Abstract:

The paper attempts to portray the 1968 software scene, by recalling the principle technical issues and concerns of the time. These are discussed under the headings Software as a Commodity, Programming Languages, Multiprogramming and Time-Sharing, Modularity and Structuring, and the Problems of Large Systems. The paper ends with an account of the major debates at the first conference ever held on the subject of "software engineering", the NATO Conference that took place in Garmisch in October, 1968.

Software Engineering in 1968

By

B. Randell

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

© 1979 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
bibliographical details

RANDELL, Brian.

Software Engineering in 1968.
By B. Randell.

Newcastle upon Tyne:
University of Newcastle upon Tyne,
Computing Laboratory, 1979.

(University of Newcastle upon Tyne, Computing Laboratory,
Technical Report Series, no. 138.)

Added entries
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)
Library of Congress:
Dewey (17th) 681.3.22.06 (091)
U. D. C. 001.64209

Suggested keywords
NATO CONFERENCE
SOFTWARE ENGINEERING

Abstract
The paper attempts to portray the 1968 software scene, by recalling the principle technical issues and concerns of the time. These are discussed under the headings Software as a Commodity, Programming Languages, Multiprogramming and Time-Sharing, Modularity and Structuring, and the Problems of Large Systems. The paper ends with an account of the major debates at the first conference ever held on the subject of "software engineering", the NATO Conference that took place in Carmisch in October, 1968.

About the author
Professor Randell is Professor of Computing Science in the Computing Laboratory of the University of Newcastle upon Tyne, a post he has held since 1969.
The future is dark, the present burdensome. Only the past, dead and finished, bears contemplation. Those who look upon it have survived it; they are its product and its victors. No wonder therefore that men concern themselves with history.

Elton (19)

1. **INTRODUCTION**

My aim in this paper is to provide a series of backward glances at the software scene as it was in 1968, rather than attempt a scholarly historical analysis; I prefer to reserve my faltering attempts at emulating the historian's craft for that perhaps more simple and surely more heroic earlier computing era, personified for me by Babbage, Turing and von Neumann. However I have in preparation for writing this paper supplemented my memories of 1968, and of the NATO Software Engineering Conference that year in Garmisch, by scanning the 1968 computing literature. This task has been at once wearisome and fascinating, dispiriting and, occasionally, uplifting - I leave you to guess which of these reactions were aroused by my finding a 1968 advertisement for a mechanical hexadecimal calculator.

It was of course the Garmisch conference that started the software engineering bandwagon rolling. One of the most notable aspects of the conference was the willingness of the participants, "about 50 experts from all areas concerned with software problems - computer manufacturers, universities, software houses, computer users, etc.", to admit the extent and gravity of those software problems. It is necessary to search the 1968 literature fairly carefully to find any comparable comments. Terms such as "software crisis" and "software failure", which were freely bandied about at Garmisch, are nowhere to be seen in the general literature. Some worries are occasionally expressed, but one can also find such statements as:
"There is a seductive fascination in software. There is a pride of accomplishment in getting a program to run on a digital computer, a feeling of the mastery of man over machine. And there is a wealth of human pleasure in such mental exercises as the manipulation of symbols, the invention of new languages for new fields of problem solving, the derivation of new algorithms and techniques, the allocation of memory between core and disk - mental exercises that bear a strong resemblance to such popular pastimes as fitting together the pieces of a jigsaw puzzle, filling out a crossword puzzle, or solving a mathematical brainteaser of widely familiar variety. And what is more, digital computer programming is an individual skill, and one which is relatively easy to learn" (37).

This attitude could hardly be in greater contrast with Dijkstra's statement (33,p.121) at Garmisch that "The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement." No wonder then that many of the participants view the 1968 Software Engineering Conference as a turning point, an event which had a significant effect on their attitudes and their subsequent work in the software field.

The conference originated from discussions in the Study Group on Computer Science, set up in 1967 by the NATO Science Committee. The conference title "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". This sentiment can be traced back earlier; the phrase, and the implied need, were it is claimed (23) discussed by Eckert at the 1965 Fall Joint Computer Conference, but the term 'software engineering' still had considerable novelty at the time of the Garmisch conference.
Needless to say, since 1968 "the term has become very fashionable, especially amongst those who believe that by inventing a new term a problem is thereby solved". (This remark was originally made by Barron (2) about 'structured programming', but is surely equally appropriate to software engineering.) Such matters are, however, to be left to the authors of the other invited papers. It is for them to assess whether and to what extent the present and the future are as described by Elton - here the only requirement is to contemplate the past.

This I will now attempt to do. My remarks - my backward glances - are given under a number of headings, chosen to reflect what were, as far as I can remember and can ascertain, the major preoccupations of the 1968 software scene.
2. SOFTWARE AS A COMMODITY

Although one writer could remark (25) that it was only two or three years earlier that the word "software" began to have meaning in normal business parlance, by 1968 there was general recognition that software was an important commodity in its own right. For example, one survey (48) listed nearly five hundred U.S. organisations concerned with selling and/or producing software. Moreover, large users were already complaining that their software costs were rivalling, or even exceeding, the sums that they were spending on hardware (25).

Much systems software still came "free" with the hardware from the computer manufacturer, but the possibility that IBM would "unbundle" its software by charging separately for it was causing great concern and confusion. By far the most reasoned contribution to the debate that I could find in the 1968 computer literature is an article by Conway in the October issue of Datamation (11). This article sought to dispel a number of then current confusions about the economics of software production and marketing, and contained some interesting statistics about System/360 hardware and software costs. It estimated that the software, whose development cost was several hundred million dollars, was contributing only about 3% to the price of a S/360 computer because of the near-zero replication cost.

At the Garmisch conference the question of software unbundling was such a hot issue that "a special session on the issue of software pricing was arranged in response to the generally expressed feeling of the importance of this topic in relation to the whole future of software engineering" (33,p.129). Indeed the whole topic was regarded as so sensitive that this was the one session for which the conference report merely records the various arguments put forward, without attributing them to individual speakers. Quoting from the report again:
"During the session it became clear that one of the major causes of divergent views on whether software should be priced separately from hardware was the fact that people had differing aims, and also differing estimates of the possible effects of separate pricing... The discussion lasted over three hours, ending after midnight, yet was well-attended throughout. At the end of the discussion the opinion of those present was tested. It was clear that a large majority were personally in favour of separate pricing of software."

Another aspect of the issue of software as a commodity concerned the protection of the proprietary rights of the owner - a matter of increasing importance given the rising sales of software packages, such as Informatics Mark IV (34). Hence one event that received considerable publicity in 1968 was the granting of the first US patent for a computer program, a sorting routine invented by Martin Goetz (50). In contrast, another news item revealed a UK scandal concerning the plagiarism of sections of BOAC's $100 million international airline reservation system (49).

This simplistic view of software as a readily priceable and saleable commodity did not however fit well with the great variations in productivity that could exist between programmers - the extent of these variations was, for most of us, first brought home by the SDC study published early in 1968 (38). The study had been aimed at evaluating the effectiveness of interactive program development, but it revealed differences between the worst and best programmers in the group that was monitored of, for example, 25:1 in coding time, 26:1 in debugging time, and 13:1 in execution speed of the resulting programs. These figures, I remember, rapidly achieved quite a degree of notoriety, even before the Garmisch conference.
3. **PROGRAMMING LANGUAGES**

By 1968 there were already a vast number of high level programming languages in existence. Jean Sammet's compendium (39), published in 1969, lists well over a hundred, although only about fifteen were in widespread use. FORTRAN apart, high level languages were in general used only on mainframe computers - a 1968 survey paper (27) states that "as of now, only one minicomputer manufacturer is known to have provided any language other than a FORTRAN dialect for a minicomputer. Hewlett-Packard has made both ALGOL and COBOL available".

Much controversy surrounded PL/I, since at least within IBM there were still hopes that it would be, so to speak, the language to end all languages, from high level languages to assembly languages. Its monumental formal definition had been produced at the IBM Vienna Laboratory (see Bandat (1)), and the first compiler, the PL/I F-level compiler had been released as far back as August 1966. (Tony Hoare was later to characterise this compiler as a superb attempt at coping with the complexities of an imperfect language, rather than, as was sometimes claimed, a sadly inadequate implementation of a marvellous new language.) However, PL/I was only making slow progress out in the field. It was reckoned at the end of 1968 that just one per cent of all US installations used PL/I (52). In commercial installations COBOL, whose first implemented version dated from 1960, reigned supreme, and in fact the language had just achieved the status of a US official standard (51). (FORTRAN standards date from 1966.) Nevertheless several articles (e.g. 46,52) attest to the worries of the COBOL community that their language would be slowly sidetracked by PL/I, which might become a de facto standard through IBM's domination of the industry. Interestingly enough, judging by the literature, FORTRAN users do not seem to have been bothered by such a possibility.
In the more rarified ALGOL world in which I moved, 1968 was something of a watershed. The IFIP ALGOL Committee (Working Group 2.1) which had been set up following the publication of the original ALGOL 60 Report, met in Munich not long after the Software Engineering Conference. A week-long debate of remarkable intensity culminated in a majority decision to approve the ALGOL 68 report that had been prepared by van Wijngaarden, Mailloux, Peck and Koster (41). In response, a renegade group (of which I was a member) produced a brief Minority Report (18), in which we stated that "it will be required from an adequate programming tool that it assists, by structure, the programmer in the most difficult aspects of his job, viz. in the reliable creation of sophisticated programs. In this respect we fail to see how the language proposed here is a significant step forward". My own recollection in fact is that several of us felt even then that the recently proposed SIMULA 67 Common Base Language (12) came closer to our ideal, and would perhaps turn out to have at least as much impact as an officially promulgated ALGOL 68.

Meanwhile what I might be permitted to term the Wirth Language Factory was in somewhat of a trough, measured that is by height of language. His earlier work on EULER and ALGOL W was followed in 1968 by PL/360 (42) - an ALGOL-like assembly language for the IBM S/360 - PASCAL and MODULA were of course yet to come. In contrast, within IBM at Yorktown Heights a distinctly high level language was gaining a large number of adherents. This of course was APL - the APL/360 system had first become available for use there late in 1966 and, I remember, by 1968 was in widespread use.

The one other controversy that I recall from those days was the issue of whether it was feasible to use a high level language for systems programming. This was a practice that at the time was sometimes preached but rarely observed - indeed PL/360 was a response to this situation, being put forward as a compromise solution to the dilemma. However, some significant inroads into the assembler
dominance of systems programming were becoming known, such as the use of EPL (a subset of PL/I) by the MULTICS project (33,p.56). Nevertheless the issue was to be quite a point of debate at the Garmisch conference.
4. MULTIPROGRAMMING AND TIME-SHARING

In May 1968, the Communications of the ACM published the proceedings of another meeting whose title could also perhaps be regarded as evidence of wishful thinking on the part of the organizers, namely the first Symposium on Operating System Principles. Held in Gatlinburg, Tennessee, I remember that it was, despite the restrictive local ordinances concerning the consumption of alcohol, an enjoyable and successful affair. The most noteworthy presentation was that by Dijkstra on 'The Structure of the 'THE'-Multiprogramming System' (16), a paper which introduced an entirely novel concept, that of an elegant bug-free operating system. Needless to say the paper was well received at the symposium, where its author was awarded a special prize consisting of two toy grizzly bears, locked in a "deadly embrace".

Multiprogramming and time-sharing featured large in the 1968 literature, in which numerous articles extolling the capabilities, or bemoaning the inadequacies, of various systems are to be found. Interestingly, one such article (24) in fact states that "remote batch has probably been the most significant capability developed commercially within the last year. This capability will drastically alter both the time-sharing and batch-processing industries within the next couple of years." A good account of the contemporary American scene is to be found in an article by Jules Schwartz, who states (40):

"It is interesting to note that in the past few years the number of systems labeled "time-sharing" has increased from around five experimental systems to about 30 different commercial systems operating in 70 installations (as of early 1968), and probably several hundred experimental or research systems... Most commercial services now are effectively one-language systems: GE's BASIC, Allen-Babcock's version of PL/I, BBN's TELCOMP
(JOSS), and IBM's QUIKTRAN (FORTRAN) are examples of such systems. Other systems stress one kind of application - for example, KEYDATA for business applications, and the IBM Text Data Service ... Systems like MULTICS and IBM's TSS, which were presumed by many to represent the next major step in large-scale interactive computing, still haven't fulfilled their promise. At this writing, MULTICS is just beginning to work in a minimal way, and the IBM system is being used in scattered installations with considerably fewer simultaneous users and services than one might have assumed for this time."

In the opening sentence of the major paper (31) describing the system, two of TSS/360's designers put it this way: "Experience with TSS/360 design, development and application has been varied and interesting" - my personal recollections of the situation regarding TSS are such as to make me treasure this example of American understatement. Incidentally, one of the few large-scale general purpose time-sharing systems that Schwartz mentions as being fully operative already is the one we still use happily at Newcastle, namely the Michigan Terminal System, which by the end of the year was supporting 58 terminals at the University of Michigan, on a two processor configuration (22). Meanwhile the European time-sharing scene, at least as surveyed by Wright (43), consisted mainly of a small number of time-sharing bureaux operating American systems, with just a few home-grown time-sharing projects in progress, based in Universities and research laboratories.

The complexity, mainly unmastered, that was typical of the latest multiprogramming and time-sharing systems, and their evident performance difficulties, were causing system performance monitoring, modelling and analysis to be quite a growth industry (e.g. 7,8,35). Much of this work concentrated on storage management issues, particularly in virtual storage systems based on the use of demand paging - a problem that I and my colleagues used to describe as that
of how to get real performance from a virtual system. Gradually though the importance of what was termed (21) the dynamic behaviour of programs, i.e. of storage referencing patterns, and the existence of the phenomenon of "thrashing" (14,29) were becoming understood, and strategies such as Denning’s working set scheme (13) were being proposed. However, all too often in actual large-scale operating systems, over-complicated strategies interacting in completely unanticipated ways, in the presence of significant "performance bugs" (8), were the rule. This led Datamation to publish a memorable anonymous parody (45) entitled "The End of OS":

"The end finally came in mid-October (1984). System Release 110.7 was distributed ... System integration was accomplished with little difficulty in no more than 504 system hours. Expectantly, customers IPLed and initiated their job streams. And nothing happened. Nothing. When it slowly dawned on everyone that nothing was going to happen, now or later, a flood of anguished telephone calls swamped the branch offices ... At last a brave man, a customer engineer, fought his way through the crowd around his system and obtained a dump. As he scanned the hex, the horrible truth came home to him. All of core, as far as the eyes could see, was filled with control blocks, each containing pointers to other control blocks. ... But no programs were being executed. No data was being read or written or processed. Operating System had taken over all the system resources and was entirely occupied with issuing supervisor calls,saving registers, restoring registers, chaining forwards and backwards and following pointers all over core. Every pointer led to some other pointer. Operating System, after several years of effort by thousands of programmers, had finally become a completely closed system."

This scenario and the 'THE' system, between them, provide a sort of survey, or at least a delimitation, of the operating system scene
which at any rate fits my own recollections of that era. This was where many of us considered most of the action to be, so to speak. Data bases, networks and distributed systems were yet to come, though work had started on implementing ARPANET. Operating systems provided the arena that seemed to attract most system programmers and computer scientists. However, looking back, one must say that though some were attracted by the challenge of reducing complexity to order, some others perhaps found the complexity itself the attraction.
5. MODULARITY AND STRUCTURING

It is perhaps the benefit of hindsight that allows us to go back to the 1968 literature and notice signs of growing interest in developing techniques and practices of system structuring. Modular programming was in vogue, based on the use of facilities for independent compilation of intercommunicating sub-programs, something which had been possible for several years in FORTRAN, but had only recently been provided in COBOL. Its proponents clearly were motivated by a belief in the need to "divide and conquer" system complexity:

"The availability of inter-program communication allows the user a greater degree of freedom in how he structures his systems. In general, large problems are most easily solved by factoring them repeatedly into smaller, more logically independent parts until the solution of each part is either available (i.e. the problem has been solved before and the result recorded) or is clear. The set of solutions thus developed forms the solution of the large problem. This particular problem-solving process is greatly facilitated by inter-program communication." (26)

Similar ideas, but much further developed, can be found in a paper by Constantine (9), in which he writes of a structural theory of programs:

"A program is an ordered set of statements and aggregates of statements defining, describing, or directing the performance of some task. A program is a system and is thus defined by its inputs, outputs, transformation function, its boundary, components, and their interrelationships. The aggregates of statements are called modules, and a module is a program ... In the sense of containing phases, segments, procedures, blocks, subroutines, macros or other externally defined aggregations,
all programs are modular. The physical structure defined by such aggregates can be considered to be superimposed on the inherent structure of the statements ... Numerous factors are influenced by the aggregation (or in practice, segmentation) scheme. The separability of the programming task and the success of such separation will depend on the number and kind of intermodular interconnections. Ease of maintenance is largely a function of limiting the scope of changes. Structures with minimal intermodular connections and maximal functional cohesiveness of module content will tend to minimise the cost of correction and changes."

Constantine also discusses what he terms a "process theory of programming", describing the normal phases of the programming task and claiming that, taken with his structural theory of programs, this provides a formal procedure for the design of modular programs which tends to produce "near optimal", highly decoupled systems. Moreover, he states that such ideas are sometimes explicitly applied, and frequently implicitly used, in industry, though very rarely taught in universities.

Certainly one can find little mention of these ideas elsewhere in the academic computing literature in 1968 though the 'THE' paper (16), in its very different way, was in part putting over the same message. Indeed today it comes as something of a shock to find that even the term "structured programming" had yet to be invented and turned into a catch phrase - in fact Dijkstra's 'Notes on Structured Programming' (17) were written in August 1969. However his now famous letter 'Go To Considered Harmful', and the first reactions to it, had been published (15,36). Otherwise though, documented discussions, in terms of programming language issues, of system structuring do not seem to have existed except, of course, in the SIMULA Common Base Language Report (12).
The discussion in the 'THE' paper was in terms of "levels of abstractions" though these were not so much a linguistic form as a set of conventions as to what aspects of the system state and activities (e.g. drum store transfers, and processor multiplexing) were deemed to be hidden from outside view. Nevertheless they enabled Dijkstra and his colleagues so to structure the system as to simplify its strategies and their description to the point that a semi-formal proof of the correctness of the system could be provided.

The notion of "level of abstraction" had also arisen in the work in which I was involved at the time, on what Frank Zurcher and I termed "iterative multi-level modelling" (44). The idea was to reduce the long delays which normally arose between the taking of a design decision in a complex computing system project and the provision of any significant feedback to the designer from attempted usage of the implemented system. The technique we proposed and investigated was to represent the evolving design as an evolving simulation program which could be experimented with as needed throughout the design process. The requirement to make continual modifications and extensions to a simulator led us to the need for improved methods of explicit program structuring, and to the notion of level of abstraction. The concept of having a simulation gradually evolve into the intended actual system is still, I regret to say, one whose time has yet to come.

Last but by no means least on the subject of modularity and structuring, circa 1968, there is the splendid article (10) entitled 'How do Committees Invent' in which what I have since known as Conway's Law is to be found. This is that organizations which design systems are constrained to produce designs which are copies of the communication structures of the organizations. The article is well-reasoned and fundamentally serious but I cannot resist ending this section by repeating one of my favourite quotes from it:
"A contract research organisation had eight people who were to produce a COBOL and an ALGOL compiler. After some initial estimates of difficulty and time, five people were assigned to the COBOL job and three to the ALGOL job. The resulting COBOL compiler ran in five phases, the ALGOL compiler ran in three.

Two military services were directed by their Commander-in-Chief to develop a common weapon system to meet their respective needs. After great effort they produced a copy of their organisation chart."
6. THE PROBLEMS OF LARGE SYSTEMS

It was the US military-industrial complex that first started to try and develop very large software systems involving man-millenia of effort. The secrecy that shrouded their purposes served also to hide the extent to which such projects were often characterised by "underestimates and overexpectations". This phrase was in fact the title of a paper (32) written in 1969 by Licklider as a contribution to the public debate about the Anti-Ballistic Missile System, a project which was clearly going to require immensely sophisticated software. This was despite the fact that, according to Licklider: "At one time, at least two or three dozen complex electronic systems for command, control and/or intelligence operations were being planned or developed by the military. Most were never completed. None was completed on time or within the budget." (Or, I would assume, to adequate reliability levels!)

I still remember the ABM debate vividly, and my horror and incredulity that some computer people really believed that one could depend on massively complex hardware and software systems to detonate one or more H-bombs at exactly the right time and place over New York City to destroy just the incoming missiles, rather than the city and its inhabitants.

By the late 1960's systems of similar complexity were not unique to the military scene. Large scale manufacturer-supplied operating systems, and specialised real-time systems such as airline reservations systems, had given a much wider computing audience comparable experiences and problems. The latest operating systems were immensely more complicated than their predecessors - release 16 of OS/360, issued in July 1968, contained nearly 4,000 modules, comprising almost one million instructions (3,30); the system was estimated by one speaker at the Garmisch conference to have already absorbed 5,000 man-years of effort. Another contemporary statistic -
its FORTRAN H compiler alone was stated to contain 2,000 distinct faults (47).

However, experience had shown that in at least some cases, given enough effort, even systems of such complexity could be made to work tolerably well, and by 1968 many OS/360 installations seemed to be coping reasonably satisfactorily. Experience with the first large scale airline reservation system, the American Airlines SABRE system, had followed the same pattern. When it was first introduced in 1965 it had caused chaos at the airports, and lots of extra business for competing airlines (32). But by 1968 the SABRE system, by now containing 300,000 instructions, was serving about 2,000 terminals, and processing close to 3,000 messages per minute at peak load times (28). It and other systems like it were providing a facility that many depended on, and took completely for granted.

However, the costs incurred in achieving these successes were immense. Moreover there were systems, very much in the computer public's eye, whose performance and reliability problems showed little sign of succumbing to the great weight of effort that was being lavished on them - TSS/360 was a prime example. Were such systems literally beyond the capabilities of the time? Was it really feasible and necessary to involve such large numbers of designers and implementors in a single project, and then to have to face such horrendous management problems? Would improved management techniques save the day, or was the problem primarily a technical one. These were the sort of questions that had to be faced - the Garmisch conference provided a badly needed opportunity for us to do so.
7. THE GARMISCH CONFERENCE

Such then was the background to the Garmisch conference, at least as far as I can recollect or reconstruct it. But what of the conference itself? Its success was I believe due mainly to the choice of participants, though its location and its timing (which enabled attendees to experience the Munich Oktoberfest en route to Garmisch) must also have helped. In line with the intentions of the organising committee the participants, though strictly limited in number, represented a wide variety of backgrounds. Nearly half came from North America, the rest from various European countries. Many were meeting each other for the first time. Yet almost from the start a tremendous rapport grew up, as the delegates found common cause despite their disparate experience, and expressed common concern at the seriousness of some of the problems of the software world.

The conference had been planned not as a conventional series of formal sessions at which papers were presented and discussed, but rather as effectively a group debate, aimed at the production of an agreed report surveying the problems and practices of "software engineering". The participants were divided into three groups corresponding to the topics of Design, Production and Service. In parallel working sessions punctuated by occasional plenary sessions they attempted to summarise the present situation in their field, and to illustrate this from their own experiences. It is difficult if not impossible to convey to anyone who was not there the sense of excitement and even missionary zeal that all this activity engendered. Thus there was a general concensus that the topics discussed and some of the experiences described and views expressed were of great importance, not just to would-be "software engineers", but also, to quote the report, to "those who have no special interest in computers and their software as such, but who are concerned with the impact of these tools on other parts of society".
The main body of the conference report was produced from the draft outline reports produced by the three groups, and the tape-recordings of discussions, together with the many working papers brought to the conference, or prepared during it aided by the burning of much midnight oil. The editing was done during a week of frantic effort immediately after the conference by a small group led by Peter Naur and myself. In an attempt to avoid undue editorial bias, we evolved an editorial style which relied heavily on the use of direct quotations to flesh out the structure that had been agreed by the meeting. The resulting report, which was widely circulated by NATO, and then later republished in book form (6) was, I am proud to say, greeted by Doug McIlroy, one of the participants, as "a triumph of mis-applied quotation".

So much for the mechanics of the conference - now let me try to describe some of the discussions and personalities which made the greatest impression on me personally. First and foremost were the discussions on what many were prepared to term the "software crisis", though some felt this overly dramatic. One of the major contributions to these discussions was provided by the position paper produced by Ed David of Bell Labs. and Sandy Fraser of Cambridge University. Quoting this paper:

"There is a widening gap between ambitions and achievements in software engineering. This gap appears in several dimensions: between promises to users and performance achieved by software, between what seems to be ultimately possible and what is achievable now and between estimates of software costs and expenditures. The gap is arising at a time when the consequences of a software failure in all its aspects are becoming increasingly serious. Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people and ultimately for nations as well."
One of the many participants obviously in full agreement with such worries was Andy Kinslow, a recent refugee from the TSS/360 project, whom I remember as still almost visibly suffering from the experience. As he put it:

"In my view both OS/360 and TSS/360 were straight-through, start-to-finish, no-test-development, revolutions. I have never seen an engineer build a bridge of unprecedented span, with brand new materials, for a kind of traffic never seen before - but that's exactly what has happened on OS/360 and TSS/360."

Another contributor to this debate whom I can still visualise quite clearly was Bob McClure, who had earlier produced some quite startling statistics as to the amount of code provided as standard programming support for a variety of computers by their manufacturers, and the rate at which this amount was growing. In his view a major cause of present problems was "the refusal of the industry to re-engineer last year's model". This point he illustrated well with further statistics, showing how a particular group's speed of implementation and ability to estimate completion date improved dramatically over a sequence of FORTRAN compiler projects. Andy Kinslow struck a chord though with the heartfelt cry:

"Personally, after 18 years in the business I would like just once, just once, to be able to do the same thing again. Just once to try an evolutionary step instead of a confounded revolutionary one."

To be fair though, it was certainly not just experience with large IBM software projects that fuelled the debate - other systems which figured large in the conference discussions included the Bell Labs. Electronic Switching System, MULTICS, the Allen-Babcock Time-Sharing System, and the NEBULA Compiler for the I.C.T. Orion. (Incidentally, I recollect someone once, rather unkindly, suggesting
that MULTICS stood for Many Unnecessarily Large Tables in Core Store.) The point was of course made that many of the less demanding uses of computers were succeeding perfectly adequately. Nevertheless, to my mind the mood of the majority was best caught by the late Stan Gill, who also expressed the need many of us felt to communicate our concerns beyond the normal confines of the software community:

"Software is as vital as hardware, and in many cases much more complex, but it is much less well understood. It is a new branch of engineering, in which research, development and production are not clearly distinguished, and its vital role is often overlooked. There have been many notable successes, but recent advances in hardware, together with economic pressures to meet urgent demands have sometimes resulted in this young and immature technology of software being stretched beyond its present limit ... It is of the utmost importance that all those responsible for large projects involving computers should take care to avoid making demands on software that go far beyond the present state of the technology, unless the very considerable risks involved can be tolerated."

A second major topic area that I well recall concerned the problems of sequencing the design process and structuring a complex design. Much of the former problem was discussed in terms of top-down versus bottom-up design, a potentially confusing distinction that Stan Gill did much to clarify. Another, to me, major contributor to the debate was Bob Barton, who made an eloquent plea regarding the need to harmonise hardware and software architecture. One remark made during the debate that particularly struck home, and which I have myself since quoted on several occasions was that by Peter Naur:

"In the design of automobiles, the knowledge that you can design the motor more or less independently of the wheels is an important insight, an important part of the automobile
designer's trade. In our field, if there are a few specific things to be produced, such as compilers, assemblers, monitors, and a few more, then it would be very important to decide what are their parts and what is the proper sequence of deciding on their parts. That is the really essential thing, what should you decide on first."

Discussions on program structuring stemmed largely from a paper by Edsger Dijkstra, summarising the lessons he had learnt from the 'THE' project, and arguing the advantages of using levels of abstraction as a structuring scheme. Two other participants whose contributions have to my mind stood the test of time, and can be read with increased understanding and sympathy now that notions of abstraction, and of abstract data types, are more widely established, are Ken Kolence and Doug Ross.

There was also extensive discussion at the conference on the problems of controlling a design and implementation activity that involved large numbers of people. Some participants were reluctant to admit that such projects were necessary, but this attitude was strongly criticised by Alan Perlis:

"We kid ourselves if we believe that software systems can only be designed and built by a small number of people. If we adopt that view this subject will remain precisely as it is today, and will ultimately die. We must learn to build software systems with hundreds, possibly thousands of people."

The most ambitious approach to this problem was that described by Bob Bemer, who envisaged the use of a "machine-controlled production environment, or software factory", which would provide an environment for all program construction, check-out and usage. There was also a detailed presentation on methods of project estimation by John Nash, and much discussion on techniques for monitoring the
progress of a software project. This discussion was brought down to
earth in memorable fashion by J.W. Smith, with the comment:

"I've only been seven months with a manufacturer and I'm still
bemused by the way they attempt to build software. SDS imposes
rigid standards on the production of software. All documents
associated with software are classified as engineering drawings.
They begin with planning specification, go through functional
specifications, implementation specifications, etc., etc. This
activity is represented by a PERT-chart with many modes. If you
look down the PERT-chart you discover that all the nodes on it
up until the last one produce nothing but paper. It is
unfortunately true that in my organisation people confuse the
menu with the meal."

Two other notable events at the conference were the specially
arranged debate on software pricing, which was discussed earlier, and
the invited talk by Doug McIlroy on "Mass-Produced Software
Components". Perhaps an extended quotation from his talk will serve
as an appropriate epilogue to this fragmentary account of the
Garmisch conference.

"We undoubtedly produce software by backward techniques. We
undoubtedly get the short end of the stick in confrontations
with the hardware people because they are the industrialists and
we are the crofters. Software production today appears in the
scale of industrialization somewhere below the more backward
construction industries. I think its place is considerably
higher, and would like to investigate the prospects for
mass-production techniques in software.

... I would like to see components become a dignified branch of
software engineering. I would like to see standard catalogues of
routines, classified by precision, robustness, time-space
performance, size limits, and binding time of parameters. I would like to apply routines in the catalogue to any one of a large class of often quite different machines, without too much pain. I do not insist that I be able to compile a particular routine directly, but I do insist that transliteration be essentially direct. I do not want the routine to be inherently inefficient due to being expressed in machine independent terms. I want to have confidence in the quality of the routines. I want the different types of routine in the catalogue that are similar in purpose to be engineered uniformly, so that two similar routines should be available with similar options and two options of the same routine should be interchangeable in situations indifferent to that option.

What I have just asked for is simply industrialism, with programming terms substituted for some of the more mechanically oriented terms appropriate to mass production. I think that there are considerable areas of software ready, if not overdue, for this approach."
8. CONCLUSIONS

Here I must end this series of backward glances. It is to be hoped that I have given those of you who have entered this strange profession of ours only recently some idea of how one of the survivors now recollects the 1968 software scene and the conference which launched "software engineering" onto an unsuspecting world. Equally I trust that those who were already active in 1968 do not find the picture I have painted too much at odds with their own memories of those days. From a personal viewpoint 1968 turned out to be a major turning point. It was the year in which, to approximate John Buxton's marvellous comment, I left the ivory towers of industry for the sordid commercial life of a university computing laboratory. The main research I have undertaken since then on system reliability was, I now see, largely provoked by what I learnt and experienced at Garmisch; and my cynical attitude to programming language design was certainly strengthened, if not caused, by attendance at the WG2.1 meeting which decided the fate of ALGOL 68.

The year after Garmisch a follow-up conference was held in Rome (5). Sadly the Garmisch spirit did not survive the journey across the Alps, and the conference was a much less successful affair. However, the story of this conference, and of all the other post-1968 developments, lies beyond the self-imposed confines of the present account. I gladly leave it to others to express opinions as to what progress has been made since 1968 in turning the phrase "software engineering" from an expression of a requirement to a description of a healthy well-developed discipline. It is for them to attempt to refute the statement expressed in my opening quotation, that "the future is dark, the present burdensome".
9. REFERENCES


47. Anon. IBM Users Praise Hardware, Boo Software. *Datamation* 14,6 (June 1968) p. 19.

48. Anon. Roster of Organisations Selling and/or Producing Software. *Computers and Automation* 17,6 (June 1968) pp. 84-87,89.


