Andreas Brennecke, Reinhard Keil-Slawik (editors)

Position Papers for Dagstuhl Seminar 9635 on

History of Software Engineering

August 26 – 30, 1996

organized by

William Aspray, Reinhard Keil-Slawik and David L. Parnas

History and Identity

In August 1996 about a dozen historians met with about a dozen computer scientists to discuss the history of software engineering. The term software engineering has been deliberately chosen as being provocative at the 1968 NATO Conference on Software Engineering. This notion was meant to imply that software manufacture should be based on the types of theoretical foundations and practical disciplines that are established in the traditional branches of engineering. This need was motivated by the so-called software crisis. Ever since, the debate whether such a crisis exists has continued within the software engineering community. It is a crucial question, because if the answer is yes, software engineering may not be called an engineering discipline yet. If the answer were no, the question would be, what is it that constitutes this discipline.

It turned out at the seminar that there may or may not be a software crisis, but there is definitely what might be called an identity crisis. A strong indicator for this phenomenon is the fact that after more than 30 years computer scientists are investigating the history of other established branches of engineering to find out (or to define?) what should be done to turn software engineering into a sound engineering discipline. In this endeavor, historians were regarded to be some kind of universal problem solvers who were called in whenever a general answer to some fundamental question was needed.

Of course, this could not work, because history is not a methodical vehicle to clarify ones own identity or to derive normative principles and guidelines for a discipline.

Furthermore, there is only little historic knowledge in the field of software engineering as compared to the “History of Programming Languages”, for instance, or the history of electronic calculating devices. Thus, a Dagstuhl seminar on the “History of Software Engineering” can only act as a starting point, providing (a) a first overview of what has been accomplished so far and (b) identify crucial issues to be studied in the future.

With respect to my own expectations about the possible outcome, I have to admit that I was too optimistic when I took the first initiative for this seminar. I underestimated the personal and disciplinary identity problems and I was expecting more historical studies already having been carried out in the field. As a consequence, this seminar report does not provide the reader with a coherent and concise account of what constitutes the history of software engineering. It contains, however, some of the many facets, issues and approaches to the field which have made this Dagstuhl seminar a very stimulating event and which may serve as a starting point for further investigations. More than before, I am convinced that studying the history of software engineering is a necessary and rewarding activity especially also for young scholars in the field.

I want to thank the people from Schloß Dagstuhl for providing such a stimulating environment. Thanks also to Bill Aspray and David Parnas who organized the seminar together with me, and finally many thanks to Andreas Brennecke who took the burden to prepare this seminar report.

Paderborn, July 1997

Reinhard Keil-Slawik

Contents

Invitation to the Dagstuhl Seminar on: "The History of Software Engineering"

William Aspray
Reinhard Keil-Slawik
David L. Parnas

Computer science is often characterized as an engineering discipline with the systematic study and development of software as its principal subject matter. Software Engineering, however, although combining both key words, has not become a central discipline in most computer science departments. In many respects, this discipline embodies the same ideosyncracies that can be observed within computer science as a whole such as:

- Highly innovative and rapidly changing field with no broadly recognised core of material that every practitioner must know;
- Few results are supported by empirical or comparative studies;
- Work within the field older than 3–4 years is rarely acknowledged or referenced;
- Old problems are given new names and old solutions overlooked;
- Evolution of the discipline is tightly coupled to economic and societal demands;
- There is a need for interdisciplinary work comprising e.g. mathematics, psychology, business or management science, … ;
- Continuing debate about whether there should be a discipline called software engineering, and if so, whether this should be treated as another discipline among the set of traditional engineering disciplines.

In a highly dynamic scientific environment, with ongoing debates about fundamental issues a historic investigation may help to better understand the issues in question and may provide some support to assess the current state of the discipline and what has been accomplished so far. It may further help to encourage scientists and young scholars to look into the past in order to built upon experiences and insights already accomplished rather than inventing the wheel anew every once and a while. Lessons learnt from the past may provide a deeper understanding of the complexity and problems encountered in software engineering up to date.

On the other hand, historians may have the chance to study the history of technology by talking directly to those who were part of it. The interdisciplinary exchange between historians and computer scientists, resp. software engineers may also yield a more accurate picture of the past than either one of these disciplines could accomplish on their own.

Thus, the aim of the seminar is to bring together software engineers and historians to discuss the history of software engineering for two reasons:

- to gain a better understanding of what has been accomplished in the past, and how it came about, and
- to try to improve on future research and practice in software engineering based on previous experiences and insights.

The conference may also help in the establishment of a recognised set of core principles that should be known by everyone who calls him/herself a Software Engineer.

Problems and questions to be discussed at the seminar should comprise the whole realm of issues ranging from purely technical to social and societal ones.

Advantages and Problems of Bringing Together the Historical and Engineering Communities to Study Software Engineering History

William Aspray

The first section of the talk discussed what historians do, and how they do history differently from engineers. The focus was on a few themes:

Technological determinism – i.e. that technology develops in a certain, deterministic way with its own autonomous logic; or in a milder form, that technology develops along certain paths (technological trajectories) that represent natural technological choices. Historians do not accept these approaches, which diminish human agency; what are seen as necessary technological directions by engineers are seen as human choices by historians.

Technological progress – i.e. that technology necessary gets better over time and makes the world a better place. Historians ask the question "better for whom?" Progress is regarded by historians as a value-laden term. Historians have studied examples of technologies being uninvented – contrary to the notion of progress.

Social construction of technology – an approach adopted by some historians of technology in more or less radical versions, and influencing most others. The basic idea is that technology does not develop autonomously, but rather it is a human artifice. More precisely, SCOT historians see people, technology, and institutions as equal actors.

Contextualization – i.e. trying to understand the various social, economic, technical, political, etc. contexts in which technology develops. It is a more moderate and widely accepted historical notion, looking for ways to get around the one-dimensionality of any one of these approaches and seeking a way of combining them without compromising theoretical principles. Many historians specialize in one historical approach, e.g. they are economic historians, while others use many tools. None of these approaches are privileged; and using multiple approaches gives multiple perspectives.

Priority claims – i.e. who invented a technology first. This was a major interest of historians in the past and continues to interest engineers. History does not give special place to what happens first; and many precursors are unimportant in that they had little impact.

Technological failures – Many engineers restrict their historical studies to technological successes. Increasingly, historians are interested in failures, for they want to explain why something with promise did not work out. Failures can reveal structure and relationships in a way that successes do not.

The second part of the talk raised a number of historical issues and possible explanations of software engineering, mainly related to its origins.

Software crisis – Was the so-called "software crisis" really a crisis? who was it a crisis for? what were its causes? was it the origin of software engineering?

Technology imbalance theory – One claim is that, during the 1960s, memory and processor speed increased markedly but software techniques improved very little. Was this a principal cause of the software crisis?

Expectation theory – More was expected of operating systems in the 1960s, e.g. multiprogramming capabilities, than had been expected in the past. Was the software crisis simply an expression of difficulty meeting heightened expectations?

Professionalization – Can the software crisis and the origins of software engineering be seen as an effort for software writers to professionalize themselves, develop professional methods, and establish ethics and procedures for safe and reliable products?

Economic – Can the origins of software engineering be explained as an economic driving force on the part of suppliers and customers who wanted to have means to control pricing and delivery timetables?

Dramatic failure theory – Was the rhetoric of crisis simply the result of a few large-scale, dramatic failures, such as OS/360 and the Mariner I spacecraft failure, rather than a general trend?

Dissemination issue – To what degree were particular books (e.g. Fred Brooks, Mythical Man-Month) or conferences (NATO on software engineering) influential in identifying/creating a problem and building a community of interested people?

Labor perspective – Can software engineering be explained as the move away from the celebration of the craft skills of the talented individual programmer to the regularization of practice so that it could be accomplished by masses of less skilled individuals?

These and a number of other issues were raised as a way of starting a general discussion for the conference.

Comparison of electrical "engineering" of Heaviside's times and software "engineering" of our times

Robert L. Baber

In the second half of the nineteenth century the field of electrical technology, in particular telegraphy and telephony, exhibited problems analogous to those in software development in the past and present. Also in many other technical fields spectacular failures were all too common, such as bridges collapsing under the weight of locomotives and, still earlier, ships sinking on their maiden voyages because of instable hull designs.

It is the thesis of this presentation that software development today is in a pre-engineering phase analogous in many respects to the pre-engineering phases of the now traditional engineering disciplines. In this talk, selected examples of experiences from some of those disciplines, especially electrical telegraphy and telephony, are presented. From some observations regarding similarities between the experiences in those disciplines in the past and software development today, some questions regarding lessons software developers might learn from those earlier experiences of others are raised. Some answers are suggested.

"Those who do not learn their lessons from the mistakes of history are doomed to repeat them." It appears to me that we are repeating those mistakes and will, I am afraid, have to relearn those lessons the hard way. We do not seem to be even trying to learn from the history of the classical engineering fields.

In 1628 the Swedish navy's magnificent new ship Wasa sank at the beginning of her maiden voyage, only a few hundred meters from the shipyard in a light squall. The loss of life and property was considerable. Analysis performed after salvaging her in 1961 showed that her hull was dynamically instable. Today we can calculate in advance of actual construction whether a ship will be stable or not. Similarly, in the last century many bridges collapsed under the weight of locomotives, again with much loss of life and property. Today, structural engineers can and do verify by calculation, before construction, that a proposed bridge design will support its own weight and the intended load under specified environmental conditions (e.g. wind, earthquakes, etc.).

In the case of electrical telegraphy and telephony before and around the turn of the century, a number of phenomena confused designers and limited the advance of the field. Insufficient understanding of relevant technical aspects of the systems in question sometimes led to considerable financial losses, e.g. the destruction of the first successfully laid transatlantic cable in 1858. Expensive experiments were conducted, often with inconclusive results, in attempts to understand the phenomena in question (e.g. the different working speeds of a telegraphic circuit in the two directions, distortion of voice signals in transmission lines, etc.). People like Oliver Heaviside solved many of these problems by analysis, using paper and pencil only, often many years before those solutions were accepted and put into practice by the "practical" men. In the eyes of these practitioners such theories were useless, even ludicrous, as evidenced by the fact that one of them, Maxwell's theory, implied self propagating electromagnetic waves, something no one had ever observed and clearly never would observe, let alone make practical use of! A few years later Hertz, knowing from theoretical considerations what to look for and how to look for it, observed such waves. A few decades later, these waves were being used for practical, commercially viable transatlantic communication.

The essential factor which accounts for the very substantial difference between design correctness yesterday and today in these fields is, it seems clear to me, is a scientific and mathematical foundation for each of the engineering fields of today and the fact that their practitioners apply that foundation regularly and as a matter of course in their work. Their models of the artefacts being designed are not only descriptive (as in pre-engineering days also), but predictive. Failures and successes can be and are predicted – calculated – in advance. Only if success is determined in advance is the artefact even built; e.g. in civil engineering such a calculation is normally required as part of the application for a building permit. Trial and error (mostly error) is no longer an accepted design technique.

The scientific and mathematical foundations referred to above are provided by Newton's laws (for the civil and mechanical engineer) and by Maxwell's laws (for electrical engineers). The comparable foundation for software engineering in the true sense of the word engineering is provided by the work of Floyd, Hoare, Dijkstra and others, but has not yet come into widespread use in practice. We are still living and working in the pre-engineering days of software development.

We have all heard of the many consequences of our non-engineering approach to software development. The many mistakes, expensive failures, even some deaths are well known. We laugh about substantial parts of the telephone system in the U.S. being out of service for hours at a time, but behind such failures are expensive, even life endangering consequences. In another moderately publicized incident several cancer patients died due to overdoses of radiation delivered by a system controlled by faulty software. Quite recently, the Johannesburg Stock Exchange has been repeatedly plagued by interruptions lasting as long as an entire trading day which were attributed to faulty and deficient software in essential computer systems.

Instead of implicitly using the pre-engineering days of such fields as nautical, civil and electrical technology as role models, we should and could be explicitly using their current engineering practice as role models. Just one example is the world wide telephone network. I can use my South African hand held mobile cell telephone while riding in an automobile on a German autobahn to talk with someone in England or in an outlying area in Thailand. The signals are transmitted through some combination of analog and/or digital land lines, terrestrial and satellite radio links. Each of the individual subsystems in this extensive network was designed with little or no information about most of the other subsystems. In fact, the designers of many of these subsystems could not even know that some day their subsystems would be used to carry signals originating or terminating in a hand held mobile telephone operating in a radio cell network. The success of such a system derives from two main and commonly employed engineering design strategies: (1) Every subsystem is rigorously designed to a precisely and well defined interface specification dealing with both logical and physical aspects of the signals being transmitted and processed. (2) Each such subsystem is correctly designed to these interface specifications, using predictive models as outlined above and based on a scientific and mathematical foundation also outlined above.

The sooner we software developers start in earnest to follow the examples set by our engineering brethren, the better off everyone will be, especially our customers and the users of our software. Developing software in the traditional way is nothing other than high tech Russian roulette.

Bibliographie

Baber, Robert L.: *Software Reflected: The Socially Responsible Programming of Our Computers*, North-Holland, Amsterdam, 1982.

Baber, Robert L.: *Praktische Anwendbarkeit mathematisch rigoroser Methoden zum Sicherstellen der Programmkorrektheit*, Reihe Programmierung komplexer Systeme / PKS, Walter de Gruyter, Berlin, 1995.

Borgenstam, Curt; Sandström, Anders: *Why Wasa Capsized*, Wasa Studies 13, Statens sjöhistoriska museum, Stockholm, no date given (1984?).

Marciniak, John J. (ed.): *Encyclopedia of Software Engineering*, John Wiley & Sons, New York, 1994.

McDermid, John A. (ed.): *Software Engineer's Reference Book*, Butterworth-Heinemann, Oxford, 1991.

Nahin, Paul J.: *Oliver Heaviside: Sage in Solitude*, IEEE Press, New York, 1988.

Recent Research Efforts in the History of Computing:

Thomas J. (Tim) Bergin

My major focus in the early part of 1996 was the organization of the ACM's *50th Anniversary Retrospective* program. The Retrospective took place on February 14, 1996, which was the 50th anniversary of the public unveiling of the ENIAC at the Moore School, University of Pennsylvania, and opened ACM's *Computing Week* which included the Computer Science Conference. For the latter, I organized a *History Track* consisting of five panels on "ENIAC," "The Army, the National Need, and the ENIAC," "Early Computer Efforts at the National Bureau of Standards," "Hardware History (with Maurice Wilkes)," "Software History," and a lecture by Alan Kay on "The History of the Personal Computer". I also served as a consultant to a team which created a large photographic exhibit for the conference and provided a complimentary exhibit of artifacts from my collection.

I am continuing this research, and working with the US Army on documenting early Army efforts to develop and use computers. As part of this effort, I gave the opening address at a two-day meeting, in November, at the Aberdeen Proving Grounds, MD. *Fifty Years of Army Computing: From ENIAC to MSRC* (the MSRC is a new supercomputer center which was dedicated at the conclusion of this meeting). This meeting allowed pioneers from the early days of Army computing to share their experiences. Present efforts involve editing the transcripts of the meeting for publication, and doing oral histories of early Army computing pioneers. I am also planning on developing a database of original ENIAC drawings and documents to assist the Army and researchers.

The major effort at this time is the creation of a Museum on The History of Computing, on the American University campus. This museum will consist of two rooms (160 square feet) and will contain a copy of the photographic exhibit created for the ACM 50 Anniversary Ret-

rospective (above) and two display cases of artifacts relating to the general history of computing. In addition to a display of mechanical calculating devices, the museum will also house examples of early microcomputers, a teletype, and a working IBM 029 keypunch. There will also be changing exhibits, which will focus on more specific topics such as the history of programming languages or the history of artificial intelligence. These exhibits will be developed with colleagues who are specialists in these areas.

Finally, I will be working with William Aspray, of the Computing Research Association, on a three year research project to examine the "History of Academic Computer Science." This effort will involve case studies of computer science, information systems, computer/electrical engineering and information science programs. as well as an examination of selected industries and professional organizations.

Report on Dagstuhl Conference: Final Session: Historians Report

Thomas J. (Tim) Bergin

The Historians met on Thursday night and formulated a summary statement to be delivered at the concluding session on Friday morning. We believe that a history of software engineering could be written, providing that boundaries, topics, themes, approaches, evidence, etc. are identified. Although historiography is usually an individual or small team effort, larger and more diverse teams could attempt to document the history of a movement such as software engineering. The team necessary to complete such a monumental task would consist of historians, software engineers, and other individuals involved in program/application development in the 1950s and 1960s (and perhaps up to the present time). In order to have a chance at succeeding, someone (presumably a historian) would have to have control of the project, so that the effort proceeded as if from a "single mind."

During the historians work session, last evening, the following points were made:

1. it is important to present crisp interpretations with appropriate contrasts;
2. we need to examine the discipline prior to the NATO conferences, i.e., what exactly lead up to the problems and the Conferences?
3. we need to examine multiple threads throughout the discipline (software development methodologies, systems analysis tools; project management techniques, etc.)
4. we need to examine the "software crisis" and document multiple perspectives (from the universities, from industry, from government, from non-American sources)
5. we see the need for relevant, in-depth, case studies:
   - **a.** by *community*: ACM, IEEE, SHARE, GUIDE, DPMA, etc.
   - **b.** by *software category*: operating systems, compilers/languages, interfaces, etc.)
   - **c.** by *life cycle stages*: systems analysis (needs assessments, requirements documentation, and methodologies/tools/techniques), systems design (architectures, data storage strategies and hardware interfaces/dependencies),

development (languages, compilers, software engineering techniques and tools), and management (project management, resource management, and software quality metrics).

6 we believe that we need to develop a distinctive view of software engineering, i.e., one that is more than a definition based on a few disparate academic papers or practitioners views.

7 we need to examine the interactions, over time, between the defense establishment, software development consulting and service organizations, and the multiple, relevant academic disciplines (computer science, information systems, computer engineering, and software engineering (where so identified)).

8 in general, therefore, we need to determine the beginning, the middle, and the end (?) of software engineering, as we would define it (and its extensions and progeny).

In conclusion, we believe that this week was a good start down a long and windy road. The history of large technical movements, if we may categorize software engineering in that way, has not ever been documented a large group, especially one dominated by practitioners. If a members of this Dagstuhl seminar decided to pursue a history of software engineering, we would need to recognize the enormity of the project, the broad range and depth of resources needed for completion, and the necessity of adopting the basic precepts of software engineering (however defined)s to provide some control over the process and the participants. The software engineering model would allow the group to:

(1)identify goals and requirements,
(2)design executable independent modules/research-projects by which to divide the effort,
(3)assign projects to cooperating historians and software engineers, and,
(4)apply project management techniques to control progress and quality.

Such a large effort by a diverse group of people would in all likelihood suffer from the same problems which the software engineering discipline attempted to answer for the applications development process. Thus, we close with these thoughts:

(1)the notions of a beginning, a middle, and an end of a living process come from us, the historians, they do not exist in the real world;
(2)history is not an agreement; we can't vote on it; it is an elucidation of facts with significant interpretation;
(3)the idea of developing a history of software engineering is of interest to the historians, but we are mindful of the problems of the software development process: (a) someone would have to exercise intellectual control of the project, and (b) that a project as complex as this would need to apply the lessons learned from the history of software engineering;

In the interest of encouraging further research we would like to point out that there are some *models* for such an approach to history. There have been two conferences devoted to the history of programming languages (HOPL). In 1978, the ACM Special Interest Group on Programming Languages (SIGPLAN) sponsored a Conference on the History of Programming Languages (HOPL) in Los Angeles, CA (USA). The Program Committee prepared material to guide prospective contributors, including a list of questions designed to elicit "good history." Also of

interest to the Dagstuhl participants, is a paper by Henry Tropp entitled "On Doing Contemporary History," and "General Questions Asked of All Authors." The results of this effort are documented in *History of Programming Languages*, edited by Richard L. Wexelblat, editor, [New York: Academic Press, 1981].

A second HOPL Conference was held in Cambridge, MA in 1993. The Program Committee followed the strategy of the earlier conference, and prepared a set of questions that was sent as guidance to all prospective contributors. These questions are documented in J.A.N. Lee, "Guidelines for the Documentation of Segments of the History of Computing," *Annals of the History of Computing*, Volume 13, Number 1 (1991). In addition, the Committee asked Michael S. Mahoney to serve as Conference Historian. Mahoney's lecture on "Making History" as well as his paper, "What Makes History?" are both of interest to the Dagstuhl participants. This second HOPL Conference is documented in Thomas J. Bergin and Richard J. Gibson, editors, *History of Programming Languages*, [New York: ACM Press/Addison-Wesley, 1996].

Another excellent reference to guide such research is R. W. Hamming's "We Would Know What They Thought When They Did It," which can be found in N. Metropolis, et al (eds), *A History of Computing in the Twentieth Century,* [Orlando: Academic Press, 1980]. In addition, there have been some fine papers on aspects of software engineering history, such as Stuart Shapiro's "Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering," *IEEE Annals of the History of Computing*, Volume 19, Number.1 (January-March, 1997), pp. 20-54, and Mary Shaw's "Prospects for an Engineering Discipline of Software," *IEEE Software*, Vol.7, No. 6 (November 1990), pp. 15-24.

In conclusion, the historians suggest that a history of software engineering would be an important contribution to our knowledge of computer science and its allied disciplines, and that a return to Schloß Dagstuhl to work on such a project would be most welcome.

SOFTWARE PROCESS MANAGEMENT: LESSONS LEARNED FROM HISTORY

Barry W. Boehm

Regarding history, George Santayana once said, "Those who cannot remember the past are condemned to repeat it."

I have always been dissatisfied with that statement. It is too negative. History has positive experiences too. They are the ones we would like both to remember and to repeat.

The three papers in this session are strong examples of positive early experiences in large-scale software engineering. The papers are:

- H.D. Benington, "Production of Large Computer Programs," *Proceedings, ONR Symposium*, June 1956.
- W.A. Hosier, "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," *IRE Transactions on Engineering Management*, June, 1961.
- W.W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings, WESCO*, August 1970.

Given the short lifespan of the software field, they can certainly be called "historic." Indeed, since many people date the software engineering field from the NATO Garmisch conference in 1968, two of them can even be called "prehistoric." They are certainly sufficiently old that most people in the software engineering field have not been aware of them. The intent of this session is to remedy this situation by reprinting them in the Conference Proceedings, and by having the authors (or, in one case, Hosier's colleague J.P. Haverty) discuss both the lessons from their papers which are still equally valid today, and the new insights and developments which have happened in the interim.

Rationale: 1961 Lessons vs. 1979 Practice

About the best rationale I can provide for the value of these papers is an exercise I performed in 1979 to compare the lessons learned in Hosier's 1961 article with a sample of 1979 software engineering practice gathered from a set of 50 term papers from a course I was giving at the time. Overall, I would say that the 1961 Hosier lessons were being applied successfully about 50% of the time. Here are some sample comparisons between the 1961 lessons and 1979 practice:[1]

- *Testable Requirements*
  Hosier: As soon as specifications for a system program are definitive, contractors should begin to consider how they will verify the program's meeting of the specifications. In fact, they should have had this in mind during the writing of the specifications, for it is easy to write specifications in such terms that conformance is impossible to demonstrate. For example: 'The program shall satisfactorily process all input traffic presented to it.'"

  1979 Experience: "A requirements spec was generated. It has a number of untestable requirements, with phrases like 'appropriate response' all too common. The design review took weeks, yet still retained the untestable requirements."

  "The only major piece of documentation written for the project was a so-called specification. Actually, the specification was written after the program was completed and looked more like a user's manual."

- *Precise Interface Specifications*
  Hosier: "The exact interpretation of digital formats, the rise an fall times of waveforms, special restrictions as to when each type of data may or may not be sent these and sundry other details must be agreed on by all parties concerned and clearly written down. Accomplishing this is apt to be a monumental and tedious chore, but every sheet of accurate interface definition is, quite literally, worth its weight in gold."

  1979 Experience: "No one had kept proper control over interfaces, and the requirements specs were still inexact."

  "The interface schematics were changed over the years and not updated, so when we interfaced with the lines, fuses were burned, lights went out...".

1.Boehm, B.W., "Software Engineering: As It Is," *Proceedings, 4th ICSE*, September 1979.

"The interface between the two programs was still not exact. When interfacing the two programs we ran into run time errors. Debugging was difficult because of the lack of documentation. We also began to forget exactly what our code did in certain situations and wished we had done more documentation."

- *Lean Staffing in Early Phases*
  Hosier: "The designer should not be saddled with the distracting burden of keeping subordinates profitably occupied…. Quantity is no substitute for quality; it will only make matters worse."

  1979 Experience: "At an early stage in the design, I was made the project manager and given three trainees to help out on the project. My biggest mistake was to burn up half of my time and the other senior designer's time trying to keep the trainees busy. As a result, we left big holes in the design which killed us in the end."

I think things have gotten somewhat better since 1979, but I daresay there are still a good many current projects which could have benefitted from the lessons in these three papers.

Software Process Insight

Given that the theme of this conference centers on the software process, it is worth examining how these early papers address the software process.

Royce's 1970 paper is generally considered to be the paper which defined the stage-wise "waterfall" model of the software process. But it is surprising to see both that the earlier Benington and Hosier papers had good approximations to the waterfall model, and that Royce's paper already incorporates prototyping as an essential step compatible with the waterfall model.

The stages in Benington's Figure 4 are very similar to those in Royce's waterfall, and actually contain more detail in the area of software and system integration and test. Hosier's Figure 2 adds even more detail on the interactions between the various stages and activities in the process. It is particularly good in identifying the steps required to prepare all the support capabilities necessary to software production: coding rules or programming standards, input-generating programs, test control and recording capabilities, data reduction programs, etc. The primary additional contribution of Royce's Figure 3 is in the explicit treatment of iteration and-feedback, and the focus on confining iterations as much as possible to neighboring phases, in order to minimize the much more expensive long-range feedback loops.

One frequent objection to the waterfall model is that it forbids prototyping. People interpret it to say, "Thou shalt not write one line of code until every detailed design specification is complete." Royce's Figure 7 shows that this was not the intent: the "Do it twice" approach emphasizes at least one round of prototyping (called "simulation" in Royce's paper) to ensure that high-risk issues are well understood. Benington's original paper does not mention prototyping, but his later Foreword indicates that SAGE did an extensive early prototype before entering full-scale software development. Hosier does not say much about prototyping, except for such occasional statements as, "Possibly, parts of the program may have been created for simulation in previous studies," indicating that prototyping was used in his context as well. Probably the major difference today is the existence of powerful tools for rapid prototyping, making prototypes much more costeffective (or "schedule-effective") for use today.

Summary

Although these three papers may be of considerable interest for historic perspective on our understanding of the software process, I do not think that is their primary value. Their main value is their continuing relevance today. Most of the specific guidance they provide on requirements analysis, prototyping, early planning, precise interface specifications, lean staffing in early phases, core and time budgeting, objective progress monitoring, integration planning and budgeting, support software preparation, documentation, test planning and control, and involving the customers and users, can be used as well today as at the time they were written. They stand today as the record of thoughtful people summarizing the lessons they had learned, in the hopes that those of us who came along later would be able to repeat the positive software engineering experiences from history rather than the negative ones. I hope you will be able to benefit from them.

Reprint of: Barry W. Boehm: SOFTWARE PROCESS MANAGEMENT: LESSONS LEARNED FROM HISTORY. *Proceedings, ICSE 9*, March-April 1987

Remarks on the history and some perspectives of Abstract State Machines in software engineering

Egon Boerger

In Gurevich88 a notion of Abstract State Machine, previously called evolving algebras, has been proposed in an attempt to sharpen Turing's thesis by considerations from complexity theory where the notion has led to important new developments (see BlaGur94). Immediately after Gurevich had defined his notion of ASM, it has been used to develop a powerful method for design and analysis of software and hardware systems (see Boerger94 and Boerger95 for a detailed historical and methodological account).

In this talk we first survey some of the major real-life case studies from software engineering and hardware engineering through which that method has been developed or to which it has been applied as a test for its practicality. We then explain that Gurevich's notion of ASM (see Gurevich95 for a detailed definition) provides a convergence of Dijkstra's notion of abstract machine (see Dijkstra68), of Wirth's notion of stepwise refinement and of the data abstraction idea which has been developed in the theory of abstract data types. In this way ASMs can be viewed as representing an epistemologically complete realization and generalization of the long standing structured programming program (see DaDiHo72). We conclude the talk by pointing to recent research showing that the ASM based design and verification method, when applied in software engineering, provides a possibility to solve the Ground Model Problem and to develop in a systematic and practical manner well documented and formally inspectable code from rigorous ASM requirement specifications.

1. A detailed survey of the ASM literature from 1988–1995 can be found in the annotated bibliography in Boerger95a; the more recent work can be traced through the two ASM WWW sites http://www.uni-paderborn.de/cs/asm.html and http://www.eecs.umich.edu/gasm/ in Paderborn and Ann Arbor. In the following we mention therefore only some outstanding examples which illustrate salient features of the ASM based specification method.

The first practical use of ASMs has been made for providing rigorous but simple definitions of the semantics of different kinds of real-life programming languages. Examples are the logic programming language PROLOG, the imperative parallel language OCCAM, the 1993 standard of the IEEE hardware design language VHDL, the object-oriented language C++, OBERON, etc. (see Boerger90, Boerger90a, Boerger92, BoDuRo94, BoeDur96, BoGlMu94, BoGlMu95, Wallace95, KutPie97). These definitions show that ASMs allow one to give succinct formal models of complete programming languages which support the way programmers think and work ("operational view") and nevertheless are simple and have a rigorous mathematical basis. As a matter of fact, the ASM definition of the logic programming language PROLOG has provided the basis for the formulation of the ISO 1995 PROLOG standard (see BoeDae90, BoeRos95). For PROLOG and OCCAM it has been shown that these ASM definitions can be refined in a natural way by a sequence of intermediate ASM models leading to a correct implementation by WAM and Transputer code respectively (see BoeRos94, BoeDur96). The correctness of the underlying compilation schemes has been proved by mathematical argument, coming in the form of lemmas accompanying the various refinement steps. In the PROLOG-to-WAM case this proof has been mechanically verified (see SchAhr97, Pusch96) and could be reused for extensions of the ASM specifications to Prolog with polymorphic types, to the constraint logic programming language CLP(R), to a sublanguage of Lambda-Prolog where implications are allowed in the goals, to a parallel distributed version of Prolog and to other Prolog extensions (see BeiBoe97, BeiBoe97a, BoeSal95, Kwon97, Araujo97).

After these experiments with ASM definitions and implementations of full-fledged programming languages, subsequent work has established that ASM specifications offer in full generality the possibility of turning intuitive design ideas into rigorous specifications and proofs for (high-level as well as low-level) system properties. This has been tried out successfully for many different computing systems, namely for protocols, pipelined RISC architectures, control software, etc. (see for example Huggins95, BoGuRo95, BoeDel96, BoeMaz97, BBDGR96, BoeMea97). Such transparent and easy to understand ASM modellings of complex real-life computing systems, relating high-level and low-level views of these systems in a natural and provably correct way, confirm that the ASM method is capable of supporting Abrial's program of accomplishing "the task of programming (in the small as well as in the large) by returning to mathematics" (see Abrial96). The ASM approach provides a still greater freedom than Abrial's B-method because it avoids any a priori prescribed combined formal language and proof system and invites the system engineer to use whatever form of mathematical language, programming notation and rigorous reasoning may be useful for defining the desired system and for justifying his construction.

2. The naturalness of the ASM models for the great variety of computing systems mentioned above could be obtained only because of the most general abstraction principle which is built into the notion of ASM. Gurevich's notion of ASM combines two fundamental ingredients which for decades have lived in separate worlds: Dijkstra's notion of abstract machines on the one side and on the other side the idea of having states as structures (in the sense of first-order logic, i.e. collections of domains with functions and predicates defined on them). Dijkstra pro-

posed the notion of abstract machine in the context of the definition of his operating system T.H.E. (see Dijkstra68, preceded by the class concept of Simula67 which has been interpreted as a form of abstract machine in Hoare76), but despite numerous variations of the concept which have been introduced later by many researchers (virtual machines, layered machines, data base machines, etc., to mention only some), nobody has given an epistemologically complete definition of the underlying notion of "abstraction" or even tried to justify whether the proposed notion is the most general one possible. On the other side, in the theoretically oriented community of abstract data types, it seems to have been recognized since the late 60-ies that the appropriate most general notion of "abstract state" is that of structure or algebra, as explained above. But unfortunately this understanding has been used only for static system descriptions (technically speaking by describing state components by syntactic terms) and has never been related to the dynamics of machines. This may explain also why the structuring (composition and refinement) principles which have been investigated were mainly of syntactical nature, guided by the syntax of the underlying formal (programming) language.

Gurevich's notion of ASM is that of a machine transforming structures. Arbitrary structures are allowed, of whatever degree of "abstraction", i.e. with any domains of objects and any operations and predicates defined on them. Gurevich argues convincingly that this concept allows us to justify a generalized Turing's thesis (see Gurevich88a) and thus guarantees that the concept of "abstraction" underlying the notion of Abstract State Machine is the most comprehensive one we can think of in terms of our current mathematical knowledge. The pragmatic consequence is that the common practice of system engineering can be accommodated where a system is defined in terms of its basic objects and of its basic operations which the system uses for its actions; such definitions can be formalized as ASM directly, without any need to introduce extraneous encodings. The dynamic machine behavior is expressed by instructions which – given certain conditions – assign new values to functions ("guarded function updates"). The freedom to choose the objects and fundamental operations guarantees that also most general forms of refinement can be realized within the ASM framework. Both data and actions can be refined separately or simultaneously, providing composition and modularization principles which go well beyond any syntactical framework and reflect semantical structurings of the system under investigation. In this way the ASM method generalizes the longstanding structural programming endeavour.

3. In relation to software engineering the abstraction and refinement principle which are built into the notion of ASM incorporate a strongest form of Parnas' information hiding principle and of modularization concepts (see Parnas72). They allow us to define abstract systems with precise interfaces, making use of the concepts of oracle functions and of externally alterable functions (see Boe95).

The refinement method used for building software in steps avoids overspecification and helps to postpone premature design decisions; sequences of stepwise refined ASMs can be used conveniently to realize modular architectural design. Industrial software can be developed using ASMs systematically, from the requirement specification to code, in such a way that the abstract ASM models reflect the code structure and provide a satisfactory documentation of the whole system at different levels of abstraction (see for example BBDGR96, BoeMea97). It is interesting to note that this method can be applied successfully also for reengineering of complex hardware systems (see the example of the APE100 parallel architecture in BoeDel95).

Pragmatically more important seems to me the fact that the notion of ASM solves the ground model problem which is fundamental for software engineering. A ground model (see

Boerger94 where I used the term primary model) of a system S – which itself is very often not a mathematically well defined system – is a mathematical model which formalizes the basic intuitions, concepts and operations of S, without encoding, in such a way that the model can be recognized by the system expert "by inspection" and thus justified as a faithful adequate precise representative of S. Good ground models play a crucial role for provably correct specifications of complex systems in general. A provably correct specification transforms a given model A into another model B–an "implementation" of A– and proves that this implementation is correct; ultimately, in a chain of such provably correct specifications, the first model must be ground in the above sense in order to provide, from the pragmatic point of view, a safe foundation for the whole specification hierarchy. This is true in particular for software systems where the ground model must be understandable to the application domain expert (the customer). How can one establish the correctness of such a first model S0 in a specification chain? The question is how we can relate the non–formal system S to the formal model S0. By definition there is no provable relation between the mathematical object S0 and the loosely given informal system S. Therefore the only thing we can hope for is a pragmatic foundation: we have to grasp the correctness of S0 with respect to S by understanding ("by inspection"). In order to make such a pragmatic foundation safe, the ground model S0 has to be flexible, simple, concise, abstract and rigorous (see the discussion in Boerger95). The flexibility provbided by the freedom of abstraction with ASMs makes it possible for the practitioner to construct satisfactory ASM ground models, thus coming up with reliable requirement specifications.

References

(Abrial96) J.R. Abrial, The B-Book. Cambridge University Press, 1996, pp. XXXIV+779.

(Araujo97) L. Araujo, Correctness Proof of a Distributed Implementation of Prolog by Means of Gurevich Machines. (manuscript)

(BeiBoe97) C. Beierle and E. Boerger, Specification and correctness proof of a WAM extension with abstract type constraints. In: Formal Aspects of Computing, Vol. 8(4), 1996, 428–462.

(BeiBoe97a) C. Beierle and E. Boerger, Refinement of a typed WAM extension by polymorphic order-sorted types. In: Formal Aspects of Computing, Vol. 8(5), 1996, 539–564.

(BBDGR96) C. Beierle, E.Boerger, I. Durdanovic, U. Glaesser and E. Riccobene, Refining abstract machine specifications of the steam boiler control to well documented executable code. In: J.-R. Abrial, E.Boerger,H. Langmaack (Eds.): Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control. Springer LNCS State–of–the–Art Survey, vol. 1165, 1996, 52-78.

(BlaGur94) A. Blass and Y. Gurevich, Evolving Algebras and Linear Time Hierarchy. In B. Pehrson and I. Simon, editors, Proc. of the IFIP 13th World Computer Congress 1994, Vol. I, pp. 383–390. Elsevier, 1994.

(Boerger90) E. Boerger, A logical operational semantics for full Prolog. Part I: Selection core and control. In: Springer LNCS, vol. 440, 1990, pp. 36-64.

(Boerger90a) E. Boerger, A logical operational semantics for full Prolog. Part II: Built-in predicates for database manipulations. In: B.Rovan (Ed.): MFCS'90. Mathematical Foundations of Computer Science. Springer LNCS, vol. 452, 1990, pp 1-14.

(Boerger92) E. Boerger, A logical operational semantics for full Prolog. Part III: Built-in predicates for files, terms, arithmetic and input-output. In: Y.Moschovakis (Ed.) Logic from Computer Science. Berkeley Mathematical Sciences Research Institute Publications, vol.21, Springer 1992, pp. 17-50.

(Boerger94) E. Boerger, Logic Programming: The Evolving Algebra Approach. In: B. Pehrson and I. Simon (Eds.), IFIP 13th World Computer Congress 1994, Volume I: Technology/ Foundations, pp.391-395, 1994, Elsevier, Amsterdam.

(Boerger95) E. Boerger, Why use of evolving algebras for hardware and software engineering. In: M.Bartosek, J.Staudek, J.Wiedermann (Eds), SOFSEM'95 22nd Seminar on Current Trends in Theory and Practice of Informatics. Springer Lecture Notes In Computer Science, vol. 1012, 1995, pp.236–271.

(Boerger95a) E. Boerger, Annotated Bibliography on Evolving Algebras. In: E. Boerger (Ed.), Specification and Validation Methods, Oxford University Press, 1995, pp.37–52

(BoeDel96) E. Boerger and G. Del Castillo, A formal method for provably correct composition of a real–life processor out of basic components (The APE100 reverse engineering project). In: Proc. First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'95). IEEE Computer Society Press, Los Alamitos, California, 1995, pp.145-148.

(BoeDae90) E. Boerger and K. Daessler, PROLOG. DIN papers for discussion. ISO/IEC JTC1 SC22 WG17 report no.58, National Physical Laboratory, Middlesex, April 1990, pp.92-114.

(BoeDur96) E. Boerger and I. Durdanovic, Correctness of Compiling Occam to Transputer Code. In: The Computer Journal, Vol. 39, No.1, pp.52-92, 1996.

(BoDuRo94) E. Boerger, I. Durdanovic and D. Rosenzweig, Occam: Specification and Compiler Correctness. Part I: Simple Mathematical Interpreters. In: E.-R. Olderog (Ed.), Proc. PROCOMET'94 (IFIP Working Conference on Programming Concepts, Methods and Calculi), pages 489-508, North-Holland, 1994

(BoGlMu94) E. Boerger, U. Glaesser, W. Mueller, The Semantics of Behavioral VHDL'93 Descriptions. In: EURO-DAC'94 European Design Automation Conference with EURO-VHDL'94. Proceedings IEEE CS Press, Los Alamitos, CA, 1994, pp.500-505.

(BoGlMu95) E. Boerger, U. Glaesser, W. Mueller, Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines. In: Carlos Delgado Kloos and Peter T. Breuer (Eds.), Formal Semantics for VHDL, pp.107–139, Kluwer Academic Publishers, 1995

(BoGuRo95) E. Boerger, Y. Gurevich and D. Rosenzweig, The Bakery Algorithm: Yet Another Specification and Verification. In: E.Boerger (Ed.), Specification and Validation Methods, Oxford University Press, pages 231–243, 1995

(BoeMaz97) E. Boerger and S. Mazzanti, A Practical Method for Rigorously Controllable Hardware Design. in: Proc. ZUM'97, Springer LNCS (in print)

(BoeRos94) E. Boerger and D.Rosenzweig, The WAM – Definition and Compiler Correctness. In: Logic Programming: Formal Methods and Practical Applications (C.Beierle,

L.Pluemer, Eds.), Elsevier Science B.V. North-Holland, Series in Computer Science and Artificial Intelligence, 1995, pp. 20–90 (chapter 2).

(BoeRos95) E. Boerger and D.Rosenzweig, A Mathematical Definition of Full Prolog. In: Science of Computer Programming 24 (1995) 249–286.

(BoeSal95) E. Boerger and R. Salamone, CLAM Specification for Provably Correct Compilation of CLP(R) Programs. In: Specification and Validation Methods (E.Boerger, Ed.), Oxford University Press, pages 97–130, 1995

(DaDiHo72) O. Dahl, E. Dijkstra, C. Hoare, Structured Programming. Academic Press, 1972.

(Dijkstra68) E. W. Dijkstra, Structures of the T.H.E. Multiprogramming System. In: Comm. ACM 11 (1968), 341–346.

(Gurevich88) Y. Gurevich, Logic and the challenge of computer science. In E. Boerger (Ed.), Current Trends in Theoretical Computer Science, pp. 1–57. CS Press, 1988.

(Gurevich88a) Y. Gurevich, Algorithms in the World of Bounded Resources. In: R. Herken (Ed.), The Universal Turing Machine. A Half-Century Survey. Kammerer and Unverzagt, Hamburg-Berlin 1988, pp. 407–416.

(Hoare76) C. Hoare, The structure of an operating system. In: Springer LNCS 46 (1976), pp.242–265.

(Huggins95) J. Huggins, Kermit: Specification and verification. In E. Boerger (Ed.), Specification and Validation Methods. Oxford University Press, 1995, pp. 247–293.

(KutPie97) P.W. Kutter and A. Pierantonio, Montages: Toward the Semantics of Realistic Programming Languages. (manuscript)

(Kwon97) K.Kwon, A structured presentation of a closure-based compilation method for a scoping notion in logic. (manuscript)

(Pusch96) C. Pusch, Verification of Compiler Correctness for the WAM. In: J. von Wright, J. Grundy, J. Harrison (eds.), Theorem Proving in Higher Order Logics (TPHOLs'96, Turku), Springer LNCS 1125, pp. 347-362, 1996.

(SchAhr97) G. Schellhorn and W. Ahrednt, Reasoning about Abstract State Machines: The WAM CaseStudy. (manuscript)

(Wallace95) C. Wallace, The semantics of the C++ programming language. In E. Boerger (Ed.), Specification and Validation Methods. Oxford University Press, 1995, pp. 131–164.

DEVELOPMENT AND STRUCTURE OF THE INTERNATIONAL SOFTWARE INDUSTRY, 1950-1990

Martin Campbell-Kelly

Abstract

The software industry has existed since the mid-1950s, but until about 1970 very little attention was paid to it, largely because the industry was too small to merit detailed analysis other than as an unquantified sector of the overall computer business. As late as 1970, the annual turnover of all U.S. software firms was less than $1/2 billion – about 3.7 percent per cent of the total computer business. The software industry began to grow significantly in the 1970s, first following IBM's unbundling decision of 1969, and towards the end of the decade from the rise of the personal computer. By 1979 annual sales of U.S. software firms were about $2 billion. The 1980s saw dramatic growth rates in the software industry of 20 per cent a year or more, so that the annual revenues of U.S. firms had grown to $10 billion by 1982, and $25 billion by 1985 – over ten times the 1979 figure. Today the global sales of software exceed $100 billion.

This paper attempts to develop a model of the software industry based on historical principles. Software firms are classified into three sectors, based on their historical evolution:

1 Software contractors
2 The packaged-software industry
3 The personal-computer software industry

The software contractors were the first programming firms, the earliest dating from the mid-1950s. The role of the software contractor was to develop one-of-a-kind programs for computer users and manufacturers. The ethos of the software contractor was analogous to that of a civil engineering contractor in terms of its corporate culture, organizational capabilities, and relationships with customers.

The second group of firms, the suppliers of packaged software, evolved in the 1960s to develop program products for computer users in public and private sector organizations. The packaged software suppliers operated in direct competition with computer manufacturers; and like them they have evolved the characteristics of firms in the capital goods sector.

The third group, the personal-computer software suppliers, became a significant sector in the late 1970s. Virtually all of the personal-computer software firms developed outside the established software industry; in some cases there was a background in developing hobby or games software, and a "techie" computer culture. This background has profoundly affected the shape of the personal-computer software industry and its products. Even though most personal-computer software is now sold to corporate users, the ethos of the industry is more akin to publishing or consumer products than to capital goods. For example, the search for a "hit" product is paramount, and this has led to analogies being drawn with the pop music business, or the Hollywood movie industry.

The paper used this historical model to address three economic and policy issues in the software industry:

1. Why does the United States dominate the software industry? And the related issues: What are the historical reasons for the weak positions of the European and Japanese software industries? Does the developing nations' software industry represent a threat?
2. What role has R&D played in successful software firms?
3. Why did IBM and the existing software firms fail to penetrate the market for personal computer software in the 1980s?

The paper concluded by discussing the place of recreational software in the software industry.

From Scientific Instrument to Everyday Appliance: The Emergence of Personal Computers, 1970-77

Paul Ceruzzi

Abstract

Among the many changes in computing since the Software Engineering Conference of 1968, the greatest has been the emergence of the Personal Computer and with it the giant software companies, like Microsoft, who supply operating systems, applications, and networking software to desktop machines. Serious studies of the way software is developed at these companies have not been done, but anecdotal evidence and a few "insider" accounts indicate that current practice at these firms has little if anything to do with the spirit of the 1968 NATO Conference.

This paper looks at the roots of Personal Computer Software and attempts to find connections between it and software development that occurred on mainframes and minicomputers that came before. It finds that there are indeed strong connections, many of which go back to the

Digital Equipment Corporation, founded in 1957 and housed in an old woolen mill on the banks of the Assabet River about 30 miles west of Cambridge, Mass. In particular, DEC's PDP-6, a large system (not a minicomputer) introduced in 1964 and intended for time-sharing, was the focus of efforts to create software that was interactive, conversation, that took up little memory space, and powerful. These efforts we recentered at MIT and at DEC, and led to systems like the "ITS," "TOPS-10," "Tenex" Operating Systems, all for a time shared large system.

The file-handling concepts developed there were later transferred to stand-alone minicomputers like the PDP-8 and PDP-11, and from there to the first Personal Computers. Evidence for this line of influence comes from interviews with some PC system programmers, but it is also found in the very commands themselves that migrated from one to the other. Thus, "PIP," "DDT," and "TECO" –system programs found on many DEC machines– were also found among the commands of CP/M, one of the most influential of PC operating systems. CP/M is widely acknowledged as a major influence on MS-DOS, perhaps the most successful commercial software ever written. But a closer analysis of MS-DOS reveals that it was a significant

extension and modification of CP/M. Still, in tracing this thread, I found the principal emphasis among developers the qualities of ease of use, power, and above all economy of memory requirements; I found no evidence of a desire to follow the tenets of good Software Engineering practice.

That conclusion is further reinforced by a parallel examination of the other piece of software that formed the basis for Microsoft's fortunes: Microsoft's version of BASIC. Here, too, I found a strong connection to DEC, especially in borrowing the commands "USR," "PEEK," and "POKE," which gave Microsoft's BASIC a clear advantage over the BASICs of its competitors. But here again: the use of these machine-language hooks, as well as the choice of the BASIC language itself, show how little these developments owed to Software Engineering.

I conclude by speculating that this may in part explain some of the problems with the current state of PC software. At the same time I also conclude that those who would promote the tenets of Software Engineering must share some of the blame for not taking a more active role in PC software development, a role that might have prevented this state of affairs from emerging.

A Synopsis of Software Engineering History: The Industrial Perspective

Albert Endres

Introduction

In this paper I will give a condensed survey of the software engineering history between 1956 and now. I have structured the 40 years into three eras and each era in two time periods. Each period is characterized by a different set of goals pursued, methods and tools unsed, lessons learned and problems identified. The periods may overlap and are less distinct from each other than the three eras. Of course, there are significant events prior to 1956. I have chosen to consider them as pre-history and neglected them.

Any attempt to describe a 40-year period on a few pages means selecting highlights and suppressing details. It is like drawing a small scale map of a country. If you are a motorist you look for roads, other features hardly count. This paper, as the title says, emphasizes the view from industry. This means, that ideas that did not become relevant in industry are ignored.

Mastering the Machine (Era I)

This era comprises the 12 years from 1956 to 1967. It is the era where the term "software engineering" had not been coined yet. The two periods considered are both strongly influenced by external forces. It is also the forming period of the computer industry.

The Batch Period

The first computers that achieved significant use in industry, be it for commercial or scientific applications were batch-oriented systems. This mode of operation resulted from the two typical I/O and storage media, namely punched cards and magnetic tape. The main goal of software development was to exploit the limited hardware resources (storage and processing power) in an optimal way. Any less than optimal use could double or triple the processing times, measured in hours, or make the total job infeasible. Therefore lots of effort was spend on manually tuning code, mainly written in Assembler. The first compilers like the 7090 FORTRAN compiler (developed by Backus and his team) proved that this could be done as well by machine. Since compile times for reasonably sized applications could also last hours, most changes were applied to the object code first. Recompilations only occurred in certain time intervals, usually at night.

At least one key lesson was learned during this period. For most applications high level languages of the FORTRAN or COBOL type could produce adequately performing code. The problems identified were the extensive compile times and the awful clerical task to keep source and object code in synchronization.

The Interactive Period

During the second half of the sixties, interest shifted away from language issues to the more general question of development tools. The main goal was to reduce the clerical aspects of coding and to eliminate the need for modifying object code.

The hardware environment that supported this goal was the availability of non-volatile random-access storage in the form of DASD or disk and the mode of computer usage referred to as timesharing (probably invented at MIT by Corbato et al.).

The software technology developed during this period was that of incremental compilation, source level editing and debugging, and automatic test data generation. These technologies were supported by the concept of on-line source code administration and version control. Although the main I/O device used for on-line development was a simple typewriter (or teletype device) the mode of operation can be referred to as interactive. From a later period's view, the tools developed during this period can be called lower CASE tools.

The achievement of this period and the lesson learned was, that it is advisable to maintain programs at the source code, rather than at the object code level. The problems identified during this period can be described by such terms as "code and fix" approach and "spaghetti code". Moreover, people started to recognize that there is more to software development than just coding. In fact, people started to realize that programming has much in common with other development cycles and that the highest risks may lie in the *pre-testing* and even in the *pre-coding* phases. This motivated a new approach to the field and started up a new era.

Mastering the Process (Era II)

It was during this era that software engineering was established as a field of study. The era roughly lasts from 1968 to 1982. Many people may argue that the ideas brought about during this era are still relevant today. This still allows us to call them historic ideas because they were most dominant at that time. During this era software pricing was introduced by hardware vendors and an independent software industry arose.

The Process Period

It is common to associate this period with the first recognition of the software crisis. Some people insist that the original crisis still exists today, others have identified a series of crisises, one following the other. The initial software crisis which turned away from coding tools to the study of the development process, was recognized first in industry. Later on, the subject was also discussed heavily in academic circles and appropriate curricula, conferences and journals were established.

The main goal pursued during this period was to reduce development risks and to improve quality and productivity. This lead to studies, identifying the causes of failing software projects, collecting data on cost spent per activity and to the analysis of software error data. Each organisation reacted with a set of development guidelines, typically expressing some phased approach to development. The waterfall model and many improved versions of it were introduced. The principle of check and balance was applied wherever feasible. This usually resulted in the creation of an independent test organisation which later may have grown into a quality assurance function. A milestone event of this period was the recognition that code inspections, if done wisely, could be more effective than testing. Code inspections (as documented by Fagan) were later extended to test case and to design inspections. Collecting and analysing process data became common practice. The first self assessment procedures were introduced (e. g. by Humphrey)

The key lesson learned during this period was that the quality of a product cannot be assured by only looking at the final product, i. e. the outcome of the development cycle. Quality assurance has to address the entire development process and here in particular the early phases. This is possible, however, only if synchronization and co-location of assurance and development activities is planned and arranged for. Clearly, tremendous progress was made in terms of raising the quality level of software shipped. Several organisations that carefully selected and analysed their process data could show quality improvement more than a factor of 10 within 5-6 years.

The Formal Period

The goal pursued by formal methods is to increase the trustworthiness of software and to improve productivity by achieving automation. This explains why practitioners put great hopes in these methods. Formal methods are applicable to software specification, transformation and verification. In the most visionary views, proponents expected that once a formal specification was found the entire remaining development process could be automated through a series of correctness preserving transformations. A less demanding solution is attempted in the case of verification. Here a formal specification would serve as yardstick for a manually derived implementation to prove its consistency with the original specification. The kind of tools developed were design languages or specification languages (including editors and checkers), transformation systems and verifiers.

In an industrial environment, the proponents of formal methods were typically confronted with two main arguments, namely education and tools. Even if considerable effort was spend to solve these problems, the acceptance was still less than expected. Formal methods and verification in particular have had their successes mainly for small security or safety critical programs.

For large projects, it usually turned out that neither the requirements nor the design can be expressed in a succinct mathematical model. To each requirements specification there belongs a whole set of non-functional requirements for which different forms of expression are relevant. Examples are the cross domain data interrelationships, the usability, performance and compatibility requirements. Another problem not addressed by formal methods is the need of the requirements analyst or the designer to communicate with non-professional users through language that is understandable for them. The more critical the application is to human organisations, the more important this aspect becomes.

While in era I it was recognized that the highest risks lie in the precoding phases, era II learned that the *pre-design* activities also needed a high amount of attention. It is therefore justified to view the following periods as a new era.

Mastering Complexity (Era III)

This is the era since 1983. The milestone event is the arrival of the PC. The first of the two periods may be considered history, the second period is the contemporary period. During this era the traditional dominance of hardware over software ended. Hardware has become subservant to software, also examplified by the stumbling of some companies (IBM, DEC) that were formerly strong as hardware vendors.

The Structured Period

The structured methods have their origin really in the preceding era (Dijkstra, Mills, Wirth), in particular as far as their application to coding and design is concerned. What made structured methods pervasive in industry was their application to requirements analysis.

What helped strongly was a change in the hardware environment, namely the spread of CRT displays, and here in particular the category of all-points-addressable devices. This opened up computers as a tool for engineers doing graphical designs. Eventually software designers wanted to become more like engineers by drawing rather than writing programs. This called for graphical design notations. Although graphical notation can be formal in nature (as shown by Harel), they are usually considered as a good means to communicate with non-programmers.

The emergence of CRTs as everybody’s user interface also created a new software entity, namely the Graphical User Interface (GUI) tools. Suddenly, software engineering seemed to be able to feed its own industry. It was the starting point of the CASE euphoria. So far only individual small tools like those of a craftsman were available, now everything could be integrated into a universal workbench. The strategic opportunity offered by the CASE technology was that programs could eventually be maintained at the design level rather than at the source code level. This would have been one possible way-out from the language struggle.

As indicated before, not all blossoms ripened into fruits. First the users of CASE tools had to struggle with two surprises: (a) picking a tool meant picking a design and analysis method. If the organisation was not ready for this decision the tool became shelfware, (b) the hope that a CASE tool would automate code generation did not materialize. If people had justified the CASE investment under the premise of automatic code generation, the effort did not pay off. As a result, the dichotomy between design and code continued to exist, meaning, if the design changed the code has to be changed as well. Or vice versa, if the code changed the design has to be updated, an even worse problem. A CASE tool that could be used as a drawing tool

only or as a repository for all kinds of non-related material soon became too expensive, i. e. the learning and maintenance costs were too high.

The structured methods showed up another serious problem. Doing data modelling and process modelling totally distinct from each other, did not lead to a system structure that was easy to maintain, as processes and data did not automatically preserve their interrelationship. Nor was the problem of code reuse and the reengineering of existing systems addressed.

The Object-oriented Period

The potential of software reuse was, as we all know, pointed out by a visionary (McIlroy) as early as 1968. It is fair, however, to say that it was not seriously addressed until recently. In fact it is the theme underlying the entire object-oriented movement, which is the buzzword of our time. What really motivates people to look at software reuse is the recognition of software as an asset. Wherever the world is dominated by hardware designers, software is always an add-on, something done quickly late in the cycle to feed the hardware. If software is planned for reuse (forgetting about unplanned reuse) it will be designed and implemented differently than before. It will be modularized, encapsulated and portable. If object-oriented concepts are applied, the additional mechanisms of inheritance and polymorphism are available.

Similarly as in the case of structured methods, the object-oriented methods were first applied to coding, then to design and later to requirements analysis. Contrary to the structured methods, there is a seamless transition possible between analysis, design and implementation. Code reuse is achieved through the exploitation of class libraries or (mainly in the case of GUI-intensive programs) through the use of application frameworks. Also design reuse is being discussed widely under the concept of design patterns.

Reengineering has also a much clearer defined role as before. It is the process of converting a non-object-oriented system to an object-oriented structure, thus building a new technological base for extension and reuse. Object-orientation has the potential to construct software systems that have a truely modular structure and are easily decomposable in natural subunits (following the criteria originally proposed by Parnas). Processes and data are lumped together.

The problems identified for this generation of development methods mainly relate to the increased demands on testing and software understanding. Also cycle time appears to become an important issue, particularly if a market is still expanding because creative people see opportunities for innovative products and services.

Concluding Remarks

The last 40 years of software development are certainly only the beginning of our field and represent a short period if measured in historic terms. Nevertheless, it is helpful to try to identify some threads that could form a road map into the future. As historians usually argue, the only purpose to study history is to learn about the future.

The model of history used in this essay is a very simplistic one. It emphasizes the learning process of the community. There certainly are much more elaboreted models like the one used by B. Boehm during the same workshop. It allows for thesis and antithesis, challenges and resistance. For a more detailed analysis such a model would by very useful.

Software Engineering – Why it did not work

Jochen Ludewig

A Concept which was never Defined

F.L. Bauer, 1968

> *The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a clean fabrication process, which it should be. What we need, is* ***software engineering.***

That means: Software Engineering was invented as a provocative word without meaning (like "threatened by peace" or "deafening with silence"). Let us check the standard glossary:

Software Engineering

(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

(2) The study of approaches as in (1).

IEEE Std. 610.12 (1990)

Again, not a definition, but a **goal**!

More Differences than Similarities

Successful Engineering

"Engineers" were not defined. Their typical profile (knowledge, behaviour, way oft thinking) developed over at least two centuries. If we want to know what an engineer is, we do not need a dictionary, we can simply watch one of them. (Note: In Software Engineering, there is a tradition to talk about engineers without knowing them. This resembles the way the Romans would talk about the heroic Germanic people[1], or the Germans about the heroic Red Indians[2])

In fact, average engineers do *not* work formally, they do *not* apply difficult mathematics, they do *not* read scientific books, etc., but tend to solve standard problems using standard approaches, and *sometimes* extend their solutions carefully in order to achieve slightly better results.

Craftsmen and Engineers

Engineering was built on top of crafts, like smith, mason, cabinet-maker.
These crafts date back to the MiddleAges. They have developed a **high level of quality**, guaranteed and defended by the **guilds**.

1.Tacitus: De origine et situ Germanorum
2.Karl May: Winnetou

There is no craft behind Software Engineering. There are no people who are used, and expected, to deliver decent quality. There are no guilds who would expel them if they break the rules.

Formality and Standards

Engineers use formalisms for drawings, datasheets etc., but rarely do difficult things in formal ways. Their formalisms are simple and well understood.

In Software Engineering, we do not have many formalisms (except few programming languages) which are generally accepted and understood. In particular, there are no agreed formalisms for communicating information about software or system requirements and design.

Engineers apply standards. Otherwise, they would not be able to communicate, to use components in their designs, to maintain broken equipment, deliver products within reasonable time.

In Software Engineering, we have few standards, which are not even well known and accepted. Therefore, we are not able to do what the engineers can do thanks to their standards.

The Role of the Customer

The overall cost of high quality goods is usually lower than the cost of less mature, less reliable, less complete products. Still, high quality goods are more expensive. Engineers can expect their customers to appreciate good quality because they know the effects of poor quality, and they are able to distinguish good quality from bad quality.

In a market where the customers are stupid, or incompetent, high quality goods are hard to sell.

Where is the "Electrical Engineering" in Electrical Engineering?

Component versus Common Denominator

In a profession like EE, there are many specialists (power generation, electrical engines, microwaves, semiconductors, etc.), but there is no "specialist" for EE, because the essentials of the field are in the **intersection** of all the specializations. Otherwise, the specialists would neither produce good quality, nor could they successfully communicate with each other.

Software Engineering is usually treated as one topic out of many. Students may specialise in compilers, or AI, or SE. This approach means that in a faculty, there are some 10 professors teaching informatics, one of which teaches SE. When students do a project, chances are 9 to 1 that they will not learn how to do it properly. Co-operation is unlikely, one side will usually not benefit.

We should make Software Engineering the common denominator of all branches of informatics, not just another branch.

The Need for a Software Culture

Engineers can deliver good products fairly reliably because they have got

- the craftsmen as their professional ancestors
- an education in which the common principles are not betrayed, but applied without thinking
- degrees which have a clear meaning, and imply certain responsibilities
- a sound theory, which has been expressed in terms simple enough for the engineers
- standards and notations which are generally accepted

- customers who appreciate good quality
- support from researchers when necessary or useful.
- products which develop slowly.

These ingredients together form a **culture**.

> **culture: (...):**
>
> **a:** the integrated pattern of human knowledge, belief, and behaviour that depends upon man's capacity for learning and transmitting knowledge to succeeding generations
>
> **b:** the customary beliefs, social forms, and material traits of a racial, religious, or social group
>
> **c:** the set of shared attitudes, values, goals, and practices that characterises a company or corporation

Merriam Webster's CollegiateDictionary

In a culture, all parts are required. If one is missing, the structure will collapse.

In Software Engineering, we must improve in all aspects!

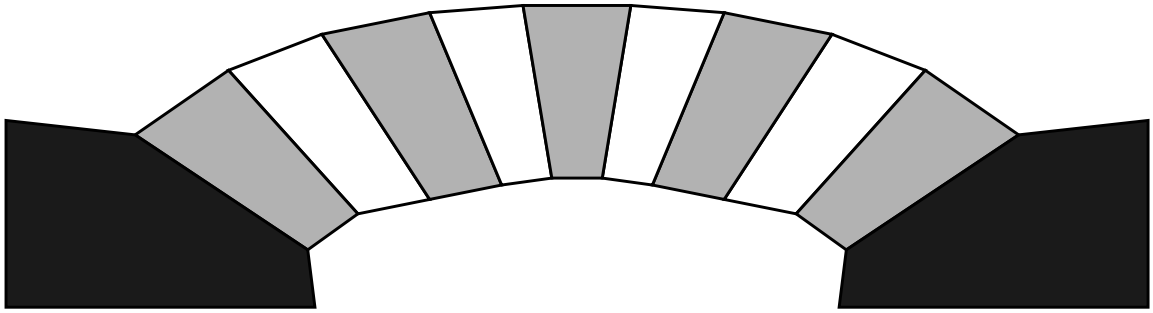

Figure 1: Building blocks of a culture

References

The papers below are available from

http://www.informatik.uni-stuttgart.de/ifi/se/se.html

J. Ludewig: Von der Software-Zivilisation zur Software-Kultur: Die Vision einer verläßlichen Software-Umgebung. H. Mayr. (Hrsg.), Beherrschung von Informationssystemen. GI-ÖCG-Jahrestagung 1996, Klagenfurt, Verlag R. Oldenbourg, Wien, S.255-266.

J. Ludewig: Der Modellstudiengang Softwaretechnik. in P. Forbrig, G. Riedewald (Hrsg.): SEUH'97 (Software Engineering im Unterricht der Hochschulen). Berichte des German Chapter of the ACM, Band 48, Teubner, Stuttgart, S.9-23.

Brief Account of My Research and Some Sample References

Donald Mackenzie

My research is on two closely related topics. One topic is the development of safety-critical and security-critical computer systems, where I have been examining processes of the development and assessment of such systems. The underlying hypothesis is that these are not simply technical matters, but also social ones. The development of a computer system is an organisational and managerial task, and its assessment falls in part into the sphere of the sociology of knowledge. When one constructs a "safety case" or "security case" for a computer system, that is in part an exercise in persuasion. Not only a systems developers, but also customers, regulators, and sometimes users and the public, have to be convinced.

The second research topic is "the sociology of mathematical proof". It relates to the first because deductive proof is increasingly being used to demonstrate the safety and security of computer systems. As is well known, the complexity of such systems means that it is frequently infeasible to test them exhaustively. So since the late 1960s computer scientists have sought ways of showing deductively that programs or hardware designs are correct implementations of their specifications. This practical interest in mathematical proof moves proof from the textbooks and lecture theatres to the world of commerce and the law. In 1991, for example, there was litigation in Britain that hinged upon what mathematical proof should be taken to mean in the context of safety-critical systems. More generally, two different notions of proof (formal proof, as analysed by logicians, and "rigorous argument" as practised by mathematicians) can be seen as co-existing, and sometimes contending, in this area.

The papers in which I have described this research are as follows:

1 'Formal Methods and the Sociology of Proof', in J.M. Morris and R.C. Shaw, eds, Proceedings of the Fourth Refinement Workshop (London: Springer, 1991), 115-124.

2. [Commentary] 'The Fangs of the VIPER', Nature, 352 (8 August 1991), 467-68.

3. 'Computers, Formal Proof and the Law Courts', Notices of the American Mathematical Society, 39 (1992), 1066-69.

4. 'Negotiating Arithmetic, Constructing Proof: The Sociology of Mathematics and Information Technology', Social Studies of Science, 23 (1993), 37-65.

5. 'The Social Negotiation of Proof: An Analysis and a further Prediction', in Peter Ryan and Chris Sennett, eds, Formal Methods in Systems Engineering (London: Springer, 1993), 23-31.

6. 'Computer-Related Accidental Death: An Empirical Exploration', Science and Public Policy, 21 (1994), 233-48.

7. 'The Automation of Proof: A Historical and Sociological Exploration', IEEE Annals of the History of Computing, 17 (3) (1995), 7-29.

8. ‘How do we Know the Properties of Artefacts? Applying the Sociology of Knowledge to Technology’, in Robert Fox (ed.), Technological Change: Methods and Themes in the History of Technology (London: Harwood, 1996), 247-63.

9. ‘Proof and the Computer: Some Issues Raised by the Formal Verification of Computer Systems’, Science and Public Policy, 23 (1996), 45-53.

10. ‘Computers, “Bugs”, and the Sociology of Mathematical Proof’, in William H. Dutton (ed.), Information and Communication Technologies: Visions and Realities (Oxford: Oxford University Press, 1996), 69-85.

11. (with Malcolm Peltu, Stuart Shapiro and William H. Dutton, ‘Computer Power and Human Limits’, in William H. Dutton (ed.), Information and Communication Technologies: Visions and Realities (Oxford: Oxford University Press, 1996), 177-95.

12. (with Margaret Tierney) ‘Safety-Critical and Security-Critical Computing in Britain: An Exploration’, accepted for publication in Technology Analysis and Strategic Management.

13. (with Garrel Pottinger) ‘Mathematics, Technology, and Trust: Formal Verification, Computer Security, and the U.S. Military’, accepted for publication in IEEE Annals of the History of Computing.

FINDING A HISTORY OF SOFTWARE ENGINEERING

Michael S. Mahoney

(Prefatory note: What follows is an abbreviated version of the talk I planned to give, reconstructed some time later from the notes. A decision at the time to proceed interactively and to open the floor to questions and comments soon generated a series of active and interesting exchanges that left time to hit only the highlights. I make no attempt to recapture the actual discussion, since I can take credit for little of its content.)

The title has at least two senses, and I intend them both. In the first instance is describes historians trying to determine just what the subject of their history might be and then deciding how to write that history. What is a history of software engineering about? Is it about the engineering of software? If so, by what historical model of engineering? Is it engineering as applied science? If so, what science is being applied? Is it about engineering as project management? Is it engineering by analogy to one of the established fields of engineering. If so, which fields, and what are the terms of the analogy?

Of what history would the history of software engineering be a part, that is, in what larger historical context does it most appropriately fit? Is it part of the history of engineering? The history of business and management? The history of the professions and of professionalization? The history of the disciplines and their formation? If several or all of these are appropriate, then what aspects of the history of software engineering fit where?

Or, to put the question in another light, is the historical subject more accurately described as “software engineering” with the inverted commas as an essential part of the title.

What seems clear from the literature of the field from its very inception, reinforced by the opening session of this conference, is that its practitioners do not agree on what software engineering is, although most of them freely confess that, whatever it is, it is not an engineering discipline. Historians have no stake in the outcome of that question. We can just as easily write a history of "software engineering" as the continuing effort of various groups of people engaged in the production of software to establish their practice as an engineering discipline. The question of interest to historians is how "software engineers" have gone about that task of self-definition.

In large part, addressing that question comes down to observing and analyzing the answers practitioners have offered to the questions stated in the opening paragraph above. That is, rather than positing a consensus among practitioners concerning the nature of software engineering, historians can follow the efforts to achieve a consensus. Taking that approach would place the subject firmly in the comparative context of the history of professionalization and the formation of new disciplines.

The second sense of the title follows from the approach through "software engineering". Efforts by practitioners to define or to characterize software engineering quite often amount to finding a history, that is, to seeking to identify the current development of software engineering with the historical development of one of the established engineering disciplines or indeed of engineering itself. Using history in this way has its real dangers; the initial conditions cannot by their nature be exactly repeated. Nonetheless, it is at the very least essential that one both have the right history and have the history right, and that is not a simple matter.

I would argue that every definition of software engineering presupposes some historical model. For example, take the oft-quoted passage from the introduction to the proceedings of the first NATO Software Engineering Conference:

> The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering.

The phrase is provocative, to be sure, if only because it leaves all the crucial terms undefined. What does it mean to "manufacture" software? What, precisely, are the "theoretical foundations and practical disciplines" that underpin the "established branches of engineering", and in what sense are they "traditional", i.e. historical?

Some, or perhaps even much, of the disagreement among the participants in the first NATO conferences rested on the different histories they brought to the gatherings. None of them was a software engineer, for the field did not exist. Rather, people came from quite varied professional and disciplinary traditions, each of which had its own history, in many cases a mythic history.

Some viewed engineering as applied science in the sense expressed in another context by John McCarthy:

> In this paper I shall discuss the prospects for a mathematical science of computation. In a mathematical science, it is possible to deduce from the basic assumptions, the important properties of the entities treated by the science. Thus, from Newton's law of gravitation and his laws of motion, one can deduce that the planetary orbits obey Kepler's laws.[1]

1."Towards a mathematical science of computation", *Proc. IFIP Congress 62* (Amsterdam: North-Holland, 1963), 21.

In another version of the paper, he changed the precedent only slightly:

> It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance.[1]

The applications of mathematics to physics had produced more than new theories. The mathematical theories of thermodynamics and electricity and magnetism had informed the development of heat engines, of dynamos and motors, of telegraphy and radio. Those theories formed the scientific basis of engineering in those fields. McCarthy looked to a mathematical theory of computation to play a similar role for programming, and he was not alone.

Yet, despite the substantial achievements of theoretical computer science during the 1960s, Christopher Strachey could still lament in a discussion on the last day of the second NATO Conference that "one of the difficulties about computing science at the moment is that it can't demonstrate any of the things that it has in mind; it can't demonstrate to the software engineering people on a sufficiently large scale that what it is doing is of interest or importance to them."[2] About a decade later, a committee in the United States reviewing the state of art in theoretical computer science echoed his diagnosis, noting the still limited application of theory to practice.[3] By the mid-'70s, moreover, it seemed clear to some that, even if existing theory had practical application, it would not quite meet the needs of software engineering. As Barry Boehm put it In a 1976 article:

> Those scientific principles available to support software engineering address problems in an area we shall call *Area 1: detailed design and coding* of *systems software* by *experts* in a relatively *economics-independent* context. Unfortunately, the most pressing software development problems are in an area we shall call *Area 2: requirements analysis design, text, and maintenance* of *applications software* by *technicians* in an *economics-driven* context.[4]

However successful the experimental systems and theoretical advances produced in the laboratory, especially the academic laboratory, they did not take account of the challenges and constraints of "industrial-strength" software in a competitive market. Those problems were, as Fritz Bauer had put it in an address at IFIP 71, "too difficult for the computer scientist".

If not applied science, then what? Others at the NATO conference had proposed models of engineering that emphasized analogies of practice rather than theory. Perhaps the most famous of these was M.D. McIlroy's evocation of the machine-building origins of mechanical engineering and the system of mass production by interchangeable parts that grew out of them. As recent studies of the American machine-tool industry during the 19th and early 20th century have shown, McIlroy could hardly have chosen a more potent model (he has a longstanding

1. "A basis for a mathematical theory of computation", *Proceedings WJCC, 9-11 May, 1961* (New York: NJCC, 1961), 225-238; repr. with corrections and an added tenth section, in P. Braffort and D. Hirschberg (eds.), Computer Programming and Formal Systems (Amsterdam: North-Holland Publishing Co., 1963), 33-70; at 69.
2. Peter Naur, Brian Randell, and J.N. Buxton (eds.), *Software Engineering: Concepts and Techniques. Proceedings of the NATO Conferences* (NY: Petrocelli, 1976), 147.
3. *What Can Be Automated? (COSERS)*, ed. Bruce W. Arden (Cambridge, MA: MIT Press, 1980), 139. The committee consisted of Richard M. Karp (Chair; Berkeley), Zohar Manna (Stanford), Albert R. Meyer (MIT), John C. Reynolds (Syracuse), Robert W. Ritchie (Washington), Jeffrey D. Ullman (Stanford), and Shmuel Winograd (IBM Research).

interest in the history of technology). Between roughly 1820 and 1880, developments in machine-tool technology had increased routine shop precision from .01" to .0001". More importantly, in a process characterized by the economist Nathan Rosenberg as "convergence", machine-tool manufacturers learned how to translate new techniques developed for specific customers into generic tools of their own. So, for example, the need to machine percussion locks led to the development of the vertical turret lathe, which in turn lent itself to the production of screws and small precision parts, which in turn led to the automatic turret lathe. It was indeed the automatic screw-cutting machine that McIlroy had in mind.

Especially as expressed by McIlroy, the idea has had a long career in software engineering. During the '70s It directed attention beyond the development of libraries of subroutines to the notion of "reusable" programs across systems, and in the '80s it underlay the growing emphasis on object-oriented programming as the means of achieving such reusability on a broad scale. It is essentially what Cox is looking for as software's "industrial revolution". More generally, the analogy with machine-building and the metaphorical language of machine-based production became a continuing theme of software engineering, often illustrated by pictures of automobile assembly lines, as in the case of Peter Wegner's four-part article in *IEEE Software* in 1984 on "Capital-Intensive Software Technology".[1] The cover of that issue bore a photograph of the Ford assembly line in the '30s, and a picture of the same line in the early '50s adorns Greg Jones's *Software Engineering* (Wiley, 1990).

As the move from machine tools to the assembly line suggests, McIlroy's model of mechanical engineering was closely akin to Bauer's proposal at IFIP 71 that "software design and production [be viewed] as an industrial engineering field".

> For the time being, we have to work under the existing conditions, and the work has to be done with programmers who are not likely to be re-educated. It is therefore all the more important to use organizational and managerial tools that are appropriate to the task.[2][

On that model the problems of large software projects came down to the "division of the task into manageable parts", its "division into distinct stages of development", "computerized surveillance", and "management". Bauer's idea was not new. In a "Position Paper for [the] Panel Discussion [on] the Economics of Program Production" at IFIP 68, also presented in substance at the NATO conference, R.W. Bemer of GE had already suggested that what software managers lacked was a proper environment, namely a "software factory", which "should be a programming environment residing upon and controlled by a computer."

Few concepts are as heavily freighted with history as that of the factory, and this was especially the case in the 1960s as exponents of automation strove to complete the revolution in production sparked by Henry Ford's assembly line and at the same time to distinguish themselves from him. Hand in hand with automation went operations research and management science, the practitioners of which similarly insisted on the essential difference between their scientific approach and the "scientific management" of the industrial engineer. Herbert Simon was not fooled and he reminded the members of the Operations Research Society that

---

4. Boehm, "Software Engineering", *IEEE Transactions on Computers*, C-25,12(1976), 1226-41 (repr. in *Milestones of Software Engineering*, ed. Paul W. Oman and Ted G. Lewis [Los Alamitos, CA: IEEE Computer Society Press, 1990, 54-69), at 1239 (67).
1. *IEEE Software* **1**,3 (July 1984), 7-45.
2. Bauer, "Software Engineering", *Information Processing 71* (Amsterdam: North-Holland Publishing Co., 1972), I, 530-538; at 532.

> Except in matters of degree (e.g., the operations researchers tend to use rather high-powered mathematics), it is not clear that operations research embodies any philosophy different from that of scientific management. Charles Babbage and Frederick Taylor will have to be made, retroactively, charter members of the operations research societies. ... No meaningful line can be drawn any more to demarcate operations research from scientific management or scientific management from management science.[1]

One does not divorce oneself from history simply by denying it. An attentive reading of most of the literature on the "software factory" reveals the continuing influence of Taylor's "one best way" and of Ford's management by mechanization. Yet, even a cursory analysis of the nature of software and its production shows how far short they fall of the technical and material preconditions of Taylor's and Ford's methods. To the extent that more recent versions of the software factory presuppose a highly trained, highly paid workforce given time and latitude to forge finely crafted software tools and parts, the concept seems to beg the question that prompted it in the first place.

What can historians looking for the history of software engineering tell software engineers looking for a history of their field? Forget the cliche about ignoring the lessons of the past. Think rather about the histories that people carry with them by virtue of their origins, education, and professional acculturation. Those histories reveal themselves in many ways, among them in the names people give to new undertakings. Software engineering began as a search for an engineering discipline on which to model the design and production of software. That the search continues after twenty five years suggests that software may be fundamentally different from any of the artifacts or processes that have been the object of traditional branches of engineering: it is not like machines, it is not like masonry structures, it is not like chemical processes, it is not like electric circuits or semiconductors. It thereby raises the question of how much guidance one may expect from trying to emulate the patterns of development of those engineering disciplines. During general discussion on the last day of the Rome conference, I.P. Sharp came at the problem of software production from an entirely different angle, arguing that one ought to think rather in terms of "software architecture" (= design), which would be the meeting ground for theory (computer science) and practice (software engineering). "Architecture is different from engineering," he maintained and then added, "I don't believe for instance that the majority of what [Edsger] Dijkstra does is theory – I believe that in time we will probably refer to the 'Dijkstra School of Architecture'."[2] That is no small distinction. Architecture has a different history from engineering, and we train architects differently from engineers.

1. Herbert A. Simon, *The New Science of Management Decision* (NY: Harper & Row, 1960), 14-15.
2. J.N. Buxton and B. Randell (eds.), *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969* (Birmingham: NATO Science Committee, n.d.), 12.

Science versus engineering in computing

Peter Naur

In order to achieve a proper view of software engineering, a view that will help clarify how and why such a thing may be usefully pursued and taught, it seems to me important, first of all, to establish a tenable understanding of the whole complex of the interrelated notions denoted science, engineering, and computing.

Achieving such an understanding in my view meets the difficulty that the discussion around these items in recent years has been entirely misguided. This misguidance has been furthered by several different confusions, including

- the claim that science is a matter of logic,
- the dominance of behaviourism in psychology,
- the claim that sciences are logical constructions upon foundations,
- the adoption of an information processor view of human thought,
- the claim that the human handling of language may be described in terms of rules.

In view of this situation I have made an examination of human knowing, so as to clarify its relations to logic, language, computing, and science. This has been published as a book: Knowing and the Mystique of Logic and Rules. As one major conclusion of this book, science and scholarship ('Wissenschaft') may properly be described to be centered around *coherent description*.

Adopting this notion, to wit that *the core of science is coherent description*, it remains merely to state that in this context *engineering* must properly be understood to be centered around the *activity of construction*. With such a view it is clear that the activity of engineering will depend on the descriptions established scientifically, but will not in itself lead to scientific contributions.

References

Datalogi som videnskab. DIKU rapport nr. 95/4, 1995. Version in English: Computing as Science has been rejected for publication by Comm. ACM

Knowing and the Mystique of Logic and Rules, Kluwer Academic Publishers, xii + 365 pages, 1995.

Software Engineering: An Unconsummated Marriage

David Lorge Parnas

Abstract

Although the first of many conferences on "Software Engineering" was held in Munich nearly three decades ago, communication between those who study software and those who work as Engineers has not been effective. Today, the majority of Engineers understand very little of the "science of programming". On the other side, the scientists who study programming understand very little about what it means to be an Engineer, why we have such a profession, how the profession is organised, and what Engineers learn during their education. In spite of this mutual ignorance, today's Engineers spend much of their time writing and using software, and an increasing number of people trained in Computer Science or Mathematics pontificate about "what Engineers do".

Studying the traditional areas of engineering we find (1) that they have faced, and developed solutions to, many of the issues that software engineering experts are discussing today. We also find that traditional engineers are beginning to recognise the need to treat software development as a new branch of Engineering. Many cannot do their jobs properly without a better knowledge of the science of software. Others cannot live up to their own professional responsibilities unless if they use software packages developed by people who do not take responsibility for their "fitness for use".

We conclude that the two fields have much to learn from each other and that the marriage of software and engineering should be consummated.

Software objects in the Deutsches Museum München – Some considerations

Hartmut Petzold

Surely software represents one of the "most important steps" in the historical development of technology, referred to in the statute of the Deutsches Museum, when it demands its illustration and documentation by "prominent and typical masterpieces".

The wished completion of the present collection and exhibition by software objects does not only mean the integration of one more class of subjects into the already existing broad spectrum of the museum. Rather it is a question of a new technological quality, which will be effective in all departments of the museum in the future. Already one can notice this tendency in all new exhibitions.

Usually the traditional western museums were seen as institutions which feel bound to the spirit of enlightenment and devoted themselves to the collection and presentation of objects with hardware quality.

The statement behind these questions applies largely to the technical activities with the museums artifacts, as to the extensive problems with its transport, its presentation in the gallery, its storage in the depositories, its lending out to external exhibitions and also its restoring and repairing. The workaday routine of the museums collaborators is nearly completely determined by problems connected with the artifacts.

All the different artifacts must be incorporated in a list of objects which can be administrated identically. It must be possible to identify them by a name and by a number and it must be possible to store them at a determined place for a very long time. This means that software only can be included into the museum collection in a defined shape as a software object.

Presumably the most important quality of artifacts in a museum lies in the fact that they can be perceived by the visitor's senses without a special qualification. The process of the perceptibility by the senses is crucial for all presentations.

Artifacts with hardware quality are immediately perceptible for the human eyes. Calculated presentations and also the individual educational background of the visitors can change the mode of the perceptibility.

The handling of software objects put these questions much more fundamentally. What does an object with software quality look like? Obviously there is no agreement.

In an exhibition which is only assembled of a certain number of computer screens in operation, the computer hardware is not much in evidence. But is that what is to be seen on the screen really the software object? I want to precise my question: Should we in the Deutsches Museum pass that what is to be seen on the screen off as the side of the software object which is perceptible by the eye and should we even define its "appearance"?

The importance of traditional museums depends for a great part on the originality and the singularity of its collected artifacts. On principle copies from hardware objects are assessed lower than the originals. When this difference is dropped the substance of the museum is touched. I think that here is a fundamental problem for the relations between software objects and the museum.

One could imagine that in the future a suitable computer would replace the depot room. A telephone call from another museum, from an exhibition maker or from a private buff could be sufficient to initiate the copymaking not only from one software object but even from the whole collection and to transfer it by the telephone line. Because all copies will be absolutely identical the importance of the museum could be derived essentially from the documentation, its scolarship and its administration. The assessment of the ratio between object and "completing materials" would be reverse.

The role of the "technical monument", in some cases of the "national monument", is important for the way the museum sees itself and is seen by the public. I am not sure if the peculiarities of a software object are oposed to its presentation in the museum in this sense. But I am sure that there will be many new possibilities how to present the software objects.

The originators of the Deutsches Museum have introduced one criterion for the selection out of the immense number of possible objects: only "masterpieces" should be included into the collection. The category "masterpiece" orients of ideas from the last centuries particularly the 19th. But it would be wrong to reduce the importance of this criterion to a historical reminiscence. Despite all technical change in our century the idea of the masterpiece continues to exist – not only in Germany. The success of the Deutsches Museum – the whole name is "Deutsches Museum von Meisterwerken der Naturwissenschaft und Technik" – is due not least to this criterion and what has been made out of that. If a technological product is to be seen as

a masterpiece the Deutsches Museum, not the Patent Office, takes over the role of the arbitration court: The object becomes a masterpiece when it is admitted to the museum collection.

For almost a century, through all political systems in Germany, the public has accepted this role. The idea of the masterpiece is still an ideal present in the public awareness and the Deutsches Museum is the only holy temple in Germany. Obviously the apparently timeless "masterpiece" has found a role as a counterpart to the industrial product which is measured by its price performance ratio. Surely a Walhalla of the heroes and the fights of technology satisfies a deep urge of the protagonists of the traditional hardware technologies and also of the public. The market already declares "winner" and "success" if a piece of software sells well. But the Deutsches Museum can, and should, judge software by conferring opon selected software the term "Masterpiece".

The 1968/69 NATO Software Engineering Reports

Brian Randell

The idea for the first NATO Software Engineering Conference, and in particular that of adopting the then practically unknown term "software engineering" as its (deliberately provocative) title, I believe came originally from Professor Fritz Bauer. Similarly, if my memory serves me correctly, it was he who stressed the importance of providing a report on the conference, and who persuaded Peter Naur and me to be the editors. (I was at the time working at the IBM T.J. Watson Research Center in the U.S.A., but had got to know "Onkel Fritz" through having been a member of the IFIP Algol Committee for several years.) As a result, it was agreed that Peter and I would stay on for an extra week after the conference in order to edit the draft report, though we arranged to move from Garmisch-Partenkirchen to Munich for this second week.

Quoting from our Report of the 1968 Conference [Naur and Randell January 1969]:

> "The actual work on the report was a joint undertaking by several people. The large amounts of typing and other office chores, both during the conference and for a period thereafter, were done by Miss Doris Angemeyer, Miss Enid Austin, Miss Petra Dandler, Mrs Dagmar Hanisch and Miss Erika Stief. During the conference notes were taken by Larry Flanigan, Ian Hugo and Manfred Paul. Ian Hugo also operated the tape recorder. The reviewing and sorting of the passages from written contributions and the discussions was done by Larry Flanigan, Bernard Galler, David Gries, Ian Hugo, Peter Naur, Brian Randell and Gerd Sapper. The final write-up was done by Peter Naur and Brian Randell. The preparation of the final typed copy of the report was done by Miss Kirsten Anderson at Regnecentralen, Copenhagen, under the direction of Peter Naur."

As I and other participants have since testified, a tremendously excited and enthusiastic atmosphere developed at the conference. This was as participants came to realize the degree of common concern about what some were even willing to term the "software crisis", and general agreement arose about the importance of trying to convince not just other colleagues, but also policy makers at all levels, of the seriousness of the problems that were being discussed. Thus throughout the conference there was a continued emphasis on how the conference could best be

reported. Indeed, by the end of the conference Peter and I had been provided with a detailed proposed structure for the main part of the report. This was based on a logical structuring of the topics covered, rather than closely patterned on the actual way in which the conference's various parallel and plenary sessions had happened to be timetabled.

Peter and I were very pleased to have such guidance on the structuring and general contents of the report, since we both wished to create something that was truly a *conference* report, rather than a mere personal report on a conference that we happened to have attended. Indeed Peter argued that we should not provide *any* additional text at all ourselves, but rather produce the main part of the report merely by populating the agreed structure with suitable direct quotations from spoken and written conference contributions. I, however, persuaded him that brief editorial introductions and linking passages would improve the continuity and overall readability of the report. So, (together with the decision that a small selection of the written texts would also be incorporated in full as appendices), we arrived at the final form of the report.

In Munich we worked from the notes taken by the rapporteurs, which we had arranged would be keyed, as they were made, to footage numbers on the recorded tapes. The tapes were not systematically transcribed, since this process typically takes five to six times real time. Rather we used the rapporteurs' notes, and our memories, to locate particularly interesting and apposite sections of the tapes and just these were transcribed. We thus built up a large set of transcribed quotations, which we supplemented with suitable quotations from the written contributions. Then, for each section of the report, one or other of us attempted to turn the relevant set of quotations into a coherent and pseudo-verbatim account of the discussion on that topic, bringing together material from quite separate sessions when appropriate since many topics had been revisited in various parallel and plenary sections.

The work in Munich was as enjoyable as it was intense, and afforded plenty of opportunity for re-hearing some of the more memorable discussions, so that many of these became etched much more deeply into my memory, and had a stronger effect on my subsequent research, than would have been the case had I merely taken part in the conference. The report was virtually complete by the end of the week in Munich, and then Peter Naur took everything back with him to Copenhagen where a complete first draft was produced using a paper tape-controlled typewriter (I assume a flexowriter) – a technique that seemed novel at the time but one that he correctly advised us would greatly aid the preparation of an accurate final text. (My memory tells me that this draft was then circulated to participants for comments and corrections before being printed, but no mention is made of this in the report so I may be wrong.)

The actual printing and distribution was done by NATO, and the Report became available in January 1969, just three months after the conference. Copies were distributed freely on request and it rapidly achieved wide distribution and attention. One of the more delightful reactions to it from amongst the participants was that of Doug McIlroy, who described it as "a triumph of misapplied quotation!". (It was only many years later did I learn from a short article by Mary Shaw that Al Perlis gave out copies of the report to the CMU computer science graduate students with the words "Here, read this. It will change your life." [Shaw 1989])

Such was the success of the first conference that the organizers sought and obtained NATO sponsorship for a second conference, to be held one year later in Italy. Peter Naur, wisely, was not prepared to repeat his editorial labours, but I – rather rashly – after some initial hesitation agreed to do so, this time in co-operation with John Buxton. As I recall it, the plans for the second conference were discussed at a meeting held in an office at NATO Headquarters. My main memory is that the office was dominated by a very large and impressive safe, which to my amusement was revealed to be completely empty when our host, at the end of the meeting,

opened it so as to put away the bottles from which drinks had earlier been served to us. During these preparatory discussions I provided, based on my hard-won experience at Munich, what I proudly considered to be a very well thought-out list of requirements regarding the facilities that we would need to have in Rome. (The most important of these was that the editorial team should have full time access to an Italian-speaker who would help sort out any difficulties that might arise – of this, more later.)

My initial (over)confidence was also in part due to the fact that this second time around, John and I had been offered the full time services of two experienced technical writers from ICL, namely Ian Hugo (who had been closely involved in the preparation of the first report) and Rod Ellis, and we had each arranged to be accompanied to Rome by an expert secretary, Margaret Chamberlain and Ann Laybourn, respectively. Ian, incidentally, went on to help found Infotech, a company that subsequently over a period of years organized a large number of technical conferences, each of which led to the publication of a State-of-the-Art Report, whose format closely matched that of the NATO reports.

In the event the second conference was far less harmonious and successful than the first, and our editorial task turned out to be very different. Quoting from our introduction to the Report of the 1969 Conference [Buxton and Randell April 1970]:

> "The Rome conference took on a form rather different from that of the conference in Garmisch and hence the resemblance between this report and its predecessor is somewhat superficial. The role played by the editors has changed and this change deserves explanation. … The intent of the organizers of the Rome conference was that it should be devoted to a more detailed study of the technical problems, rather than including also the managerial problems which figured so largely at Garmisch. … The resulting conference bore little resemblence [sic] to its predecessor. The sense of urgency in the face of common problems was not so apparent as at Garmisch. Instead, a lack of communication between different sections of the participants became, in the editors' opinions at least, a dominant feature. Eventually the seriousness of this communications gap, and the realization that it was but a reflection of the situation in the real world, caused the gap itself to become a major topic of discussion. … In view of these happenings, it is hardly surprising that the editors received no clear brief from the conference as to the structure and content of the report."

Thus the task of producing a report which was both respectable and reasonably accurate was much more difficult than I could have imagined – and was not aided by all sorts of difficulties that we suffered, almost all of which would have been much more easily dealt with if a local organizer had been provided as agreed. Nevertheless, a number of the participants expressed pleased surprise at our report, when they afterwards received a draft for checking, and evidently thought more highly of it than of the conference that it purported to document.

The conference had been held outside Rome in a rather charmless American-style hotel whose facilities and cuisine I'm sure did little to engender a harmonious atmosphere. It had been agreed beforehand that we would move to a (particular) hotel in Central Rome for the report writing – only during the conference did we discover that no attempt had yet been made to reserve accommodation at this hotel. Needless to say, the hotel turned out to be full, and so last minute arrangements had to be made, and our offices and families alerted to the change of plans.

On the Saturday morning following the conference the six of us, plus all our luggage, and a very impressive set of typewriters, tape-recorders, boxes of paper and other office supplies, etc., were transported by minibus to Central Rome to the very pleasant substitute hotel,

which was situated just across from the main entrance to the Roman Forum. In fact we arrived rather too early for the hotel, since only the small suite we were to use as an editorial office was available, our bedrooms not yet having been vacated and cleaned. We thus had to accept the hotel receptionist's suggestion that we all be initially installed in this one suite until our own rooms were ready.

I still treasure the memory of our arrival, which was watched open-mouthed by the various hotel staff and guests in the lobby. This was not just because of our number and our mountain of luggage, and the small army of porters – just one of whom had a door key – that were being employed to move it. It was undoubtedly also due to the interesting appearance the six of us must have made – in particular the fact that Margaret Chamberlain was wearing an extremely short miniskirt. This fashion apparently had yet to spread from London to Rome, where it was still regarded at least by all the Italian men as quite sensational. And Rod Ellis was wearing a splendid long black leather jacket and the sort of thick-soled suede shoes that at that time were known, in Britain at least, as "brothel-creepers". But most memorable of all was John Buxton's remark when the last of the porters had bowed himself out of our suite, and the six of us were standing around our luggage mountain wondering what to do first. He suddenly said, "I've had a great idea. Let's phone down to the front desk and ask for two thousand foot of colour film and a stronger bed, please."

This provided a wonderful start to a week in which we managed to find continual solace in humour despite the pressure of work and the many adversities we had to face. For example, by mid week, almost all of the original typewriters and tape recorders were no longer operational, and we were threatening to abandon Rome and to move to Brussels in order to complete the work at NATO Headquarters. Even the stapler had broken. As Ian Hugo has reminded me, "the suite had a bathroom which was surplus to requirements and the bath became the final resting ground for dead typewriters, tape recorders, etc; by the end of the week it was full to overflowing!" However we soldiered on, though in the end half of the Report had to be bravely typed by Ann Laybourn on a totally-unfamiliar German-keyboard typewriter that we had managed to borrow ourselves from the hotel.

All these adversities – whose impact would have been much less had we had the promised local assistant – in fact helped to bind us together as a team. Rod Ellis' brilliant gift for mimicry also helped by providing many welcome moments of general hilarity as, suiting his choice to the topic at hand, he switched effortlessly in conversations with us between the voices of Edsger Dijkstra, Fritz Bauer, and many of the other participants whose conference comments had been captured for posterity by our tape recorders.

We did in fact finish the report by early on the Friday evening – in good time for a final celebration dinner, once Rod and Ian had returned from the University of Rome where they had made copies of the draft report (and, rather fittingly, broken the photocopier). It was in keeping with the rest of the week, though, that nearly all the restaurant waiters in Rome chose that moment to go on strike – indeed, we saw a large procession of them march right past our windows shouting and waving banners – so that we had to content ourselves with an in fact excellent dinner in the hotel.

Something I had completely forgotten until I reread the introduction to the 1969 Report while preparing this brief account was that this second report was typeset at the University of Newcastle upon Tyne, to where I had moved from IBM in the interim. In fact some of the world's earliest work on computerized type-setting had been done at Newcastle. Quoting from the report: "The final version of the report was prepared by the Kynock Press, using their computer type-setting system (see Cox, N.S.M and Heath, W.A.: "The integration of the publishing

process with computer manipulated data". Paper presented to the Seminar on Automated Publishing Systems, 7–13th September 1969, University of Newcastle upon Tyne, Computer Typesetting Research Project), the preliminary text processing being done using the Newcastle File Handling system …". (However, I perhaps should also mention that this second report took three months longer to produce than its predecessor report.)

Unlike the first conference, at which it was fully accepted that the term software engineering expressed a need rather than a reality, in Rome there was already a slight tendency to talk as if the subject already existed. And it became clear during the conference that the organizers had a hidden agenda, namely that of persuading NATO to fund the setting up of an International Software Engineering Institute. However things did not go according to their plan. The discussion sessions which were meant to provide evidence of strong and extensive support for this proposal were instead marked by considerable scepticism, and led one of the participants, Tom Simpson of IBM, to write a splendid short satire on "Masterpiece Engineering".

John and I later decided that Tom Simpson's text would provide an appropriate, albeit somewhat irreverent, set of concluding remarks to the main part of the report. However we were in the event "persuaded" by the conference organizers to excise this text from the report. This was, I am sure, solely because of its sarcastic references to a "Masterpiece Engineering Institute". I have always regretted that we gave in to the pressure and allowed our report to be censored in such a fashion. So, by way of atonement, I attach a copy of the text as an Appendix to this short set of reminiscences.

It was little surprise to any of the participants in the Rome conference that no attempt was made to continue the NATO conference series, but the software engineering bandwagon began to roll as many people started to use the term to describe their work, to my mind often with very little justification. Reacting to this situation, I made a particular point for many years of refusing to use the term or to be associated with any event which used it. Indeed it was not until some ten years later that I relented, by accepting an invitation to be one of the invited speakers at the International Software Engineering Conference in Munich in 1979. The other invited speakers were Barry Boehm, Wlad Turski and Edsger Dijkstra. I was asked to talk about software engineering as it was in 1968, Barry about the present state, Wlad about the future of software engineering, and Edsger about how it *should* develop. I had great fun in preparing my paper [Randell 1979] since I included numerous implied challenges to Barry, whose talk was scheduled immediately after mine, to justify claims about progress since 1968. He studiously ignored all these challenges, or perhaps failed to recognize them, I'm sorry to say.

In my 1979 attempt at describing the 1968/9 scene I did not feel it appropriate to dwell on my experiences in helping to edit the two NATO Reports – so I am very pleased to have had cause to complete my personal software engineering reminiscences, so-to-speak. I thank the organizers of this conference for giving me this opportunity and, in particular, a belated means for me to publish the text that was so sadly censored from the Report of the 1969 Conference.

References

J.N. Buxton and B. Randell, (Ed.): *Software Engineering Techniques: Report on a Conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969,* Brussels, Scientific Affairs Division, NATO, April 1970, 164 p.

P. Naur and B. Randell, (Ed.): *Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968,* Brussels, Scientific Affairs Division, NATO, January 1969, 231 p.

B. Randell: “Software Engineering in 1968,” in *Proc. of the 4th Int. Conf. on Software Engineering*, pp. 1–10, Munich, 1979.

M. Shaw: “Remembrances of a Graduate Student (for panel, “A Twenty Year Retrospective of the NATO Software Engineering Conferences”),” in *Proc. 11th Int. Conf. on Software Engineering*, vol. 11, pp. 99–100, 1989. (Reprinted in *Annals of the History of Computing*, Anecdotes Department, 11, 2, 1989, pp.141–143.)

Plex – A Scientific Philosophy

Douglas T. Ross

Plex is a scientific philosophy – not a philosophy of science, but a philosophy which itself is scientific in its character and practice.

> **science** n 1a: possession of knowledge as distinguished from ignorance or misunderstanding Webster’s Collegiate 1979

Plex has no assumptions or axioms – only definitions – expressed in ordinary language – but used with extraordinary purpose and precision. No formal logic or mathematics appears. Even counting is not allowed, at first.

The entire Epistemology of Plex is the Formal Saying:

> Only that which is known-by-definition
> is *known* – by definition.

where the Definition of Definition follows from

> If D defines X, then Y satisfies D if <Y> is <X>.

where “is” is the ontological copula, and to define “satisfies” needs a longer derivation than is suitable for presentation here. The self-definition of X is X itself; “X is X” *expresses* self-definition of X. {The < >s are called Meaning Quotes; “ ”s are Naming Quotes.}

The Possibility Definition is:

> A possibility is that which may, but *need not* BE.

and	<may not BE>	and	<may BE>	are	the two States of Possibility
while	ISN’T	and	IS	are	the two States of BEing
so	<may not BE>/ ISN’T	and	<may BE>/IS	are	the two **States of Reality**

Notice that BEing itself is a possibility (for it satisfies the Possibility Definition). IS is its *realization*.

To complete these few Plex foundation definitions, given that “Nothing” is our proper name for <that which does not exist> – here is the

First Definition of Plex:

Nothing doesn't exist.

where "existence" is BEing and is that which is *defined* by the First Definition – as pure non-Nothing-ness – i.e. IS itself. [Notice that the First Definition does *not* define Nothing, which has no definition at all – for <it> ISN'T! – having no self-definition.]

By the Plex Epistemology, *without* any definition, Nothing is known.

I.e. <> = <Nothing> = <"Nothing">
= <that which is known without definition>
so Nothing is <The Ultimate Knowledge>
by means of which all else is known.

Finally, the **Definition of Meaning** arises in the Saying:

Every possibility is meaningful –
its meaning BEing its possibility.

– a most marvelously recursive discovery. Yes, this *is* our ordinary every-day meaning of "meaning", although it is pure Plex and is not to be found in any dictionary. To understand it it is merely necessary to appreciate the Possibility Definition, for by *this* definition, that which there is defined – (the <may BE>/<may not BE> dichotomy of the States of Possibility) – IS the meaning (of that possibility).

What *difference* does it make
whether that possibility is realized or not?

I.e. in the overall scheme of things,

What is the <*before*>-*to*-<*after*> *effect* of its existence?
– What *role* does it fulfill?

Surely that is what matters about <it>. That is what gives it meaning.

The meaning of any word (which also is the definition of Superposition)

Let <point> = <any word>. Then prove the following Propositions.

P1) Let points be such that, except for identity,
they all are indistinguishable.
P2) Let there be only points.
P3) Let the world be the collection of all points.
P4) Then the identity of a point
is the collection of all other points,
P5) And every point is the whole world.

Proof of P5 by Mathematical Induction

I n = 1: A world of one point is the whole world.

II Assume the theorem true for n-1 points. (n>1)
I.e. for any collection of n-1 points, every point is the whole world.

III To prove the theorem for n points given its truth for n-1 (n>1)

a) The identity of any one point, p, in the collection is a collection of n-1 points, each of which is the whole world, by II.

b) The identity of any other point, q, i.e. a point of the identity of p, is a collection of n-1 points, each of which is the whole world, by II.

c) The identity of p and the identity of q are identical except that where the identity of p has q, the identity of q has p. In any case, p is the whole world by b) and q is the whole world by a).

d) Hence both p and q are the whole world, as are all the other points (if any) in their respective identities (and shared between them).

e) Hence all n points are the whole world.

IV For n=2, I is used (via II) in IIIa and IIIb, q.e.d.

V Q.E.D. by natural induction on the integers.

The true import of the meaning definition and of existence itself is embodied in the Propositions, P1 through P5, which resolve Plato's "ideals" ("general elements" in modern parlance) – Is there an ideal cat through which (by some relationship) I can know that *my* cat *is* a cat? Propositions P1-P5 define and demonstrate how the meaning of any word *works*. Every usage of the word is *just as perfect* as any other. This also lies behind the Bose-Einstein State of Matter, recently demonstrated. Time going forward in a expanding universe of <now> – <all that exists> is a consequence.

This 1975 proof was first presented in public in an abortive Graduate Seminar on Plex in the Fall of 1984 in the MIT EECS Department. It has since then appeared in several publications, including

D.T. Ross: From Scientific Practice to Epistemological Discovery.
In: C. Floyd, H. Züllighoven, R. Budde, R. Keil-Slawik (Eds.): Software Development and Reality Construction. Springer-Verlag, 1992, pp. 60-70

D.T. Ross: Understanding /:\ The Key to Software.
In: Scaling Up: A Research Agenda for Software Engineering. National Research Council NRC-04131-7, 1989, pp. 66-73

Research Abstract

Stuart Shapiro

In the 1960s, the efficient and timely production and maintenance of reliable and useful software was viewed as a major problem. In the 1990s, it is still considered a major problem. The "software crisis" which was declared three decades ago persists, assuming it makes any sense to speak of a thirty year crisis. Although most would admit to some amelioration of the "crisis," steadily increasing requirements and ambitions have helped sustain it.

At the NATO conferences of the late sixties, the solution to the "crisis" was declared to be "software engineering." This, however, begged a number of questions. What is the nature of software as a technological medium? How does software development compare and contrast with other areas of technological practice. What is engineering? Is it sensible to speak of engineering software?

Answering these questions has been a difficult and tempestuous process which continues to this day. This becomes abundantly clear when one examines the practitioner discourse which has taken place in a number of prominent computing and software trade and professional journals over the years in the form of articles, news and conference reports, book reviews, and letters. The participants in this discourse represent a wide range of nationalities and educational and employment backgrounds. Thus, while the journal-based discourse in which they have engaged over time does not constitute the complete social and intellectual history of software technology, it is certainly a prime component of that history.

This discourse suggests that software as well as computing in general have characteristics which render them fundamentally different from other technologies. As in fields such as chemical engineering, would-be computing and software professionals have pursued that professionalism in a highly deliberate fashion and have found the process difficult and controversial. Unlike practitioners in those other fields, however, they have been unable to forge a consensus on an appropriate model—art, craft, science, engineering, etc.—for their activities. In attempting to achieve that consensus, moreover, they have engaged in frequent and often heated debate concerning the epistemology and praxis of computing and software development. Complicating matters further has been widespread confusion regarding the nature of engineering and science. Especially prevalent has been the misconception that engineering is essentially applied science.

Historians and sociologists (as well as some insightful technological practitioners) have discredited the notion that engineering is applied science, but it is not surprising that this myth has proven so attractive to computing and software technologists. In the aftermath of the Second World War, many areas of engineering fell under the spell of this myth. With science sitting at the top of the epistemological totem pole and, in the U.S., lecture halls filling with ever increasing numbers of students, emphasizing analysis over design and formulae over judgment made a certain kind of sense. However, this emphasis ended up in large part hindering rather than enhancing effective practice. This realization has been slowly dawning on many communities of technological practitioners and the pendulum now appears to be swinging back in the other direction.

Misconceptions of the nature of engineering aside, though, computing and software appear fundamentally different from other areas of technological practice owing to their wide ranging applicability. Computers are general-purpose problem-solving devices and their wide

utility is a function of this. However, their utility in a specific context is due to the software which turns them into special-purpose problem-solving devices. Software can play this role because it is abstract and thus unusually malleable. With this abstractness, however, comes a complexity which challenges both the cognitive processes of the individual and the degree to which the software development process can be automated.

Because computer systems span a virtually limitless number of problem domains but must function within specific ones, fundamental problem-solving processes are of exceptional concern in computing and this is one reason for the seeming inadequacy of any one model of professional activity. Moreover, this irreducible tension between specificity and generality marks both software development techniques as well as software applications. Software technologists must find a balance between sophisticated and powerful context-dependent features usable in a narrow domain and less sophisticated and powerful features amenable to more general usage. This is one reason why a software "industrial revolution" seems quite unlikely, as it suggests the difficulty of producing high-level yet widely usable standard software components. Together with the irrelevance of manufacturing-based analogies for software and an appreciation of the history and role of standardized components in the Industrial Revolution, this indicates how any great faith in the ameliorative powers of standardized software parts is probably misplaced. Standards of practice, i.e. software process standards, are similarly affected by the tension between specificity and generality. Moreover, the cultural history of programming and the resulting emphasis on controlling practitioners have served to produce process standards based on problematic assumptions regarding the ability and/or willingness of practitioners to respond to variations in circumstance (including application domain) with variations in procedure.

As a result of this broad range of application areas, a key issue in the epistemology and practice of software development involves the mapping of archetypal or model problems and solutions onto actual problems across the wide expanse of problem domains addressed by computing. As a result, I am currently turning my attention toward exploring the nature of engineering epistemology by comparing software with other areas of technological practice. Because of my focus on the development, organization, and use of archetypal problems and solutions, I am particularly interested in psychological and philosophical issues surrounding analogies, categories, and professional judgment.

References

"Degrees of Freedom: The Interaction of Standards of Practice and Engineering Judgment," *Science, Technology & Human Values* (forthcoming).

"Splitting the Difference: The Historical Necessity of Synthesis in Software Engineering," *IEEE Annals of the History of Computing* 19 (1, 1997).

"Escaping the Mythology that Plagues Software Technology," in *The Responsible Software Engineer: Selected Readings in IT Professionalism,* Colin Meyers et al., eds. (London: Springer-Verlag, 1996), 249-257.

Stuart Shapiro and Steve Woolgar, "Understanding Software Development Standards in Commercial Settings: a project report," (Uxbridge: Brunel University, 1995), CRICT Discussion Paper No. 54

"Boundaries and Quandaries: Establishing a Professional Context for IT," *Information Technology & People* 7 (1, 1994): 48-68.

Research Abstract

Richard Sharpe

Richard Sharpe continues to work on the subject of software agents. At the Dagstuhl 1996 History of Software Engineering Conference he analysed the recent history of software agents since the visionary statement of Karl Hewitt [Viewing Control structures as patterns of passing messages – Artificial Intelligence 8(3) 1977]. He identified two strands of development, both extensively funded by US ARPA sources:

Strand one: The AI communities stemming from the Thinking Machines Conference 1956; continuing from Hewitt but weakened by the Donner Pass of AI in 1987 when funding cut;

Strand two: The network-driven approach where the Internet provides the environment for agents:

Software agents are produced by "heroic nerd", "tiger team" programming organisations in metropolitan locations with close teams of developers/testers/others.

Software agents are:

- autonomous-acting software objects; complete tasks for users with a greater degree of independence and scope than other software.
- a piece of software which will act for its client, human user or system, to perform tasks on behalf of client so that client does not have to.

Four benefits to users:

- should enhance personal productivity as agents filter, prompt and gather information for users;
- should extend reach of user or user(s) by gathering and collating information which user could not connect to at all or without considerable effort;
- should help user(s) analyse data and reveal patterns they did not know were within data; and
- could coordinate complex systems simply, without development of equally complex centralised scheduling and control software.

Initial military and private backing in AI programmes, than Internet interest revival, developed software agent technology to:

- a style of software – agent-oriented software;
- software product internally or externally developed; and
- a feature within other software products or projects, internally or externally developed.

Today many different names: intelligent brokers, actors, softbots, knowbots and userbots; 'bot' from robot.

Software agents in enterprises can:

- bring customers closer to suppliers of products and services;
- support enterprise's continued demand for change;
- inject further intelligence into enterprise; and
- simplify environment for both customers and employees.

Recent history shows:

- no single, stable theory of software agent behaviour;
- a 20-year history of expectations for role of agents; and
- advent of World Wide Web (WWW) provided latest boost to interest in agent technology.

Five main elements of software agents:

- initiator – human or system on whose authority agent acts;
- task – set by initiator;
- domain – area in which agent has to complete task;
- target system(s) – in a domain; and
- resources – employed by agent to complete task.

Software agents have attributes:

- task oriented, not process oriented;
- autonomous, not heavily supervised by initiator;
- tailorable, to needs of initiator;
- heuristic, learn about initiator, domain & strategies; and
- combinable, collaborate to complete task.

Eight categories of software agent:

1 Gatherer and/or distributor
2 Seeker and combiner
3 Trader and negotiator
4 Co-ordinator and scheduler
5 Analyser
6 Filter
7 Displayer
8 Prompter: wide or local

Five strands of IT in agents:

- development of GUI,
- World Wide Web,
- Object technologies,
- Digital libraries, and
- Distributed AI.

Unification movements:

- Language-based unification – search for a meta language;
- Project-based unification – pragmatic unification;
- Standards-based unification – de jure or de facto; and
- Package-based unification – packages achieve unity.

Three components:

Objects: special instance of objects.

Agent communications to:

- become “mobile code”;
- interact when remote;
- interact with other agents.

Five communications approaches:

- hand-build low level transport mechanism;
- inside a confined system;
- agent-specific communications subsystems;
- services of existing system software; and
- extend existing communications software.

Development languages and tools:

- pure agent-orientated languages;
- languages and environments developed for WWW applications;
- languages and environments from artificial intelligence; and
- languages and environments from Corba world.

Agent links to legacy software:

- WWW;
- transaction;
- database;
- GUI; and
- node on a communications network.

Agent standards:

- common agent platforms;
- inter-agent communications;
- internal agent standards between layers; and
- development languages and environments.
- Standardisation efforts:
- object design standards; and
- communications standards for underlying Internet communication links.

Business case: Friedman [Computer Systems Development Wiley 1989] fourth phase:

- replace or mask complex, centralised scheduling software;
- IT get away from stand-alone functions;
- “watch” actions of users and include heuristic qualities; and
- build on a spiral of productivity using heuristic software agents.

Technical level:

- implemented in layers;
- co-operative hierarchies to achieve appropriate level of supply to a varied demand;
- may get deadlocked; methods can be implemented;
- state-style communication force enriched client/server relationship;
- state-style communication fixes hard; and
- resolve agent conflict – negotiation or arbitration.

Push and pull

Promoting factors:

- Enterprise – what is agenda for enterprise?
  - Customer personalisation
  - Continued enterprise organisational changes
  - Continual extraction of value from operational data.
- Information – how can it be exploited?
  - overload in servers
  - overload at clients.
- Technology – what does its use entail and how can it be used?
  - spare client and server capacity can be exploited by new forms of software.
- Vendors – what agendas do vendors have?
  - vendor value-added search
  - vendor search for differentiation.

Retarding factors:

- strategic information identification – hard to select strategic information;
- co-operative relations strain;
- resistance to IT-based business in SME and SOHO;
- standards and protocols not fixed; and
- security – agent or virus?; trust?

Vendors have own promoting and retarding factors.

Enterprise objectives:

- becoming customer focused and driven;
- creating additional value;
- creating simple organisations and responses to complex events; and
- making change permanent and self-driven.

Suitable for:

- for users overburdened with information or reluctant to use IT because of fear of overburdening;
- for users who could only achieve more understanding from information by use of agent technology;
- applications where request-bid-select approach of coordination agents could mask complexity of control systems;

- for regular distribution of information, updates or software;
- to keep customers informed automatically of changes in schedules, services, products, prices or key fritters;
- for matching of fluctuating demand with various sources of supply; and
- for use in management of networking resources;
- for use on those applications which have high support costs.

Primary enterprise implementation ways:

- through developing software agents and/or software agent systems for its own use;
- through commissioning others to develop software agents and/or software agent systems;
- through purchasing software agents and/or software agent systems for its own use;
- through purchasing other software components with software agents embedded or implemented as features within them; and
- through adopting an agent-orientated software approach to all development.

Conform to:

- client/server,
- GUI-centric,
- DBMS-centric,
- intermediate-representation-centric,
- Blackboard, and
- subsumption: made up of smaller, simple parts.

Avoid mismatches.

Future

Pragmatically absorbed by enterprises in IT agenda.

Improve productivity of individual users; useful tool for analysis.

Few strategic adoptions of agent-oriented software.

Future technical problems:

- build agent communities remotely at remote sites;
- name agents across networks;
- locate active agents in world-wide networks with, potentially, billions of active agents; and
- interrogate agents about their resources, tasks and routes to target systems so as to tell “real” agent from potential virus.

Three Patterns that help explain the development of Software Engineering

Mary Shaw

The term “software engineering” came to prominence when it was used as the name of a NATO workshop in 1968 [NaRan69]. It was used then to draw attention to software development problems. It was then, as to a large extent it remains now, a phrase of aspiration, not of description.

In the intervening years, the focus of the academic community (though not so much the industrial software development community) has shifted from simply writing programs to analyzing and reasoning about large distributed systems of software and data the come from diverse sources. Figure 1 lays out the highlights of these shifts.

	1960 ± 5 Programming-any-which-way	*1970 ± 5 Programming-in-the-small*	*1980 ± 5 Programming-in-the-large*	*1990 ± 5 Programming-in-the-world*
Specifications	Mnemonics, precise use of prose	Simple input-output specifications	Systems with complex specifications	Distributed systems with open-ended, evolving specs
Design Emphasis	Emphasis on small programs	Emphasis on algorithms	Emphasis on system structure, management	Emphasis on subsystem interactions
Data	Representing structure, symbolic information	Data structures and types	Long-lived databases	Data & computation independently created, come and go
Control	Elementary understanding of control flow	Programs execute once and terminate	Program systems execute continually	Suites of independent processes cooperate

Figure 1: Highlights of academic attention in software engineering

I see three simple patterns that have guided this development. Each of these provides partial explanations, but none is either comprehensive enough or rich enough to be in and of itself a full model.

(1) Evolution of engineering disciplines.

Technologies evolve from craft through commercial practice before they integrate scientific knowledge and become true engineering disciplines. Figure 2 illustrates this pattern. Software engineering has been following this pattern; it helps to explain the role of software process improvement. [Shaw90]

Exploitation of a technology begins with craftsmanship: practical problems are solved by talented amateurs and virtuosos, but no distinct professional group is dedicated to problems of this kind. Intuition and brute force are the primary problem-solving strategies. Progress is haphazard, and the transmission of knowledge is casual. Extravagant use of materials may be tolerated, and manufacture is often for personal or local use.

At some point, the products of the technology gain commercial significance, and economies of manufacture become an issue. At this point, the resources required for systematic commercial manufacture are defines, and the expertise to organize exploitation of the technology is introduced. Capital is needed to acquire raw materials or invest in manufacture long before sale, so financial skills become important. Scale increases over time, and skilled practitioners are needed for continuity and consistency. Pragmatically-derived procedures are replicated carefully without necessarily having knowledge of why they work. Management and economic strategies may assume as large a role as the development of the technology. Nevertheless, problems with the technology often stimulate the development of an associated science.

When the associated science is mature enough to yield operational results – that is, results that are cast in the form of solutions to practical problems, not as abstract theories – an engineering discipline can emerge. This allows technological development to pass limits previously imposed by relying on intuition; progress frequently becomes dependent on science as a forcing function.

Software engineering is in the process of moving from the craft to the commercial stage. It has only achieved the stature of a mature engineering discipline in isolated cases.

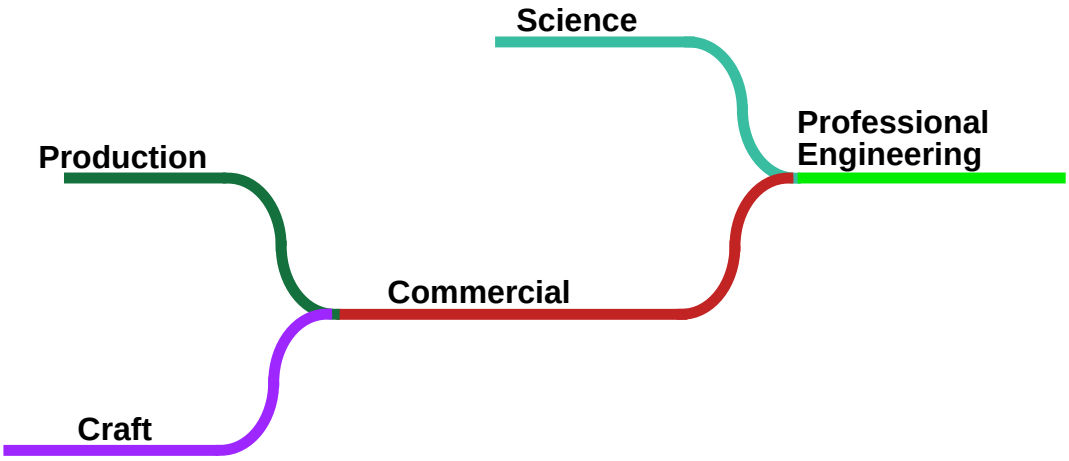

Figure 2: Evolution of engineering disciplines (after [Finch51])

(2) Abstraction and its coupling to specifications

The granularity of our abstractions – the intellectual size of a chunk we treat as atomic – increases over time. Abstractions are supported by formal specifications, but formal specifications will be used in practice only to the extent that they provide clear payoff in the near term. [Shaw80]

This pattern can be seen in the development of data types and type theory. In the early 1960's, type declarations were added to programming languages. Initially they were little more than comments to remind the programmer of the underlying machine representation. As compilers became able to perform syntactic validity checks the type declarations became more meaningful, but "specification" meant little more than "procedure header" until late in the decade. The early 1970s brought early work on abstract data types and the associated observa-

tion that their checkable redundancy provided a methodological advantage because they gave early warning of problems. At this time the purpose of types in programming languages was to enable a compile-time check that ensured that the actual parameters presented to a procedure at runtime would be acceptable. Through the 1980s type systems became richer, stimulated by the introduction of inheritance mechanisms. At the same time, theoretical computer scientists began developing rich theories to fully explain types. Now we see partial fusion of types-in-languages and types-as-theory in functional languages with type inference. We see in this history that theoretical elaboration relied on extensive experience with the phenomena, while at the same time practicing programmers are willing to write down specifications only to the extent that they are rewarded with analysis than simplifies their overall task.

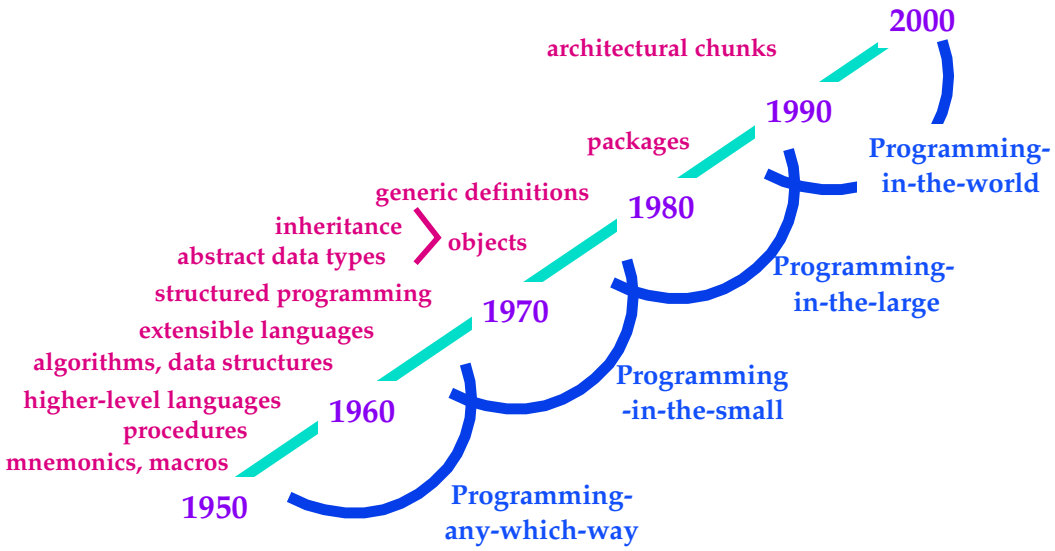

Figure 3: Language constructs and phases of software engineering development

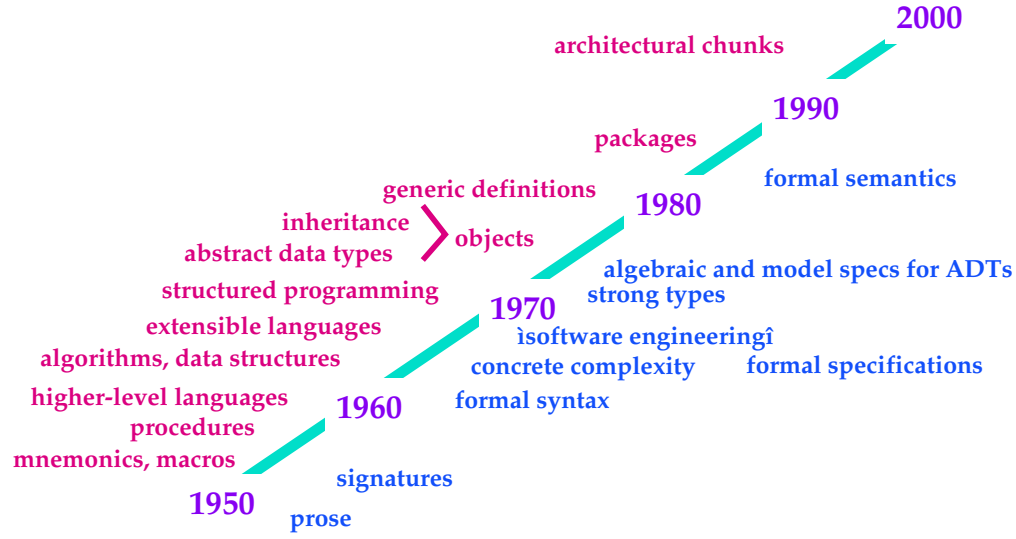

Figure 4: Coupled development of abstraction and specification

(3) Progressive codification

Specification techniques evolve in parallel with our understanding of the phenomena they specify. [ShGar95] We begin by solving problems any way we can manage. After some time we discover in the ad hoc solutions some things that usually work well. Those enter our folklore; as they become more systematic we codify them as heuristics and rules of procedure. Eventually the codification becomes crisp enough to support models and theories. These help to improve practice; they also allow us to address new problems that were previously unthinkable.

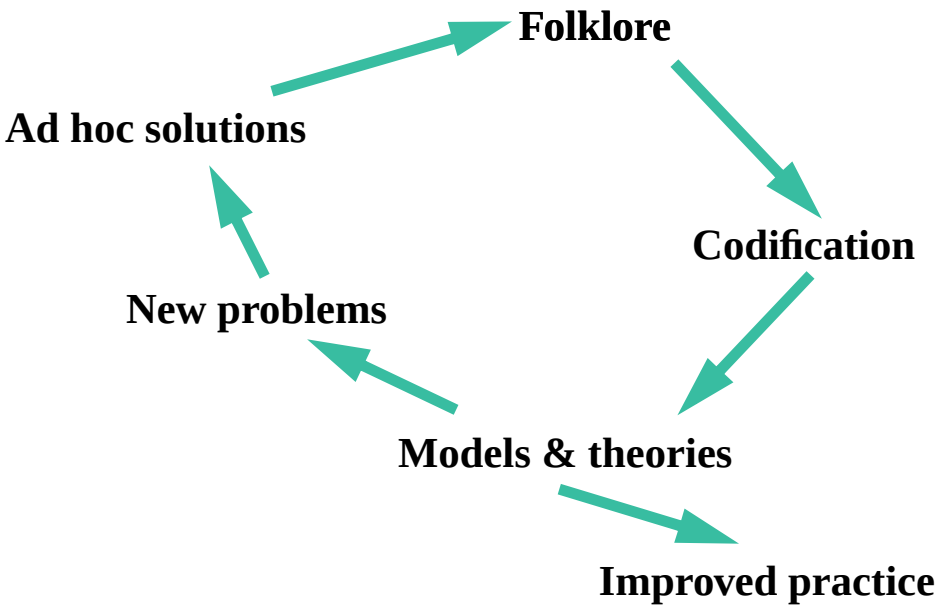

Figure 5: Cycle of progressive codification

Thus, as some aspect of software development comes to be better understood, more powerful specification mechanisms become available, and they yield better rewards for the specification effort invested. We can characterize some of the levels of specification power:

> Ad hoc: implement any way you can
> Capture: simply retain and explain a definition
> Construction: explain how to build an instance from parts
> Composition: explain how to compose parts and their specifications
> Selection: guide designer's choices for design or implementation
> Verification: determine whether an implementation matches specification
> Analysis: determine the implications of the specification
> Automation: construct an instance from an external specification

When describing, selecting, or designing a specification mechanism, either formal or informal, it is useful to be explicit about which level it supports. Failure to do so leads to mismatches between user expectations and specification power.

Brooks proposes recognizing three kinds of results, together with criteria for judging the quality of those results [Brooks88]:

findings	well-established scientific truths	truthfulness and rigor
observations	reports on actual phenomena	interestingness
rules-of-thumb	generalizations, signed by an author but perhaps not fully supported by data	usefulness
all three		freshness

Bibliography

[Brooks88] Frederick P. Brooks, Jr. Grasping Reality Through Illusion – Interactive Graphics Serving Science. *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference,* May 1988, pp. 1-11.

[Finch51] James Kip Finch. Engineering and Western Civilization. McGraw-Hill 1951.

[NaRan69] Peter Naur and Brian Randell (eds). Software Engineering: report on a conference sponsored by the NATO Science Committee, Garmisch Germany 1968. NATO 1969.

[Shaw80] Mary Shaw. The Impact of Abstraction Concerns on Modern Programming Languages. *Proc IEEE*, September 1980, also *IEEE Software* Oct 1984.

[Shaw90] Mary Shaw. Prospects for an Engineering Discipline of Software. *IEEE Software*, November 1990.

[ShGar95] Mary Shaw and David Garlan. Formulations and Formalisms in Software Architecture. Computer Science Today (LNCS 1000), Jan van Leeuwen (ed), Springer-Verlag 1995.

Paradigms and the Maturity of Engineering Practice: Lessons for Software Engineering

James E. Tomayko

There is a reluctance to include software engineering among the engineering disciplines. This is because many computer scientists and others see engineering in a binary model: fully formed, or non-existent. The truth is that different fields of engineering represent a wide range of maturity levels. One way of defining the maturity of a particular form of engineering is to study its paradigms (in the Kuhnian sense of commonly-held practices, plus process attributes). At first glance, software engineering seems a riotous collection of idiosyncratic techniques. In reality, it has a few paradigms, and these are no fewer than other immature engineering disciplines at similar stages of development. Yet, the other disciplines did not go through nearly the same level of wrenching internal debate and open derision by critics.

Paradigms serve as the basis for practice. They clearly represent successful solutions, and are evidence of practice maturity. As an example, consider the fundamental paradigm of heavier-than-air flight: a fuselage with wings and cruciform tail, moving rapidly through the air. Once George Cayley defined such a design, other aeroplane designers adopted it en masse. By 1914, subtle variations such as tractor biplanes, pusher biplanes, and tractor monoplanes were in the aeronautical engineering toolkit, yet the basic paradigm remained to guide design decision-making.

Many are confused by the fact that software engineering does not shape physical artifacts (it is most commonly a component of a larger system), but that does not impede the development of paradigms. The stored program concept, subroutines, information hiding, and evolutionary development lifecycles are all conceptual paradigms that serve to guide software engineering practice. When software engineering education is based on recognition and use of paradigms, stronger engineers will result.