerade die „hidden agenda" der Organisatoren der Rom-Tagung, darunter Bauer als Co-chairman, „persuading NATO to fund the setting up of an International Software Engineering Institute. However things did not go according to their plan". Denn der Institutsplan stieß während zweier Sitzungen überwiegend auf Skepsis und wurde, da „in the main nontechnical", im zweiten Report völlig ausgeklammert.[30] Thomas H. Simpson, einer der führenden Betriebssystementwickler bei IBM, karrikierte den quälenden Entscheidungsprozess in Rom in Form einer Satire über eine „International Working Conference on Masterpiece Engineering": Auf ihr wollten die führenden Maler der italienischen Renaissance anhand der „Mona Lisa" die Designkriterien für Meisterwerke festlegen und die Methoden herausfinden, wie der Malprozess von Meisterwerken verwissenschaftlicht und mit Hilfe von „more-efficient tools" rationalisiert

---

27 Naur, Randell 1969, Garmisch-Report, S. 19, 50 ff., 181-185.

28 Naur, Randell 1969, Garmisch-Report, S. 19, 55, 89 f., 94 f., 165 ff., 220; Bauer 1971, Software Engineering, S. 532 f.

29 Mitteilung von Albert Endres bei der CWI-Konferenz, November 2006; Zitat von Manfred Paul in: ICSE 2018 - Plenary Sessions, Discussion.

30 Randell 1993, The 1968/69 NATO Software Engineering Reports, S. 41; Buxton, Randell 1970, Rome Report, S. 5.

werden könnte. So sollte etwa die Produktivität der Maler durch die Zahl der Pinselstriche pro Tag gemessen oder eine Leinwand schnell von Maler zu Maler weitergeben werden, um dann am Ende, da dies nichts half, alles der alleinigen Entscheidung Leonardo da Vincis zu überlassen, eine deutliche Anspielung auf das „chief-programmer" Konzept, das Joel D. Aron bereits in Rom vorgestellt hatte.[31] Da nach dem Scheitern einer Verständigung von der NATO keine Mittel mehr für ein Software-Zentrum zu erwarten waren, initiierte Bauer um 1970 die Gründung eines gemeinsamen französisch-englisch-deutschen Software-Engineering-Instituts in München. Doch an „britischen Eifersüchteleien" scheiterte auch dieser letzte Versuch, so dass ein Zentrum vergleichbarer Größenordnung erst 1984 mit dem vom Pentagon errichteten „Software Engineering Institute" an der Carnegie Mellon University zustande kam.[32]

Trotz der gescheiterten Institutspläne schienen die ersten Erfolge bei der Disziplin-Genese die Erwartungen der Initiatoren voll zu bestätigen, denn mit der Leitmetapher und der Inszenierung eines Paradigmenwechsels stießen sie schnell auf eine breite Resonanz. Die beiden Konferenzen wurden Ausgangspunkt einer Kette auf einander aufbauender softwaretechnischer Innovationen, die Ende der 60er Jahre mit „Structured Programming" und „Hierarchical Structuring" (Dijkstra) begann und sich in der ersten Hälfte der 70er Jahre fortsetzten mit „Stepwise Refinement" (Wirth), „Modul-Oriented Programming" und „Information Hiding" (Parnas), „Chief Programmer Teams" (Mills), „Incremental Build" und „Software Evolution" (Mills, Lehmann, Belady).[33] Als Verfahrensmodell übernahm man in Ergänzung des „Iterative Multilevel Modelling" für die Design-Strukturierung (Zurcher, Randell) Spezifikations-getriebene Lifecycle-Modelle aus dem militärischen Bereich. Dabei bezog man sich vor allem auf die beiden einfachsten, wasserfallartigen Modelle des Leiters vom Lockheed Software Technology Center Winston W. Royce, die als normative Empfehlungen missverstanden wurden, obwohl er ihre Implementierung selber als riskant und fehlerträchtig bezeichnet hatte. Seine weniger disziplinierenden Modelle mit zusätzlichen Iterations- und Prozess-Schleifen wurden dagegen weitgehend übersehen und erst verspätet im Rahmen von inkrementellen und evolutionären Methoden-Ansätzen zur Kenntnis genommen.[34]

Die Literalisierung des neuen Faches setzte mit ersten Fachbüchern ab 1977, Handbüchern und Einführungen ab 1979/80 ebenfalls recht bald ein, relativ wenige speziel-

---

31 Simpsons Sketch in: Randell 1998, Memories of the NATO Software Engineering Conferences, S. 53 f.; Buxton, Randell 1970, Rome Report, S. 50 f.

32 Manfred Broy in Bauer 2007, Geburt der Informatik in München, S. 48.

33 Siehe hierzu bes. Endres 1996, A synopsis; Shapiro 1997, Splitting the Difference; Raccoon 1997, Introduction. Fifty Years of Progress, S. 89 ff.; Boehm 2006, A View of 20th and 21st Century, S. 16.

34 Naur, Randell 1969, Garmisch-Report, S. 204 ff.; Royce 1970, Managing the Development, Fig. 2 u.3; Boehm 1981, Software Engineering Economics, S. 35; Ruparalia 2010, Software Development Lifecycle Models, S. 8 ff.; Kneuper 2017, Sixty Years of Software Development.

le Fachzeitschriften begannen jedoch erst in den 80er und vor allem 90er Jahren zu erscheinen. Nach einer anfangs vorwiegend akademischen Debatte erfolgte in der zweiten Hälfte der 70er Jahre vor allem bei Softwarehäusern die Einführung strukturierter Methoden und Entwicklungswerkzeuge in die Praxis.[35] Auch die Institutionalisierung als Unterdisziplin bzw. Randdisziplin der Computer Science bzw. Informatik seit der Mitte der 70er Jahre erhielt durch die beiden Konferenzen entscheidende Anstöße (1975 *International Conference on Software Engineering* und *IEEE Transactions on Software Engineering (ICSE)*, 1976/77 *ACM Special Interest Group on Software Engineering*, 1981 *GI-Fachgruppe Software Engineering*). Die Etablierung im Forschungsbereich und eine teilweise disziplinäre Verankerung an Hochschulen gelangen zwar relativ bald, aber meist nur als ein einzelner Kurs im bestehenden Curriculum und nicht „as a discipline in itself", die die Breite des erforderlichen multidisziplinären Wissens abgedeckt hätte.[36] Der „Master of Software Engineering" blieb die Ausnahme, das Fach wurde von der ACM nicht als Bestandteil des regulären Curriculums akzeptiert. Die lange Zeit bindenden ACM-Curriculums-Empfehlungen, die bereits im März 1968 bekannt gemacht wurden, sahen lediglich „true-to-life"-Projekte am Rande des Lehrbetriebs vor.[37]

Die Professionalisierung von Software-Ingenieuren verlief insgesamt nur äußerst schleppend, denn die etablierte Computer Science lehnte die Anerkennung von Software Engineers als Profession ab und sperrte sich nahezu zwei Jahrzehnte gegen eine Zertifizierung, Registrierung oder Lizensierung.[38] Im Jahr 1994 wies die „National Society of Professional Engineers" der USA sogar darauf hin, dass der Gebrauch des Titels „Software Engineer" illegal sei, da Software Engineering nicht zu den anerkannten 36 Ingenieurdisziplinen gehöre.[39] Und trotz intensiver Bemühungen um eine Sammlung und Strukturierung des „Software Engineering Body of Knowledge" lehnte die ACM noch im Jahr 2000 die „license" für einen „Professional Engineer" mit der Begründung ab, das Software Engineering als Disziplin noch keine ausreichende Reife habe: „a software engineering license would be interpreted as an authoritative statement that the licensed engineer is capable of producing software systems of consistent reliability, dependability, and usability. The ACM Council concluded that our state of knowledge and practice is too immature to give such assurances."[40] Trotz der mannigfachen wissenschaftlichen Vorleistungen wurde die Subdisziplin damit auch nach über

35 Denert 1991, Software-Engineering, S. 435 ff.

36 Freeman, Wasserman, 1976, Essential elements, S. 120.

37 Freeman, Wasserman 1976, A proposed Curriculum; Tomayko 1998, Forging a discipline, S. 5 ff.; Ensmenger 2010, The Computer Boys, S. 173 ff.

38 Boehm 1979, Software Engineering, S. 13; Wasserman 1996, Toward a Discipline, S. 30 f.

39 Raccoon 1998, Introduction. Toward a Tradition, S. 105.

40 Association for Computing Machinery, A summary of the ACM position on software engineering as a licensed engineering profession (2000), zit. nach Seidman 2008, The Emergence, S. 63 ; White, Simons 2002, S. 91 f..

dreißig Jahren vom eigenen Dachverband noch immer nicht angenommen, während sie in der gewerblichen Praxis als zu theoretisch galt und nur nur sehr partiell rezipiert wurde. Auch in Deutschland, wo das Faches besser angesehen war als in den USA und als Ingenieurwissenschaft halbwegs akzeptiert wurde, führten Bemühungen um eine Anerkennung der Softwaretechniker als Ingenieure nicht zum Ziel, die Zertifizierung blieb ein Desiderat. Noch nach 2000 nutzten nichtzuletzt wegen der unzureichenden empirischen Ausrichtung nur ein Drittel der Software entwickelnden Firmen in Deutschland systematische Methoden und Entwicklungsprozesse, in keiner anderen Ingenieurdisziplin ist mithin „die Diskrepanz zwischen dem Stand der Forschung und dem Stand der Praxis" derart groß.[41]

1.2 Jubiläumsrückblicke auf die Garmisch-Konferenz: Die allmähliche Entzauberung eines Mythos

Die historischen Rückblicke der Software Engineering-Community spiegeln das fundamentale Akzeptanzproblem in den folgenden Jahrzehnten zunächst nur teilweise. Stattdessen bekräftigen sie die akademische Binnensicht auf die NATO-Konferenzen als dem Angelpunkt der Software-Geschichte immer aufs Neue. Da man frühere Software Engineering-Ansätze und die bereits seit Beginn der 60er Jahre laufende Softwarekrisen-Debatte nicht zur Kenntnis nahm (siehe dazu weiter unten), etablierte sich ein recht einfaches Argumentationsmuster: Das Methodenbewusstsein in der Softwareentwicklung ist nahezu aus dem Stand heraus entstanden. Die von wenigen Computer Scientists in der zweiten Hälfte der 60er Jahre diagnostizierte „software crisis" bildete den Auslöser für die Konferenz-Initiative, aus der die Disziplin „Software-Engineering" hervorging, durch die in der Folgezeit die Krise weitgehend bewältigt wurde. Dieses einseitige Bild fand auch Eingang in die meisten Abrisse und historischen Darstellungen.[42] Doch betrachtet man die Bestandsaufnahmen in den jeweiligen Jubiläumsjahren genauer, so wird deutlich, dass hierin bei den Pionieren wie in der neu entstandenen Software Engineering-Community zunehmend Problemwahrnehmungen und enttäuschte Erwartungshaltungen zum Ausdruck kamen. Und dies vor allem, weil sich angesichts der unerwarteten Entwicklungsdynamik des Softwaresektors die Erreichung des zentralen Zieles einer anerkannten Disziplin Software Engineering immer weiter hinausschob.

Die Zehnjahresrückblicke von Antony I. Wassermann und László A. Belády im Oregon-Report sowie von Dijkstra und Randell belegen noch ganz die Es-ist-erreicht-Sichtweise: Das Jahr 1968 war danach der „major turning point" der Software-Geschichte,

---

41 Lewerentz, Rust 1998, Sind Softwaretechniker Ingenieure?; Broy, Rombach 2002, Software Engineering, S. 446 f. ; Broy, Jarke, Nagl, Rombach 2006, Manifest, S. 215 f.

42 Siehe hierzu Haigh 2010 a, Crisis, S. 2 ff.

durch den die Software-Herstellung das Stadium des „individual craftsman“ und „software architect“ hinter sich gelassen hatte und auf eine „component production“ hinarbeitete. In den zehn Jahren wurde „a recognizable body of knowledge“ über den „software life cycle“ zusammengetragen, das mit den Methoden und Praktiken traditioneller Ingenieurwissenschaften vergleichbar wäre und das im kommenden Jahrzehnt zu einem systematischen Konzept der Software-Entwicklung zu integrieren sei.[43] Der in Garmisch noch kaum erkennbare Trend zu interaktiven Systemen veranlasste Wasserman aber schon Anfang der 80er Jahre zur Erweiterung des Instrumentariums mit dem Ziel eines „User Software Engineering“.[44] Bei der 4. ICSE in München unter Bauers Vorsitz zogen Randell, Dijkstra und Barry Boehm eine insgesamt sehr positive Bilanz über das erste Jahrzehnt nach Garmisch. Nach Randell war 1968 der „software engineering bandwagon“ ins Rollen geraten und habe sich seitdem von einer „expression of a requirement to a description of a healthy well-developed discipline” entwickelt.[45] Für Dijkstra bestand der „turning point in the history of programming“ vor allem darin, dass hier zum ersten Mal die Existenz der Softwarekrise und der „proliferation of the error-loaded software“ offen ausgesprochen wurde. Der Übergang vom zünftigen Handwerk zu einer wissenschaftlichen „Programming Methodology“ sei in den zehn Jahren erfolgreich begonnen, die Leitziele eines „scientific designer“ und der „production line“ seien allerdings wegen des Festhaltens an der traditionellen Arbeitsteilung der „programming guild“ noch nicht verwirklicht worden. Für die Zukunft forderte er das „computer programming“ zu einer exakten mathematischen Disziplin zu machen und fest in der Computer Science zu verankern, während alle Aspekte des Managements, der Ergonomie- und Wirtschaftlichkeit außenstehenden Spezialisten vorbehalten seien.[46] Dagegen mahnte Barry Boehm eine stärker Business-bezogene Methodenentwicklung an. Der Software-Bereich habe zwar einen soliden Fortschritt gemacht, doch hielten die Softwaretechniken mit dem schnellen Wachstum der Software nicht Schritt. Man habe anfangs auch zu sehr auf „simplistic panaceas“ wie Experten-Design und Prüftechniken gesetzt und die Anforderungen von Anwendungssoftware in einem „economic-driven context“ vernachlässigt.[47]

Weitaus kritischer war Tony Hoares Einschätzung des bisher Erreichten, denn er registrierte eine eher oberflächliche Aufnahme der Konzepte des „structured“ und „modular programming“. Er sah sogar eine „startling contradiction“ im Leitbegriff „software engineering“ und noch eine große Differenz zwischen den professionellen Idealen von Ingenieuren und den Qualitätsansprüchen von Software-Entwicklern und er warnte:

---

43 Wasserman, Belady 1978, The Oregon Report, S. 30 ff.

44 Wasserman 1981, User Software Engineering.

45 Randell, 1979, Software Engineering, S. 1, 8.

46 Dijkstra 1979, Software Engineering, S. 442 ff.; E. W. Dijkstra Archive: EWD 556, 1977, S. 2 f., 9 f.; EWD 641, 1978, S. 0-2; EWD 690, 1987, S. 0 (https://www.cs.utexas.edu/users/EWD/).

47 Boehm 1979, Software Engineering, S. 13, 18 f.

„The attempt to build a discipline on such shoddy foundations must shurely be doomed." Deshalb sollten sich die Programmierer strikt an den „sound methods" von Ingenieuren orientieren, um sich den Titel „Software Engineer" erst noch zu verdienen.[48] Völlig resigniert über die geringen Erfolge waren zwei weitere NATO-Konferenzteilnehmer. Robert McClure kam zu dem Urteil, die durch die Garmisch Software Engineering-Initiative propagierten Methoden hätten in der Praxis kaum Widerhall gefunden: „the great masses of programmers have not changed their habits. Their code is unstructured as it ever was."[49] Robert W. Floyd schlug in seiner Turing-Lecture 1978 sogar den Begriff der „software depression" als den angemesseneren Begriff vor, denn die Software sei trotz allgemeiner Anerkennung des „structured programming paradigm" noch immer in demselben deprimierenden Zustand wie zu Zeiten der „famous ‚software crisis' " 10 bis 15 Jahre zuvor.[50] Die Gründe für die mangelhafte Akzeptanz des „methodical design" in der Industrie, der hohe Zeitaufwand und die evolutionäre Dynamik von Entwicklungsprozessen, die sich durch den grundlegenden Wandel des Softwaresektors noch intensivierte,wurden jedoch erst im folgenden Jahrzehnt bewusst.[51]

In den 80er Jahren versprachen die Objektorientierten Programmiertechniken mit ihrem „natural design", ihren Reuse-Strategien und Design Patterns sowie vor allem der große Hoffnungsträger CASE und in der Folge eine Welle von Software Factory-Konzepten einen neuen Anlauf zur Überwindung der Softwareprobleme. Begleitet wurden die Innovationen von einer schrittweisen Flexibilisierung verbindlicher Vorgehensmodelle vom rigiden Wasserfallmodell über das Spiralmodell zu verschiedenen iterativen und Prototyping-Ansätzen.[52] Bei der „Twenty Year Retrospective of the NATO Software Engineering Conference" auf der 11. ICSE 1989 betonten die Panelists noch einmal die stimulierenden Effekte der NATO-Konferenzen auf die Forschung, die positiven Wirkungen auf das allgemeine Methodenbewusstsein in der Softwaretechnik und die hohe Anerkennung von Software Engineering als Disziplin. Allerdings sei Software Engineering noch immer keine „description of fact as a statement of aspiration".[53] Insgesamt überwogen bei den 20-Jahr-Rückblicken die kritischen Stimmen, angestoßen vor allem durch Frederick P. Brooks Generalabrechnung mit dem Software Engineering in seinem berühmten IFIP-Vortrag „No Silver Bullet". Wegen grundsätzlicher Komplexitätsprobleme in der Software-Entwicklung, aber auch weil die Methoden mit dem Wachstum und der Komplexitätssteigerung der „fat

48 Hoare 1978a, The Engineering of Software, S. 37 f., 40; 1978b, Software Engineering, a Keynote, S. 1-4.

49 McClure 1976, Software – The Next Five Years, S. 6.

50 Floyd 1979, The Paradigms, S. 455 f.; Mahoney 1990, The Roots of Software Engineering, S. 9.

51 Wirth 1995, A Plea for Lean Software, S. 66.

52 Osterweil 2011, A Process Programmer.

53 Siehe die Beiträge von Galler, S. 97; Gries, S. 98 und Shaw 1989b, S. 100 bei der 11. ICSE 1989.

software“ nicht schritthielten, plädierte er implizit dafür, die Engineering-Metapher zugunsten der „Software Architecture“ aufzugeben.[54] Dijkstra nahm nun Abschied von seiner Hoffnung auf eine streng logikbasierte Science, bei der die funktionale Spezifikation als „logical firewall“ zwischen der Sorge um „correctness“ und um „pleasentness“ fungiert. Software Engineering war für ihn damit eindeutig, wie es Tony Hoare schon zehn Jahre zuvor angedeutet hatte, eine „doomed discipline“, die ihr in sich widersprüchliches Ziel nicht erreichen werde.[55] Auch Belády ging nun auf Distanz zum „misnomer“ „Software Engineer“, den er mit Blick auf das wesentlich komplexere Aufgabenfeld durch „Information System Engineer“ ersetzen wollte.[56]

Weitere Probleme des Methodeninstrumentariums, die die Pioniere bislang ausgespart hatten, wurden nun Schwerpunkte der Kritik in der Software Engineering Community. Dewayne E. Perry von den AT&T Bell Laboratories bemängelte bei der 9. ICSE 1987 den allzu langsamen Fortschritt im Management der Software-Entwicklung: „the underlying models used in managing the evolution of software have not changed significantly in the past 20 years.“[57] Larry E. Druffel vom „Software Engineering Institute“ (SEI) sah in seiner Einführungsrede bei der 10. ICSE 1988 die entscheidenden Defizite der Disziplin in der einseitigen Fokussierung auf analytische Methoden, übergeordnete Fragen der „software architecture“ seien dagegen vernachlässigt worden. Insgesamt kam er zu dem ernüchternden Resultat, dass im Vergleich mit etablierten Ingenieurdisziplinen „software engineering still had a significant amount of maturing to do.“[58] Auch Mary Shaw vom SEI sah in der fehlenden Beachtung des „Software Architecture Level of Design“ den Hauptgrund für den Entwicklungsrückstand der Disziplin. Die „Software Architecture“ sollte daher als vorgeschaltete Teildisziplin mit neuen Abstraktionen auf der Ebene der Systemorganisation den noch fehlenden Schlussstein zu einer exakten Ingenieur-Wissenschaft setzen, ein Ziel, das seit der Lektüre des Garmisch-Reports im Jahre 1969 ihre Forschung bestimmte.[59] Andere kritische Stellungnahmen sahen die entscheidenden Mängel des Software Engineering aber nicht in den bereitgestellten Methoden, Tools, und Techniken, sondern in ihrer fehlenden Validierung in der Praxis. So wiesen empirische Feldstudien vom „Software

54 Brooks 1987, No Silver Bullet.

55 Dijkstra 1989, The Cruelty of Really Teaching Computing Science, S. 1400; E. W. Dijkstra Archive EWD 799 (https://www.cs.utexas.edu/users/EWD/ewd07xx/EWD799.PDF); siehe auch sein späteres resigniertes Resümee, dass er mit seinem Eintreten für strukturierte Programme und Systeme kaum Einfluss auf das „industrial programming“ gehabt habe, auch sein Berufsleitbild eines „mathematical engineer“ sei nicht angenommen worden: „I have had no influence in the teaching of programming as a tough but rewarding intellectual discipline; this view is hardly tolerated.“ Dijkstra 2001, Interview, S. 22.

56 Belády 1986, Software Engineer, S. 53 f.

57 Perry 1987, Software Interconnection Models S. 61.

58 Druffel, 10th ICSE 1988, S. 44.

59 Shaw 1989a, Larger scale systems, S. 143; 1989b, Remembrances, S. 99.

Engineering Laboratory“ der NASA nach, dass die Anwendung entsprechender Softwaretechniken die Zuverlässigkeit der Software nur um 30% verbesserte, die Produktivität aber überhaupt nicht erhöhte.[60] Sie und andere forderten daher eine Neuausrichtung des Forschungs- und Lehrbetriebes vom Produkt auf den Prozess, vom „technology to problem-driven research“, denn kognitive und soziale Praktiken der Softwareentwicklung, insbesondere „bevavioral factors“, „team group dynamics“ hätten einen bedeutenden Einfluss auf den Designprozess.[61] Das SEL gab den Anstoß zu einer systematischen Forschung über die Designpraxis, woraus unter ihrem Pionier Victor R. Basili das Gebiet des „experimental software engineering“ hervorging.[62]

Das 25-jährige Jubiläum der Garmisch-Konferenz wurde dann zum Höhepunkt der Methoden- und Grundlagenkritik am Software Engineering, denn es herrschte der Eindruck vor, dass sich angesichts verschärfter Softwareprobleme die Disziplin selber in einer Krise befand. Nicht nur die meisten Software-Zeitschriften und das Informatik-Spektrum brachten ausführliche Bestandsaufnahmen und Diskusionsbeiträge, sondern selbst der „Scientific American“ widmete sich der „Software's Chronic Crisis“: „The vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently. […] Academic computer scientists are starting to address their failure to produce a solid corps of software professionals.“[63] Ein internationales Dagstuhl-Seminar im Februar 1992 über die Zukunft der Disziplin geriet zu einem einzigen Desideraten-Katalog: Es gäbe keinen Konsensus über die angemessene Forschungs-Methodologie und über anerkannte Qualitätsmaßstäbe, es fehlten wegen der vorherrschenden präskriptiven Methoden deskriptive und vor allem empirische Ansätze, um die Wirksamkeit der vielen formalen Modelle Techniken und Werkzeuge zu evaluieren, ja es herrsche sogar Unklarheit darüber, was Software überhaupt ist.[64]

Angesichts der Krisensymptome konzentrierte sich die Kritik am relativ geringen Einfluss der Software Engineering-Forschung auf die industrielle Praxis und an der Qualität der Software noch mehr auf Grundsatzfragen: die generelle Praxisferne, die Überschätzung der Wissenschaft als Lehrmeister der Praxis, die Unangemessenheit des Engineering-Leitbildes für den Software-Bereich und vor allem die unzureichende Einbeziehung der User in den Design-Prozess. Colin Potts vom Georgia Tech machte für die fehlende Durchschlagskraft der teilweise beachtlichen Forschungsleistungen

---

60 Card, McGarry, Page 1987, Evaluating Software Engineering, S. 845 f.

61 Shneiderman, Carroll 1988, Ecological Studies, S. 1256; Curtis, Krasner, Iscoe 1988, S. 1268 f.

62 Basili 2006, The Past, Present, and Future of Experimental Software Engineering; ders., Briand et al. 2018, Software Engineering Research and Industry.

63 Gibbs 1994, Software’s Chronic Crisis, S. 86.

64 Siehe die Abstracts von Adrion, Kaiser, Young, Barstow, Nagl, Mahler und Rombach in: Tichy, Haberman, Prechelt, 1992, Future Directions in Software Engineering, S. 5 f., 8, 12, 27 f.

den vorherrschenden „*research-than-transfer approach*“ verantwortlich. Da hierbei die grundlegenden Praxisprobleme oft gar nicht in den Blick gerieten, forderte er einen komplementären „*industry-as-laboratory*“-Ansatz, bei dem die empirische Beobachtung realer Software-Projekte, der Kommunikationsprobleme in Teams und allgemeine organisatorische Probleme zum zentralen Forschungsgegenstand werden.[65] David L. Parnas fand, dass Wissenschaft und Ingenieurpraxis im Software Engineering noch gar nicht zusammengekommen waren, da die Mehrheit der Ingenieure nichts von der „science of programming“ verstünde und die Wissenschaftler wenig darüber wüssten, was es heißt, ein Ingenieur zu sein. Infolge der entstehungsbedingten Etablierung als ein „subfield of computer science“ hätten wissenschaftliche Methoden im Zentrum gestanden und dadurch eine akkreditierte „engineering education“ behindert.[66]

Es wurden nun selbst die wissenschaftlichen Grundlagen und das Wissenschaftsverständnis in Frage gestellt. Gerhard Goos, der als Oberassistent F. L. Bauers an den NATO-Konferenzen teilgenommen und mit ihm Standard-Lehrbücher für das Top-Down-Teaching verfasst hatte, kritisierte nun besonders die Überschätzung der mathematischen Grundlagen und Spezifikationen in der Disziplin seit Garmisch: „Die Wissenschaft trägt Einsichten bei, liefert aber noch keine ausreichende Einsicht in die Zusammenhänge zwischen Systemzielen, Konstruktionsmethoden, Produktionsmethoden, Kosten und Terminen. […] Es ist eigentlich bedauerlich, daß man 25 Jahre nach der Garmischer Konferenz noch nicht sehen kann, wie eine solche Konstruktionslehre für Software im einzelnen aussieht.“[67] Kontrovers wurde auch die Orientierung an ingenieurwissenschaftlichen Vorbildern diskutiert. Robert Baber wollte mit den „engineering design strategies“ und den mathematisch rigorosen Methoden des Electrical Engineering die Softwarekonstruktion endlich aus ihrer „pre-engineering phase“ herausführen.[68] Herbert Klaeren, Vertreter einer gesellschaftsbezogenen Informatik, betonte dagegen die generellen Unterschiede zum Softwarebereich, der keine physikalischen Größen, stetige Funktionen und Naturgesetze kenne. Die Softwareentwicklung verlaufe zwar „disziplinierter, messbarer und prüfbarer“, doch eine Herstellung nach „ingenieurmäßigen Regeln“ sei nach wie vor nicht möglich.[69] Nach Mary Shaw hatte Software Engineering noch einen weiten Weg zur Ingenieurwissenschaft vor sich, sie hielt aber ungeachtet der prinzipiellen Kritik von Michael Mahoney eisern an dem Leitbild fest. Für sie entwickelte sich die Disziplin auch noch ganz übereinstimmend mit dem Evolutionsmodell für Ingenieurwissenschaften von James

---

65 Potts 1993, Software Research Revisited, S. 19 f.

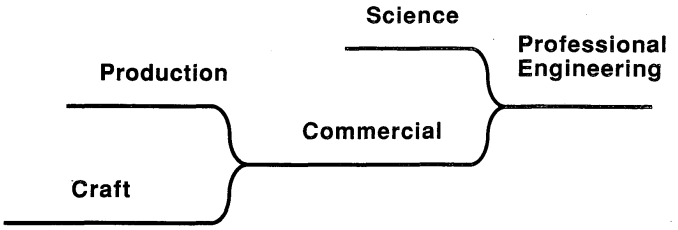
66 Parnas 1996, Software Engineering: An Unconsummated Marriage, S. 35; ders. 1998, Software engineering programmes, S. 19, 22 ff.

67 Goos 1994, Programmiertechnik, S. 12, 19.

68 Baber 1997, Comparison of electrical “engineering”, S. 5; ders. 1998, Software engineering education, S. 43 f.

69 Klaeren 1994, Probleme des Software-Engineering, S. 21, 28.

Kip Finch. Diesem folge bereits das „Programming-in-the-small“, in Zukunft auch das „Programming-in-the-large“, so dass in etwa zehn Jahren der „level of an industrial-age engineering discipline“ erreicht werde.[70]



Finch-Modell der „Evolution of an Engineering Discipline“
als Vorbild für Software Engineering, Shaw 1990, S. 17

Zentraler Kritikpunkt wurde jetzt auch der „gulf between developer and user“ und die fehlende Berücksichtigung des Software Designs als „predominantly social activity“.[71] Da die „iterative and incremental design cultures“ nicht mit den traditionellen Prozess-Standards des Software Engineering vereinbar waren, entwickelten die Bell Labs Verfahren zur Eigenbeobachtung der realen sozialen und kommunikativen Verhaltensmuster in Software-Entwicklerteams.[72] Auch die Informatikerin an der TU Berlin Christiane Floyd, die bei Softlab die weltweit erste Entwicklungsumgebung geschaffen hatte, erklärte sich die fehlende Akzeptanz des Software Engineering vor allem mit organisatorischen, kollaborativen und kommunikativen Defiziten. In Teilbereichen wie der Modellierung, dem Softwareentwurf und der Programmierung seien zwar ungeheure Fortschritte erzielt und dadurch die Softwarekrise der 60er Jahre im Prinzip überwunden worden. Doch die aus dem Engineering-Leitbild abgeleitete „Produktionssicht“ habe über das verabsolutierte lineare Wasserfallmodell zu einer „Softwarebürokratie“ geführt, die das menschliche Arbeitshandeln, die Kooperation und die nutzerbezogene Perspektivität ausblendete. Der Leitidee der „Softwarefabrik“ setzte sie nach skandinavischen Vorbildern die alternativen Leitbilder einer manufakturartigen „Software-Werkstatt“ und einer User-zentrierten evolutionären Systementwicklung entgegen.[73] Ganz ähnlich sah Jack W. Reeves, der mit seinem Manifest „What Is Software Design?“ von 1992 die agile Craftsmanship-Bewegung mit angestoßen hatte, in der Software Entwicklung wesentlich ein erfahrungsbasiertes Handwerk bzw. ein „Agile Manufacturing“.[74] Auch für Jan Witt von der Digital-PCS-Systemtechnik war Software Engineering noch keine „Disziplin des anwendungs-

---

70 Shaw 1990 Prospects, 17, S. 21 ff.; Shaw 1996, Three Patterns, S. 52 ff.; Mahoney 1990, The Roots of Software Engineering; 1996; Gibbs 1994, Software’s Chronic Crisis.

71 Potts, Software Research Revisited, S. 21.

72 Cain, Coplien 1996, Social patterns, S. 259 ff.

73 Floyd 1994, Software-Engineering, S. 29 ff., 35 f.

74 Reeves 1992, What Is Software Design? S. 2 ff.

orientierten Designs“. Durch das „extrem konservative Personenbild des Software-Ingenieurs“, der nach dem Idealbild des Hardware-Ingenieurs „Vorgehensmusterzwang“ und „One-best Way“-Lösungen anstrebt, sei die nutzerbezogene Gestaltung vernachlässigt worden. Statt den „längst obsoleten Schraubenschlüssel-Ingenieurs-Träumen“ weiter zu folgen, sollte sich die Zunft „ein Beispiel an den großen amerikanischen Software-Manufakturen“ nehmen.[75]

Die folgenden Jubiläumsbetrachtungen von 1998, 2008 und 2018 sahen sich mit radikal veränderten technischen, sozialen und professionellen Umgebungsbedingungen für die Software-Entwicklung konfrontiert, die kaum noch mit der Entstehungskonstellation der Disziplin vergleichbar waren. Die 90er Jahre und das Millennium wurden zur „Era of Disruptions“.[76] Die seit 1968/69 propagierten Methodenkonzepte hatten jeweils auf einander aufgebaut und das Instrumentarium kontinuierlich ausgeweitet.[77] Doch da es die *eine* alles lösende Methodik nicht gab[78] und sich Computer-Technologien -und -anwendungen grundlegend gewandelt hatten, trat in den 90er Jahren an die Stelle der softwaretechnischen Innovationskette[79] ein interdisziplinärer Methodenpluralismus mit teils konkurrierenden, teils kooperierenden Methodenschulen: Neben eher traditionellen szientistischen und Objekt- und Pattern-orientierten Richtungen entstanden Design-, Art- und Stakeholder-fokussierte Architektur-Ansätze sowie verschiedene selbstorganisierte inkrementelle Lightwave-Methoden der „Agile Software Development“-Community. Im Software Engineering vollzog sich damit eine ähnliche Entwicklung, wie sie Mahoney bereits 1992 für die Gesamtdisziplin konstatiert hatte: Computer science has taken „the form more of a family of loosely related research agendas than of a coherent general theory validated by empirical results. So far, no one mathematical model had proved adequate to the diversity of computing, and the different models were not related in any effective way.“[80]

Dementsprechend stellte man 30 Jahre nach Garmisch bei der 20. ICSE 1998 fest, dass es keinen disziplinären „main focus“ auf der Konferenz mehr gab: „As a result, technical issues relating to software construction have spread over a broad spectrum of sciences and technologies, and software engineering is difficult to be identified as a single engineering discipline. Current software engineering is better to be understood as a multi-disciplinary collection of sciences and technologies.“[81] Das seit Ende der 80er

---

75 Witt 1994, Praxis des Software-Engineering, S. 53-55.

76 Booch 2018, The History of Software Engineering, S. 112.

77 Shapiro 1997, Splitting the Difference.

78 Brooks 1987, No Silver Bullet.

79 Raccoon 1997, Introduction. Fifty Years of Progress, S. 89 ff.

80 Mahoney 1992, Computers and Mathematics, S. 361, vgl. auch Peter Denning 1997/99, Computer Science: The Discipline, S. 24 f., der denselben Trend beim Software Engineering beobachtete.

81 Futatsugi, 2018, ICSEs before and after ICSE98, S. 94.

Jahre wiedererstandene Konzept der „Software Architecture“ diente dabei als Diskursplattform und Auffangbecken für alle ungelösten Probleme und Versprechen des Software Engineering. Angesichts der völlig veränderten Disziplin bekam die Rückbesinnung von fünf ehemaligen Teilnehmern auf die Garmisch-Konferenz vor 40 Jahren bei der 30. ICSE weitgehend nostalgische Züge. Brian Randell beklagte in seinem Statement, dass trotz großer Fortschritte vor allem bei Softwarekomponenten es noch viele sicherheitsrelevante Bereiche gäbe, in denen „the field has not advanced significantly“.[82] Bei großen komplexen Softwaresystemen stehe man noch immer vor denselben Herausforderungen, was Kosten, Zeitplaneinhaltung, Performanz und vor allem Zuverlässigkeit betreffe. Die gewaltige Ausweitung der Computeranwendung durch „useable and useful software systems, utilities and applications“, die Computervernetzung mit all ihren schädlichen Begleiterscheinungen seien dagegen bei den NATO-Konferenzen überhaupt noch nicht vorstellbar gewesen.[83]

Bei der feierlichen Begehung des 50. Jubiläums der NATO-Konferenzen und der 40. ICSE im Frühjahr 2018 wurde dann die große Distanz zwischen den gegenwärtigen und den völlig abweichenden technologischen Verhältnissen der Entstehungszeit der Disziplin überdeutlich. In den aktuellen softwaretechnischen Debatten der ICSE spielten die einstigen Methodenstreite und das Ringen um einen anerkannten Ingenieurstatus keine Rolle mehr. Spezialisierung, Methodenmix und das Verschwimmen der einst kontroversen softwaretechnischen Richtungen bestimmen weitgehend den disziplinären Diskurs. Dem noch immer nicht entschlüsselten Geheimnis der mentalen Prozesse von Softwareentwicklern versucht man in neuesten Forschungen mit Methoden des Data Mining und Deep Learning, der „Sentiment analysis“ zur Erforschung der emotionalen Einstellungen und sogar mit EEG-Vergleichen der Hirnaktivitäten von Programmierern beim Pair Programming zu erkunden.[84]

In der total veränderten Software-Landschaft versuchten die sieben anwesenden Software Engineering-Pioniere in ihren Rückblicken auf die NATO-Konferenzen eine Quintessenz aus der 50-jährigen Entwicklung zu ziehen. Für Brooks, der selbst kein Teilnehmer der Konferenzen war, gingen zwar von Garmisch wichtige konzeptionelle Anstöße für weitere Softwaretechniken aus, doch am Ende zog er das skeptische Fazit: „the term had been coined but the idea of writing programs as an engineering discipline was not generally accepted.“[85] McIlroy verwies bedauernd darauf, hin dass die „formal scientific methods“ in den USA nie als Standardpraxis anerkannt waren und überhaupt wissenschaftliche Erkenntnisse kaum absorbiert wurden. Möglicherweise sei aber die Software viel zu sehr als Gegenstand der Wissenschaft angesehen worden, die große

82 Dwyer, Gruhn, Remembering ICSE 2008, S. 143 f.
83 Randell 2008, Position Statement - How Far Have We Come?
84 Siehe die Papers im Track New Ideas and Emerging Results, ICSE–NIER 2018, S. 49, 85, 94, 105.
85 Brooks Talk bei ICSE 2018.

Hoffnung von Garmisch, die Schaffung einer „real engineering discipline“, habe sich nicht realisieren lassen. Für Bob McClure war die stark szientistische Orientierung sogar ein folgenreicher Fehler gewesen, denn die Herstellung von Software beruhe wie jedes Ingenieurprodukt auf Kompromissen, und dafür, was ein guter Kompromiss ist, gebe es keine wissenschaftliche Technik.[86]

Randell zog seine ausführliche historische Bilanz anhand seiner früheren Rückblicke und Bestandsaufnahmen, um so der Gefahr der Überlagerung von Erinnerungen nach so langer Zeit zu entgehen. Erneut kam er auf den die spätere Disziplin belastenden Gegensatz zwischen den beiden Konferenzen zurück, dem enthusiastischen Aufbruch in Garmisch und dem frustrierenden Treffen in Rom, bei dem die „academia“ das Konzept okkupierte und sich darüber zerstritt, ein Vorbote der späteren Zersplitterung. Neben der Begründung der Disziplin sah er den größten Erfolg von Garmisch in McIlroys Idee der Software-Komponenten, die sich in den Unix-Systemen allgemein durchgesetzt hätten. Er musste aber auch eingestehen, dass sich in der industriellen Softwareindustrie in einer „Darwinian-style evolution“ ohne Methoden-getriebenes Vorgehen beste Ergebnisse erzielen ließen. Als zwiespältiges Gesamtresultat sah er zwar bedeutende Fortschritte beim Software Engineering, aber mit großem Missfallen registrierte er die Fragmentierung der Softwaretechnik in „a ridiculous number of different languages, methodologies, platforms, and standards.“[87] Er schloss sich deshalb Ian Sommervilles Einschätzung an, dass Software Engineering ein „umbrella term“ für eine große Zahl und Vielfalt von Subdisziplinen geworden sei: „There are no universal software engineering methods and techniques that are suitable for all systems and all companies. Rather, a diverse set of software engineering methods and tools has evolved over the past 50 years.”[88] Dem in Garmisch formulierten ingenieurwissenschaftlichen Leitbild wehmütig nachblickend, dessen Realisierung mit zunehmender Ausdifferenzierung und interdisziplinärer Ausrichtung in kaum noch erreichbare Ferne rückte, zog Randell nun sein Fazit: „A lot of great work is done. What I am questioning how far we are from calling having something worthy from being called an engineering discipline.“[89]

Der Vergleich der Jubiläen der NATO-Konferenzen zeigt aus der Perspektive der Pioniere und Repräsentanten der Software Engineering-Community die sukzessive Entzauberung des Garmisch-Mythos. Er belegt aber auch das lange Festhalten an einem vagen Ingenieur-Leitbild und einer Industrialisierungs-Metapher, die, wie Mahoney

86 Siehe hierzu die Panel Discussion der anwesenden Garmisch-Pioniere bei der ICSE 2018 - Plenary Session – Panel.
87 Randell, Fifty Years of Software Engineering, S. 1, 4 f.
88 Sommerville 2011, Software Engineering S. 10.
89 Randells Talk bei der ICSE 2018

mehrfach dargelegt hat, nie gründlicher wissenschaftlich fundiert wurden.[90] Es setzt sich jetzt mehr und mehr die Einsicht von Brooks und Mahoney in die „uniqueness of Software Engineering“ durch, das wegen unvermeidbarer Komplexität, ständigen Wandels, Kontextabhängigkeit sowie Perspektivgebundenheit der Entwickler nie den Status einer reifen Ingenieurwissenschaft erreichen wird. Was die Pioniere als Defizite ansahen, wird in aktuellen Rückblicken geradezu als Qualitätsmerkmal gewertet, auf der Ideenreichtum und Innovation beruhen: „fuzziness“, „diversity“ und „lack of a coherent community“: „Software engineering is also a unique brand of engineering because, as a discipline, we’re not always ... disciplined.“[91] Manfred Broy sieht in der explodierenden Vielseitigkeit der Disziplin, die sich in immer weitere Subdisziplinen aufspaltet und daher heute im Unterschied zu 1968 von keinem mehr in allen Details überschaut werden könne, nun die Anzeichen für die Entwicklung zu einer Leitdisziplin für alle übrigen Ingenieurwissenschaften: „It is about to become the key interdisciplinary approach for all the established engineering techniques, and it has the most significant economic impact in engineering.“[92]

Engineering-Metapher und -Leitbild hatten eine wichtige Anstoßfunktion für die Institutionalisierung, reichten aber offenbar nicht aus, um die Entwicklung der Softwaretechnik als einen durchgängigen Verwissenschaftlichungs- und Industrialisierungsprozess zu begründen. So entstanden aus unterschiedlichen Perspektiven neben diesem Narrativ[93] weitere historische Entwicklungsmodelle: eine Kette von Programmierparadigmen und epistemischen Leitmetaphern,[94] Shapiros diskursdynamisches Modell wiederholter theoretischer Kontroversen, pragmatischer Annäherungen und Herausbildung eines technologischen Pluralismus,[95] eine innovationstheoretisch begründete Lernkurventheorie von Raccoon,[96] ein szientistischer Bürokratisierungs- und Niedergangsprozess, der durch den Aufstieg der „Agilen Softwareentwicklung“ überwunden wurde[97] und eine dialektische „Hegelian View of Software Engineering‘s Past“, bei der sich voranschreitende Phasen, Gegenbewegungen und Synthesen jeweils ablösen.[98] Die folgenden Kapitel wollen und können nicht entscheiden, welches der Narrative die Entwicklung am adäquatesten wiedergibt, dies ist angesichts der

90 Mahoney 1990, The Roots of Software Engineering, 1996, Finding a History for Software Engineering, 2004, Finding a History.

91 Erdogmus, Medvidovic´, Paulisch 2018, 50 Years of Software Engineering, S. 21.

92 Broy 2018, Yesterday, Today, and Tomorrow. 50 Years of Software Engineering, S. 39, 42.

93 Shaw, 1996, Three Patterns that help explain the development; Cusumano, 1991, Factory Concepts and Practices.

94 Brooks 1987, No Silver Bullet; danach Pflüger 2004, Writing, Building, Growing.

95 Shapiro 1997, Splitting the Difference.

96 Raccoon 1997, Introduction. Fifty Years of Progress.

97 Cockburn 2004, The End of Software Engineering, S. 14-18; DeMarco 2009, Software Engineering.

98 Boehm 2006, A View of 20th and 21st Century Software Engineering.

unzureichenden empirischen Aufarbeitung der Disziplingeschichte seit den 80er Jahren auch noch nicht möglich. Sie möchten aber durch Einbeziehung der softwaretechnischen Methodenentwicklung vor den NATO-Konferenzen stärker als bisher den diskontinuierlichen Lernprozess herausarbeiten, in dem frühere Erkenntnisse über die Grenzen der Engineering-Metapher und erste Ansätze zu iterativen und evolutionären Designmethoden aus dem Blick gerieten und Jahrzehnte später erst wieder neu entdeckt werden mussten.

2 Software-Manufakturen im militärischen Contractor-Sektor unter dem Einfluß von Operations Research und Systems Engineering

Die Sichtweise von den NATO-Konferenzen als Epochenwende der Software-Entwicklung wurde bereits von Teilnehmern durch den Rekurs auf wirtschafts- und universalhistorische Entwicklungsmodelle untermauert. So bedeutete für Dijkstra die Garmisch-Konferenz beim Programming „the End of the Middle Ages" und Beginn des „Age of Enlightment" der Softwaretechnik, aber auch den Übergang „from craft to scientific discipline" bzw. von der vorindustriellen „guild " zur „production line".[99] Auch andere Teilnehmer griffen auf die Entwicklungsstufen der industriellen Produktion zurück, um den angestoßenen Transformationsprozess der Software-Produktion zu kennzeichnen. Dabei fällt auf, dass man auf der einen Seite den historischen Vorbildern folgte, sich auf der anderen aber durch einen ruckartigen Aktionismus von ihnen löste. So propagierten McIlroy die „industrialization" der Software und eine direkte Ablösung der „crofters" durch „mass-production techniques" nach dem Modell von Ford und John N. Buxton den Übergang von der „Cottage Industry" zum „Heavy Engineering".[100] F. L. Bauer beklagte 1967, die Software „is not made in a clean *fabrication process*, which it should be."[101] Ihm ging es daher darum, „to *switch* from home-made software to manufactured software, from tinkering to engineering" bzw. zu einer „modern *software plant*".[102] Auch ihm erschien die Überwindung der „ ‚software' crisis of the sixties" durch ein planvolles „top-down program development" in einem „real industrial or engineering approach" als eine „*revolution* in programming".[103] Es gibt in diesen Entwicklungsmodellen wie auch in den meisten späteren historischen Abrissen keine Übergangsperiode, wie sie Adam Smith, Charles Babbage

---

99 Dijkstra, 1977, Programming: from craft to scientific discipline; ders. 1979, Software Engineering, S. 442 ff.; Valdez 1988, A Gift from Pandoras Box, S. 175; MacKenzie 2001, Mechanizing Proof, S. 107.

100 Naur, Randell 1969, Garmisch-Report, S. 138 f.; Mahoney, Michael (1990): The Roots of Software Engineering, S. 7 f.; Buxton 1978, Software Engineering, S. 24 f.

101 Bauer 1967, zit. in Bauer 1993, Software Engineering, S. 259.

102 Bauer 1969, Software Engineering, S. 189; Bauer 1972a, Software Engineering, S. 533, meine Hervorhebungen.

103 Bauer 1972b, Training Future Software Engineers S. 355; Bauer, 1976, Programming as an Evolutionary Process, S. 231.

und Karl Marx mit der Manufaktur als notwendiger logischer Zwischenphase von der handwerklichen zur industriellen Produktionsweise formulierten und zwar entweder als Zusammenwirken verschiedenartiger Handwerker in einer Werkstatt oder als arbeitsteilige Kooperation gleichartiger sich spezialisierender Handwerker. Die manufakturelle Herstellung erfolgt dabei zwar arbeitsteilig, doch das Handwerk mit voll qualifizierten Arbeitskräften bleibt die Grundlage eines teamartigen Arbeitsprozesses.

Im Folgenden soll nun gezeigt werden,

- dass die Transformation der Software-Herstellung alles andere als ein „switch“ bzw. „step“ war, dass es auch gerade im Softwarebereich eine Manufakturperiode der Software-Konstruktion gegeben hat und dass diese sich zwischen 1955 und 1970 im Bereich der grosskaligen Software-Systeme im militärischen bzw. halbstaatlichen Contractor-Sektor entwickelt hat und als Produktionsform noch lange nachwirkt,

- dass auch die Software-Entwickler des Contractor-Sektors zunächst einen schnellen Einstieg in die ingenieurmäßige industrielle Software-Entwicklung erhofften, aber sehr bald die Besonderheiten des Produktes Software erkannten, die auch in arbeitsteiligen Entwicklungsprozessen einen hohen Bedarf an Kooperation und Kommunikation erforderlich machten,

- dass sich der Verzicht auf eine rein hierarchische Ablaufsteuerung des Software-Entwicklungsprozesses zugunsten einer arbeitsteiligen teamartigen Kooperation in den Modellierungswerkzeugen niederschlug, so dass in dieser Periode Phasenmodelle entstanden, die sich deutlich von den rigiden Wasserfall-Modellen der 70er und 80er Jahre unterschieden.

2.1 Das Luftraumüberwachungssystem SAGE: Ein Pionierprojekt für das Engineering-Konzept in der Softwareproduktion

Das Ideenreservoir des Software Engineering reicht bereits zurück in die Anfänge des modernen Computers. John von Neumann entwickelte bereits 1946/47 ein 6-Phasenmodell des ‚Programming’ und ging dabei, ähnlich wie gleichzeitig Alan Turing, von einem arbeitsteiligen Entwicklungsprozess aus. Zusammen mit Herman H. Goldstine führte er „flow diagrams“ nach dem Vorbild der Process Charts im Industrial Engineering ein. George R. Stibitz modellierte schon 1947/48 die Programm- und Rechnerabläufe als ein hierarchisches Ebenenmodell und versprach sich davon eine Erleichterung arbeitsteiliger Entwicklungs-, Test- und Änderungsprozesse. Und Alwin Walther stellte 1946/52 wohl als erster eine direkte Parallele zwischen der Programmherstellung und der Planung eines Fabrikationsprozesses her. Über derartige rein gedankliche Analogiebildungen gingen Wilkes, Wheeler und Gill mit ihrem Konzept einer Programm-Montage aus Standard-Subroutinen hinaus. Sie übertrugen 1951 die

„modular design philosophy“ von der Elektronik- auf die Software-Konstruktion. Doch selbst auf dieser niedrigen Stufe der Baustein-Montage sahen sie bereits das Problem der Ermittlung des „best way to construct subroutines“ und der Auswahl der am besten geeigneten Varianten.[104]

Einen wirklichen Durchbruch erlebten Engineering-Konzepte im Softwarebereich aber erst Mitte der 50er Jahre im Rahmen der von Software-Contractors entwickelten großen Softwaresysteme im Militär- und Luftfahrt-Bereich. Die zwischen 1952 und 1961 entstandenen Betriebs- und Auswertungsprogramme für das Luftraumüberwachungssystem SAGE waren mit nahezu einer halben Million Befehle sprunghaft in völlig neue Größenordnungen vorgestoßen. Als „SAGE program contractor“ fungierte zunächst das MIT-Lincoln Laboratory, dann eine Abteilung der RAND Corporation, aus der 1956 die System Development Corporation (SDC) hervorging.[105] Um den Umfang und die Komplexität der Aufgabe zu bewältigen, griffen die Projekt-Verantwortlichen auf das in der Hardware-Entwicklung eingesetzte Engineering-Methodeninstrumentarium und Projektmanagement zurück. In dem direkten Methodentransfer von der Hardware- in die Software-Konstruktion sah Herbert D. Benington im Nachhinein den entscheidenden Erfolgsfaktor: „It is easy for me to single out the one factor that I think led to our relative success: We were all engineers and had been trained to organize our efforts along engineering lines.“[106] Der Methoden-Import bestand vor allem in Ansätzen des Operations Research und des Systems-Engineering, die seit ca. 1943 in den Bell-Laboratorien zur Steuerung komplexer nachrichtentechnischer Entwicklungsvorhaben entstanden war. Hinzu kamen aus dem Taylorismus stammende Instrumente des Industrial Engineering wie Flowcharts und Gantt Diagrams zur Verfolgung der definierten "milestones".

Aus diesem Methoden-Mix ging das erste Vorgehensmodell für den Entwicklungsprozess von Large-scale Software-Systems hervor. Das bekannte 9-Phasenmodell wurde 1956 im Wesentlichen vom „head of programming“ im Lincoln Lab des MIT John F. Jacobs konzipiert. Es wird meist allein dem Gesamtprojektleiter Benington zugeschrieben, da der das Konzept im Juni 1956 auf einem Symposium des Office of Naval Research (ONR) über „programming methods“ zuerst vortrug und es erneut in der SAGE-Nummer der „Annals of the History of Computing“ präsentierte.[107] Die detaillierte Entstehungsgeschichte in Jacobs' SAGE-Erinnerungen, die den Eindruck eines von Anfang an konsequent durchgeführten wasserfallartigen Entwicklungsprozesses in

---

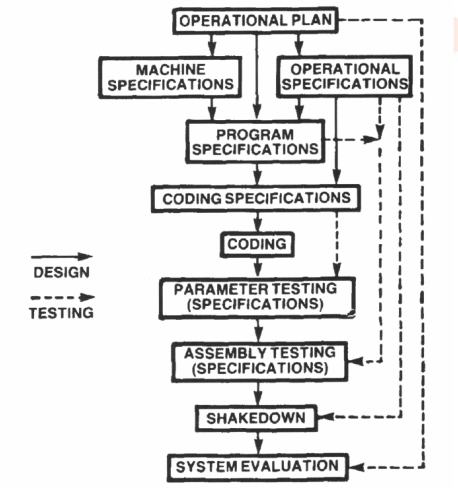
104 Siehe die Belege zu diesem Absatz in Hellige 2003, Zur Genese des informatischen Programmbegriffs, S. 57 ff.; Hellige 2004b, Die Genese von Wissenschaftskonzepten, S. 423 ff.

105 Siehe hierzu Baum 1981, The System Builders, Kap.1.

106 Benington 1983, Production of Large Computer Programs, S. 351, meine Hervorhebung.

107 Benington 1956, Production of Large Computer Programs, Wiederabdruck 1983.

Beningtons Darstellung korrigierte, wurde dagegen kaum zur Kenntnis genommen.[108] Dabei entsprach das von Benington vorgestellte Vorgehensmodell nach Jacobs in vielen Punkten nicht dem wirklichen historischen Ablauf, sondern stellte ein normatives Idealmodell dar, das erst mehr als zwei Jahre nach Beginn der SAGE-Software-Entwicklung am Lincoln Lab in einer kritischen Phase des Gesamtprojekts entstand. Mit ihm sollte der teilweise verschlungene Lernprozess eines derart umfassenden und komplexen Software-Systems mit Blick auf die Fachöffentlichkeit nachträglich begradigt und vor allem auch gegenüber den militärischen Auftraggebern gerechtfertigt werden. Denn die waren wegen der Kostenexplosion und Zeitüberschreitung bei der Software-Entwicklung äußerst beunruhigt und erwogen bereits eine grundlegende Neuorganisation.[109]



Das 9-Phasenmodell von John F. Jacobs und Herbert D. Benington von 1956

Den Ausgangspunkt des Entwicklungsprozesses bildete nach Benington eine weitgehend Hardware-orientierte Modularisierung des Gesamtprogramms in „large decentralized programs“ für Input, Output, Bookkeeping, Control und Processing, die ihrerseits nach den „data-processing functions“ in einzelne Blöcke unterteilt wurden. Der Entwicklungprozess wurde danach dann in neun Phasen unterteilt. Am Beginn stand der von den „system engineers“ und den Anwendern gemeinsam zu entwickelnde „general operational plan“. Aus dessen „requirements“ wurden die „machine“ und „operational specifications“ abgeleitet. Diese betrachteten die Computer, Terminals und Systemprogramme noch als reine „blackboxes“, d.h. als abstrakte Funktionen, die erst in der folgenden Stufe in detaillierten Programm-Spezifikationen konkretisiert wurden. Die darauf in quasi-industrieller Arbeitsteilung produzierten „component subprograms“ sollten anschließend für sich getestet und dann stufenweise zu immer größeren „subassemblies“ und schließlich zum „mainprogram“ zusammenmontiert

108 Jacobs 1986, The SAGE Air Defense System, S. 108 ff., 165 ff., 180-201.

109 Zur Entstehung der MITRE Corporation siehe Redmond, Smith 2000, S. 404 ff., 418 ff.

werden. Am Ende stand der „shake down“, der integrierte Test von Hard- und Software im „operational environment“.

Das am industriellen Fertigungs- und Montageprozess orientierte Vorgehensmodell hätte, so Benington, vor allem darauf gezielt, den Kommunikationsaufwand zwischen den Programmierern zu reduzieren und den Einsatz einer großen Zahl von „relatively inexperienced programmers“ zu ermöglichen: „This considerably simplifies the design problem; after the blocks have been documented, groups of programmers can be assigned to each part with the assurance that little communication between these programmers will be necessary.“[110] Der vorgestellte Entwicklungsplan entsprach so einem relativ starren Top-down-Ansatz und einem rigiden Wasserfallmodell, mit dem auch die „nostalgic view“ von Computerprogrammierern als einer einzigartigen Profession überwunden werden sollte. Benington interpretierte den SAGE-Software-Entwicklungsansatz nachträglich noch als Small-Team- bzw. Chief-Programmer-Konzept mit strukturiertem Ansatz. Er sah sich selber als „chief engineer who was cognizant of these activities and responsible for orchestrating their interplay. In other words as engineers, anything other than structured programming or a top-down approach would have been foreign to us.“ [111]

Die selbstkritischen Rückblicke von Jacobs und Benington drei Jahrzehnte später geben dagegen ein ganz anderes Bild von dem „disciplined approach“ der SAGE-Softwareentwicklung. Danach hatte man anfangs keine realen Vorstellungen von Umfang, Dauer, Kosten und organisatorischem Ablauf des Gesamtvorhabens. Da man beim Gesamtvorhaben zunächst von einfachsten Konfigurationen ausging unterschätzte man völlig das Ausmaß der militärischen Anforderungen und damit die Programm-Größe und -Komplexität.[112] So glaubte man nach allerersten Einschätzungen, dass 10 bis 15 Programmierer das Programm nach den „Operational Specs“ produzieren könnten. Nach und nach musste man registrieren, dass allein das Betriebssystem auf 100.000 Befehle anwuchs und auch die „utility and instrumentation programs“ einen ähnlichen Umfang annahmen, so dass am Ende über 800 Programmierer eingesetzt wurden, zu denen noch 1400 Hilfskräfte hinzukamen. Auch die Kosten stiegen von dem ursprünglich geplanten unter 1$ pro Befehlszeile auf mehr als das 50fache.[113] Der Gesamtaufwand für die Erstellung, Wartung und Dokumentation der Software war so am Ende genauso hoch wie der für die gesamte „specialized hardware“ im SAGE-System.[114]

---

110 Benington 1983, Production of Large Computer Programs, S. 356.

111 Ebda., S. 351 f.

112 Sackman 1967, Computer, System Science and Evolving Society, S. 267.

113 Benington 1983, Production of Large Computer Programs, S. 351, 356 f.; Jacobs 1986, The SAGE Air Defense System, S. 108, 180.

114 Jacobs 1986, The SAGE Air Defense System, S. 130, 155.

Vor allem aber musste man einsehen, dass das Wissen über das System bei der Aufstellung des „Operational Plan“ und der „hierarchy of specifications“ noch gar nicht vorhanden war, sondern sich erst im Laufe der parallel aufenden Entwicklung von Hardware und Anwendungskonzeptionen herausbildete.[115] An der fundamentalen Wissenslücke änderte auch der erstmalige Einsatz von Prototyping nichts, denn beim „prototype program“ vom „Experimental SAGE Sector“ (ESS) waren viele Skalierungs- und Integrationsprobleme noch gar nicht aufgetreten.[116] Zudem erfolgte der Entwicklungsprozess bei dem Prototypen und auch sonst in „an entirely iterative manner“: [The] development of the new concept and its embodiment in the Cape Cod System relied heavily on iterative cycles of experiment-learn-improve.“[117] Unter diesen Bedingungen war an eine konsequente Durchführung des von Benington propagierten „disciplined approach“ gar nicht zu denken, der „Operational Plan“ selber musste vielmehr immer wieder an die Lernerfahrungen angepasst und die sauber konzipierte Stufenfolge durchbrochen werden. Erst aus der Rückschau erkannte man, dass sich wesentliche Entscheidungen überhaupt nicht vorab regulieren und top-down organisieren ließen.[118]

So musste man die Erfahrung machen, dass sich die anfängliche Aufteilung der Programmentwicklung nach Hardware-Komponenten im Nachhinein als Sackgasse erwies: „As we progressed through the component/subsystem development in SAGE and finally the overall system development, people gradually gained an understanding of ‚the system.’ They tended to stop thinking about vacuum tube characteristics and to start thinking about air surveillance.“ Man machte hier bereits die Erfahrung, dass die Zerlegungskriterien eines Softwaresystems stärker variieren können und somit selber ein Designproblem bilden. Entgegen dem ursprünglichen Plan stellten sich auch bei der Entwicklung große Unterschiede zwischen den einzelnen Subsystemen heraus, so dass hier keine halbwegs zuverlässige Regulierung und Zeitplanung möglich war. Überhaupt ließ sich die intendierte strikt arbeitsteilige Entwicklung von „decentralized programs“ bei einem Real-Time-System mit hoher Interdependenz der Einzelprozesse nicht aufrechterhalten. Änderungen am „Operational Plan“ ergaben sich auch aus der Einbeziehung der anfangs nicht berücksichtigen Waffensysteme und vor allem der nachträglichen Regelung des militärischen Kommandosystems.[119]

---

115 Ebda., S. 109 f.,165.

116 Benington 1983, Production of Large Computer Programs, S. 351; Redmond, Smith, 2000, From Whirlwind to MITRE, S. 374-378.

117 Wieser 1985, The Cape Cod system, S. 364; vgl. Schmidt 2011, Cooperative Work S. 316 ff.

118 Jacobs 1986, The SAGE Air Defense System, S. 165 f.; Benington 1983, Production of Large Computer Programs, S. 352 f.

119 Jacobs 1986, The SAGE Air Defense System, S. 130, 165.

Als besonders großer Fehler erwies sich die Verschiebung der Integrationsentscheidungen auf die Endphase der Systementwicklung, denn die Erstellung des viele Funktionen integrierenden „master computer program“ machte eine von Anfang an integrale Sicht der „subsystems as a single system“ erforderlich, was sich als weitaus komplexer erwies als man es vom Hardware-Engineering her kannte. Denn der „integration process“ führte nicht nur zu Veränderungen am Programmsystem, sondern erzwang auch Abwandlungen der Hardware in Form von neuen Peripheriegeräten und modifizierten Input-Output-Systemen. Die Systemintegration unter Einbeziehung der Befehls-und Kommando-Systeme führte im Herbst 1956 zu einer ersten umfassenden Revision der SAGE-Software. Es wurde sogar der Vorschlag eines völlig neuen Software-Systems erwogen, doch man beließ es bei einem „redesigned and rewritten computer program“, das alle technischen Neuerungen integrierte.[120] Eine strikte Regulierung des Entwicklungsablaufes, wie sie Benington vorschwebte, hätte derartige Chancen für Innovationen erschwert oder gar verhindert.[121]

Als besonders dysfunktional erwies sich schließlich das Bestreben, die Kommunikation zwischen den beteiligten Entwicklergruppen durch das Vorgehensmodell vorab zu kappen. Denn sowohl bei dem alles integrierenden Kontrollprogramm wie auch bei beim „command system development“ war ein hoher, Abteilungen und Disziplinen übergreifender Abstimmungsaufwand erforderlich. Es handelte sich hierbei nämlich nicht mehr nur um ein rein technisches „systems engineering in the small“, sondern um ein „systems engineering in the large“, bei dem neben technischen Entscheidungen auch „sociological, financial, budgetary, and other such considerations“ in die Systemgestaltung eingingen.[122] Im Laufe des Entwicklungsprozesses traten so die Grenzen und Probleme des „Operational Plan“ immer deutlicher hervor. Denn da Änderungen an einem „component subprogram“ und die Hinzufügung neuer Komponenten eine Kette von Revisionen nach sich zog, war schließlich die Hälfte der 800 Programmierer nur noch mit der Programm-Anpassung beschäftigt. Die neuen Spezifikationen wurden jeweils durch besonders qualifizierte „ad-hoc teams“ erstellt, die direkt mit dem von Jacobs für Koordinations-Meetings geschaffenen „Systems Office“ zusammenarbeiteten. Die Probleme des gewählten Vorgehensmodells führten bereits während des Projektes zu Modifikationen des Projektmanagements. Man ließ die zuvor gekappte Kommunikation zwischen den Programmierteams wieder zu, denn nur so ließ sich der Änderungsbedarf in Grenzen halten. Die Fertigstellung der Software verzögerte sich dadurch um ein weiteres Jahr und brachte den Fahrplan des Gesamtprojektes durcheinander.[123]

---

120 Redmond, Smith 2000, From Whirlwind to MITRE, S. 418 f.
121 Jacobs 1986, The SAGE Air Defense System, S. 130, 165 f.
122 Ebda., S. 165 f., 170.
123 Ebda., S. 108, 130, 180; Redmond, Smith, S. 378 f.

Dass das seinerzeit beispiellose Software-Pionierprojekt doch abgeschlossen werden konnte, führte Benington in der Rückschau entscheidend darauf zurück, dass der im Engineering übliche „disciplined approach“ nicht eins zu eins auf die Programmherstellung übertragen worden war. Man habe nämlich erkannt, „that computer programming and the computer programmer were ‚different‘. They could not work and would not prosper under the rigid climate of engineering management.”[124] Der stark verspätete Abschluss des größten Software-Projektes seiner Zeit wurde so nur dadurch möglich, dass man ziemlich bald auf die starre Orientierung am Stufenplan des Program Production Plan verzichtete und wie bisher das „general purpose flow diagram“ als Arbeitsgrundlage und Verständigungsmittel zwischen den Entwicklergruppen verwendete. Der „Operational Plan“ selbst wurde durch den weitaus flexibleren „Operational Employment Plan“ ersetzt, der Spielraum für neue Soft- und Hardware-Spezifikationen und für die abteilungsübergreifende Kooperation schuf.[125] Damit hatte sich die anfängliche Hoffnung, „that there was a Ford in our future“ zerschlagen, statt einer industriellen Software-Produktion blieb es bei einem teamartigen manufakturellen Produktionsprozess.[126] Das 9-Phasenmodell stellte mithin nur noch einen groben Orientierungsrahmen für die SAGE-Softwareentwicklung dar, sein Erfolg beruhte letztlich gerade auf seiner ständigen Durchbrechung und Anpassung an neue Gegebenheiten. Es kann daher nicht wie vielfach üblich als der erfolgreiche Beginn der Geschichte des Wasserfallmodells gewertet werden.[127]

Dennoch erlangten die Software-Entwicklungsmethoden des SAGE-Projektes, die in der Folgezeit durch die System Development Corporation ausgebaut und verfeinert wurden, Vorbildcharakter für die folgenden Large-Scale-Informationssysteme von Militär, Raumfahrt und Airlines. Auch diese versuchten der durchweg beklagten Komplexitätssteigerung bei Online-, Real-Time- und Time-Sharing-Systemen mit Phasenmodellen, Meilensteinen und der Aufteilung der Riesenprogramme in arbeitsteilig zu bearbeitende „blocks“ oder „modules“ zu begegnen. Doch ging man bei der Strukturierung von Produkt und Prozess meist nicht so straff nach dem Muster des Industrial Engineering vor, wie es im SAGE-Projekt intendiert war. Auch die Literatur über Software-Entwicklungsmethoden empfahl in der Regel keine rigide Arbeitsteilung und Ablaufplanung. Dies belegen herausragende Traktate zu Methoden des „Systems Engineering“ und der Programm-Entwicklung in diesem Zeitraum.

---

124 Benington 1983, Production of Large Computer Programs, S. 351.

125 Jacobs 1986, The SAGE Air Defense System, S. 130, 185.

126 Ebda., S. 197; vgl. hierzu auch Mahoney 2004, Finding a History for Software Engineering.

127 Vgl. u.a. Boehm, 2006, A View of 20th and 21st Century Software Engineering, S. 13; Kneuper 2017, Sixty Years of Software Development S. 43 f.

2.2 Strukturierungskonzepte und Managementmethoden für Software-Manufakturen in der SAGE-Nachfolge

Schon das erste Lehrbuch über Systems Engineering, das 1957 von Harry H. Goode und Robert E. Machol vor allem auf der Basis von Erfahrungen bei den Bell-Labs und AT&T vorgelegt wurde, bot ein idealtypisches Phasenmodell des „Systems Design" für Computer- und Automatisierungsvorhaben. Doch die Autoren verstanden die „well-defined phases" nicht als eine streng chronologische Ordnung, denn in der Praxis sei eine Phase „often unrecognizable until it has past".[128] Als Vorbild für ihr Darstellungsmodell wählten sie dementsprechend nicht wie spätere Phasenmodelle das bereits 1911 von Henry Lawrence Gantt eingeführte Balkendiagramm, sondern ein Flow Diagram nach dem Vorbild der elektrischen Stromlaufpläne. In dieses wurden auch Systembestandteile und Lösungsalternativen integriert, der Übergang vom Phasenmodell zum Aufgabenplan war also fließend. Ähnlich wie Goode und Machol kamen wenig später auch die leitenden Systemingenieure William C. Tinus und Henry G. Och aus den Bell-Labs zu dem Ergebnis, dass zur Beherrschung der wachsenden Komplexität von Automatisierungs- und Informationssystemen zwar "mileposts" für die Hauptentwicklungsschritte und definierte Phasen sowie Zeitpläne für die einzelnen Einheiten erforderlich seien, dass sie in der Praxis jedoch an Grenzen stoßen: „Generally, these steps follow in chronological order. However, many of the detailed steps actually overlap and even recur during the development program." Ebenso sei zwar eine klare Strukturierung in Subsystems zwar wichtig, um die Kontrolle über den Arbeitsprozess zu behalten, doch müsste wegen der Wechselbeziehungen zwischen den Teilkomponenten und der dafür nötigen ständigen Kommunikation der Entwickler die Arbeitsteilung immer wieder durchbrochen werden.[129] Bei der IBM Federal Systems Division, die 1957/58 das Betriebssystem für das Project Mercury entwickelte, war man nach Gerald M. Weinberg sogar der Meinung, dass „waterfalling of a huge project was rather stupid, or at least ignorant of the realities" und praktizierte zu diesem frühen Zeitpunkt bereits ein „incremental development".[130]

Auch die ab 1960 von Methodik-Spezialisten im SDC-Umfeld und von Projektmanagern großer militärischer Softwaresysteme formulierten Software-Entwicklungslehren verstanden Phasenmodelle in der Regel nur als Leitlinien für die Programm-Entwicklung, an die man sich im konkreten Entwicklungsablauf nicht sklavisch halten dürfe. Dies zeigten gleich die beiden ersten State-of-the-Art-Reports über Managementtechniken für die Programmentwicklung von Realzeitsystemen, die von William A. Hosier und Thomas A. Holdiman. Beide Texte wurden, obwohl in einschlägigen Fachzeitschriften publiziert, von der frühen Software-Engineering-Community gar nicht zur Kenntnis

---

128 Goode, Machol 1957, System Engineering, S. 35 ff.; Zitat S. 35.

129 Tinus, Och 1959, Systems Engineering for Usefulness and Reliability, S. 8, 10.

130 Larman, Basili 2003, Iterative and Incremental Development, S. 3

genommen. Erst später strich Barry W. Boehm heraus, „how many of today's software engineering hot topics had already been understood in 1961 in Bill Hosier's IRE article.“[131] Hosier bündelte in seiner Methodenlehre von 1961 für die „system genesis“ und „program genesis“ von digitalen Realzeitsystemen seine langjährigen Erfahrungen in der SAGE-Programmentwicklung am Lincoln Lab sowie im System Engineering von Aircraft Control Systems und des „Anti-Missile Systems Ballistic Missile Early Warning System“ (BMEWS) bei Sylvania. Er legte besonderes Gewicht auf eine enge Kooperation der verschiedenen an der Systementwicklung beteiligten Fachvertreter. Vor allem in dem kleinen und sehr kompetenten Team für den „tentative design plan“ sollten alle Hardware-Designer und Programmer „be able to communicate which each other“, notfalls unter Zuhilfenahme von „one or two men of more general system experience to bridge these gaps and to help resolve differences of viewpoint.“[132]

Auch das Programm selber war für Hosier in erster Linie ein „product of team effort“. Daher hielt er es auch bei arbeitsteiligem Vorgehen für unabdingbar, „that each programmer must clearly see his role in the system and understand all the rules.“ Aber ähnlich wie später Frederick Brooks, forderte er, dass selbst bei sehr großen Programm-Systemen der erste Entwurf von einem kleinen Team oder einer Koryphäe erstellt werden sollte, denn nirgendwo sei das Sprichwort „Viele Köche verderben den Brei“ mehr angebracht als hierbei. Doch auch auf den folgenden Stufen der Programmentwicklung wollte Hosier die Kooperation und Kommunikation keinesfalls ausschalten, um die Gefahren des Kirchturmhorizontes zu vermeiden: „Parochialism is a common fault both of subdivisions of a system programming team and of the teams as a whole.“ Er plädierte deshalb für mehrmalige Treffen der Teams mit „external consultants“, wobei sich abwechselnde „selected programmers review their progress and problems as speakers to the group.“ Der Denkprozess beim Entwurf von Systemprogrammen sollte dabei vor allem graphisch visualisiert werden, um „intact from one mind to another“ transferiert zu werden.[133]

Als Grundlage der Organisation und Kontrolle des Ablaufes des „real-time program development“ diente Hosier eine „over-all flow chart“ mit den Hauptphasen des Prozesses. Obwohl er sich im Begriff und in der Notation an dem tayloristischen Instrument der „flowchart“ orientierte, ähnelte seine Darstellungsform mit den verschiedenen „stages“, Aufgabenpaketen und Zuständigkeitsbeziehungen eher einem Netzplan. Er verstand die Flowchart auch nicht im Sinne eines „paragraphing of the whole program“, d.h. als eine hierarchisch kontrollierte sequenzielle Ablaufplanung. Denn

131 Brief Boehms an Benington, zit. in Benington 1983, Production of Large Computer Programs, S. 351.
132 Hosier, W. A. (1961): Pitfalls and Safeguards, S. 101.
133 Ebda, S. 108f., 114.

„the entire structure can not be 100 per cent foreseen, and must evolve through considerable experiment. But if a sound beginning is made, and if it is kept clear what is frozen and what is fluid, it will greatly lessen the conflicts and re-work that always accompany the assembly process.“ [134] Die Programm-Entwicklung ist nach Hosier ein sowohl durch angemessene Meilensteine gerichteter als auch ein teilweise offener evolutionärer Prozess, so dass es bei ihm zu einem Nebeneinander von dem offiziellen und dem internen Fahrplan kommt. Das koordinierte Vorgehen soll eher über gemeinsame "tools and procedures for program production and testing" erreicht werden als durch strikte Zeitpläne. Er wollte die Kreativität der Programmer nicht durch ein Projektmanagement à la SAGE gängeln, aber doch durch ein Bündel von Team-orientierten Instrumenten des Systems Engineering sowie durch standardisierte „programming techniques“ lenken, die er im Report jedoch nicht behandelte. Denn er kannte die Psyche der Programmer nur zu gut: „Programmers differ little from engineers, in general, in their reluctance to stop tinkering with and improving their creations. This is a laudible trait; but as delivery dates approach and time grows short, it has to be restrained.“[135]

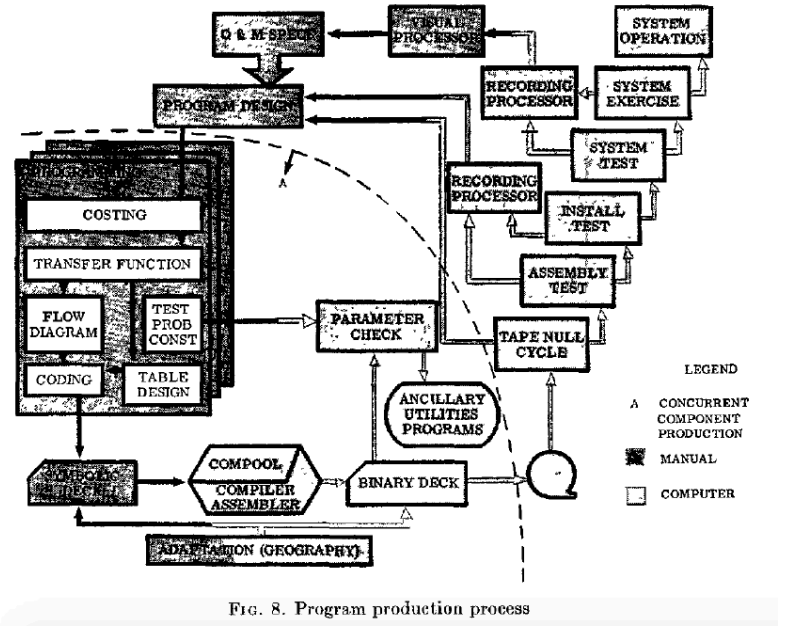


FIG. 8. Program production process

Modell des Program production process von Thomas A. Holdiman (1962, S. 396)

Auch Thomas A. Holdiman verarbeitete in seinem Bericht von 1962 über „Management Techniques for Real Time Computer Programming“ im ACM Journal Erfahrungen als Projektmanager bei General Dynamics/Electronics und bei der aus dem Lincoln Lab entstandenen MITRE Corporation. Sein Traktat, der auf den ersten Blick mehr an

134 Ebda, S. 110.
135 Ebda, S. 114.

Benington als an Hosier anknüpfte, richtete die Programm-Erstellung ganz auf den industriellen Fertigungsprozess aus. Er begriff den „program production process“ ausdrücklich als eine in definierten Phasen ablaufende Komponenten-Fertigung und eine „*program assembly*“: „Programming the central computer in a large system amounts to custom *fabricating* a number of ‚think‘ pieces, each created by a different craftsman and fashioned to fulfill a broad program design. The separate pieces (*program components*) are then united into a coherent whole [...].“[136] Auch Holdimans Modelldarstellung der „program assembly“ lehnte sich eng an die Engineering-Metapher von Benington an, seine Darstellungsmodelle für das „program component design“ und den gesamten „program production process“ ähnelten so auch stark deren Wasserfallmodell. Doch bei näherem Hinsehen wird deutlich, dass Holdiman den konkreten Ablauf der Software-Entwicklung nicht als strikt arbeitsteilige Komponenten-Montage begriff, sondern ähnlich wie Hosier als einen in hohem Maße kooperativen Prozess. Denn die Programmierer müssten wegen der Wechselbeziehungen der Programmkomponenten untereinander die Prozesse der Programm-Komponenten verstehen, die mit seinem eigenen Programmteil interagieren. Darüber hinaus müssten sie einen Einblick in den Gesamtzusammenhang besitzen. Der „*component designer*“ bedürfe daher „considerable freedom for design“, um sein Produkt zu optimieren: „Part of his art is to reconcile his piece of the program with those being produced by the other programmers“.[137]

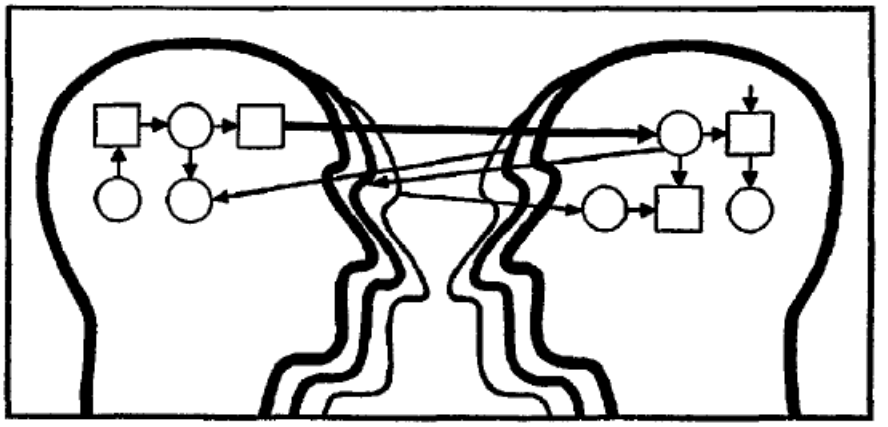


FIG. 4 Matching think pieces

Kooperatives Modellieren bei Programmierern nach Holdiman (1962, S. 393)

Demgemäß hielt Holdiman eine serielle Fertigung von Programmen für die Erstellung von Real-time-Programmen für unökonomisch, vielmehr müssten die Komponenten in „concurrency“ produziert werden, doch: „concurrency naturally demands a great deal

136 Holdiman, 1962, Management Techniques, S. 392, meine Hervorhebung.

137 Ebda., S. 393, meine Hervorhebung.

of coordination among programmers working on interacting components“.[138] Die Tätigkeit der Programmierer ist für Holdiman deshalb am Ende nach wie vor „craftmanship“ oder „art“. Die im Entwicklungsprozess verwendete Folge von immer konkreteren Modellen sieht er weniger unter dem Aspekt der Rationalisierung und Mechanisierung der Programmerstellung, als vielmehr unter dem der besseren Gesamtübersicht und der Erleichterung späterer Veränderungen: „The division of the programming task into a succession of models permits change, and even though it implies much reworking of the program from model to model, yet it maintains a working situation manageable from every standpoint.“ Leitend für die Programmorganisation wird für Holdiman daher nicht ein Wasserfall-Modell, sondern sein „Model Concept“, das mit der Formalisierung der Anforderungen und Spezifikationen beginnt und in einer Serie von „progressive program models“ schrittweise konkretisiert und verfeinert wird. Holdiman nimmt damit Ideen des „hierarchical modelling“ und der „levels of abstraction“ vorweg, die Edsger Dijkstra mit der „system hierarchy“ seines „THE Multiprogramming System“ (1968a) sowie Frank W. Zurcher und Brian Randell (1968) mit ihrem „concept of iterative multi-level modelling“ bei der IBM in der zweiten Hälfte der 60er Jahre vorstellten, die ein wesentliches Element des Software Engineering werden sollten.[139]

An die Softwareentwicklungs-Lehren von Hosier und Holdiman schlossen sich in den Folgejahren eine große Zahl weiterer Erfahrungsberichte, Designreflexionen und Projektmanagement-Betrachtungen aus dem Contractor-Sektor an. So entstand im Umkreis von SDC, der Air Force und in den Contractor-Unternehmen in den 60er Jahren eine reichhaltige Traktatliteratur, die die Erfahrungen mit dem „large-scale programming“ in den Central-Command und Control-Systemen der Air Force, der Flugüberwachung und Raumfahrt verarbeitete. Gebündelt wurden die verstreuten Aufsätze, Proceedingsbeiträge bei ACM und AFIPS, Berichte und Studien schließlich in einer von Perry E. Rosove, dem Leiter der „Advanced Systems Division“ der SDC herausgegebenen Software-Entwicklungsmethodik. Der ab 1963 entstandene, 1967 erschienene Sammelband „Developing Computer-Based Information Systems“ taucht im Unterschied zu dem viel zitierten Kurzbeitrag von Winston W. Royce von 1970 in historischen Darstellungen kaum auf, obwohl sich hierin viel grundsätzlichere Betrachtungen über Vorgehensmodelle finden und in ihm schon wesentliche Momente eines iterativen evolutionären Entwicklungsprozesses vorweggenommen wurden.

---

138 Ebda., S. 394.

139 Ebda., S. 401 f.; Larman, Basili 2003, Iterative and Incremental Development, S. 3; Kneuper 2017, Sixty Years of Software Development, S. 44 ff.

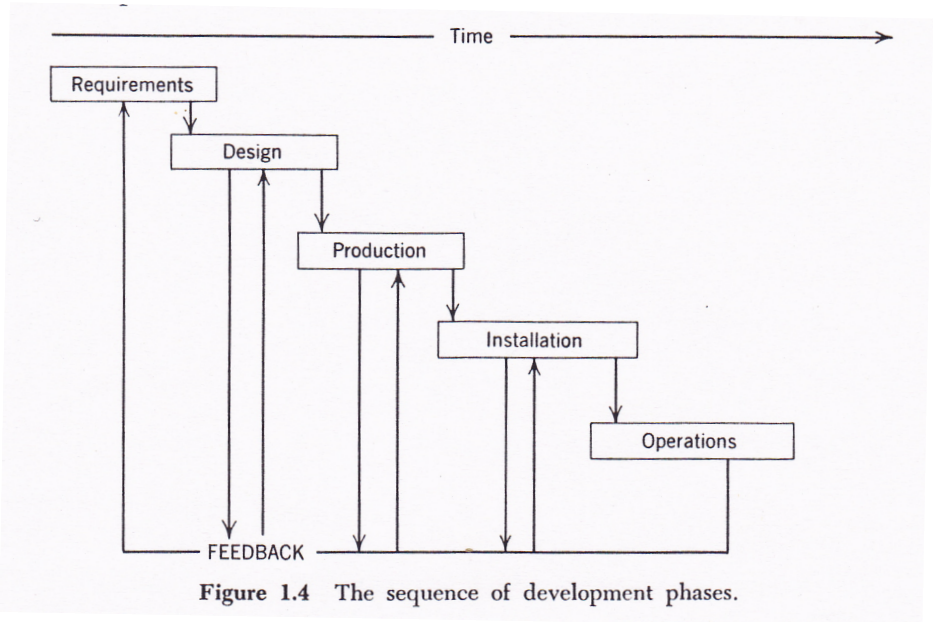


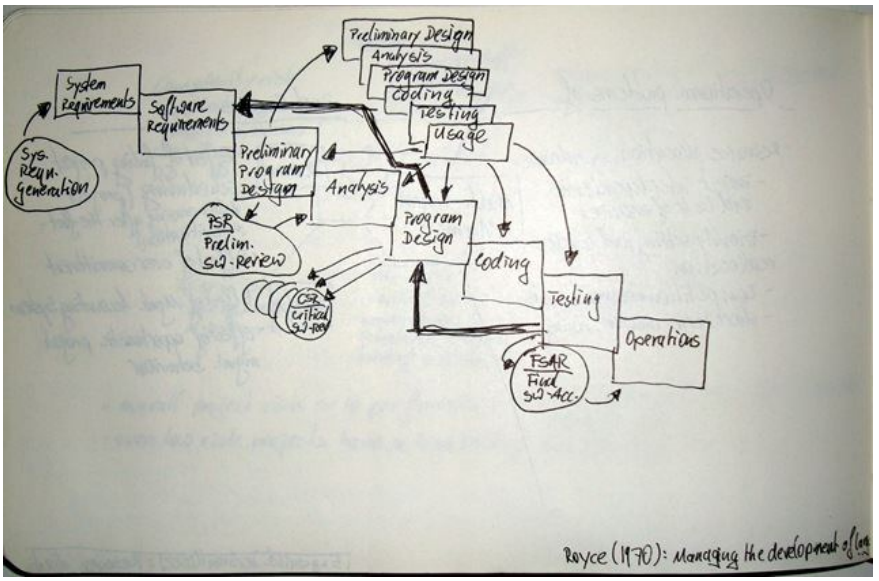
Figure 1.4 The sequence of development phases.

Rosoves Phasenmodell (Rosove 1967, Developing Computer-Based Information Systems, S. 18).

Auch für Rosove bildete ein 5-Phasenmodell den Ausgangspunkt für die Organisation des Software-Entwicklungsprozesses. Es ähnelt in der Darstellungsform infolge der direkten Anknüpfung an das Ganttsche Balkendiagramm schon sehr weitgehend dem Wasserfallmodell Barry Boehms in seiner meist rezipierten Grundform.[140] Es unterscheidet sich von diesem aber dadurch, dass der „iterative problem-solving process" Feedback-schleifen nicht nur jeweils zur vorhergehenden Stufe, sondern über den gesamten Software-Life-Cycle hinweg zulässt: „The feedback arrows indicate that the development history is a closed-loop cycle".[141] Denn Rosove hatte aus der Beobachtung der Software-Entwicklung in den großen Informationssystemen der Air Force die Erkenntnis gewonnen, dass sich die Systemanforderungen nicht ausschließlich in der dafür vorgesehenen ersten Phase festlegen ließen, sondern dass sich im Laufe der Projekte die anfänglichen „requirements" immer wieder änderten und dass sich durch Technologiewechsel selbst noch in späten Stadien neue Anforderungen ergeben konnten. Daher war für Rosove das „system design" einerseits die logische und zeitliche Stufe nach der „requirement phase", aber zugleich „a function which is carried out repetitively at different levels of a system development process".[142] Aus ähnlichen Erwägungen sah auch Winston W. Royce in seinem erweiterten Wasserfallmodell von 1970 durchgängige „design iterations" vor, während die Software Engineering Community meist nur sein vereinfachtes 7-Phasen-Modell rezipierte, das nur Iterations-

140 Boehm 1981, Software Engineering Economics, S. 36.
141 Rosove 1967, Developing Computer-Based Information Systems, S. 18.
142 Ebda., S. 18 f.

schleifen zum vorhergehenden Schritt zuließ.[143] Hier wurden Einsichten vorweggenommen, die Basili und Turner 1975 (S. 43) zur Grundsatzkritik am „top-down approach“ und zum Vorschlag des „iterative enhancement“ veranlassten.



Das von der Software Engineering Community kaum wahrgenommene erweiterte Wasserfallmodell von Winston W. Royce von 1970 (Alchetron, The Free Social Encyclopedia, 29.5.2018)

Der Hauptgrund für ein flexibles Vorgehen war für Rosove aber die durch alle Entwicklungsphasen hindurch bestehende „close relationship between the user and the software developer“. Er plädierte deshalb für ein von ihm „*evolutionary approach*“ genanntes Verfahren, bei dem sich „design und production iterations“ notwendigerweise überlappen.[144] Ein vom starren Phasenmodell abweichender „evolutionary development process“ ist nach Rosove auch erforderlich, weil, im Unterschied zur Produktion von Maschinen, die als unabhängige Elemente auf der Basis deterministischer Prinzipien gefertigt werden könnten, die Software von Informationssystemen jeweils eine nicht-deterministische „particular configuration of independent elements“ darstellten, die viel komplexerer Rückkopplungen und Abwägungen zwischen Hard- und Software-Design-Alternativen bedürften.[145] Die Methode des „evolutionary design“ wurde vom US Department of Defense aufgegriffen und allen Entwicklungsteams von Informationsverarbeitungs-Programmen für Command and Control-Systeme empfohlen. Um die Nutzeranforderungen wirklich umzusetzen, sollten die Anforderungsermittlungen

143 Royce 1970, Managing the Development of Large Software Systems, S. 4 f.; Kneuper 2017, Sixty Years of Software Development S. 45.
144 Rosove 1967, Developing Computer-Based Information Systems, S. 43 f.
145 Ebda., S. 31 ff.

nicht auf den Beginn der Entwicklung beschränkt werden: „The user of the system should participate in every step of the evolution. He is a vital part of the system and if he delegates his responsibility to an agency outside his organization, there is danger that the user will depend on automated decision aids without realizing the extent to which human judgement of operational parameters has been built into such aids by an outside developer.“[146] Auch dies Erkenntnisse, die in der Software Engineering Community erst wieder in den 80er Jahren auftauchten.

Die Strukturierungskonzepte und Managementmethoden von Holdiman, Hosier und Rosove für die Softwareherstellung erscheinen aufgrund der Vorstellung eines quasi-industriellen Fertigungs- und Montageprozesses von Komponenten bzw. Modulen auf den ersten Blick als eine Vorwegnahme der Software-Engineering-Ansätze am Ende der 60er Jahre. Doch bei näherem Hinsehen werden charakteristische Unterschiede zum Konzept eines monolithischen sequenziellen Zyklus deutlich, die sie eher als einen manufakturellen Produktionsprozess erscheinen lassen:

Charakteristika der Software-Manufakturen

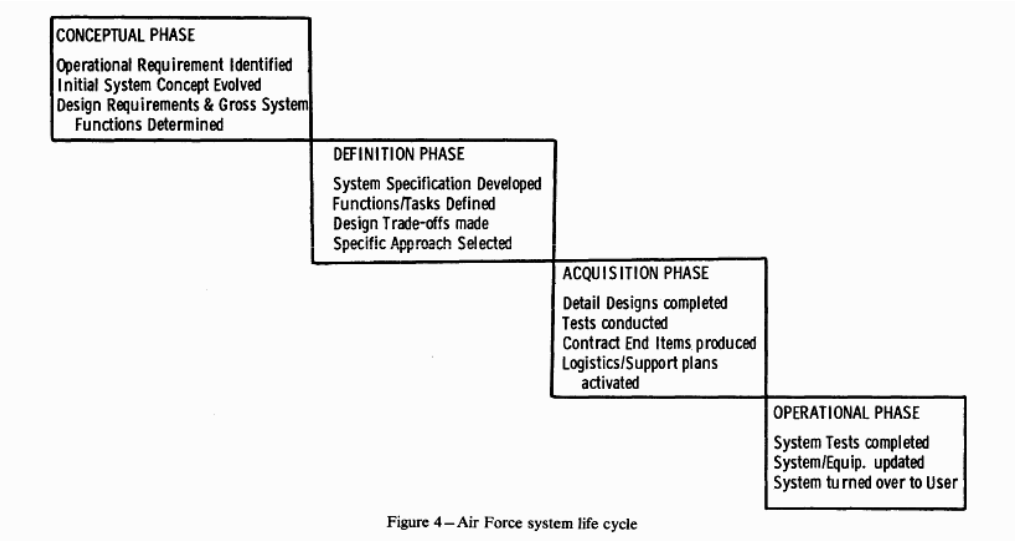
- Die Strukturanalogie zum Austauschbau wird nur angedeutet, die interdependente Aufgaben- und Problemstruktur bleibt trotz der Modularisierungsbestrebungen erhalten.
- Das Phasenmodell zielt nicht auf eine strikt ‚tayloristische‘ Arbeitsteilung, sondern bleibt eingebettet in ein aus heutiger Sicht modern erscheinendes Konzept des Concurrent Software-Development mit inkrementellen und evolutionären Ansätzen, in dem die Programmsysteme in enger Abstimmung mit den Usern evolutionär entstehen.
- Die Programmerstellung bleibt letztlich „art“, ein teamartiger Designprozeß, in dem Real-Time-Programmer „creative work in a highly specialize environment“ leisten.

Im Jahrzehnt vor den NATO-Konferenzen von 1968/69 waren so im Bereich der militärischen Programm- und Systementwicklung großer Software- und Informationssysteme eine Reihe von Design-, Projekt- und Entwicklungsmanagement-Methoden entstanden, die schon in vielem die späteren evolutionären Ansätze von Meïr M. Lehman und Laszlo A. Belady[147] (1974, 1980, 1985) sowie die ‚ganzheitlichen‘, auf Teamarbeit, Entwickler-Kreativität und Benutzerorientierung ausgerichteten Methoden der partizipativen und agilen Software-Entwicklung vorwegnahmen. Die Software-Konstruktionsmethodik dieser Zeit entspricht damit vom Typ her noch in hohem

146 Christie, Kroger 1962, Information Processing for Military Command, S. 59.

147 Belady, Lehman 1974, Programming Systems Growth Dynamics; Lehman. 1980. Programs, life cycles, and laws of software evolution; Lehman, Belady 1985, Program Evolution.

Maße den im und unmittelbar nach dem Zweiten Weltkrieg entstandenen allgemeinen Ingenieurmethodiken, dem Operations Research, dem System-Engineering, der Wertanalyse, den Prognosemethoden der RAND Corporation und anderer Denkfabriken (Delphi-Methode, Szenario-Technik) sowie den Concurrent-Engineering-Konzepten in der deutschen und amerikanischen Rüstungswirtschaft. Sie alle waren, obwohl weitgehend aus militärischer Umgebung hervorgegangen, betont teamorientiert, sie standen in der Tradition der interdisziplinären Expertenteams des New Deal und der fach- und professionsübergreifenden Braintrusts und Thinktanks der Kriegs- und Nachkriegszeit, und diese Tradition wurde in den großen Softwareprojekten des militär-industriellen Komplexes noch lange Zeit beibehalten.[148]



Ikonisches Wasserfallmodell des System Life Cycle der US Air Force
nach Milton V. Ratynski, 1967, S. 37

Doch Ende der 60er Jahre wurde dieses Team-orientierte Software-Manufacturing abrupt gestoppt, denn auch der finanziell und institutionell privilegierte „User" Militär geriet angesichts steigender Softwareausgaben unter stärkeren Kostendruck. Auf den jährlichen ACM-Konferenzen und den AFIPS-Treffen häuften sich ab 1966 deshalb Klagen über die teuren „one-of a-kind software systems", die Unzuverlässigkeit, Kostenexplosion und den Eigensinn der Programmierer, die stark an die Äußerungen auf den NATO-Konferenzen von 1968/69 erinnern. Man forderte nun eine striktere Reglementierung des „system life cycle" durch verbindliche „milestones", ständige Erfolgskontrollen und „computer program management techniques" mit dem Ziel „reuse", „compatiblity" und „transferabilty". In diesem Kontext setzte sich seit 1967 das keine Rekursionen zulassende Wasserfallmodell immer mehr durch. Durch eine

148 Warren Weaver sah in dem „mixed-team approach of operations analysis" neben dem Computer die zweite große Errungenschaft des Weltkrieges zur Lösung von Komplexitätsproblemen in der Wissenschaft (Weaver 1948, Science and Complexity).

weitgehende Angleichung des „'software' management process" an den Hardware-Entwicklungsprozess sollte sich das Management endlich die Kontrolle über den Produkt-Lebenszyklus auch im Softwarebereich sichern.[149] Der US-Delegierte im NATO Science Committee Isidor Isaac Rabi initiierte deshalb Anfang 1967 die „Study Group Computer Science", die den Anstoß für die beiden Software Engineering-Konferenzen im Folgejahr gab.[150] Das Leitbild „Software-Manufaktur" wurde damit im Contracting Sector endgültig von den Leitbildern der industriellen Softwareproduktion abgelöst. Die „component assembly", der „orderly process of software production" und ab 1971 bei SDC die „software factory" waren nun nicht mehr unverbindliche Metaphern, sondern strikte Leitlinien für die Software-Entwicklung. Mit der zunehmenden Durchsetzung des „Software Engineering" gerieten die Design-Traktate der Manufaktur-Periode bis auf einige wenige sehr bald in Vergessenheit.

3 Strukturierungs- und Designkonzepte im Bereich der Corporate Software: Software Engineering versus Software Architecture

3.1 Die „Software Crisis-Debate" und die Anfänge des Begriffs „Software Engineering" von 1960-1967

Auch im privatwirtschaftlichen Corporate-Sektor kam es im Laufe der 60er Jahre zu Bestrebungen, die wachsende Komplexität von Betriebssystemen und großen Anwendungsprogrammen durch einen konzeptionellen Wandel von der handwerklich-manufakturellen zur industriellen Software-Entwicklung zu bewältigen. Betriebssysteme waren um 1960 durch die Abwicklung des Ressourcen-Managements im Batch-Processing, die komplizierte Steuerung von Realtime-Systemen und die Regelung der Benutzerzugriffe in Time-Sharing-Systemen die größten und komplexesten Softwaresysteme, die bis dahin überhaupt entwickelt worden waren. Hier kam es auch zu den spektakulärsten Zeitüberschreitungen, Fehlerhäufungen und Projektdesastern. Nachdem bereits 1953 der Effizienzrückstand des Programming als chronischer Flaschenhals im Computing thematisiert worden war,[151] häuften sich bereits um 1960 Klagen über ein „software gap" bzw. eine "software crisis", deren Entdeckung von Organisatoren und Teilnehmern der NATO-Konferenzen beansprucht wurde.[152] In der Zeitschrift „Datamation" erschien eine Reihe von Artikeln u.a. von Herb Grosch über explodierende Softwarekosten, die Probleme beim „software project management" und die

---

149 Ratynski 1967, The Air Force computer program acquisition concept, S. 33 f.; Liebowitz 1967, The technical specification, S. 51 ff.; Ward 1969, Program transferability.

150 Bauer 1993, Software Engineering - wie es begann, S. 259; siehe dazu oben Kapitel 1.

151 Mahoney 1990, The Roots of Software Engineering, S. 3.

152 Naur, Randell 1969, Garmisch-Report, S. 120 ff.; Goos 1994, Programmiertechnik, S. 13; Buxton, Randell 1970, Rome Report, S. 7, 13.

großen Niveau-Unterschiede zwischen Hardware- und Software-Experten, ja er sprach sogar davon, die kranke Software gehöre auf die Couch: "a few miles between software and hardware boys is healthy, but a hundred is too much."[153] Bei der Zentenarfeier des MIT 1961 konstatierten der Computer-Konstrukteur George W. Brown und der „Corporate Director of Programming" bei der IBM David Sayre, dass „prevailing programming philosophies" und das „programming bottleneck" zum entscheidenden „limiting factor" der Nutzung von Computern geworden seien. Die Arbeit der „software designer" riesiger Programme bedürfe daher dringend der Ökonomisierung durch formalisierte Methoden und spezifizierte „atomic processes".[154] Der Computer-Experte und Software-Berater Ascher Opler stellte im selben Jahr fest: „the software crisis was at hand", aber auch, „that some of the solutions currently proffered will bring the chaotic situations in automatic programming under control by 1963 or 1964."[155] Frederick Brooks berichtete von zwei erlebten Softwarekrisen in den Jahren 1961-65, die „trouble building big systems" hätten schon vor 1968 zu großen Softwareproblemen geführt. Die „software crisis" wurde offenbar in den USA bereits breiter diskutiert und ist nicht erst eine ‚Erfindung' von Bauer, Dijkstra und Kollegen im Interesse der Durchsetzung strukturierter Methoden.[156]

Die Softwarekrise führte auch schon in der ersten Hälfte der 60er Jahre zu verstärkten Forderungen, zu ihrer Überwindung Ingenieurmethoden auf die Software-Produktion zu übertragen. Bereits 1964 forderte der Computer-Linguist Antony G. Oettinger als Folgerung aus den vielfachen Klagen über die ausufernden Kosten von „programming and reprogramming" und den Kontrollverlust des Managements bei der Software-Produktion: „The practice of *program engineering* with the ethical and intellectual standards and obligations of an engineering profession should be encouraged." Das Management müsse lernen, bei Informationsproblemen, Programmierung und Software generell dieselben Standards und Kontrollen anzuwenden wie beim Engineering. Die Programmierer dürften sich nicht länger als verhinderte Mathematiker oder als Aristokraten der Buchführung verstehen, sondern als Ingenieure, denn „programming is, if anything, a major new incarnation of the vanishing older engineering profes-

153 Grosch 1961a, Software in sickness and health, S. 32-33; Grosch 1961b, Software on the Couch, S. 23 f.; Head, 1963b, The Programming Gap; siehe hierzu besonders Haigh 2002, Software in the 1960s as Concept, S. 6 ff.; Ensmenger, Aspray 2002, Software as Labor Process, S. 139 ff., 146 ff.; Ensmenger 2010, The Computer Boys, S. 137 ff.

154 Brown 1962, A New Concept of Programming und Sayres Kommentar dazu in: Greenberger, 1962, Management, S. 255-259, 273 ff.

155 Opler 1971, Current Problems in Automatic Programming, S. 13-15; ders. 1967, The Receeding Future, in: Datamation 13, 9, S. 31-32, weitere Belege bei Haigh, 2002, Software in the 1960s as Concept S. 6.

156 Brooks 2018, Talk bei ICSE 2018; Ensmenger 2010, The Computer Boys, S. 10, 195; Haigh 2010a, Crisis, What Crisis?; ders. 2010 b, Dijkstra's Crisis.

sions."[157] Nach mündlicher Überlieferung soll die MIT-Mathematikerin Margaret Hamilton, die schon beim SAGE-System in der Softwareentwicklung tätig war, den Begriff „software engineering" bereits 1963/64 verwendet haben, auf jeden Fall hat sie durch ihre führende Rolle als Softwaredesignerin beim Apollo-Programm viel zur Propagierung des Begriffes beigetragen.[158] Nicht ganz sicher verbürgt ist auch die erstmalige Verwendung durch den ENIAC-Designer John Presper Eckert. Er soll einer Grußadresse an die Eastern Joint Conference 1965 davon gesprochen haben, dass der wachsende Konflikt zwischen den Programmierern und ihren Vorgesetzten in den Software-Abteilungen „would be only manageable when we could refer to it as software engineering."[159] Dokumentiert ist jedenfalls die Verwendung des Begriffs im selben Jahr durch die New Yorker Consulting-Firma Abacus Information Management Co. als Anbieter von Beratungsdiensten für „systems software engineering".[160] Im August 1966 führte dann Oettinger als neuer ACM-President den Begriff offiziell in die Computer Science ein. Gegen deren vorherrschendes Selbstverständnis als „abstract pure science" charakterisierte er sie als eine „engineering profession" und zwar besonderer Art, da zu ihr neben dem klassischen Gebiet des „hardware engineering" auch das neuartige „software engineering" gehöre, dem „engineering with symbols".[161] Bereits ein Jahr später verkündete er im President's Letter to the ACM Membership: „The notion of software engineering is, thank goodness, beginning to be heard of more and more". Er sah darin ein Umdenken von den „snobish attitudes" der brillianten reinen Mathematiker, deren Fixierung auf Abstraktionen und Formalismen bei der Lösung von Problemen des „dirtiest of unconventional engineering" nicht helfen würden: „Unless economic and engineering criteria are brought into the picture, sterile monsters result."[162] Der Begriff Software Engineering und die mit ihm transportierten Anschauungen waren somit schon Jahre vor der Garmisch-Konferenz in der fachöffentlichen Debatte der USA präsent, wobei hier offenbar stärker als bei dieser die ingenieurpraktische Ausrichtung dominierte.

---

157 Oettinger 1964, A bull's eye view of management and engineering information systems S. 1-10, meine Hervorhebung.

158 Booch 2018, The History of Software Engineering, S. 108; Computer History Museum, Margaret Hamilton 2007 Fellow. (https://www.computerhistory.org/fellowawards/hall/margaret-hamilton/).

159 Ensmenger 2010, The Computer Boys, S. 196 nach Gordon 1968. Dessen Quellenangabe stimmt nicht und in den infrage kommenden AFIPS-Bänden ist die Grußadresse nicht abgedruckt.

160 Sie die Liste der Consulting Services zu „Systems Engineering" in Computers and Automation, Juni 1965, S. 44.

161 Oettinger 1966, President's Letter to the ACM Membership, S. 546.

162 Oettinger 1967, The Hardware-Software Complementarity, S. 605 f.

3.2 Frühe Software Engineering Ansätze im Corporate Sektor von 1960-1967

Die unbewältigte Komplexität großer Software-Systeme wurde auch im Corporate Sector des Software-Marktes Anlass für die Suche nach Methoden einer klareren Strukturierung von Programmsystemen und Entwicklungsprozessen. Man fand sie in Strategien der Bildung von Abstraktionsebenen und der arbeitsteilungsgerechten Strukturierung der Systeme. Leiter großer Softwareprojekte bei IBM und General Electric wie Robert V. Head und James Martin sahen in der Hierarchisierung und Modularisierung schon bald den entscheidenden strategischen Lösungsansatz für das Management der Entwicklung großer Programm- und Informationssysteme. Head zog im Rahmen seiner Tätigkeit im „Systems Research Institute" der IBM im Jahre 1963/64 aus den Erfahrungen mit den Großprojekten SAGE, SABRE usw. die Folgerung, dass die Komplexität von Realzeitsystemen als Folge des gewachsenen Umfangs und der hochgradigen Programminteraktionen nur durch eine hochstrukturierte Programmspezifikation und ein „disciplined approach to system analysis and maintenance" zu bewältigen sei: „This dictates a requirement for some scheme for artificially segmenting the programs and increases the need for well-defined program-to-program linkage." Seine Forderung lautete daher fünf Jahre vor den NATO-Konferenzen: „Break the problem down in true programmable chunks". Da der Programmierergruppe nicht einfach ein „job" von 50 bis 150 Tausend Maschineninstruktionen vorgesetzt werden könne, seien Spezifikationen erforderlich, „to compartmentalize this task into easily manageable units of work".[163] Neben der horizontalen Arbeitsteilung durch „chunks" und „units" spricht er bereits von der vertikalen Gliederung der Gesamtaufgabe durch „levels". Denn um, wie in den Großprojekten erforderlich, möglichst viel unerfahrenes Personal verwenden zu können und zugleich den Kommunikations- und Abstimmungsaufwand zwischen den Programmierern zu reduzieren, müsste eine klare Abgrenzung der Einzelaufgabenbereiche erfolgen: „While informal communication among programmers on technical problems is no doubt desirable, it can in a large system seriously impair productivity. One purpose of specifications is to organize and formalize such communication."[164]

Head orientierte sich zwar am „Operational Plan" von SAGE, doch übernahm er nicht die „tabular form" der „System Flow Chart", die schon zu Beginn der „programming specifications" vorliegen und den ganzen Ablauf regulieren sollte. Stattdessen plädierte er für die „conventional form" der Flowchart in der Art des „General Logic Diagram" bei SABRE, die erst im Laufe der Programmentwicklung erarbeitet wurde. Denn so Head, „it may be a general rule that a System Flow Chart which is *meaningful to the programming staff* cannot be produced until after considerable effort has been invested

163 Head 1963a: Real-Time Programming Specifications, S. 376, 377 f.
164 Ebda., S. 378.

in specifications."[165] Trotz der Anlehnung an die Engineering-Metapher blieb die „software production" für Head mehr „art than science", denn „computer programming, unlike computer engineering, cannot be classified as a science or even as a well-disciplined profession. What is good for hardware development [...] is not necessarily good for the programming."[166] Doch schon 1965 vollzog Head eine Kehrwende mit seinem Szenario künftiger ‚Softwarefabriken' bei Banken, großen Industrieunternehmen und vor allem bei Computerherstellern. Obwohl deren Programmiererstäbe bereits den Charakter von „programming factories" hätten, sei die Programmentwicklung derzeit noch keine „well disciplined profession". Um endlich Organisation in die „the troubled waters of programmer productivity" zu bringen, solle künftig die gesamte Programmproduktion einer straffen industriellen Arbeitsteilung mit definierten Arbeitsstandards und den in Großbetrieben üblichen Methoden der Planung, Produktionssteuerung und Arbeitskontrolle unterworfen werden: „In a well run *programming factory*, there should not be extensive modification of the product once the specifications prepared by systems engineering have been frozen."[167] Head gehörte damit zusammen mit Bob Bemer und M. Douglas McIlroy zu den ersten, die das „Component" bzw. „Software Factory Principle" in die Software-Community einführten.[168]

Mitte der 60er Jahre kündigte sich allgemein ein Übergang von den in der Tradition des team-orientierten Systems Engineering stehenden Ansätzen der manufaktuerellen Software-Entwicklung zu einer am Industrial Engineering angelehnten Vorgehensweise an. Sie zielte auf eine striktere Abgrenzung der Teilaufgaben und eine zunehmend vom Projektmanagement kontrollierte Arbeitsteilung. Man gab sich nun nicht mehr mit einer Phasenbildung zufrieden, die eher Appellcharakter hatte, und ebenso wenig mit dem bisherigen Kompromiss zwischen horizontaler modularer Aufgabenteilung und interdependenter Problemstruktur, der den Kommunikationsaufwand nicht entscheidend zu verringern vermochte. Vielmehr suchte man die Lösung in einer vertikalen Systemstrukturierung und einer interferenzfreien Zerlegung der komplexen Programm- und Informationssysteme. Besonders deutlich wird dieser Methoden- und Strategiewechsel an der Designlehre von James Martin für Realzeit-Computersysteme von 1965/67, die auf eigenen Erfahrungen bei der IBM als „original designer" des britischen Flugreservierungs-Systems BOADICEA und der Mitarbeit am amerikanischen elektronischen Buchungssystem SABRE beruhte.

Der bei seinem Kollegen im „IBM Systems Research Institute" Head nur angedeutete Zusammenhang zwischen der Produkt- bzw. Aufgabenstruktur und der Arbeitsorgani-

---

165 Ebda., S. 377, Hervorhebung im Original.

166 Head 1964, Real-time Business Systems, S. 178, 282.

167 Head 1965, Planning for Generalized Business Systems, S. 153, 155 f., 159, meine Hervorhebung

168 Vgl. Cusumano 1991, Factory Concepts and Practices in Software Development, S. 5 f.

sation wurde bei Martin ins Grundsätzliche gehoben. Er legte dar, dass die Interdependenzen zwischen den Segmenten größerer Programmsysteme bei horizontaler Arbeitsteilung unweigerlich zu einer nicht mehr beherrschbaren Hyperkomplexität führen mussten. Das entscheidende Problem war für ihn das „problem of control“: „How can the programs that make up a real-time system be developed concurrently by separate programmers when there are so many interactions between them? How can control over the interactions be maintained?"[169] Die Lösung des Komplexitätsproblems interdependenter Problemstrukturen sah auch Martin in einer Annäherung an die Methoden der klassischen Ingenieurwissenschaften und industriellen Produktionsweisen: „It is interesting to pursue the analogy between this type of system and a complex piece of mechanical or electrical engineering. Some of the techniques that have become traditional in convential engineering point the way here also.“[170]

Dem Engineering-Leitbild und dem Muster des Austauschbaus in der Mechanik und der Komponentenstruktur der Elektrokonstruktion folgend, wollte Martin die Interdependenzen der Programmsegmente und die komplexen Interaktionen zwischen den Programmierern durch eine „division into sub-systems“ bereits vor Beginn des Entwicklungsprozesses weitgehend reduzieren: „The unit is insulated as completely as possible from the other units. The interface between the unit and the rest of the system must be very clearly defined. Every attempt will be made not to modify the interfaces between the the units. This means that, before programming begins, the functions of the units must be very clearly thought out.“[171] Wie bei Thomas Holdimann stand auch bei Martin ein experimenteller Prototyp am Anfang, der dann in ein stufenweises „model-by-model built up“ überging, mit dem durch kontinuierliches „rewriting“ und „cleaning“ das Programmsystem schrittweise verfeinert würde. Er legte hier bereits den Grundstein für sein späteres prototypisches Entwicklungskonzept, das „Rapid Application Development“ (RAD) von 1991. Für sein Vorgehensmodell entscheidend war die Einfachheit der einzelnen Programmmechanismen, sie hatte unbedingten Vorrang vor der individuellen Brillianz: „Being a member of a real-time programming team may not be the right place for a programming prima donna.“[172] Zur Vermeidung vertikaler Interdependenzen und Rekursionen in frühere Entwicklungsphasen sollten dann nach der Fertigstellung eines „model“ dessen funktionale Spezifikationen eingefroren werden. Schließlich sah er eine „central control group“ bzw. eine „central of authority“ vor, die sich ein „complete knowledge of the system under development“ verschafft, die alle doch noch erforderlichen Änderungen kommuniziert und den gesamten Produktionsprozess in Taylor'scher Manier kontrolliert.

---

169 Martin 1965, Programming Real-Time Computer Systems, S. 327.

170 Ebda., S. 327; Martin 1967, Design of Real-Time Computer Systems, S. 562.

171 Martin 1965, Programming Real-Time Computer Systems, S. 329; ders. 1967, Design of Real-Time Computer Systems 567.

172 Martin 1967, Design of Real-Time Computer Systems, S. 569.

Eine dezidierte Orientierung am Engineering-Vorbild hielt Martin indessen nur bei der Programmentwicklung im Batch Processing und im konventionellen wissenschaftlichen Rechnen für sinnvoll, nicht dagegen bei den weitaus komplexeren Programmsystemen des Realzeit-, Time-Sharing- und interaktiven Online-Computing. Denn hierbei kommt es im Unterschied zum klassischen Engineering wegen nicht vorausplanbarer Wechselwirkungen zwischen den Programmblöcken zu zahlreichen Änderungen am „original plan". Eine „simple bar chart" in der Art eines Wasserfallmodells erschien ihm daher für die Ablaufplanung als ein „too clumsy tool because of the intricate relationships that exist between different events." Er entwarf zwar selber eine detailliertere „programming schedule" in Chart Form, doch war sie seiner Meinung nach nur für einfach strukturierte On-line-Systeme geeignet. Bei einer höheren Programmkomplexität versagten dagegen die üblichen Engineering-Instrumente zur strikten Abgrenzung und Ablaufsteuerung der Arbeitspakete, denn hierbei handle es sich um „a tightly integrated piece of teamwork", für das sich „this type of discipline" nicht eigne. „Die „programming group" müsse vielmehr als ein Team aufgebaut und als Team gemanaged werden, in dem die Mitglieder zu einer „careful cooperation" ermuntert werden, um so wechselseitig die Probleme der anderen wahrzunehmen und zu erkennen, wie die einzelnen Elemente sauber zusammenarbeiten.[173] Dabei würden Spezifikationen und Schnittstellen-Dokumente zwar hilfreich sein, doch diese „formal means of control of interactions" bedürften dringend der Ergänzung durch den „informal contact between programmers", um als „well-knit team" die komplexen Probleme zu bewältigen.[174]

Ohne den Begriff „Software-Engineering" zu verwenden, war damit die Engineering-Metapher in Martins Designlehre durchgängig präsent, er wollte ihr Potenzial voll ausschöpfen, doch er sah nach wie vor strukturelle Unterschiede zwischen der Mechanik-/Elektrokonstruktion und der Programmentwicklung, die der Analogie Grenzen setzten. Ingenieurmäßige Methoden und lineare Phasenmodelle waren nach ihm nur in Anwendungskontexten mit geringer Komplexität und Interaktivität angebracht, nicht dagegen in organisationsbezogenen Verarbeitungs- und Informationssystemen mit nicht vorab definierbaren Programmabläufen. Da der Entwicklungsstand der Entwurfstechniken für derart komplexe Systeme noch wenig ausgereift war, hielt auch er bei ihnen weiterhin an der Sichtweise der „craftmanship of programming" bzw. der „art of programming" fest.[175] Damit wurde hier bereits in Ansätzen Christiane Floyds „Unterscheidung zwischen kontextfrei definierter, technisch eingebetteter und sozial eingebetteter Software" vorweggenommen.[176]

---

173 Ebda., S., 563, 567 f.

174 Ebda., S. 569.

175 Martin 1965, Programming Real-Time Computer Systems, 8; 1967, S. 562 f., 565 ff.

176 Floyd 1994, Software-Engineering - und dann? S. 31.

3.3 Die Anfänge des Software Architecture-Konzeptes von 1959-1975

Neben derartigen Kompromissformen zwischen handwerklich-manufaktureller Kooperation und quasi-industrieller Software-Produktion entstand in der ersten Hälfte der 60er Jahre ebenfalls im IBM-Umfeld als grundsätzliche Alternative zur Engineering-Metapher das Konzept der „Software Architecture". Entwickelt wurde es von Frederick P. Brooks, der als einer der Chefdesigner des IBM Stretch-Computers und als leitender Programmmanager der Hardware- und Software-Entwicklung des IBM System /360 erkannt hatte, dass ingenieurwissenschaftliche und systemtechnische Dekompositionsstrategien nicht ausreichten. Diese lieferten zwar Ansatzpunkte zu einer arbeitsteiligen Bewältigung der Designkomplexität, klammerten dabei aber wesentliche Designentscheidungen, User-Aspekte und Managementprobleme aus. Anfang der 60er Jahre führte er daher das Architektur-Leitbild in die Computer Community ein. Mit ihm lenkte er den Fokus auf die bisher unterbelichteten Syntheseaspekte und technisch-organisatorischen Zusammenhänge des Designs und der Struktur eines Computersystems und die Interfaces zum Programmer und User. Aufgabe des Designers war es, analog zu einem Architekten alle beim Entwurf eines Hardware- und Softwaresystems auftretenden grundlegenden Benutzeranforderungen, Designkonflikte und Allokations-Entscheidungen im Zusammenhang zu betrachten, auf der Grundlage des vorhandenen Erfahrungsschatzes zu bewerten und zu einem konsistenten Gesamtentwurf hoher Qualität zu integrieren.[177]

Als Leitbegriff für diesen erweiterten Ansatz der Komplexitätsbewältigung im Design von Computing-Systemen verwendete er bereits seit 1959 in internen IBM-Dokumenten und ab 1961 in Publikationen die Begriffe „computer" bzw. „system architecture". Nach Brooks' eigener Darstellung kamen die ersten Anstöße für die Verwendung der Metapher eigentlich aus dem Software-Bereich: „I still remember the jolt I felt in 1958 when I first heard a friend talk about *building* a program, as opposed to *writing* one. In a flash, he broadened my whole view of the software process. The *metaphor shift* was powerful, and accurate."[178] Die erste Darlegung seiner „Architectural Philosophy" von 1961/62 bezog sich jedoch auf den Designprozess des gesamten Computersystems, in dessen „architectural phase" zu Beginn die Nutzererfordernisse ermittelt und die „rationales", „features" und die dem entsprechenden „resources" und „interfaces" des Systems festgelegt werden. Hier taucht auch bereits erstmals der Gedanke auf, dass die „guiding principles" der Architektur das Rollenverständnis der Entwickler leiten und heterogene Teams orientieren sollte, um so die konzeptionelle Integrität des Designs zu sichern.[179] Für Brooks blieben die Programmierer- und Enduser-Sicht auf die

---

177 Siehe hierzu ausführlich Hellige 2004, Die Genese von Wissenschaftskonzepten der Computerarchitektur, S. 436-448.

178 Brooks 1987, No Silver Bullet, S. 18, meine Hervorhebungen.

179 Brooks 1962, Architectural Philosophy, S. 5 ff.

Rechner-Funktionalität, die Ausgestaltung des „instruction set“ und die Interfaces für den Endnutzer immer Kern seiner Architekturauffassung. Doch in den Folgejahren rückten bei ihm wie im IBM-Konzern das äußere Erscheinungsbild der Computerfamilie, ein abgestimmtes Industrial und Interface Design und eine vereinheitlichte Produktpalette ins Zentrum.[180] Erst Mitte der der 60er Jahre verschob sich bei Brooks der Fokus wieder auf die Architektur von Softwaresystemen, als er 1965, d.h. im Jahr der ersten Verwendung des Software Engineering-Begriffs, in seinem berühmten IFIP-Vortrag „The Future of Computer Architecture“ den programmatischen Begriff "*software architecture*" einführte.[181] Mit ihm zog er seine Lehren aus dem problematischen Verlauf der Entwicklung des OS/360, der auch der Anlass für sein Ausscheiden aus dem IBM-Konzern war.

Das OS/360 der IBM war 1962 unter seiner Leitung als „single operating system“ für die gesamte Rechnerfamilie des 360/-Systems konzipiert und 1964 angekündigt worden aber erst Mitte 1967, das heißt drei Jahre nach der Hardware, auf dem Markt verfügbar.[182] Da das OS/360 eine Vielzahl von Maschinen mit beträchtlichen Unterschieden in der Größe, Leistungsfähigkeit, Funktionalität und im Hauptspeichervolumen mit vollkompatiblen Steuer-, Dienst- und Anwendungsprogrammen sowie mit Compilern für eine ganze Reihe von möglichen Programmiersprachen auszustatten hatte, bestand es am Ende aus einem Satz von zehn Betriebssystemen mit hunderten von Programmbausteinen und hatte einen Umfang von mehr als einer Million Codezeilen. Die Gesamtfunktionalität wurde nach dem „concept of functional, that is horizontal ‚modularity' “ strukturiert und die einzelnen Systemkomponenten jeweils gesonderten Programmierergruppen in verschiedenen IBM-Laboratorien in den USA und Europa übertragen.[183] Die Architektur des Software-Systems bildete so die Organisationsstruktur ab und entsprach damit, wie Brooks später bewusst wurde, genau dem Convay’s Law.[184]

Das Resultat des „distributed software design“ war ein kaum noch überschaubares „kit of tools, techniques, and functions“, die sich z. T. sogar überschnitten und aus denen sich die Anwender erst ein ihren Anforderungen gemäßes Betriebssystem zurecht schneidern mussten.[185] Da die Forderung nach Kompatibilität innerhalb der gesamten Produktfamilie zu einer hochgradigen Interdependenz der Teilkomponenten führte, zog eine Änderung an einer Stelle Folgen an sehr vielen anderen Stellen nach sich.

---

180 Vgl. hierzu Hellige 2004, Die Genese von Wissenschaftskonzepten der Computerarchitektur, S. 446 ff.; Halsted 2018, The Origins of the Architectural Metaphor in Computing.

181 Brooks 1965, The Future of Computer Architecture, S. 87.

182 Mealy, Witt, Clark 1966, The Functional Structure of OS/360; S. 2-51; Weizer 1981, A History of Operating Systems, 122.; Campbell-Kelly, Aspray 1996, Computer, S. 196-200.

183 Cusumano 1991, Factory Concepts and Practices in Software Development, S. 8 f.

184 Brooks 1975/95, The Mythical Man-Month, S. 111.

185 Lynch 1972, Operating System Performance, S. 582.

Personalprobleme in einzelnen Abteilungen, mangelnde Kommunikation und lokale Fehler lösten schnell Kettenreaktionen aus, die den gesamten Fahrplan durcheinanderwarfen und die Auslieferung immer weiter verzögerten. So kam es, dass sich das anfangs favorisierte „small, sharp team concept“ mit 200-Mann zu einem „human wave approach“ mit über 1000 Mann entwickelte, so dass das OS/360 zwischen 1963 und 1966 insgesamt 5000 Mannjahre für Design, Entwurf und Dokumentation beanspruchte.[186] Die Entwicklung des Betriebssystems für die erfolgreichste Produktfamilie der Computergeschichte wurde am Ende zu einem ihrer größten Debakel.

Brooks zog aus dem OS/360-Desaster die Konsequenz, dass Dekompositionsmethoden, wie sie auch im Software Engineering propagiert wurden, zur Komplexitätsbewältigung in Softwaresystemen nicht ausreichen. Da es in diesen darum gehe, divergierende Designanforderungen nutzer- und nutzungsgerecht zu einem konsistenten und möglichst kostengünstigen Systementwurf zu integrieren, beruhe besonders die Software-Entwicklung wesentlich auf Syntheseleistungen und Managementmethoden, die eine Software-spezifische Architekturlehre erforderten. Sein Konzept der Software Architecture verzichtete dabei nicht auf strukturierende Methoden, ergänzte sie aber in den strukturbildenden Phasen um integrierende Designmethoden, die sich vor allem auf die Wünsche der „user“ sowie Qualität und Konsistenz des Designs bezogen. Nach Brooks bildete die „software architecture“ zusammen mit der „architecture of input/output systems“ sogar den Schwerpunkt der architektonischen Systemgestaltung, nachdem bis ca. 1960 die Hardware-zentrierte „pure computer architecture“ dominiert habe. Das künftige Hauptaufgabengebiet der „architects“ sah er stärker im Design des Gesamtsystems, in der „integrated hardware and software architecture“ bzw. in der noch anwendungsnäheren „information system architecture“.[187]

Die Konzeption seiner Softwarearchitektur entwickelte Brooks nach 1965 in seinen 1972 abgeschlossenen und 1975 unter dem Titel „Mythical Man-Month“ erschienenen „Essays on Software Engineering“. In ihnen arbeitete er die Lehren aus den Managementfehlern bei der OS/360-Entwicklung systematisch aus und setzte sich zugleich mit den quantitativen Planungs- und Kontrollmethoden des Software Engineering auseinander. Denn die Probleme des „tar pit of software engineering“[188] waren für ihn wesentlich qualitativer Natur und nur durch vom klassischen Engineering abweichende Design- und Managementmethoden zu bewältigen. Rigide Terminplanungen, die von einer Austauschbarkeit von „man“ und „month“ ausgingen, mussten unweigerlich zu „scheduling desasters“ führen.[189] Indirekt kritisierte er auch wasserfallartige

186 Brooks 1975, The Mythical Man-Month, S. 30 ff.
187 Brooks 1965, The Future of Computer Architecture, S. 88.
188 Brooks 1975, The Mythical Man-Month, S. 177.
189 Ebda., S. 13 ff.

Vorgehensmodelle mit strikten Meilensteinen, die zwar als Frühwarnsystem sehr sinnvoll seien, aber als Zwangsmittel meist zur Fehlsteuerung führten. Schon die Hardware-Entwicklung des System/360 hatte nämlich gezeigt, dass sie keinem starrem Stufenschema folgte, sondern eher in rekursiven Zyklen verlief: „they tend to spiral in developing series of partial approximations."[190] Die strukturierte Programmierung hielt er für einen „major step forward", doch das „Top-down design" mit „concrete milestones" markierten lediglich „the vague phases of planning, coding, debugging": „The process of step-wise refinement does not mean that one never has to go back, scrap the top level, and start the whole thing again as he encounters some unexpectedly knotty detail."[191] Mit seinem „Throw away"-Prinzip und Empfehlungen für „constant changes" oder gar einen Neustart plädierte er implizit bereits für inkrementelle Methoden.[192]

Dem analytischen Software Engineering mit seinen formalen Methoden setzte Brooks seine qualitative Design- und Management-Lehre der „Software Architecture" entgegen, die er in Anlehnung an klassische Architekturtraktate als eine Sammlung von hochkondensierten Design-Erfahrungen anlegte. Aus ihnen leitete er Design-Prinzipien ab, die wesentlich um die Maxime der konzeptionellen Integrität des Architekturentwurfs kreisten. Diese war für ihn nur zu erreichen, wenn wenige „senior architects" das Konzept nach den Vorgaben des Benutzers und den Belangen anderer betroffener Akteure festlegten: „Conceptual integrity does require that a system reflect a single philosophy and that the specification as seen by the user flow from a few minds."[193] Mit der Anlehnung an den professionellen Status von Bauarchitekten artikulierte er eine Mittlerrolle zwischen den Stakeholdern und zugleich ein neues Rollenverständnis und Selbstbewusstsein einer Designerelite. Sein zunehmend aristokratischerer Ansatz, der weitgehend mit dem in einem IBM-Projekt erprobten und bereits von Joel D. Aron 1969 bei der NATO-Konferenz in Rom vorgestellten „surgical team"- und "chief-programmer" Konzept von Harlan D. Mills und F. Terry Baker übereinstimmte, zielte daher auf eine Neugewichtung der Arbeitsteilung im Entwicklungsprozess. Der „chief programmer" ist danach der „prime architect and key coder of the system", der mit seinem „team nucleus" die Systementwicklung leitet, die damit aber auch ganz von seinen Designerfahrungen und seiner Synthesefähigkeit abhängig wird.[194]

---

190 Siehe Ruth Mack 1971, Planning on Uncertainty mit der Vorwegnahme des Spiralmodells, S. 136-148, 158.

191 Brooks 1975, The Mythical Man-Month, S. 143 f., 154 ff.

192 Ebda, S. 116, 144.

193 Ebda., S. 49.

194 Ebda., Kap. 3; siehe Joel D. Aron in: Buxton, Randell 1970, Rome Report, S. S. 38 f.; Baker 1972, System quality through structured programming S. 339 f.; Baker, Mills 1973, Chief Programmer Teams.

Gegenüber der horizontalen Arbeitsteilung gewann die vertikale damit eine entscheidende Bedeutung und das hieß eine Aufteilung des Entwicklungsprozesses in die drei Hauptphasen "architecture", "implementation" und "realization". Diese stellten zugleich voneinander getrennte Zuständigkeitsbereiche dar, wobei dem "architecture manager" neben dem Systementwurf die Oberaufsicht zufiel: "In effect, a widespread horizontal division of labor has been sharply reduced by a vertical division of labor, and the result is radically simplified communication and improved conceptual integrity."[195] Im Gegensatz zu dem bald vorherrschenden Leitbild einer wissenschaftlich-exakten oder ingenieurmäßigen Software-Entwicklung mit einer verbindlichen Strukturierung von Produkt, Programm und Prozess blieb Brooks aber weiter bei seinem Ideal des aus seiner Erfahrung schöpfenden Architekten, der den „creative process" der Softwareentwicklung durch die konzeptionelle Integrität der Software-Architektur und eine auf Überzeugung bauende qualitätsorientierte Teamsteuerung leitet. Im Unterschied zur Computer Architecture, wo Brooks in jahrzehntelanger Arbeit zusammen mit Gerrit A. Blauuw anhand von historischen Musterarchitekturen noch eine umfassende Darstellung der Bauformen- und Konzept-Evolution in einem grandiosen Alterswerk fertigstellen konnte, kam es im Softwarebereich nicht mehr zu einer entsprechenden Systematisierung der „architectural styles", „development qualities" und „design methods".[196]

3.4 Software Architektur-Traktate in der Brooks-Nachfolge

Brooks war mit seiner Kritik an szientistischen Auffassungen des Software-Designs keineswegs allein. Selbst in der angeblich so harmonisch verlaufenen NATO-Konferenz in Garmisch bildete sich nach Aussagen von Teilnehmern eine stille Opposition gegen den vorherrschenden Theorieanspruch der akademischen Software-Engineering-Protagonisten. Doch selbst den im ersten Report bereits erkennbaren kritischen Statements schenkte man lange Zeit wenig Beachtung, sie wurden erst im Zuge der Grundsatzkritik am „Engineering Model" seit den 90er Jahren wiederentdeckt. So verwies der Begründer agiler Softwaremethoden Alistair Cockburn (2004) auf Alexander G. Frasers dezidierte Ablehnung der simplen Vorstellung, die Softwareproduktion erfolge in linearer Progression bis zum Endprodukt als der „sum of many sub-assemblies". Ebenso auf das leidenschaftliche Plädoyer des IBM-Computer-Konstrukteurs Hollis Andy Kinslow für ein inkrementelles Vorgehensmodell: „The design process is an iterative one."[197] Das Thema „incremental systems" war ursprünglich in der Garmisch-Konferenz als eigener Programmpunkt vorgesehen, entfiel dann aber. Peter Naur

---

195 Brooks 1975/95, The Mythical Man-Month, S. 43 f., 50.

196 Blauuw, Brooks 1997 Computer Architecture; Hellige 2004, Die Genese von Wissenschaftskonzepten der Computerarchitektur S. 441-447 und unten Kap. 4.

197 Fraser und Kinslow in Naur, Randell 1969, Garmisch-Report, S. 19, 21; Cockburn 2004, The End of Software Engineering, S. 22.

brachte auch bereits unter Verweis auf Christopher Alexanders „Notes oft the Synthesis of Form“ neben dem „civil engineer“ den „*architect*“ als eines Designers von "large heterogenous constructions" als Vorbild für das Profil des „software designer“ ins Spiel.[198] Doch das Architektur-Konzept als Gegenmodell zum Software Engineering trat erst Rahmen der sich verschärfenden Kritik in Rom in Erscheinung.

Wohl am radikalsten kritisierten dort Joel D. Aron (IBM Gaithersburg) und Robert M. Needham den Wissenschaftsanspruch des neuen Leitkonzeptes: Der praktische Software-Ingenieur habe es nicht mit den klaren Prinzipien und konsistenten Theorien der Computer Science zu tun, sondern mit einem heterogen Geflecht technischer, wirtschaftlicher, betrieblicher und sozialer Designentscheidungen: „In practice large systems implementation is influenced by many factors: available personnel, management structure, and so on. [...] Much theoretical work appears to be invalid because it ignores parameters that exist in practice. [...] Theorists have not learned to cope with this randomly discrete set of events in an uncertain environment. Therefore, their impact on the engineers is minimal.“[199] Für Adin D. Falkoff vom Watson Research Center der IBM, der mit Ken Iverson die Programmiersprache APL entwickelt hatte, war „software engineering“ als Disziplin viel zu eng konzipiert, denn neben technischen Problemen gehe es besonders auch um Ressourcen- und Wirtschaftsfragen. Ähnlich wie für Brooks waren für ihn aber letztlich Symmetrie, Balance und „general rightness“ des Designs entscheidend: „In engineering or architectural design, aesthetic criteria are the ultimate basis for design decisions that are not dictated by strictly technical considerations. Good designers are distinguished from others by their exercise of aesthetic judgment.“[200]

Auch der Chef-Programmierer der kanadischen Ferranti-Division und Firmengründer Ian P. Sharp forderte unter Hinweis auf Brooks und dessen schlechte Erfahrungen beim OS/360, den „theoretical architect“ durch den „practical architect“ zu ergänzen und brachte als einziger den Begriff „Software Architecture“ in die Debatte ein: „I think that we have something in addition to software engineering: [...] This is the subject of software architecture. Architecture is different from engineering. [...] The reason that OS is an amorphous lump of program is that it had no architect. Its design was delegated to a series of groups of engineers, each of whom had to invent their own architecture. And when these lumps were nailed together they did not produce a smooth and beautiful piece of software.”[201] Doch derartige Versuche, die theorielastige

198 Naur in Naur, Randell 1969, Garmisch-Report, S. 35; Alexander 1964, Notes on the Synthesis of Form.

199 Buxton, Randell 1970, Rome Report, S. 113.

200 Ebda., S. 63, 90.

201 Ebda., S. 12; sein Working Paper “Systems design for a changing environment" ist nicht im Report enthalten.

Ingenieurauffassung des stark mathematisch-theoretisch bzw. naturwissenschaftlich geprägten Software-Engineering-Leitbildes in Garmisch und Rom mit der pragmatischen Sicht des Ingenieurs bzw. Architekten zu konfrontieren, blieben insgesamt erfolglos, zumal der Hauptpromotor der „Software Architecture“ Frederick Brooks nicht eingeladen worden war.

Im Gegensatz zur Computer- und Netzwerkarchitektur, die sich als Gebiets-, Teildisziplin- und als Strukturbezeichnung erstaunlicherweise schon Ende der 60er bzw. in der ersten Hälfte der 70er Jahre fest etablierte, bestanden im Softwarebereich lange Zeit massive Vorbehalte gegen den Softwarearchitektur-Begriff. Die Ablehnung gegen ihn war dabei besonders deutlich gerade bei den Informatikern, die die zentralen Strukturierungskonzepte der Hierarchien, Abstraktionsebenen und Module entwickelt hatten. Die bestimmenden Begriffe waren bei diesen „layering approach“, „structuring concept“, „system structure“ und besonders „system hierarchy“. Der Grund für die Abstinenz gegenüber dem Architekturbegriff lag zum einen in dem eher akademischen Ursprungsmilieu der neuen Strukturierungskonzepte begründet und zum anderen in der massiven Krisenwahrnehmung des Zustands der Softwareproduktion. Angesichts des schwer beherrschbaren Wildwuchses bei großen Softwaresystemen ging es diesen Informatikern nicht mehr primär um eine Entfaltung von Designkomplexität und eine kunstvolle, anwendungsspezifische Software-Gestaltung, sondern vorrangig um eine Eingrenzung des „design space“, um „shortening correctness proofs by simplifying control logic“. Auch ihrem anvisierten Leitbild einer stark formalisierten „science of software production“ widersprach die Architektur-Metapher.[202]

Dennoch erlangte das Softwarearchitektur-Konzept Ende der 60er und in der ersten Hälfte der 70er Jahre inner- und außerhalb der Computer Science eine breitere Resonanz. Es finden sich in diesem Zeitraum nicht nur im IBM-Umkreis, sondern bei Betriebssystem-Spezialisten in den USA und vor allem in England diverse Belege für einen Design-orientierten Softwarearchitektur-Begriff, vor allem im Kontext von Strukturvergleichen von Betriebssystemen. So verwendete der Computer Scientist vom Brooklyner Pratt Institute und Fachbuchautor Harry Katzan Jr. 1970 den Begriff "operating systems architecture" im Sinne des "overall design of hardware and software components and their operational effectiveness as a whole."[203] In England hatte Peter D. Jones, der seit 1961 am Betriebssystem des von Thomas Kilburn entwickelten wegweisenden Atlas-Multiprogramming-Systems der Universität Manchester mitgewirkt hatte, die Architekten-Metapher bereits 1968 eingeführt. Ganz ähnlich wie bei Brooks drei Jahre zuvor betonte Jones anhand eines Vergleiches des „Atlas Supervisor“ mit dem OS 360 und dem Chipewa OS des Supercomputers CDC 6600 „die

---

202 Dijkstra in Buxton, Randell 1970, Rome Report, S. 85; Buxton in Naur, Randell 1969, Garmisch-Report, S. 89.

203 Katzan 1970, Operating Systems Architecture, S. 109

entscheidende Rolle des „architect of the operating system“: „The role of the system architect is to provide an integrated overall system structure which is designed in parallel with the computer organisation and hardware.“[204] Er beanspruchte deshalb implizit für die Architekten von Betriebs- bzw. Softwaresystemen einen ähnlichen Rang, wie sie der „chief computer designer“ bei der Hardware genieße. Die Analogie mit dem Architekten zielte bei Jones aber weniger auf eine Mittlerposition zum Kunden als vielmehr auf die Durchsetzung qualitativer Designprinzipien wie „minimum system concept“, „dynamic growth pattern“, „protection hierarchy“, „simplicity“, „modularity“. Der „system architect“ werde durch seine auf Vergleichen gelungener und misslungener Systeme beruhende Kenntnis des Zusammenhangs der „chief characteristics“ und der „architectural features“ zum Beherrscher der immer komplexer werdenden Systeme und damit entscheidend für den Erfolg oder Misserfolg eines Softwaresystems.[205]

Etwa zur gleichen Zeit wurde im weiteren Umfeld der Atlas-Projekte die Reichweite des Architektur-Begriffs nochmals ausgeweitet. Chris R. Spooner, der sowohl an der "central architecture" des Atlas-1-Supervisor als auch an dessen Redesign für das Atlas-2-System mitgewirkt hatte, transferierte die Ansätze aus Manchester in die Entwicklung der Betriebssysteme bei Control Data. Er bündelte 1970/71 die englischen Architektur-Reflexionen in dem Begriff „Software Architecture“, den er zu einem Schlüsselbegriff der Software Community erheben wollte. In dem programmatischen Artikel „A Software Architecture for the 70's“, zugleich der Eröffnungsbeitrag der neu gegründeten englischen Zeitschrift „Software – Practice and Experience“, entwickelte Spooner im Gegensatz zu der „large-scale system structure“ von Dijkstras „THE Multiprogramming System“ die Software-Architektur als eine integrale „design philosophy“. Diese umfasste sowohl die allgemeinen Prinzipien, die Methoden und das grundlegende Erfahrungswissen sowie die „software structure“, d.h. die funktionale Arbeitsteilung und Strukturierungsansätze in Softwaresystemen. Von der konsequenten Anwendung des „new software framework“ versprach sich Spooner „a good software architecture“, die die Anforderungen nach Modularität, Flexibilität und Effizienz erfüllt und damit zugleich Kosten und Entwicklungszeiten senke.[206]

Im Zentrum dieses Architektur-Konzeptes standen die Beziehungen zwischen den Designkriterien, die nicht als Kriterienbaum oder Zielhierarchie gesehen wurden, sondern als ein Geflecht von Trade-offs, die von den Designern sorgfältig austariert werden müssten. Die Modularisierung war zwar mit Blick auf spätere Änderungen in einem „open-ended development“ ein zentraler Bestandteil des Architekturkonzeptes,

---

204 Jones 1969, Operating System Structures, S. 525 ff.

205 Jones, Lincoln, Thornton 1972, Whither Computer Architecture? S. 730.

206 Spooner 1971, A Software Architecture for the 70's, S. 6.

sie diente jedoch nicht als Grundlage der Phasenstrukturierung des Entwicklungsprozesses oder als Instrument der Disziplinierung der Programmierer.[207] Spooner kritisierte sogar die starren Ebenenkonzepte der Zwiebelschalenmodelle („onion-ring approach"), in denen eine Schicht immer nur mit der über- oder untergeordneten Nachbarschicht kommunizieren darf. Statt der umständlichen Instanzenwege sollten jenseits des Kernbereiches (Kernel) je nach Funktion auch Verknüpfungen von „activities" unterschiedlicher Schichten möglich sein. Dieses umfassende Konzept einer Softwarearchitektur war somit ausgesprochen design- und erfahrungsorientiert und beanspruchte trotz des Verzichtes auf Verwissenschaftlichung und strikte Ingenieurmethoden gleichwohl eine „architectural theory" zu sein.[208]

Die von Brooks inspirierten Bestrebungen für die Entwicklung von designzentrierten Softwarearchitektur-Konzeptionen in der englischen und amerikanischen Software-Community verebbten jedoch im Laufe der 70er Jahre. Die Architekturkonzepte gerieten nun zunehmend unter den Einfluss des Software Engineering-Hypes. Mitte der 70er Jahre bemühten sich die Computer Scientists John R. White und Taylor L. Booth von der University of Connecticut, „software architecture" und „software engineering" in einem Phasenmodell des „software development" zu integrieren. Denn die „structured programming period" von 1970-1975 hätte die Kostensteigerung in der Softwareentwicklung nicht aufhalten können, da die eigentlichen Probleme auf der Ebene der „fundamental design issues" lägen. Als Beginn des Softwarezyklus setzten sie deshalb die strukturbildende Phase des „software design", d.h. „the application of sound engineering principles to the evolution of an architecture for a software system". Das „design" ist hierbei ein iterativer, inhärent kreativer Prozess, in dem vage Anforderungen in ein wohl organisiertes und definiertes System übersetzt werden. Dabei müssten verschiedene Architekturvarianten noch Hardware-unabhängig durchgespielt und die Beziehungen der funktionalen Komponenten interferenzfrei organisiert werden: „We must recognize that software design is a very complex and demanding profession which requires a high level of professional competence. […] The design process, unlike implementation, is very non-algorithmic and relies heavily on the expertise and insight of the designer(s)." In den folgenden Phasen würde dann die zu schaffende „software architecture" an die spezifischen Restriktionen der Maschine und Prozesse angepasst und schließlich nach der Wahl der Programmiersprache die ausführbare Software im „program design" implementiert.[209] Es wurde somit bereits 1976 eine Arbeitsteilung zwischen Software Architektur und Software Engineering vorgeschlagen, wie sie erst in den 90er Jahren zum Tragen kam.

---

207 Ebda., S. 7.

208 Ebda., S. 29.

209 White, Booth 1976, Towards an engineering approach to software design, S. 214 ff., 222.

Doch Mitte der 70er Jahre bestimmten Abstraktion, Standardisierung und Modularisierung zunehmend das „new architectural model“ und das „standard procedure environment“, in dem Prozesse auf verschiedenen Ebenen kommunizieren, um dem Nutzer unterschiedliche „virtual machines“ zur Verfügung zu stellen. Nicht mehr das „overall system design“ und die nutzungsbezogene Bauform standen nun im Zentrum, sondern das einheitliche Design für Standardsoftware und Systemfamilien. Die Erörterung übergeordneter Ziele und Methoden der Software-Gestaltung einschließlich aller Designkonflikte wich den Gliederungsprinzipien von rationellen modularen Baukastenstrukturen. So stellte die Software-Architektur-Lehre des Schöpfers der Programmiersprache für numerische Steuerungen APT Douglas T. Ross vom MIT die Architektur in Form eines dreidimensionalen Modells von „building blocks of software engineering“ dar, das schon ganz an die späteren CASE-Methodenbaukästen erinnert. Alle Teile-System-Beziehungen werden hier zu einer widerspruchsfreien, sequenziell abarbeitbaren Schrittfolge der „rigorous decomposition“, deren Kern die Strukturierungsprinzipien der Modularität, Abstraktion, des Hiding bzw. der „hierarchical decomposition“ bilden: „So like all other architectures the structure of functional architecture is both modular and hierarchic.“[210] Der Architekturbegriff ging damit in einer an industriellen Produktionsmethoden orientierten Strategie des Software-Engineering auf, er entwickelte sich von einer Kompositions- zu einer reinen Dekompositionslehre. Der „software architect“, der seinen „individual programmers“ zu viele Freiheiten lässt und damit die Durchsetzung der standardisierten Komponenten-Produktion behindert, erschien mit Blick auf die Konvergenz digitaler Hardware und Software nun sogar als Negativ-Leitbild.[211] Seit der zweiten Hälfte der 70er Jahre wurden dann Begriff und Konzept der Software Architecture für nahezu zwei Jahrzehnte von der Leitmetapher Software Engineering an den Rand gedrängt.

---

210 Ross, Goodenough, Irvine 1975, Software Engineering, S. 17 ff.; Ross, Schomann 1977, Structured Analysis for Requirements Definition, S. 8, 11.

211 Wasserman, Belady 1978, The Oregon Report Software Engineering, S. 31.

4 Die Überwindung der Krise des Software Engineering durch die „Software Architecture Renaissance“

Eine Rückbesinnung auf das Architektur-Konzept kündigte sich jedoch ab Mitte der 80er Jahre an, als bei der Suche nach den Ursachen der noch immer unzureichenden Akzeptanz des Software Engineering in der Praxis dessen grundlegende Defizite in den Blick kamen. Man erkannte jetzt, dass zwar eine Vielzahl von analytischen Methoden für die strukturierte Software-Entwicklung und Tools für die Modulbildung und Ablaufplanung entwickelt worden waren, doch dass es nicht gelungen war, mit der wachsenden Komplexität von Softwaresystemen Schritt zu halten. Vor allem entdeckte man die große Lücke bei der Methodenunterstützung für die entscheidende Phase der Anforderungs-Ermittlung, des strukturbildenden „high level design“, das die „software architecture“ und den „architectural style“ entwirft: „ we have trained carpenters and contractors, but no architects.“[212] Führende Repräsentanten der Software Engineering-Community schlossen sich nun Brooks‘ Anschauungen an, dass die Grundlegung der Software-Architektur einen kreativen Prozess darstellt, der ein auf Erfahrungswissen beruhendes Urteilsvermögen von „heterosystem designers“ voraussetzt: „This can be done only by persons well versed in software and hardware in order the make the right choices and trade-offs, both from the economic and technical points of view.“[213] Wie Brooks sah auch der für IBM tätige John A. Zachmann den „architect“ eines Informationssystems als Mediator zwischen den beteiligten Akteuren. Er vermittelt mithilfe unterschiedlicher Modellrepräsentationen zwischen divergierenden Perspektiven der Nutzer, Designer und Auftraggeber und konfiguriert analog zu einem Bauarchitekten das „framework for information systems architecture”.[214]

Brooks selber schaltete sich auf dem IFIP-Kongress im September 1986 mit einer Generalkritik an dem Theorieverständnis und der Methodenentwicklung des Software Engineering in seinem schnell berühmt gewordenen Vortrag „No Silver Bullet“ in die Debatte ein. Zwei Jahrzehnte vergeblicher Umsetzungsbemühungen hätten gezeigt, dass die „*building* metaphor“ ihre Nützlichkeit verloren habe. Der vorab geplante konstruktive Ingenieurbau mit den ihn konstituierenden Elementen „specifications, assembly of components, and scaffolding“ sei die falsche „metaphor“ gewesen. Auch das tayloristische „Waterfall Model“ in einem linearen „downstream movement“ hielt er, wie er in einem Rückblick auf die Geschichte des Software Engineering näher ausführte, für einen grundlegend falschen methodischen Ansatz. Softwaresysteme sollten stattdessen, wie es in natürlichen komplexen Systemen geschehe, in einem

212 Perry, Wolf 1989, Software Architecture, S. 22, 24; dies., 1992, S. 52.
213 Belady 1986, Software Engineer, The System Designer, S. 53, f.
214 Zachman 1987, A Framework for Information Systems Architecture.

„incremental development“ wachsen: „*Growing* software organically, adding more and more function to systems as they are run, used, and tested.“[215]

Grundsätzlich wehrte sich Brooks gegen eine Verwissenschaftlichung der Software-Entwicklung, da „Science“ und „Design“ vom Prinzip her grundsätzlich verschiedene Aktivitäten seien. Die formale Exaktheit des „linear, step-by-step Rational Model“ greife nicht bei sozial bezogener Software, denn die Komplexität anwendungsnaher System-Architekturen und von „software objects” sind nicht vorübergehender, sondern prinzipieller Natur: „Our complexities are arbitrary, because they are fruits of many independent minds acting independently. [...] This complexity is compounded by the necessity to conform to an external environment that is arbitrary, unadaptable, and ever changing.“[216] Software Development, System Design und System Architecture folgten daher nach Brooks nicht dem Paradigma der Mathematik oder der Physik. Eine Überwindung von Komplexität durch abstrakte Modellierung müsse ebenso scheitern wie der Versuch, Heterogenität und Vielfalt auf eine „fundamental unified theory” zurückzuführen.

Das auffallend schnelle Revival der „software architecture“ auf US-Konferenzen nach 1986/87 wurde dann jedoch weniger von dem qualitativen „architectural concept“ geleitet als von stark abstrahierenden Architektur- und Stil-Metaphern. Treibend war hier vor allem das vom DoD gegründete und auch großenteils finanzierte „Software Engineering Institute“ (SEI), das die „leadership“ in der Disziplin sowie bei der Einführung neuer Methoden und des „software factory process“ beanspruchte.[217] Mary Shaw und Larry E. Druffel okkupierten 1988 den Architektur-Begriff als Ersatz für „Programming-in-the-Large“ und forderten nun auch formale Spezifikationsmethoden für den „software architecture level“ des Designs. Für Druffel bildete die Software-Architektur die „SEI vision for the future of the profession“ und er schlug vor, „that in addition to analytic methods, we needed focus on software architectures.“[218] Das SEI begründete so ab 1990 eine stark szientistische Richtung, die die unterschiedlichen „architectural styles“ und „design patterns“ von „software architectures“ klassifizieren, mit Abstraktionstechniken formalisieren und so den Bereich bislang informeller Design- und Strukturierungs-Entscheidungen in eine „Software Architecture Science“ und ein „Software Architecture Engineering“ überführen sollte.[219] Davon versprach man sich vor allem eine Reduzierung von Mehrfach-Entwicklungen, eine zuverlässigere Vorhersehbarkeit des Entwicklungsablaufes und ein verbessertes „intel-

215 Brooks 1987, No Silver Bullet, S. 18, Brooks 1975/95, The Mythical Man-Month, S. 180, 266 ff.; vgl. hierzu bes. Pflüger 2004, Writing, Building, Growing, S. 286 ff.

216 Brooks 1987, No Silver Bullet, S. 6; ders., 2008, A Science of Design is a Misled, S. 1 ff.

217 Barabacci, Habermann, Shaw 1985, The Software Engineering Institute, S. 5, 13, 16 f.

218 Druffel, 10th ICSE 1988, S. 44.

219 Kogut, Clements 1994, The Software Architecture Renaissance

lectual control“ des Managements über die Entwickler.[220] Architektur wird bei ihnen im Gegensatz zur Brookschen Auffassung vorwiegend als eine formale Strukturlehre verstanden und dem Software Engineering-Paradigma untergeordnet. Als „important subfield“ soll „Software Architecture“ das bisherige Lehrgebäude vollenden und mit dessen systematisiertem und kodifiziertem Wissen endlich das noch immer nicht erreichte Garmisch-Postulat einer theoriebasierten Engineering-Disziplin einlösen.[221]

Um dem SEI-Architektur-Konzept allgemeine Geltung zu verschaffen und auf dieser erweiterten Grundlage endlich die Institutionalisierung und Professionalisierung der „Engineering Discipline of Software“ abzuschließen, konstruierten Shaw und Clements 2006 ein weiteres historisches Narrativ der softwaretechnischen Entwicklung. Angelehnt an das „technology maturation model“ von Samuel Redwine und William Riddle, das vom qualitativ-deskriptiven zu formalisiertem systematischen Wissen führt, entwarfen sie ein Entwicklungsmodell, dessen Vorgeschichte sie mit Dijkstra und Parnas beginnen und dessen Endstadium sie mit dem weitgehend von der SEI-Philosophie bestimmtem „Golden Age of Software Architecture“ enden lassen.[222] Nachdem in der „Basic Research Phase“ von 1985-94 die qualitativen Deskriptionen der „architectural styles“ hervorgebracht und diese in der folgenden „Concept Formulation Phase“ von 1992-1996 mithilfe von „architecture description languages“ formalisiert wurden, komme es nach 2000 über „Top-down teaching“ zur Popularisierung und dadurch dank verbesserter Softwaretechniken zu einer „period of prosperity and excellent achievement“. In diesem „Golden Age“ wird es, so hoffen sie, endlich auch zum Abschluss der Professionalisierung mit „certified software architects“ kommen. Eine Initiative hierfür startete das SEI 2010 in Verbindung mit ähnlichen Bestrebungen von Boeing, weiteren Rüstungsfirmen und der weitgehend von Großfirmen gegründeten „International Association of Software Architects“ (IASA). Zertifizierung, Standardisierung von „high quality architectures“ und Flexibilisierung gehen dabei Hand in Hand: „Certification enforces common concepts and practices across your business units, making your architects more portable and able to pitch in where needed.“[223]

Neben diesem, Dominanz anstrebenden szientistischen Software Architektur-Konzept entstanden mehrere erfahrungsorientierte Richtungen. Eine bildete sich seit Ende der 80er Jahre in der OOP- und Pattern-Community unter dem Einfluss der Architekturtheorie Christopher Alexanders, der den Verlust des konkreten Gestaltens in den auf

220 Shaw 1990a, Toward higher-level abstractions for software systems, S. 127.

221 Shaw 1990b, Prospects for an Engineering Discipline of Software, S. 21 ff.; Shaw, Garlan 1994, An Introduction to Software Architecture, CMU-CS-94-166 ; Shaw 2010, Research toward an Engineering Discipline for Software Architecture; Shaw., Garlan, 1996, Software Architecture. Perspectives on an Emerging Discipline.

222 Shaw, Clements 2006, The Golden Age of Software Architecture.

223 Clements 2010, Certified Software Architects, S. 8

Software-Abstraktion fokussierten „architecture description languages“ kritisiert hatte. Bei der Bestimmung der Rolle des „architect“ im Entwicklerteam und der Kritik am rigorosen Formalismus der SEI-Theoretiker orientierte man sich hier auch an dem qualitativen Architektur-Konzept und dem „piecemeal growth of design“ von Brooks.[224] Die „Pattern-Oriented Software Architecture“ erweiterte die bisherigen Mikroarchitekturen der Objektklassen und Pattern-Typen um „highest-level patterns“ für die übergeordnete Organisation von Softwaresystemen. Als Ergebnisse der kollektiven Erfahrung in partizipatorischen Entwicklungsprozessen zielen formalisierte „architectural patterns“ und „architectural styles“ vor allem auch auf wiederverwendbare Designlösungen, die über OO-Plattformen verfügbar sind.[225] Diese Richtung war auch Ausgangspunkt für Produktfamilien- und Produktlinien-bezogene Ansätze, denen es besonders um wiederverwendbare Infrastrukturmuster für ganze Lebenszyklen von Softwareprodukten geht, um mit ihnen die naive „Lego-brick perspektive“ zu überwinden.[226]

Seit Anfang der 90er Jahren erschienen auch qualitative Stakeholder- und Organisations-orientierte Design-Konzepte, die sich noch deutlicher auf Frederick Brooks bezogen. Architektur wurde in ihnen als Resultat technischer, geschäftlicher und sozialer Einflüsse auf ein Softwaresystem gesehen. Im Zentrum stehen hier die unterschiedlichen Interessen, Perspektiven und Qualitätsanforderungen der beteiligten User und Akteure, die daraus resultierenden Zielkonflikte und Designprobleme, die ein sozial reflektierter Architekt auszubalancieren und zu integrieren hat. Die erste umfassende Designlehre dieses Typs wurde 1991 und in erweiterter Fassung mit Mark W. Maier 1997 von Eberhardt Rechtin in dem Lehrbuch „System Architecting. Creating and Building Complex Systems“ vorgelegt. Auf dem Hintergrund eigener Erfahrungen in der Softwareentwicklung in der Luft- und Raumfahrt und anhand historischer Fallstudien führte Rechtin die unvollendete Designlehre von Brooks weiter. Das „System *Architecting*“ ist für ihn das nicht-analytische Komplement zum analytischen „System Engineering“, das aufgrund seiner inhärenten Begrenzung auf quantitativ Messbares zum Systementwurf und zur Synthese des Gesamtsystems nicht in der Lage ist. Das Architecting hat es dagegen wegen der zentralen Klienten- und Realwelt-Orientierung mit „ill-structered complexity“ zu tun, für die es keine deterministischen Lösungen gibt, es beruht daher vor allem auf Erfahrungen und informellem Wissen, heuristischen Methoden sowie kreativen Synthesefähigkeiten der „configurator“.[227] Rechtin bündel-

---

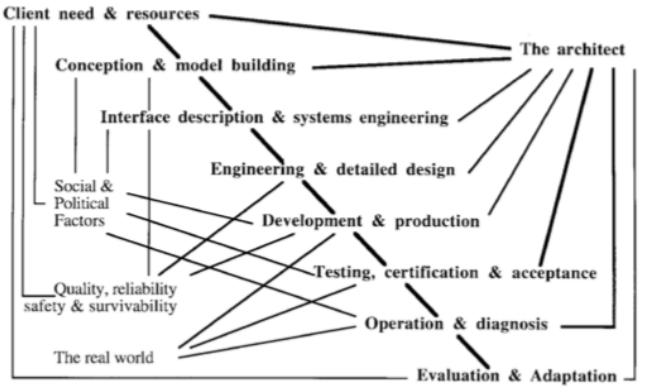
224 Alexander 1977, A Pattern Language; Coplien 1999, Reevaluating the Architectural Metaphor, S. 41; Booch 1996, Managing object-oriented software development, S. 252 ff.

225 Buschman, Meunier, Rohnert 1996, Pattern-Oriented Software Architecture; Laine 2001, The Role of SW Architecture.

226 Bosch, Design & Use of Software Architectures, 2000, S. 6 ff.

227 Rechtin 1991, Systems architecting, S. 1 ff., 12, 49 f.; Maier, Rechtin 1997, The Art of Systems Architecting, S. 7 ff.; vgl. auch Bass, Clements, Kazman 1998, Software Architecture in Practice, S. 5-12.

te seine Software Architecture-Lehre in einem auf die Klienten- und Architektenrolle fokussierten Wasserfallmodell, das mit seinen durchgängigen Iterationsschleifen stark an die Modelle von Rosove und Royce von 1967 bzw. 1970 erinnert.[228]



Erweitertes Wasserfallmodell im „System Architecting“ von Eberhardt Rechtin 1991, S. 4.

Wegen der großen Bedeutung von „good practice, domain knowledge, management and soft skills“ für die Qualität und konzeptionelle Integrität des Designs betonen die Vertreter des Architecting-Ansatzes wie Brown, McDermid und Albin besonders den Kunstcharakter des Softwaredesigns: „Software is an intellectual artifact – producing software is essentially a “pure design activity“. Gerade die Integration der Methoden aus sehr unterschiedlichen Bereichen verleihe der „Art of Software Architecture“ den Rang eines krönenden Paradigmas der Software-Evolution.[229] Der Wandel der sozialen Architekturen durch den Übergang von monolithischen zu verteilten und zu Internet- und Cloud-basierten Softwarearchitekturen und die dadurch in den Vordergrund gerückten Stakeholder- und Community-Aspekte haben im letzten Jahrzehnt schließlich noch einen ganz auf soziale und organisationale Dimensionen spezialisierten Nebenzweig der Softwarearchitektur entstehen lassen.[230]

Bei diesen OOP-nahen und in der Tradition des Brooks’schen Architektur-Konzeptes stehenden Ansätzen wird der „software architect“ als eine besondere Profession gesehen und die „Software-Architecture“ als eine eigenständige Disziplin neben dem Software-Engineering verstanden. Auch hier bemühte man sich, deren Institutionalisierung und Professionalisierung durch ein historisches Narrativ untermauern. So lassen Kruchten, Obbink, Stafford die Geschichte der Software Architektur mit der Erwähnung des Begriffs in der Rom-Konferenz von 1969 beginnen, an die sich die

228 Siehe oben Kap. 2.

229 Brown, McDermid 2007, Software Architecture as Art, S. 250; Albin 2003, The Art of Software Architecture S. 3 ff.

230 Rozanski, Woods 2008, Software Systems Architecture; Tamburri, Kazman, Fahimi 2016, The Architect’s Role in Community Shepherding; Woods 2016, Software Architecture in a Changing World.

Formulierung des Architektur-Konzeptes vor allem durch Brooks (1975), Rechtin sowie Walker E. Royce und Winston W. Royce (1991) anschließt, das dann durch Kruchten und anderere mit dem multiperspektivischen iterativen Entwicklungsansatz kombiniert wurde. Der vorläufige Abschluss der Institutionalisierung wird hier in der Gründung der IFIP Working Group 2.10 on Software Architecture und des „Worldwide Institute of Software Architects" (WWSIA) im Jahre 1999 gesehen. Das mit der SEI-IASA-Initiative konkurrierende WWISA lehnt sich eng an das Vorbild des „American Institute of Architects" an und sieht im „architect" ganz im Sinne von Brooks den „client advocate", der dessen Bedürfnisse softwaretechnisch umsetzt und ihn dabei durch den Konstruktionsprozess geleitet.[231] Obwohl dabei auch die formalen Methoden des SEI-Architekturkonzeptes zum Einsatz kommen, ist die Software Architektur hier eher als ein multidisziplinärer Verbund aus vielen „subareas", anderen Disziplinen und Communities angelegt. Ein goldenes Zeitalter sieht diese historische Erzählung noch nicht, dafür sei das Grundverständnis von Software Architektur noch viel zu offen, denn „despite the maturing of the discipline, we're far from having a consensus on a satisfying, short, crisp answer to this simple question—no widely accepted definition exists."[232] „Software Architecture" entwickelte sich am Ende infolge dieser Auffächerung in verschiedene Richtungen zu einem Umbrella-Konzept und Sammelbecken aller ungelösten Probleme und nicht erfüllten Versprechen des Software-Engineering, ja es wurde zu „one oft the most overused and least understood terms in software development circles."[233]

Nach 2000 scheinen die Konflikte zwischen den verschiedenen Richtungen und Philosophien allerdings abgeflaut zu sein und die anwendungsspezifische Arbeitsteilung zwischen den verschiedenen Schulen auf der einen Seite und die sukzessive Durchmischung von plan-driven und agilen Softwaretechniken auf der anderen das Feld zu bestimmen. In der Tendenz zu Kompromissen und zu einem generellen „technical pluralism"[234] hat auch der alte und seit den 90ern neu entfachte Streit „architecture" oder „engineering", „art" oder „science" seine Schärfe verloren: „The future of software engineering clearly lies somewhere in a parallel world that's neither pure art nor pure science.[235] Es verwirklicht sich somit nach drei Jahrzehnten, was Ian P. Sharp unter Hinweis auf die Erfahrungen beim OS/360 bereits 1969 bei der NATO-Konferenz in Rom verkündet hatte:

---

231 WWISA 1999, Role of the Software Architect (http://www.wwisa.org/wwisamain/role.htm) abgerufen 15.3.2006

232 Kruchten, Obbink, Stafford 2006, The Past, Present, and Future for Software Architecture, S. 23; ähnlich bereits Baragry, Reed 2001, Why we need a different view of software architecture, S. 12 ff.

233 Gorton 2006, Essential Software Architecture, S. 1.

234 Shapiro 1997, Splitting the Difference, S. 43 f., 48 ff.

235 Hurlburt, Voas 2016, Software Is Driving Software Engineering? S. 104.

I believe that a lot of what we construe as being theory and practice is in fact architecture and engineering; you can have theoretical or practical architects: you can have theoretical or practical engineers. I don't believe for instance that the majority of what Dijkstra does is theory—I believe that in time we will probably refer to the "Dijkstra School of Architecture".[236]

Logo des 1999 gegründeten „Worldwide Institute of Software Architects"
(http://www.wwisa.org/ abgerufen 15.3.2006)

236 Sharp in Buxton, Randell 1970, Rome Report, S. 12.

Literatur

(Alle Online-Zitate zuletzt überprüft am 10.5.2019)

Albin, Stephen T. (2003): The Art of Software Architecture: Design Methods and Techniques. Indianapolis.

Alexander, Christopher (1964): Notes on the Synthesis of Form, Cambridge, MA.: Harvard University Press.

Alexander, Christopher (1977): A Pattern Language: towns, buildings, construction. New York: Oxford University Press.

Baber, Robert L. (1997): Comparison of Electrical 'Engineering' of Heaviside's Times and Software 'Engineering' of Our Times, in: Annals of the History of Computing, 19, 4, S. 5-17.

Baber, Robert L. (1998): Software engineering education: issues and alternatives, in: Annals of Software Engineering 6 (1998) 39–59.

Baker, F. Terry (1972): System quality through structured programming, in: AFIPS '72 (Fall, part I): Proceedings of the FJCC December 5-7, S. 339-343.

Baker, F. Terry and Harlan Mills (1973): Chief Programmer Teams. In: Datamation 19,12, S. 58-61.

Barabacci, Mario R.; Habermann, A. Nico; Shaw, Mary (1985): The Software Engineering Institute: Bridging Practice ans Potential, in: IEEE Software , 2, 6, S. 4-21.

Baragry, Jason and Karl Reed (2001): Why we need a different view of software architecture, in: Proceeding Working IEEE(IFIP Confernce on Architecture Aug 28-31, S. 125-134.

Basili, Victor R. (2006), The Past, Present, and Future of Experimental Software Engineering, in: Journal of the Brazilian Computer Society 12, 3, S. 7-12.

Basili, Victor R.; Briand, Lionel et al. (2018): Software Engineering Research and Industry, in: IEEE Software, 35, 5, S. 44-49.

Bass, Len; Clements, Paul and Rick Kazman, Software Architecture in Practice. SEI Series in Software Engineering. 1998: Addison-Wesley.

Bauer, Friedrich L. (1969): Software Engineering. A Conference Report. In: Datamation 15, 10, S. 189-192.

Bauer, Friedrich L. (1970): Der *computer* in unserer Zeit, Festrede bei der Bayer. Akademie der Wissenschaften, München 6.6.1969, München.

Bauer, Friedrich L. (1971): Software Engineering, in: IFIP Congress 1971, S. 530-538.

Bauer, Friedrich L. (1972a): Software Engineering, in: Proceedings of the IFIP Congress 71, hrsg. von C. V. Freiman, Amsterdam, London, S. 530-538.

Bauer, Friedrich L. (1972b): Training Future Software Engineers, in: INFOTECH Report 11, Software Engineering, Maidenhead 1972, S. 355-367.

Bauer, Friedrich L. (1973), Software Engineering, in: F. L. Bauer, (Hrsg.) Software Engineering. An Advanced Course, Heidelberg, New York: Springer-Verlag, S. 522-545.

Bauer, Friedrich L. (1976): Programming as an Evolutionary Process, in: Proc. 2nd. Intern. Conf. Software Engineering (ICSE), IEEE Computer Society, S. 223-234.

Bauer, Friedrich L. (1980): Mathematik und Informatik, in: Dörfler, Willibald; Schauer, Helmut (Hrsg.), Wechselwirkungen zwischen Informatik und Mathematik, Wien, München: Oldenbourg, S. 17-39

Bauer, Friedrich L. (1993): Software Engineering - wie es begann. In: Informatik-Spektrum, 16, 5, S. 259-260.

Bauer, Friedrich L. (2004): Die Algol-Verschwörung, in: Hellige, H. D. (Hg.), Geschichten der Informatik. Visionen, Paradigmen und Leitmotive, Berlin, Heidelberg, New York: Springer-Verlag, S. 237-254.

Bauer, Friedrich L. (2007): Geburt der Informatik in München, in: ders. (Hrsg.) 40 Jahre Informatik in München 1967-2007, Fakultät für Informatik (IN) der TU München.

Bauer, Friedrich L. (2009): Informatik – Geburt einer Wissenschaft (1983), wiedergedr. in: ders., Historische Notizen zur Informatik, Berlin, Heidelberg, New York: Springer Verlag, S. 21-29.

Baum Claude (1981): The System Builders: The Story of SDC. System Development Corp., Santa Monica, Calif.

Belady László A. (1986): Software Engineer, The System Designer, in: Proceedings 1986 ACM 14th Annual Conference on Computer Science, S.53-55.

Belady, László A.; Lehman, Meïr M. (1974): Programming Systems Growth Dynamics, in: Computer Reliability, Infotech State of Art Lecture 20, 1974, S. 391-412.

Benington, Herbert D. (1956): Production of large computer programs, in: Proc. Symposium on Advanced Programming Methods, Washington, D.C. ONR Symposium Report ACR-15, June 1956, S. 15-27.
Benington, Herbert D. (1983): Production of Large Computer Programs, in: Annals of the History of Computing 5, 4, S. 350-361.
Blaauw, Gerrit A.; Brooks Jr., Frederick P. (1997): Computer Architecture. Concepts and Evolution, Reading, MA, Harlow, Menlo Park, CA: Addison Wesley.
Boehm, Barry W. (1979): Software Engineering – As it is, in: 4th ICSE 1979, München, S. 11-21.
Boehm, Barry W. (1987): Software Process Management: Lessons Learned from History, in: ICSE '87 Proceedings of the 9th international conference on Software Engineering, S. 296-298.
Boehm, Barry W. (1981): Software Engineering Economics, Upper Saddle River, N. J.: Prentice Hall.
Boehm, Barry W. (2006): A View of 20th and 21st Century Software Engineering, in: ICSE'06, May 20–28, 2006, Shanghai, S. 12-29.
Boehm, Barry W. (2011): Some Future Software Engineering Opportunities and Challenges, in: Nanz, Sebastian (Hg.): The Future of Software Engineering, Berlin Heidelberg: Springer-Verlag, S. 1-31.
Booch, Grady (1996): Managing object-oriented software development, in: Annals of Software Engineering 2, S. 237-258.S.
Booch, Grady (2018): The History of Software Engineering, in: IEEE Software, 35, 5, S. 108-114.
Bosch, Jan (2000): Design & Use of Software Architectures. Adopting and Evolving a product-line approach, Harlow, London, New York: Addison-Wesley.
Brooks Jr., Frederick P. (1962): Architectural Philosophy, in: Buchholz, W. (Hg.), Planning a Computer System. Project Stretch, New York: Mcgraw-Hill.
Brooks Jr., Frederick P. (1965) The Future of Computer Architecture, in: Information Processing 1965, Proceedings of IFIP Congress '65, 2 Bde. Washington, D. C., London 1965, Bd. 2, S. 87-91.
Brooks Jr., Frederick P. (1975): The Mythical Man-Month. Essays in Software Engineering, Reading, MA, Menlo Park, CA., London: Addison Wesley.
Brooks Jr., Frederick P. (1975/95): The Mythical Man-Month. Essays in Software Engineering, 20 Anniversary Editon Reading, Ma, Menlo Park, Cal., London: Addison Wesley.
Brooks Jr., Frederick P. (1987): No Silver Bullet. Essence and Accidents of Software Engineering, in: IEEE Computer 20, S. 10-19.
Brooks Jr., Frederick P. (2008): A Science of Design is a Misled and Misleading Goal (Perspectives Workshop: Science of Design: High-Impact Requirements for Software-Intensive Systems, Dagstuhl Perspective Workshop 2008 http://drops.dagstuhl.de/opus/volltexte/2009/1976/pdf/08412.BrooksFrederick.Paper.1976.pdf.
Brooks Jr., Frederick P. (2018): ICSE 2018 - Plenary Sessions - Frederick P. Brooks Jr. Talk (https://www.youtube.com/watch?v=StN49re9Nq8).
Brown, George W. (1962): A New Concept of Programming, in: Greenberger, Martin (Hrsg.), Management and the Computer of the Future, Cambridge, Mass., New York, London: MIT Press, Wiley, S. 250-271.
Brown, Alan W.; McDermid, John A. (2007): Software Architecture as Art, in: Oquendo, F. (Hg.): The Art and Science of Software Architecture. ECSA 2007, LNCS 4758, S. 237 – 256.
Broy, Manfred (2007): Von der Ingenieurmathematik zur Informatik, in: Bauer, Friedrich L. (Hrsg.) 40 Jahre Informatik in München 1967-2007, Fakultät für Informatik (IN) der TU München, S. 239-246.
Broy, Manfred (2017): Informatik als Wissenschaft an der Technischen Universität München und ihre Anwendung in Wirtschaft und Gesellschaft, in: Informatik-Spektrum 40, 2, S. 201-204.
Broy, Manfred (2018): Yesterday, Today, and Tomorrow. 50 Years of Software Engineering, in: IEEE Software, 35, 5, S. 38-43.
Broy, Manfred; Rombach, Dieter (2002): Software Engineering. Wurzeln, Stand, Perspektiven, in: Informatik-Spektrum, 25, 6, S. 438-451.
Broy, Manfred; Jarke, Matthias; Nagl, Manfred; Rombach, Dieter (2006): Manifest: Strategische Bedeutung des Software Engineering in Deutschland, in: Informatik-Spektrum, 29, 3, S. 210-221.
Buschman, Frank; Meunier, Regine; Rohnert, Hans. et a. (1996): Pattern-Oriented Software Architecture. A System of Patterns, New York: Wiley.
Buxton John N. (1978): Software Engineering. In: Gries D. (Hg.) Programming Methodology. Texts and Monographs in Computer Science. New York, NY: Springer, S. 23-28

Buxton John N. und Brian Randell (1970): Software Engineering Techniques. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969, NATO Science Committee. (Online-Version: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.)

Cain, Brendan G.; Coplien ; James O. and Nell B. Harrison (1996): Social patterns in productive software development organizations, in: Annals of Software Engineering 2, S. 259-286.

Campbell-Kelly, Martin; Aspray, William (1996): Computer. A History of the Information Machine, New York: Basic Books.

Card, David N. ; McGarry, Frank E. und Gerald T. (1987): Page 1987, Evaluating Software Engineering Technologies, in: IEEE Transactions on Software Engineering, 13, 7, S. 845-851.

Christie, Lee S. und Kroger, Martin G. (1962): Information Processing for Military Command, in: Datamation 8, 6, S. 58-61.

Christie, Lee S.; Kroger, Martin G. (1962): Information Processing for Military Command, in Datamation June, S 58-61.

Cockburn, Alistair (2004): The End of Software Engineering and the Start of Economic-Cooperative Gaming, in: Computer Science and Information Systems (ComSIS), 1, 1, S. 1-32.

Coplien, James O. (1999): Reevaluating the Architectural Metaphor: Toward Piecemeal Growth, in: IEEE Software, 16, 5, S. 40-44.

Coy, Wolfgang (2004): Was ist Informatik? Zur Entstehung des Faches an den deutschen Universitäten, in: Hellige, H. D. (Hg.), Geschichten der Informatik. Visionen, Paradigmen und Leitmotive, Berlin, Heidelberg, New York 2004, S. 473-498.

Curtis, Bill; Krasner, Herb und Neil Iscoe 1988: A Field Study Of The Software Design Process For Large Systems, in: Communications of the ACM, 31, 11, S. 1268

Cusumano, Michael A. (1991): Factory Concepts and Practices in Software Development, in: Annals of the History of Computing 13, 1, S. 3-32.

DeMarco, Tom (2009): Software engineering: An Idea Whose Time Has Come and Gone? In: IEEE Software, 26, 4, S. 95 f.

Denert, Ernst (1991): Software-Engineering, Berlin, Heidelberg, New York : Springer-Verlag

Denning, Peter (1997/99): Computer Science: The Discipline. (http://denninginstitute.com/pjd/PUBS/ENC/cs99.pdf).

Dijkstra, Edsger W. (1968a): The Structure of the “THE"-Multiprogramming System, in: Communications of the ACM, 11 5, S. 341-346.

Dijkstra, Edsger W. (1968b): Verslag von het bezouk aan de NATO Conference on Software Engineerind, Typoskript, E. W. Dijkstra Archive, EWD246 (https://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD246.html).

Dijkstra, Edsger W. (1972): The Humble Programmer, in: Communications of the ACM, 15, 4, S. 859-866.

Dijkstra, Edsger W. (1977): Programming: From Craft to Scientific Discipline, in: Proceedings of the International Computing Symposium, Apr. 4-7 1977, S. 23-30 (Ms. in EWD 566).

Dijkstra, Edsger W. (1979): Software Engineering: as it should be, in: 4th ICSE 1979, München, S. 442-448.

Dijkstra, Edsger W. (1989): The Cruelty of Really Teaching Computing Science. In: Communications of the ACM, 32,12, S. 1398-1414.

Dijkstra, Edsger W. (2001): An Interview with Edsger W. Dijkstra. Conducted by Philip L. Frana on August 2, 2001, Charles Babbage Institute, OH 330 (https://conservancy.umn.edu/bitstream/handle/11299/107247/oh330ewd.pdf?sequence=1&isAllowed=y).

Druffel, Larry (2018): 10th ICSE 1988, in: Bianculli, Domenico; Medvidovi´c, Nenad und David S. Rosenblum (Hg.)40 Editions of ICSE: the ruby anniversary celebration, 2018, S. 44.

Endres, Albert (1996): A synopsis of software engineering history: the industrial perspective, in: History of Software Engineering, Position Papers for Dagstuhl Seminar 9635, S. 20–24.

Ensmenger, Nathan (2010): The Computer Boys Take over: Computers, Programmers, and the Politics of Technical Expertise, Cambridge, Mass.–London: MIT Press.

Ensmenger, Nathan; Aspray, William (2002): Software as Labor Process, in: Hashagen, Ulf; Keil-Slawik, Reinhard; Norberg, Arthur L.: Mapping the History of Computing Software Issues, Berlin, Heidelberg, New York: Springer-Verlag, S. 139-165.

Erdogmus, Hakan; Medvidovic´, Nenad; Paulisch, Frances (2018): 50 Years of Software Engineering, in: IEEE Software, 35, 5, S. 21.

Floyd, Christiane (1994): Software-Engineering - und dann? In: Informatik-Spektrum 17, 1, S. 29-37.
Floyd, Robert W. (1979): The Paradigms of Programming, in: Communications of the ACM, 22,8, S. 455-460.
Freeman; Peter; Wasserman Anthony I. und Richard E. Fairley (1976): Essential elements of software engineering education, in: ICSE '76 Proceedings of the 2nd international conference on Software engineering (ICSE '76), S. 56-62.
Futatsugi, Kokichi (2018): ICSEs before and after ICSE 98, in: Bianculli, Domenico; Medvidovi´c, Nenad und David S. Rosenblum (Hg.) 40 Editions of ICSE: the ruby anniversary celebration, 2018, S. 93-95.
Galler, Bernhard (1989): Thoughts on Software Engineering, in: 11th ICSE 1989, S. 97.
Garlan, David (2000): Software Architecture: A Roadmap. In: Finkelstein, Anthony (Hg.), The Future of Software Engineering: ACM Press.
Gibbs, W. Wayt (1994): Software's Chronic Crisis, in: Scientific American, 271, 3, S. 80-95.
Gill, Stanley (1972): The Origins and Meaning of Software Engineering, in: INFOTECH Report 11, Software Engineering, Maidenhead, S. 218-242.
Goode, Harry H.; Machol, Robert, E. (1957): System Engineering: An Introduction to the Design of Large-scale Systems, New York u.a.: McGraw-Hill.
Goos, Gerhard (1994): Programmiertechnik zwischen Wissenschaft und industrieller Praxis, in: Informatik-Spektrum 17, 1, S. 11-20.
Goos, Gerhard (2008): Die Informatik in den 70er Jahren, in: Reuse, Bernd; Vollmar, Roland (Hg.): Informatikforschung in Deutschlan, Berlin, Heidelberg, New York: Springer-Verlag S. 133-150.
Gordon, Robert (1968): Review of Charles Lecht „The Management of Computer Programming Projects", in: Datamation 14, 4, S. 2000-2002.
Gorton, Ian (2006): Essential Software Architecture, Berlin, Heidelberg, New York: Springer-Verlag.
Gries, David (1989): My Thoughts on Software Enineering in th Late 1960s, in: 11th ICSE 1989, S. 89.
Grosch, Herb R. J. (1961a): Software in sickness and health, in: Datamation, 7,7, S. 32-33.
Grosch, Herb R. J. (1961b): Software on the Couch, in: Datamation, 7, 11, S. 23-24.
Haigh, Thomas (2002): Software in the 1960s as Concept, Service and Product, in: IEEE Annals of the History of Computing, 24, 1, S. 5-13.
Haigh, Thomas (2010a): "Crisis, What Crisis?" Reconsidering the Software Crisis of the 1960s and the Origins of Software Engineering. DRAFT Version for discussion at the 2nd Inventing Europe/Tensions Of Europe Conference, Sofia, June17-20 2010. (http://www.tomandmaria.com/tom2/Writing/SoftwareCrisis_SofiaDRAFT.pdf).
Haigh, Thomas (2010b): Dijkstra's Crisis: The End of Algol and Beginning of Software Engineering, 1968-72. Draft for discussion in SOFT-EU Project Meeting, September 2010. (http://www.tomandmaria.com/Tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf).
Halsted, David G. (2018): The Origins of the Architectural Metaphor in Computing. Design and Technology at IBM, 1957–1964, in: IEEE Annals of the History of Computing, 40, 1, S. 61-70.
Head, Robert V. (1963a): Real-Time Programming Specifications, in: Communications of the ACM, 6,7, S. 376-379.
Head, Robert V. (1963b): The Programming Gap in Real-time Business Systems, in: Datamation 9, 2, S. 39-41.
Head, Robert V. (1964): Real-time Business Systems, New York: Holt, Rinehart and Winston.
Head, Robert V. (1965): Planning for Generalized Business Systems, in: AFIPS 27, FJCC 1965, Part I, S. 153-159.
Hellige, Hans Dieter (2003): Zur Genese des informatischen Programmbegriffs. Begriffsbildung, metaphorische Prozesse, Leitbilder und professionelle Kulturen, in: Karl-Heinz Rödiger (Hg.): Algorithmik – Kunst – Semiotik. Hommage für Frieder Nake, Heidelberg: Synchron Wissenschaftsverlag der Autoren, S. 42–73.
Hellige, Hans Dieter (2004): Die Genese von Wissenschaftskonzepten der Computerarchitektur: Vom „system of organs" zum Schichtenmodell des Designraums, in: ders. (Hg.), Geschichten der Informatik. Visionen, Paradigmen und Leitmotive, Berlin, Heidelberg, New York: pringer-Verlag, S. 411-471.
Hoare, Charles Antony Richard (1978a): The Engineering of Software: A Startling Contradiction, in: Gries, David (Hg.): Programming Methodology: A Collection of Articles by Members of IFIP WG2.3. New York, Heidelberg, Berlin: Springer-Verlag, S. 37-41.
Hoare, Charles Antony Richard (1978b): Software Engineering: a Keynote Adress, in: 3 rd ICSE, Atlanta, GA Mai 1978, New York, S. 1-4.

Holdiman, Thomas A., (1962): Management Techniques for Real Time Computer Programming, in: Journal of the ACM 9, S. 387-404.
Hosier, W. A. (1961): Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming, in: IRE Trans. Engineering Management, EM-8, S. 311-327.
Hurlburt, George und Jeffrey Voas (2016): Software Is Driving Software Engineering? In: IEEE Software, 33,1, S. 101-104.
ICSE2018 - Plenary Session - Panel: 50 years of Software Engineering & Celebrating the 40th ICSE (https://www.youtube.com/watch?v=5HssVXx7xy8&list=PLMr4f0qPvkXfzFNpLXXITK3TnjCZbZTYS&index=4).
ICSE–NIER 2018: ACM/IEEE 40th International Conference on Software Engineering: New Ideas and Emerging Results. Gothenburg, Sweden — May 27 - June 03, 2018. (https://conferences.computer.org/icse/2018/#!/toc/2).
Jacobs, John F.(1986) The SAGE Air Defense System: A Personal History. Bedford, Mass: MITRE Corp.
Jones, Peter D. (1969): Operating System Structures, in: Information Processing 68, Amsterdam 1969: North-Holland Publishing Co, S. 525-530.
Jones, Peter D.; Lincoln, Neil R.; Thornton, James E. (1972): Whither Computer Architecture ? in: Information Processing 71, Amsterdam 1972: North-Holland Publishing Co, S. 729-763.
Katzan, Harry, Jr. (1970): Advanced Programming. Programming and Operating Systems, New York, Cincinnati, Toronto, London: Van Nostrand Reinhold.
Klaeren, Herbert (1994): Probleme des Software-Engineering. Die Programmiersprache - Werkzeug des Entwicklers, in: Informatik-Spektrum 17, 1, S. 21-28.
Kneuper, Ralf (2017) Sixty Years of Software Development Life Cycle Models, in: Annals of the History of Computing 39, 3, S. 41-54.
Kogut, Paul und Paul Clements (1994): The Software Architecture Renaissance. (ftp://itin.sei.cmu.edu/pub/sati/Papers_and_Abstracts/SW_Arch_Renaissance.ps).
Kruchten, Philippe; Obbink, Henk und Judith Stafford (2006): The Past, Present, and Future for Software Architecture, in: IEEE Software 23,2, S. 22-30.
Krückeberg, Fritz (20020): Die Geschichte der GI (Veröffentlichungen und Dokumente zur Geschichte und Entwicklung der GI), 2. Aufl.
Laine, Petri K. (2001): The Role of SW Architecture in Solving Fundamental Problems in Object-Oriented Development of Large Embedded SW Systems, in: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WlCSA'O1), S. 14-23.
Larman, Craig and Victor R. Basili (2003): Iterative and Incremental Development: A Brief History, in: IEEE Computer, 36, 6, S. 2–11.
Lehman, Meïr M. (1980): Programs, Life Cycles, and Laws of Software Evolution, in: Proceedings of the IEEE, 68,9, S. 1060-1076.
Lehman, Meïr M., and Belady, László (1985): Program Evolution: Processes of Software Change, New York: Academic Press.
Leimbach, Timo (2011): Geschichte der Softwarebranche in Deutschland, Stuttgart: Fraunhofer Verlag.
Lewerentz, Claus; Rust, Heinrich (1998): Sind Softwaretechniker Ingenieure? Lehrstuhl Software Systemtechnik, BTU Cottbus, Internal Report I-12/1998; engl. Version: Are software engineers true engineers? in: Annals of Software Engineering 10 (2000) S. 311–328.
Liebowitz, Burt H. (1967): The technical specification – key to management control of computer programming, in: AFIPS 30, SJCC, S. 51-59.
Lynch, W. C. 1972, Operating System Performance, in: Communications of the ACM, 15, 7, S. 579-585
Mack, Ruth P. (1971): Planning on Uncertainty: Decision Making in Business and Government Administration, New York: Wiley-Interscience.
MacKenzie, Donald (2001): Mechanizing Proof. Computing, Risk, and Trust, Cambridge, Mass., London: MIT Press.
MacKenzie, Donald (2002): A View from the Sonnenbichl. On the Historical Sociology of Software and System Dependability, in: Hashagen, Ulf; Keil-Slawik, Reinhard; Norberg, Arthur L.: Mapping the History of Computing Software Issues, Berlin, Heidelberg, New York: Springer-Verlag, S. 97-116.
Mahoney, Michael (1990): The Roots of Software Engineering. In: CWI Quarterly 3,4, S. 325-334 (http://www.princeton.edu/~mike/articles/sweroots/sweroots.htm).
Mahoney, Michael (1991): Software and the Assembly Line, in: Oskar Blumtritt, Hartmut Petzold (Hg.), Technohistory of Electrical Information Technology. Preliminary Papers, München

Mahoney, Michael (1992): Computers and Mathematics: The Search for a Discipline of Computer Science, in: Echeverría, Javier; Ibarra, A.; Mormann, T. (Hg.): The Space of Mathematics, Berlin, New York De Gruyter, S. 347-61.
Mahoney, Michael (1996): Finding a History for Software Engineering, in: Brennecke, Andreas und Reinhard Keil-Slawik (Hg.): History of Software Engineering, Dagstuhl Seminar 9635, 1993, S. 29-33.
Mahoney, Michael S. (2004): Finding a History for Software Engineering, in: Annals of the History of Computing 26, 1, S. 8-19.
Maier, Mark W.; Rechtin, Eberhardt (1997): The Art of Systems Architecting: CRC Press.
Martin, James (1965): Programming Real-Time Computer Systems, Englewood Cliffs, N.J.: Prentice Hall.
Martin, James (1967): Design of Real-Time Computer Systems, Englewood Cliffs, N.J.:Prentice Hall.
Martin, James (1991), Rapid Application Development : Macmillan Coll Div.
McClure, Robert (1976): Software – The Next Five Years, in: Wasserman, Anthony und Peter Freeman (Hg.): Software Engineering Education. Needs and Objectives. New York, Heidelberg, Berlin: Springer-Verlag, S. 5-9.
McIlroy, M. Douglas (1969): Mass Produced Software Components, in: Naur, Peter; Randall, Brian, Software Engineering: Report on a Conference Sponsered by the NATO Science Committee , Garmisch, Okt. 1968, NATO Scientific Affairs Division, Brüssel, S. 138-150.
McIlroy, M. Douglas (1972): The Outlook for Software Components, in: INFOTECH Report 11, Software Engineering, Maidenhead, S. 243-252
Mealy, G. H.; Witt, B. I.; Clark, W. A. (1966): The Functional Structure of OS/360, in: IBM Systems Journal 5, 1, S. 2-51.
Naur, Peter and Brian Randell (1969): Software Engineering. Report on a conference sponsored by the NATO Science Committee Garmisch, Germany, 7th to 11th October 1968. NATO Scientific Affairs Division, Brüssel. (Online-Version: http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF).
Nofre, David; Priestley, Mark; Albert, Gerard (2014): When Technology Became Language. The Origins of the Linguistic Conception of Computer Programming, 1950–1960, in: Technology and Culture, 55, 1, S. 40-75.
Oettinger, Antony G. (1964): A bull's eye view of management and engineering information systems, in: ACM Proceedings of the 19th National Conference, Bd 1, S. 1-14. 1964, Baltimore, Md: Spartan Books.
Oettinger, Antony G. (1966): President's letter to the ACM Membership, in: Communications of the ACM, 9, 8, S. 545-546.
Oettinger, Antony G. (1967): The Hardware-Software Complementarity, in: Communications of the ACM, 10, 10, S. 604-606.
Opler, Ascher (1960): Current Problems in Automatic Programming, in: AFIPS Bd. 17, WJCC, S. 365-369.
Opler, Ascher (1961): Trends in Programming Concepts, in: Datamation 7, 1, S. 13-15.
Opler, Ascher (1967): The Receeding Future, in: Datamation 13, 9, S. 31-32.
Osterweil, Leon J. (1986): A process-object centered view of software environment architecture, in: Conradi, Reidar; Didriksen, Tor M. und Dag H. Wanwik (Hg.): Advanced Programming Environments. Proceedings of an International Workshop Trondheim, Norway, June 16–18, (LNCS, Bd. 244).
Osterweil, Leon J. (2011): A Process Programmer Looks at the Spiral Model: A Tribute to the Deep Insights of Barry W. Boehm, in: International Journal of Software and Informatics, 5, 3, S. 457-474.
Parnas, David Lorge (1996): Software Engineering: An Unconsummated Marriage, in: Brennecke, Andreas und Reinhard Keil-Slawik (Hg.): History of Software Engineering, Dagstuhl Seminar 9635, 1993, S. 37-42.
Parnas, David Lorge (1998): Software engineering programmes are not computer science programmes, in: Annals of Software Engineering 6, S. 19-37.
Perry, Dewayne E. (1987): Software Interconnection Models, in: 9th ICSE 1987, S. 61-71.
Perry, Dewayne E.; Wolf, Alexander L. (1989): Software Architecture (http://users.ece.utexas.edu/~perry/work/papers/swa89.pdf).
Perry, Dewayne E.; Wolf, Alexander L. (1992): Foundations for the Study of Software Architecture, in: ACM Software Engineering Notes, 17, 4, S. 40-52.

Pflüger, Jörg (2004): Writing, Building, Growing: Leitvorstellungen der Programmiergeschichte, in: Hellige, Hans Dieter (Hrsg.), Geschichten der Informatik. Visionen, Paradigmen und Leitmotive, Berlin, Heidelberg, New York, S. 275-320.
Potts, Colin (1993): Software Research Revisited, in: IEEE Software, 10, 5, S. 19-28.
Raccoon, L. B. S. (1997): Introduction. Fifty Years of Progress in Software Engineering, in: Software Engineering Notes, 22, 1, S. 88-104.
Raccoon, L. B. S. (1998): Introduction. Toward a Tradition of Software Engineering, in: Software Engineering Notes, 23, 3, S.105-110.
Randell, Brian (1979): Software Engineering: As it was in 1968, in: 4th ICSE 1979, München, S. 1-10.
Randell, Brian (1993): The 1968/69 NATO Software Engineering Reports, in: Brennecke, Andreas und Reinhard Keil-Slawik (Hg.): History of Software Engineering, Dagstuhl Seminar 9635, 1993, S. 37-42.
Randell, Brian (1998): Memories of the NATO Software Engineering Conferences, in: IEEE Annals of the History of Computing, 20, 1, S. 51-54.
Randell, Brian (2008): Position Statement - How Far Have We Come? In: 32nd COMPSAC 2008, Turku, S. 8.
Randell, Brian (2018): ICSE 2018 - Plenary Sessions – Brian Randell Talk (https://www.youtube.com/watch?v=YdEGNpbD8FY).
Randell, Brian (2018): Fifty Years of Software Engineering or The View from Garmisch, in: ICSE 2018 (https://arxiv.org/pdf/1805.02742.pdf)
Ratynski, Milton V. (1967): The Air Force computer program acquisition concept, in : AFIPS 30, SJCC, S. 33-44.
Rechtin, Eberhardt (1991): Systems architecting: creating and building Complex Systems, Englewood Cliffs, NJ: Prentice-Hall.
Redmond, Kent C.; Smith, Thomas M. (2000): From Whirlwind to MITRE: The R&D Story of The SAGE Air Defense Computer. Cambridge, Mass. : MIT Press.
Reeves, Jack W. (1992/2005): What Is Software Design? In: Ders. : Code as Design. Three Essays by Jack W. Reeves, developer.* Magazine, S. 1-11.
Rosove, Perry E. (Hrsg.) (1967): Developing Computer-Based Information Systems, New York, London, Sydney : Wiley.
Ross, Douglas T.; Goodenough, John B. und C A Irvine (1975): Software Engineering: Process, Principles, and Goals, in: IEEE Computer 8, 5, S. 17-27.
Ross, Douglas T. und Kennet E. Schomann Jr. (1977): Structured Analysis for Requirements Definition, in: IEEE Trans. Software Engineering, 3, 1, S. 6-15.
Royce, Winston W. (1970): Managing the Development of Large Software Systems, in: Proceedings of IEEE WESCON 26, S. 1–9.
Royce, Winston W. und Walker Royce (1991): Software Architecture: Integrating Process and Technology, in: TWI Quest, 14, 1, S. 2–15.
Rozanski, Nick ; Woods, Eoin (2008): Software Systems Architecture. Working with Stakeholders Using Viewpoints and perspectives, 2. Aufl.: Addison-Wesley.
Ruparelia, Nayan B. (2010): Software Development Lifecycle Models, in: ACM SIGSOFT Software Engineering Notes, 35, 3, S. 8-13.
Sackman, Harold (1967): Computer, System Science and Evolving Society. The Challenge of Man-Machine Digital Systems. New York, London, Sydney: Wiley.
Schmidt, Kjeld (2011): Cooperative Work and Coordinative Practices. Contributions to the Conceptual Foundations of Computer-Supported Cooperative Work (CSCW), London, Dordrecht, Heidelberg: Springer-Verlag.
Seidman, Stephen B. (2008): The Emergence of Software Engineering Professionalism, in: IFIP 20th World Computer Congress , E-Government; ICT Professionalism and Competences. Service Science. Boston: Springer-Verlag, S., 59–67.
Shapiro, Stuart (1997): Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering, in: IEEE Annals of the History of Computing, 19, 1, S. 20-54.
Shaw, Mary (1989a): Larger scale systems require higher-level abstractions, in : ACM SIGSOFT Software Engineering Notes 14, 3, S. 143-146.
Shaw, Mary (1989b): Remembrances of a Graduate Student, in: 11th ICSE 1989, S. 99-100.
Shaw, Mary (1990a): Toward higher-level abstractions for software systems, in: Data & Knowledge Engineering 5 (1990) 119-128.

Shaw, Mary (1990b): Prospects for an Engineering Discipline of Software, in : IEEE Software, 7, 6, S. 15-24

Shaw, Mary (1996): Three Patterns that help explain the development of Software Engineering, in: Brennecke, Andreas und Reinhard Keil-Slawik (Hg.): History of Software Engineering, Dagstuhl Seminar 9635, 1993, S. 52-56.

Shneiderman, Ben und John M. Carroll (1988): Ecological Studies of Professional Programmers, in: Communications of the ACM, 31, 11 S. 1256-1258.

Sommerville, Ian (2011): Software Engineering, 9. Aufl. Boston, Columbus, Indianapolis: Addison Wesley.

Spooner, Chris R. (1971): A Software Architecture for the 70's: Part I - The General Approach, in: Software - Practice and Experience. 1, 1, S. 5-37.

Tamburri, Damian A.; Kazman, Rick; Fahimi, Hamed (2016) : The Architect's Role in Community Shepherding, in: IEEE Software, 33, 6, S. 70-79.

Tichy, Walter F. ; Haberman, Nico ; Prechelt, Lutz (1992) : Future Directions in Software Engineering: First Dagstuhl Seminar Febr. 17–21, 1992, Schloß Dagstuhl (https://pdfs.semanticscholar.org/e140/a58885cc2ed8c9aaf2cc80bc0e6182ea9779.pdf).

Tinus, William C.; Och, Henry G. (1959): Systems Engineering for Usefulness and Reliability, in: IRE Transactions on Military Electronics MIL-3, 1, S. 8-12.

Tomayko, James E. (1998): Forging a discipline: An outline history of software engineering education, in: Annals of Software Engineering 6, S. 3–18.

Valdez 1988, A Gift from Pandoras Box : The Software Crisis, Phil .Diss., University of Edinburgh.

Ward, James A. (1969): Program transferability, in: AFIPS 34, SJCC, S 605-606.

Wasserman, Anthony (1981): User Software Engineering and the design of interactive systems, in : 5th ICSE '81 Proceedings, S. 387-393.

Wasserman, Anthony (1996): Toward a Discipline of Software Engineering, in: IEEE Software, 13, 6, S. 23-31.

Wasserman, Anthony und Belady, László (1978): The Oregon Report Software Engineering: The Turning Point, in: IEEE Computer, 11,9, S. 30-41.

Weaver, Warren (1948): Science and Complexity, in: American Scientist 36, S. 536–544.

Weizer, Norman (1981): A History of Operating Systems, in: Datamation 27, 1, S. 118-126.

White, John R. and Taylor L. Booth (1976): Towards an engineering approach to software design, in: Proceedings of the 2nd international conference on Software engineering, ICSE '76, S. 214-222.

White, J.; Simons, B. (2002): ACM's Position on Licensing of Software Engineers, in: in: Communications of the ACM, 45, 11, S. 91-92.

Wieser, C. Robert (1985): The Cape Cod system, in: IEEE Annals of the History of Computing, 5,4, S. 362–369.

Wirth, Niklaus (1971): Program Development by Stepwise Refinement, in: Communications of the ACM,14,4, S. 221-227.

Wirth, Niklaus (1995): A Plea for Lean Software, in: IEEE Computer, 28,2, S. 64-68.

Witt, Jan (1994): Praxis des Software-Engineering - heute und morgen, in: Informatik-Spektrum 17, 1, S. 53-56.

Zachman, John A. (1987): A Framework for Information Systems Architecture, in: IBM Systems Journal, 26, 3, S. 276-292.

Zurcher, Frank W. und Brian Randell (1969): Iterative Multi-Level Modeling. A Methodology for Computer System Design, in: Proceedings IFIP Congress 68, Amsterdam: IEEE CS Press, S. 138–142.