SOFTWARE ENGINEERING

Report on a conference sponsored by the

NATO SCIENCE COMMITTEE

Garmisch, Germany, 7th to 11th October 1968

Chairman: Professor Dr. F. L. Bauer

Co-chairmen: Professor L. Bolliet, Dr. H. J. Helms

Editors: Peter Naur and Brian Randell

January 1969

The present report is available from:
Scientific Affairs Division
NATO
Brussels 39 Belgium

Note for the current edition: The version of this report that you are reading was prepared by scanning the original edition, conversion to text through OCR, and then reformatting. Every effort has been made to do this as accurately as possible. However, it is almost certain that some errors have crept in despite best efforts. One of the problems was that the OCR software used kept trying to convert the original British spellings of words like 'realise' to the American spelling 'realize' and made other stupid mistakes. Whenever the OCR program was unsure of a reading, it called it to the attention of the operator, but there were a number of occasions in which it was sure, but wrong. Not all of these instances are guaranteed to have been caught.

Although the editor tried to conform to the original presentation, certain changes were necessary, such as pagination. In order that the original Table of Contents and Indices would not have to be recalculated, an artifice was used. That is the original page breaks are indicated in the text thusly: 49 indicates that this is the point at which page 49 began in the original edition. If two such indicators appear together, this shows that there was a blank page in the original.

The figures have been redrawn to improve legibility. The original size and scale was not changed. In order to accommodate the new pagination, the figures may have been shifted slightly from their position in the original document.

Finally, it should be noted that the effort required to produce the current edition was a tiny fraction of the effort required for the original. The current editor therefore wants to express his appreciation to the original editors, Peter Naur and Brian Randell, for producing what was clearly a landmark effort in the Software Engineering field.

Robert M. McClure
Arizona 2001

HIGHLIGHTS

The present report is concerned with a problem crucial to the use of computers, viz. the so-called software, or programs, developed to control their action. The report summarises the discussions at a Working Conference on Software Engineering, sponsored by the NATO Science Committee. The Conference was attended by more than fifty people, from eleven different countries, all concerned professionally with software, either as users, manufacturers, or teachers at universities. The discussions cover all aspects of software including

- relation of software to the hardware of computers
- design of software
- production, or implementation of software
- distribution of software
- service on software.

By including many direct quotations and exchanges of opinion, the report reflects the lively controversies of the original discussion.

Although much of the discussions were of a detailed technical nature, the report also contains sections reporting on discussions which will be of interest to a much wider audience. This holds for subjects like

- the problems of achieving sufficient reliability in the data systems which are becoming increasingly integrated into the central activities of modern society
- the difficulties of meeting schedules and specifications on large software projects
- the education of software (or data systems) engineers
- the highly controversial question of whether software should be priced separately from hardware.

Thus, while the report is of particular concern to the immediate users of computers and to computer manufacturers, many points may serve to enlighten and warn policy makers at all levels. Readers from the wider audience should note, however, that the conference was concentrating on the basic issues and key problems in the critical areas of software engineering. It therefore did not attempt to provide a balanced review of the total state of software, and tends to understress the achievements of the field.

CONTENTS

8 9

PREFACE

In working out the present report on the Conference on Software Engineering organised by the NATO Science Committee, every attempt was made to make it useful to a wide circle of readers. Thus parts of it are written for 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. This class includes, for example:

- civil servants
- politicians
- policy makers of public and private enterprises.

These readers should find material of interest in Section 1 (Background of Conference) and Section 2 (Software Engineering and Society).

A somewhat narrower readership for the report includes those who need an understanding of the nature of software engineering, although they are not themselves working in the field. These readers are typically:

- managers of business enterprises using computers
- researchers in fields other than software engineering and computer science
- university officials
- computer marketing personnel.

These readers should find points of interest in Section 3 (Software Engineering), Section 7.1 (Software: the State of the Art), Section 7.2 (Education), and Section 7.3 (Software Pricing) as well as in Sections 1 and 2.

Finally, a large part of the report addresses itself to those directly engaged in the design, production (implementation), and service of software. These technical areas are first given an approximately 10 uniform coverage in Sections 4 (Design), 5 (Production), and 6 (Service). The succeeding chapters 7 (Special Topics), 8 (Invited Addresses) and 9 (Working Papers), present more detailed treatment of a selected set of topics.

The main problem in deciding on a structure for the report was to decide between one of two different basic classifications, the one following from the normal sequence of steps in the development of a software product, from project start, through design, production or development, to distribution and maintenance, the other related to aspects like communication, documentation, management, programming techniques, data structures, hardware considerations, and the like. The final structure is based on the first of these two classifications. However, in many places an unavoidable influence from the second type of classification has crept in. The editors are only too aware of this problem and have attempted to mitigate its effects by provision of a detailed index.

The text of the report derives mainly from two sources, viz. the working papers contributed by the participants before or during the conference (mostly in June 1968), and the discussions during the conference. The discussions were recorded by several reporters and most were also recorded on magnetic tape. The reporters' notes were then collated, correlated with footage numbers on the magnetic tape, and typed. Owing to the high quality of the reporters' notes it was then, in general, possible to avoid extensive amounts of tape transcription, except where the accuracy of quotations required verification. However, to give an impression of the editors' task, here is an example, albeit extreme, of the typed notes:

```
536    DIJKSTRA
F      -
H      --
P      --?--
```

(here '536' is the tape footage number, and the letters F,H and P identify the reporters). This section of tape was transcribed to reveal that what was actually said was: 11

»There is tremendous difference if maintenance means adaptation to a changing problem, or just correcting blunders. It was the first kind of maintenance I was talking about. You may be right in blaming users for asking for blue-sky equipment, but if the manufacturing community offers this with a serious face, then I can only say that the whole business is based on one big fraud. [*Laughter and applause*]«.

For use in the report the source texts, and some additional transcribed material, have been sorted out according to the subject classification developed during the conference. Whenever possible the material in the working papers has been integrated into Sections 3 to 7 of the report. However, in some cases it has been found more convenient to place the working material in Section 9, and merely to make the appropriate references in the main sections of the report.

To avoid misinterpretations of the report it must be kept in mind that the participants at the conference were acting as individuals, and in no sense as representatives of the organizations with which they are affiliated.

In order to retain the spirit and liveliness of the conference, every attempt has been made to reproduce the points made during the discussion by using the original wording. This means that points of major disagreement have been left wide open, and that no attempt has been made to arrive at a consensus or majority view. This is also the reason why the names of participants have been given throughout the report.

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 Angermeyer, 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 the 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 12 done by Peter Naur and Brian Randell, assisted by Ian Hugo. The preparation of the final typed copy of the report was done by Miss Kirsten Andersen at Regnecentralen, Copenhagen, under the direction of Peter Naur,

Peter Naur
Brian Randell

13

1. BACKGROUND OF CONFERENCE

Discussions were held in early 1967 by the NATO Science Committee, comprised of scientists representing the various member nations, on possible international actions in the field of computer science. Among the possible actions considered were the organising of a conference, and perhaps, at a later date, the setting up of an International Institute of Computer Science.

In the Autumn of 1967 the Science Committee established a Study Group on Computer Science. The Study Group was given the task of assessing the entire field of computer science, and in particular, elaborating the suggestions of the Science Committee.

The Study Group concentrated on possible actions which would merit an international, rather than a national effort. In particular it focussed its attentions on the problems of software. In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. 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.

It was suggested that about 50 experts from all areas concerned with software problems — computer manufacturers, universities, software houses, computer users, etc. — be invited to attend the conference. It was further suggested that every effort should be made to make the conference truly a working conference, whose discussions should be organised under the three main headings: Design of Software, Production of Software, and Service of Software.

Prominent leaders in the field were appointed as group leaders to direct the work within each of the three working groups. Dr. Arnth-Jensen, of the Scientific Affairs Division of NATO, was put 14 in charge of conference arrangements. At a meeting held in Brussels in March 1968 the group leaders and the Study Group met and agreed on the final details of the conference.

The Conference was to shed further light on the many current problems in software engineering, and also to discuss possible techniques, methods and developments which might lead to their solution. It was hoped that the Conference would be able to identify present necessities, shortcomings and trends and that the findings could serve as a signpost to manufacturers of computers as well as their users.

With this hope in mind the present report is made widely available.

15

2. SOFTWARE ENGINEERING AND SOCIETY

One of the major motivations for the organizing of the conference was an awareness of the rapidly increasing importance of computer software systems in many activities of society. Thus, although much of the conference was concerned with detailed technical questions, many of the discussions were of a more general nature, and should be of interest to a wide spectrum of readers. It is for the benefit of this wider audience that representative discussions of various points relating to the impact of software engineering on society have been abstracted from later sections of this Report, and collected in this introductory section.

First, three quotations which indicate the rate of growth of software:

Helms: In Europe alone there are about 10,000 installed computers — this number is increasing at a rate of anywhere from 25 per cent to 50 per cent per year. The quality of software provided for these computers will soon affect more than a quarter of a million analysts and programmers.

David: No less a person than T.J. Watson said that OS/360 cost IBM over $50 million dollars a year during its preparation, and at least 5000 man-years' investment. TSS/360 is said to be in the 1000 man-year category. It has been said, too, that development costs for software equal the development costs for hardware in establishing a new machine line.

d'Agapeyeff: In 1958 a European general purpose computer manufacturer often had less than 50 software programmers, now they probably number 1,000-2,000 people; what will be needed in 1978?

Yet this growth rate was viewed with more alarm than pride.

David: In computing, the research, development, and production phases are 16 often telescoped into one process. In the competitive rush to make available the latest techniques, such as on-line consoles served by time-shared computers, we strive to take great forward leaps across gulfs of unknown width and depth. In the cold light of day, we know that a step-by-step approach separating research and development from production is less risky and more likely to be successful. Experience indeed indicates that for software tasks similar to previous ones, estimates are accurate to within 10–30 percent in many cases. This situation is familiar in all fields lacking a firm theoretical base. Thus, there are good reasons why software tasks that include novel concepts involve not only uncalculated but also uncalculable risks.

This is not meant to indicate that the software field does not have its successes.

Hastings: I work in an environment of some fourteen large installations using OS/360. These are complex systems, being used for many very sophisticated applications. People are doing what they need to do, at a much lower cost than ever before, and they seem to be reasonably satisfied.

Buxton: Ninety-nine percent of computers work tolerably satisfactorily. There are thousands of respectable Fortran-oriented installations using many different machines, and lots of good data processing applications running steadily.

However, there are areas of the field which were viewed by many participants with great concern.

Kolence: The basic problem is that certain classes of systems are placing demands on us which are beyond our capabilities and our theories and methods of design and production at this time. There are many areas where there is no such thing as a crisis — sort routines, payroll applications, for example. It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy.

David and Fraser: 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.

Dijkstra: The dissemination of knowledge is of obvious value — the massive dissemination of error-loaded software is frightening.

17

There was general agreement that 'software engineering' is in a very rudimentary stage of development as compared with the established branches of engineering.

McIlroy: We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with 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.

Kolence: Programming management will continue to deserve its current poor reputation for cost and schedule effectiveness until such time as a more complete understanding of the program design process is achieved.

Fraser: One of the problems that is central to the software production process is to identify the nature of progress and to find some way of measuring it. Only one thing seems to be clear just now. It is that program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as the sum of many sub-assemblies.

Graham: Today we tend to go on for years, with tremendous investments to find that the system, which was not well understood to start with, does not work as anticipated. We build systems like the Wright brothers built airplanes — build the whole thing, push it off the cliff, let it crash, and start over again.

Of course any new field has its growing pains:

Gillette: We are in many ways in an analogous position to the aircraft industry, which also has problems producing systems on schedule and to specification. We perhaps have more examples of bad large systems than good, but we are a young industry and are learning how to do better.

Many people agreed that one of the main problems was the pressure to produce even bigger and more sophisticated systems.

Opler: I am concerned about the current growth of systems, and what I expect is probably an exponential growth of errors. Should we have systems of this size and complexity? Is it the manufacturer's fault for producing them or the user's for demanding them? One shouldn't ask for large systems and then complain about their largeness.

18

Buxton: There are extremely strong economic pressures on manufacturers, both from users and from other manufacturers. Some of these pressures, which are a major contributory cause of our problems, are quite understandable. For example, the rate of increase of air traffic in Europe is such that there is a pressing need for an automated system of control.

This being the case, perhaps the best quotation to use to end this short section of the report is the following:

Gill: 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 technology) unless the very considerable risks involved can be tolerated.

19

3. SOFTWARE ENGINEERING

3.1. THE NATURE OF SOFTWARE ENGINEERING

The content of software engineering was explained in several ways during the conference. Nash and Selig provided figures 1 and 2 indicating the various activities of a software project. These diagrams also indicate some of the terminology used in the field.

The need for feedback was stressed many times.

Perlis: Selig's picture requires a feedback loop, for monitoring of the system. One must collect data on system performance, for use in future improvements.

The project activity was described in more detail by Fraser.

Fraser: (from *The nature of progress in software production*)

»Design and implementation proceeded in a number of stages. Each stage was typified by a period of intellectual activity followed by a period of program reconstruction. Each stage produced a useable product and the period between the end of one stage and the start of the next provided the operational experience upon which the next design was based. In general the products of successive stages approached the final design requirement; each stage included more facilities than the last. On three occasions major design changes were made but for the most part the changes were localised and could be described as 'tuning'.

The first stage did not terminate with a useable object program but the process of implementation yielded the information that a major design change would result in a superior and less expensive final product. During the second stage the entire system was reconstructed; an act that was fully justified by subsequent experience. The second major design change had its origin in the more usual combination of an inelegant system and a demanding environment. Over a period of time we discovered 22 the failure characteristics of the hardware that was being used and assembled a list of techniques designed to overcome these. The final major design change arose out of observing the slow but steady escalation of complexity in one area of the system. As is often the case, this escalation had its origins in the conceptual inadequacy of the basic design. By replacing the appropriate section of the system kernel we simultaneously removed an apparent need for a growing number of specialized additions to the superstructure and considerably enhanced the quality of the final product«

The disadvantages of not allowing feedback were commented on by Galler.

Galler: Let me mention some bad experiences with IBM. One example concerns a request to allow user extensions of the PL/1 language. After a week of internal discussion at IBM it was decided that this could not be done because the language designers were not to tell the implementers how to implement the desired extensions. Another example: the OS/360 job control language was developed without any users having the chance to see the options beforehand, at the design stage. Why do these things happen?

External and internal design and their mutual feedback were described by Selig.

Selig: External specifications at any level describe the software product in terms of the items controlled by and available to the user. The internal design describes the software product in terms of the program structures which realize the external specifications. It has to be understood that feedback between the design of the external and internal specifications is an essential part of a realistic and effective implementation process. Furthermore, this interaction must begin at the earliest stage of establishing the objectives, and continue until completion of the product.

20

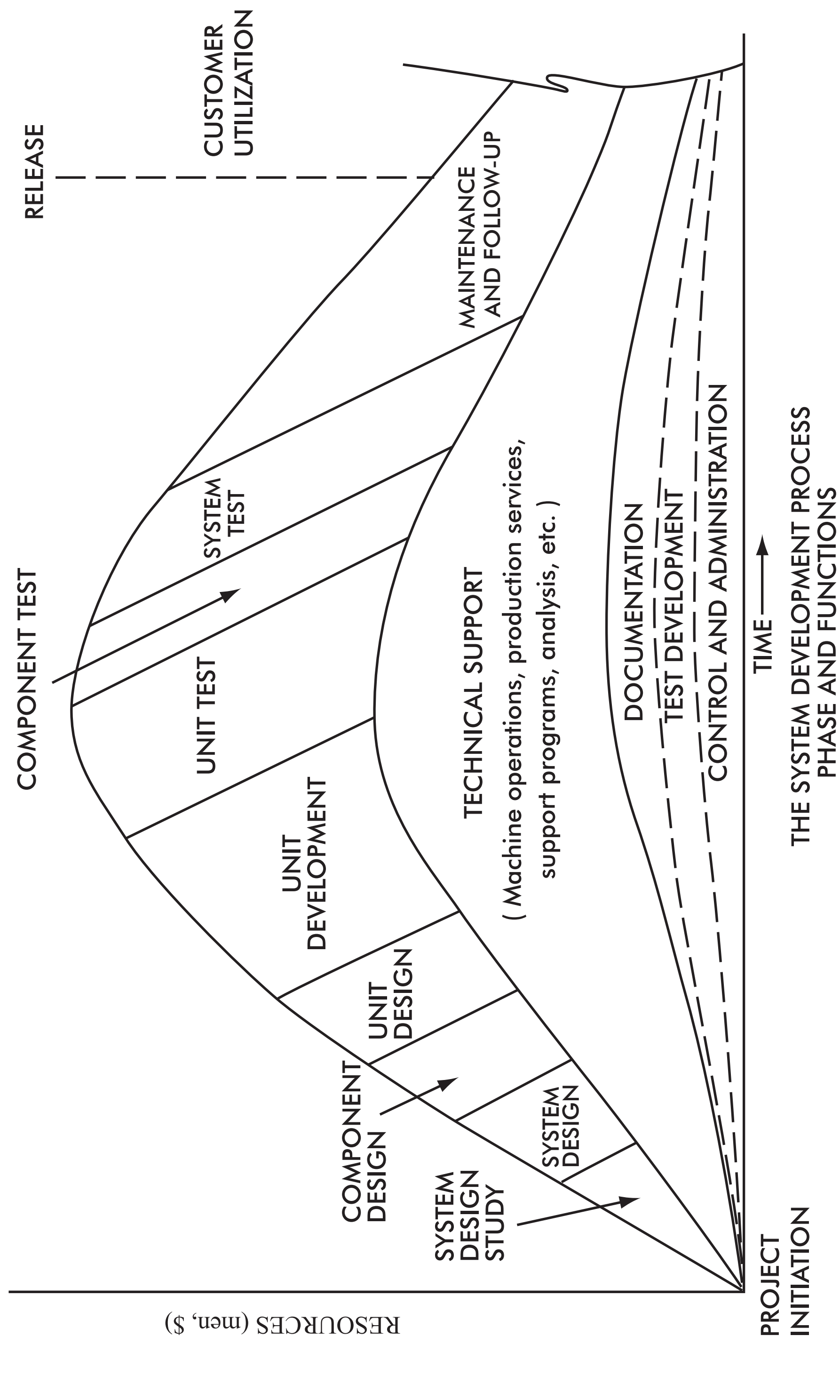

Figure 1. From Nash: Some problems in the production of large-scale software systems.

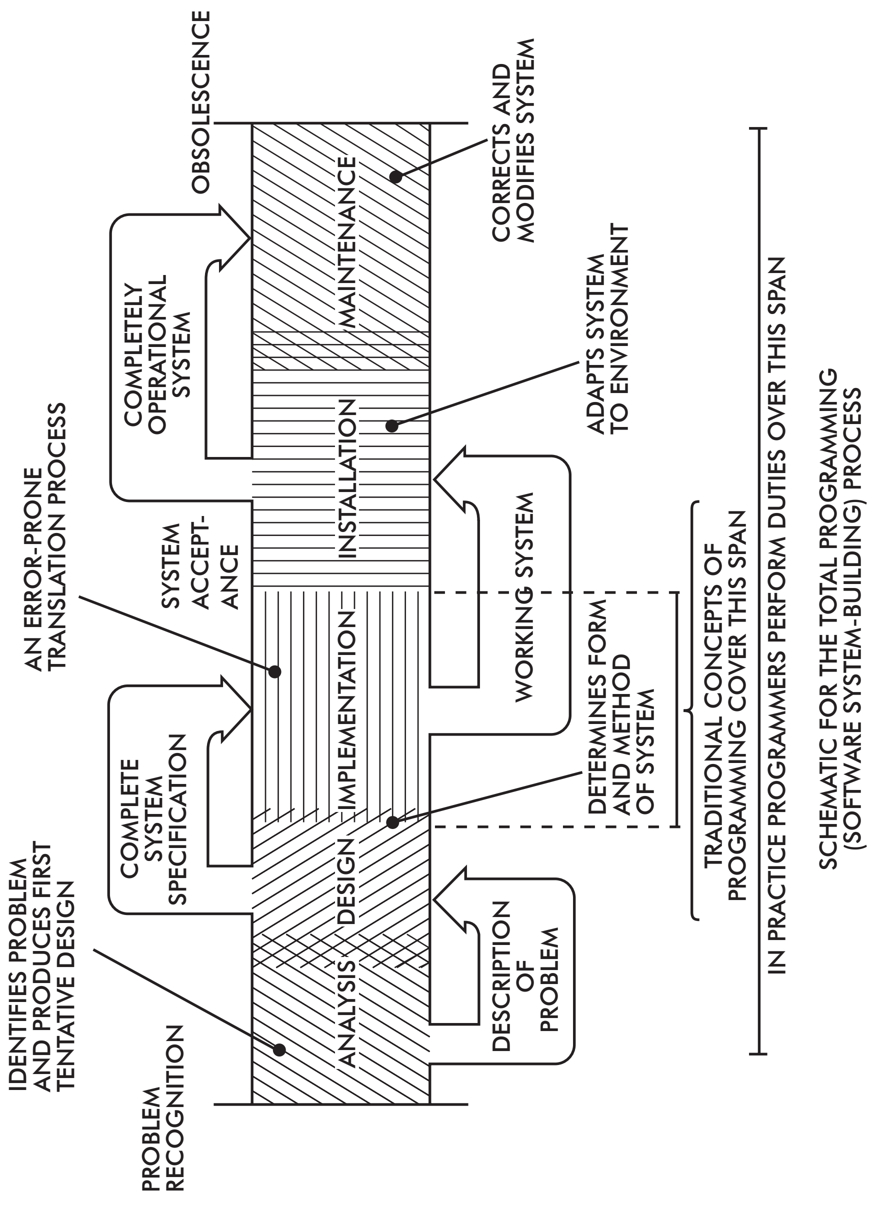
PROBLEM RECOGNITION
IDENTIFIES PROBLEM AND PRODUCES FIRST TENTATIVE DESIGN
ANALYSIS
DESIGN
DESCRIPTION OF PROBLEM
COMPLETE SYSTEM SPECIFICATION
IMPLEMENTATION
DETERMINES FORM AND METHOD OF SYSTEM
AN ERROR-PRONE TRANSLATION PROCESS
SYSTEM ACCEPT-ANCE
WORKING SYSTEM
INSTALLATION
COMPLETELY OPERATIONAL SYSTEM
ADAPTS SYSTEM TO ENVIRONMENT
MAINTENANCE
CORRECTS AND MODIFIES SYSTEM
OBSOLESCENCE
TRADITIONAL CONCEPTS OF PROGRAMMING COVER THIS SPAN
IN PRACTICE PROGRAMMERS PERFORM DUTIES OVER THIS SPAN
SCHEMATIC FOR THE TOTAL PROGRAMMING (SOFTWARE SYSTEM-BUILDING) PROCESS

Figure 2. From Selig: Documentation for service and users. Originally due to Constantine.

Another over-all view of the substance of software engineering was given.

d'Agapeyeff: An example of the kind of software system I am talking about is putting all the applications in a hospital on a computer, whereby you 23 get a whole set of people to use the machine. This kind of system is very sensitive to weaknesses in the software, particular as regards the inability to maintain the system and to extend it freely.

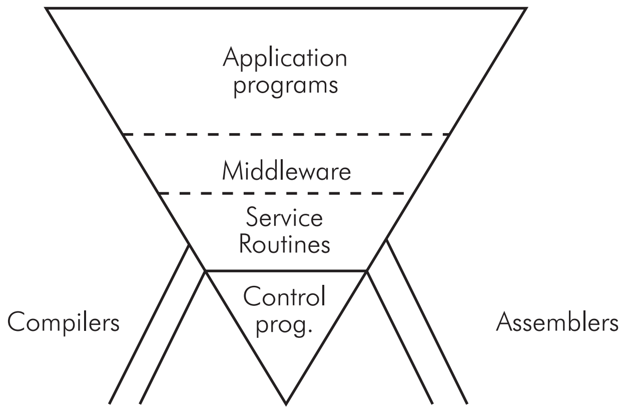

Figure 3. d'Agapeyeff's Inverted Pyramid

This sensitivity of software can be understood if we liken it to what I will call the inverted pyramid (see figure 3). The buttresses are assemblers and compilers. They don't help to maintain the thing, but if they fail you have a skew. At the bottom are the control programs, then the various service routines. Further up we have what I call middleware.

This is because no matter how good the manufacturer's software for items like file handling it is just not suitable; it's either inefficient or inappropriate. We usually have to rewrite the file handling processes, the initial message analysis and above all the real-time schedulers, because in this type of situation the application programs interact and the manufacturers, software tends to throw them off at the drop of a hat, which is somewhat embarrassing. On the top you have a whole chain of application programs.

The point about this pyramid is that it is terribly sensitive to change in the underlying software such that the new version does not contain the old as a subset. It becomes very expensive to maintain these systems and to extend them while keeping them live.

24

3.2. SOFTWARE ENGINEERING MANAGEMENT AND METHODOLOGY

Several participants were concerned with software engineering management and methodology, as exemplified by the following remarks.

Kolence: (from *On the interaction between software design techniques and software management problems*)

» A software design methodology is composed of the knowledge and understanding of what a program is, and the set of methods, procedures, and techniques by which it is developed. With this understanding it becomes obvious that the techniques and problems of software management are interrelated with the existing methodologies of software design.«

d'Agapeyeff: (from *Reducing the cost of software*)

»Programming is still too much of an artistic endeavour. We need a more substantial basis to be taught and monitored in practice on the:

(i) structure of programs and the flow of their execution;

(ii) shaping of modules and an environment for their testing;

(iii) simulation of run time conditions.«

Kinslow: There are two classes of system designers. The first, if given five problems will solve them one at a time. The second will come back and announce that these aren't the real problems, and will eventually propose a solution to the single problem which underlies the original five. This is the 'system type' who is great during the initial stages of a design project. However, you had better get rid of him after the first six months if you want to get a working system.

Berghuis: (from *The establishment of standard programming and management techniques throughout the development and production of software and their enforcement*)

» Independent software packages don't exist; they run on an equipment (hardware), they need procedures by which to be operated and that indicates that we have to define what a system, project, phase of a project, releases, versions, etc., mean. Also we have to consider the organisation from the point of view of developing systems and in fact we are faced with the differences between functional and project organisation. We are also faced with the difficulties of system-engineering. «

25

Detailed information related to one particular, large project is given below.

Harr: (from: *The design and production of real-time software for Electronic Switching Systems*):

»In order to set the stage for a discussion on techniques for management of program design and production, I would like to first review with you the program design process. By program design process I mean all of the activities required to produce a documented, tested and working program. Regardless of how large or small a programming task is, it requires essentially the following sequence of steps:

1. The design process
   a. Specification of the complete hardware-software system.
   b. Definition of the functions to be performed by the program.
   c. Design and documentation of the master (overall) program plan.
   d. Subdivision of the large system program into manageable program blocks.
   e. At this point, the interfaces between the program blocks must be precisely defined and documented. Since the usual means of passing data between program jobs is via the use of data in the memory of the system, the fundamental program progress

data in memory should be formulated and defined for the interfaces between each program block.

f. Basic program subroutines defined and documented.

g. Detail design, coding and documentation of each program block.

h. Design and documentation of test methods for each program block in parallel with step (g).

i. Compilation and hand check of each program block.

j. Simulation of each program block using test methods planned during the design of the program.

k. Test and evaluation of the program blocks in the system.

l. Integration of complete program in the system.

m. Final load testing of the complete software-hardware package to see that the program meets all of its design requirements.

2. Organization used in the program design process

a. Structure of the programming groups as below.

26

ORGANIZATION OF PROGRAM GROUPS (1961)

	NO. OF PROGRAM DESIGNERS	
SYSTEM PLANNING AND REQUIREMENTS DEPT.		36
OPERATION	13	
MAINTENANCE	13	
ADMINISTRATION	9	
SYSTEM PROGRAM DESIGN DEPT		34
OPERATION AND ORDER STRUCTURE	2	
COMPILER	11	
SIMULATION	10	
TOTAL		70

ORGANIZATION OF PROGRAM GROUPS (1963)

	NO. OF PROGRAM DESIGNERS	
ESS REQUIREMENTS		38
SUBURBAN OFFICES	19	
METROPOLITAN OFFICES	11	
PROCESSOR AND ADMINISTRATION	17	
ESS PROGRAM DESIGN		45
SYSTEM	18	
NETWORK	14	
COMPILER	12	

ESS MAINTENANCE PROGRAM		47
PROCESSOR	10	
MEMORY	21	
PERIPHERAL	15	
	TOTAL	130

27

b. Types of personnel used as program designers include two-year trade school graduates, B.S., M.S., and Ph.D.'s in Electrical Engineering and Mathematics.

c. Communications within and between programming groups as below.

COMMUNICATIONS WITHIN AND BETWEEN PROGRAMMING GROUPS

GROUP MEETINGS ON DESIGN SPECIFICATIONS

FORMAL PROGRAM DESIGN SPECIFICATIONS

INFORMAL ENGINEERING AND PROGRAMMING MEMORANDA

PROGRAM CHARACTERIZATION DATA

FORMAL PROGRAM DOCUMENTATION

d. Control and measurement of a programmer's output. See curves (figures 4 and 5) and summary below.

SUMMARY OF FOUR NO. 1 ESS PROGRAM JOBS

The data covers design through check out of an operational program.

	PROG UNITS	NO. OF PROGRAMMERS	YRS	MAN YEARS	PROG WORDS	WORDS MAN YR
Operational	50	83	4	101	52,000	515
Maintenance	36	60	4	81	51,000	630
Compiler	13	9	2 1/4	17	38,000	2230
Translation Data Assembler	15	13	2 1/2	11	25,000	2270

30

3. Program Documentation

   a. Methods and standards used in preparation and editing of: Program descriptions, flowcharts, listings and program change notices.

   b. Preparation and updating of program user's manuals covering operational and maintenance procedures.

4. Development of general purpose computer tools i.e.:

   a. A macro-compiler, assembler and loader.

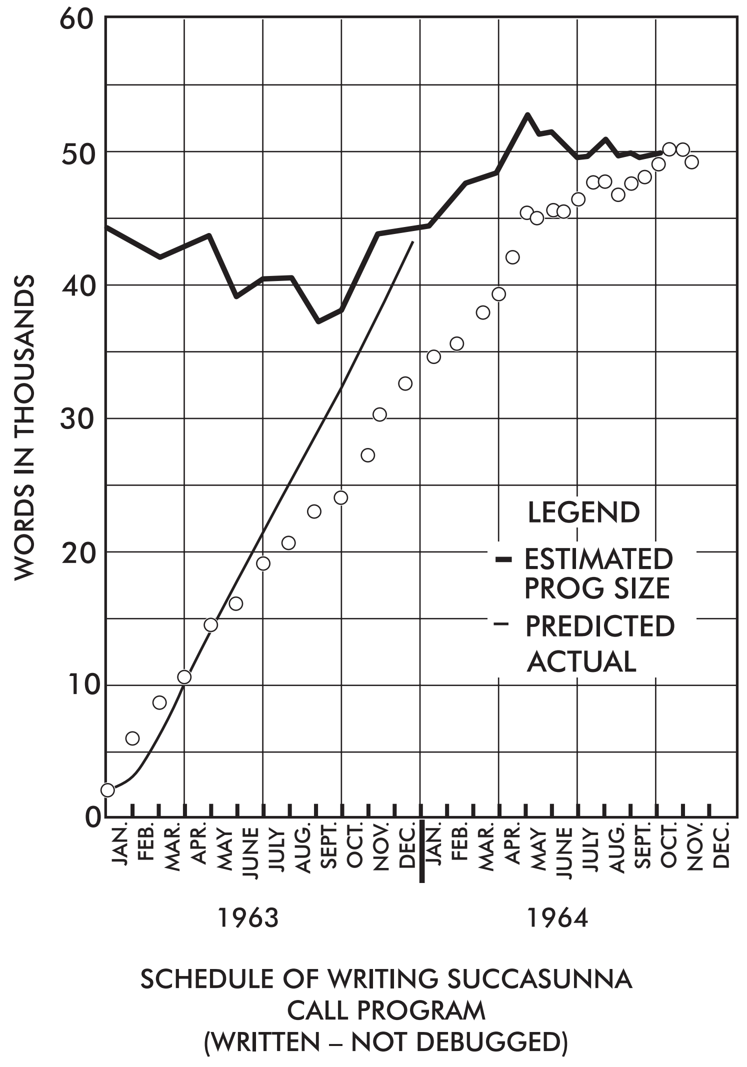

Figure 4. From Harr: The design and production of real-time software for Electronic Switching Systems.

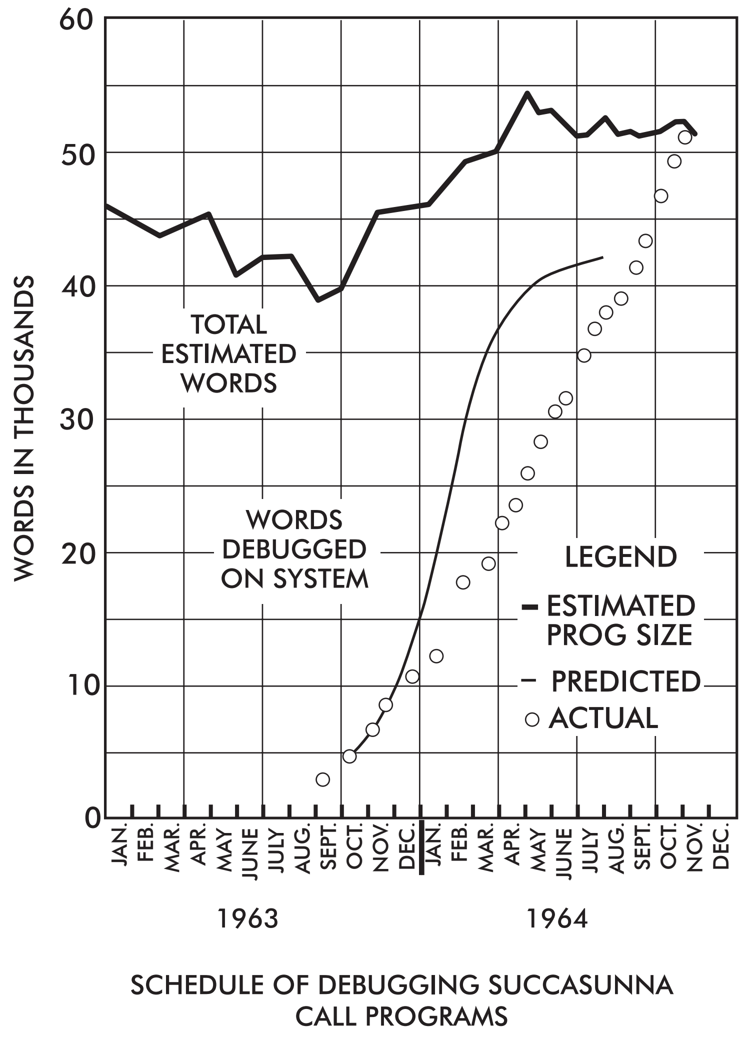

Figure 5. From Harr: The design and production of real-time software for Electronic Switching Systems.

   b. Parameter compiler.

   c. Translation data assembler.

   d. Simulators for:

      (i). Studying traffic and real-time capabilities

      (ii). Testing hardware design

   e. Summary of general purpose computer support programs.

5. Program test and evaluation support programs.

   a. By simulation on a general purpose machine.

   b. On the system itself:

      (i) First by program blocks

      (ii) Second by program functions

      (iii) Finally under many load tests as a complete hardware software system.«

Inspired by a proposal by Fraser in 'Classification of software production methods,' a working party was formed to work on the classification of the subject matter of software production methods. This working party, with the members Bemer, Fraser, Glennie, Opler, and Wiehle, submitted the report titled 'Classification of subject matter,' reproduced in section 9.

3.3. DESIGN AND PRODUCTION IN SOFTWARE ENGINEERING

The difficulties associated with the distinction between design and production (or implementation) in software engineering was brought out many times during the conference. To clarify this matter it must first be stressed that the difficulty is partly one of terminology. 31 *Indeed, what is meant here by production in software engineering is not just the making of more copies of the same software package (replication), but the initial production of coded and checked programs. However, as evidenced by quotations below, the appropriateness of the distinction between design and production was contested by several participants.*

Added to these basic difficulties is that the conference was organized around the three concepts, design, production, and service. In organizing the report this distinction was retained, mostly for expediency.

First we quote an attempt to clarify the distinction.

Naur: (from *The profiles of software designers and producers*)

»Software production takes us from the result of the design to the program to be executed in the computer. The distinction between design and production is essentially a practical one, imposed by the need for a division of the labor. In fact, there is no essential difference between design and production, since even the production will include decisions which will influence the performance of the software system, and thus properly belong in the design phase. For the distinction to be useful, the design work is charged with the specific responsibility that it is pursued to a level of detail where the decisions remaining to be made during production are known to be insignificant to the performance of the system.«

The practical dangers of the distinction are stressed in the two following remarks.

Dijkstra: Honestly, I cannot see how these activities allow a rigid separation if we are going to do a decent job. If you have your production group, it must produce something, but the thing to be produced has to be correct, has to be good. However, I am convinced that the quality of the product can never be established afterwards. Whether the correctness of a piece of software can be guaranteed or not depends greatly on the structure of the thing made. This means that the ability to convince users, or yourself, that the product is good, is closely intertwined with the design process itself.

32

Kinslow: The design process is an iterative one. I will tell you one thing which can go wrong with it if you are not in the laboratory. In my terms design consists of:

1. Flowchart until you think you understand the problem.

2. Write code until you realize that you don't.

3. Go back and re-do the flowchart.

4, Write some more code and iterate to what you feel is the correct solution.

If you are in a large production project, trying to build a big system, you have a deadline to write the specifications and for someone else to write the code. Unless you have been through this before you unconsciously skip over some specifications, saying to yourself: I will fill that in later. You know you are going to iterate, so you don't do a complete job the first time. Unfortunately, what happens is that 200 people start writing code. Now you start through the second iteration, with a better understanding of the problem, and it is too late. This is why there is version 0, version 1, version N. If you are building a big system and you are writing specifications) you don't have the chance to iterate, the iteration is cut short by an arbitrary deadline. This is a fact that must be changed.

Ross: The most deadly thing in software is the concept, which almost universally seems to be followed, that you are going to specify what you are going to do, and then do it. And that is where most of our troubles come from. The projects that are called successful, have met their specifications. But those specifications were based upon the designers' ignorance before they started the job.

3.4. MISCELLANEOUS NOTES ON THE SUBJECT OF THE CONFERENCE

A few remarks reflect on the subject of the conference as a whole.

Barton: The scope of this conference is not large enough. In design we should start by designing hardware and software together. This will require a kind of general-purpose person, 'a computer engineer'.

33

Babcock: (from *Variations on software available to the user*)

»The future of new operating system developments in the future may depend more and more on new hardware just on the horizon. The advent of slow-write, read-only stores makes it possible for the **software** designer to depend on a viable machine that has flexible characteristics as opposed to rigid operation structures found in most of today's machines. If the software designer had access to a microprogramming high-level language compiler, then systems could be designed to the specific problem area without severe hardware constraints. Major U. S. manufacturers including IBM, UNIVAC, Standard Computer, RCA are building or seriously considering such enhancements.«

34 35

4. DESIGN

4.1. INTRODUCTION

4.1.1. SOURCES OF TECHNIQUES

Part of the discussion was concerned with the sources of the right attitude to design. The suggestion that designers should record their wrong decisions, to avoid having them repeated, met the following response:

McClure: Confession is good for the soul —

d'Agapeyeff: — but bad for your career.

A more general suggestion was made:

Naur: (from *The profiles of software designers and producers*)

»… software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous constructions, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem. As one single example of such a source of ideas I would like to mention: Christopher Alexander: *Notes on the Synthesis of Form* (Harvard Univ. Press, 1964)«

4.1.2. NEED FOR HARDWARE BASED ON PROGRAM STRUCTURE

This point was elaborated in two contributions.

d'Agapeyeff: (from *Reducing the cost of software*)

»Instead of crude execution of absolute code, allow the machine to use a higher level intermediate code (perhaps through interpretation via a fast read-only store) in order to:

1. increase run time checks and therefore program reliability;

36

2. provide more facilities for program development;
3. reduce the problem of conversion to new machines;
4. allow all communication with the programmer to be in source language.

Another point is that programming would be simplified if certain processes could be declarative rather than procedural. This is an extension of the idea of channel command words. It appears this could be applied to much of I/O, file processing and checking. The functions must, however, operate in a uniform manner, indicate their progress (e.g. by separate clocks) and allow for dynamic alteration of the declarations.

Finally we need more redundant information, available in the machine, safely recording what has just happened (e.g. last operand addresses and contents, etc.) so that there can be greater automatic retrieval after an error. The current tendency for programs to be arbitrarily thrown off must be reversed.«

Barton: In design you have to start at the level of organization of programs and machines, with the design of hardware and software together. This must be in the hands of one or a few individuals and will require a kind of general purpose person. What do these persons know? As far as programming is concerned they can only know what the state of the art is, what can be learned from existing systems, the extrapolations that people have made on these, and what happens in languages, in operating systems, in control programs. They are not expert programmers or experts in things like syntax directed compilers. They must be people who are interested in synthesizing, extracting essentials out of what has been learned. On the hardware side they are not hardware designers either, but they are concerned about putting together the gross logic of machines. These are the only parts of the design that can be done by just a few people. These people have to rely on two other groups. Their decision making stops on the programming side with the choice of a basic language to use, just as in conventional design practice you start with the engineers handing over a machine language. In this approach the designers can hand over a language at a higher level, but it need not pay any attention to the standards that exist in the field at the given time, it should just embody their best understanding of the tools, concepts and ideas of programming. If ultimate users use that language they would use it in the same way they 37 would use an assembly language furnished by the manufacturer. At any rate, the systems designers can then hand over to compiler implementers something which is more complete and takes account of more problems before the languages demanded by the market place are implemented. The systems designers, on the other hand, hand over to the engineers a gross logical organization. Then logical detailing and circuit development takes place. Circuit development and logic nowadays are getting closely intertwined. With the programmers and engineers, there will be feedback going on for a considerable time, for months or even a year. But this is feedback into one place, the place where there is some central, authoritative, decision making. That is not at the time the machine has been designed; it is before you have anything at all.

There is one more source of feedback, besides the logic designers and compiler implementers, namely from the world of application. Whenever an application acquires a uniqueness, enormous size perhaps, or extraordinary complexity, that does not seem to fit into the existing languages, there are three paths of feedback, one is into the area of language design, which will take care of the question of expression. The other is into the area of logic design, because completely new logic devices may be required. Third, into the one point which is right at the top, the area of system design, where people are concerned with the organization of hardware and programs together.

In all this, we don't have to wait for theory; in fact, unless you look at the problem in this way we will never get a theory.

4.1.3. RELATION TO MATHEMATICS

A few statements were made on the importance of mathematics in software design.

Perlis: Software systems are mathematical in nature. A mathematical background is not necessary for a designer, but can only add to the elegance of the design.

Bauer: What is needed is not **classical** mathematics, but **mathematics**. Systems should be built in levels and modules, which form a mathematical structure.

38

Kolence: (from *On the interaction between software design techniques and software management problems*')

»At the abstract level, a concise mathematical notation is required by which to express the essential structures and relationships irrespective of the particular software product being implemented. For example, in the area of computer central processor design, the notation that satisfies this requirement is Boolean Algebra. The notation of Ken Iverson is an attempt to provide an equivalent notation for software.«

4.2. DESIGN CRITERIA

4.2.1. GENERAL DESIGN CRITERIA

Several general design criteria were stressed.

Perlis: I adopt as a basic principle of design that the user should be able, in principle, though perhaps not shown how by the manufacturer, to reach every variable which actually appears in the implementation. For example, in formatting, if the system uses a variable that sets margins, then that should be available. You may perhaps not find it in the first volume of system description, but you should come across it later on.

Letellier: (from *The adequate testing and design of software packages*)

»… it will always be found that extensions are to be made after some time of use and a software package must be thought of as **open-ended**, which means enough syntactic flexibility in the input and modularity in the implementation«

Smith: There is a tendency that designers use fuzzy terms, like ‘elegant’ or ‘powerful’ or ‘flexible’. Designers do not describe how the design works, or the way it may be used, or the way it would operate. What is lacking is discipline, which is caused by people falling back on fuzzy concepts, instead of on some razors of Occam, which they can really use to make design decisions. Also designers don’t seem to realize what mental processes they go through when they design. Later they can neither explain, nor justify, nor even rationalize, the processes they used to build a particular system. I think that a few Occam’s razors floating around can lead to great simplifications in building software. Just the 39 simple: how something will be used, can often really sharpen and simplify the design process. Similarly, if we use the criterion: it should be easy to explain, it is remarkable how simple the design process becomes. As soon as one tries to put into the system something that takes more than a paragraph or a line to explain, throw it out — it is not worth it. We have applied this to at least one system and it worked beautifully. As a result we had documentation at the same time as we had the system. Another beautiful razor is: the user should have no feeling that he is dealing with an inanimate object, rather he should feel that he is dealing with some responsive personality, who always responds in some reasonable way. Criteria like these have nothing to do with ‘powerful linguistic features’ or ‘flexibility’ and right at the beginning I think we should throw these fuzzy terms out of the window.

David: (in *Some thoughts about production of large software systems*)

»… experience… has led some people to have the opinion that any software systems that cannot be completed by some four or five people within a year can never be completed; that is, reach a satisfactory steady-state. While no one has stated this opinion as an immutable ‘law’, the question of what techniques can lead to a counter-example arises…

Define a subset of the system which is small enough to bring to an operational state within the ‘law’ mentioned above, then build on that subsystem. This strategy requires that the system be designed in modules which can be realized, tested, and modified independently, apart from conventions for intermodule communication.«

Gillette: (from *Aids in the production of maintainable software*)

»Three fundamental design concepts are essential to a maintainable system: **modularity, specification,** and **generality**. Modularity helps to isolate functional elements of the system. One module may be debugged, improved, or extended with minimal personnel interaction or system discontinuity. As important as modularity is specification. The key to production success of any modular construct is a rigid specification of the interfaces; the specification, as a side benefit, aids in the maintenance task by supplying the documentation necessary to train, understand, and provide maintenance. From this viewpoint, specification should encompass from the innermost primitive functions outward to the generalized functions 40 such as a general file management system. Generality is essential to satisfy the requirement for extensibility.«

4.2.2. USER REQUIREMENTS

Several remarks were concerned with the influence of users upon the design.

Hume: One must be careful to avoid over-reacting to individual users. It is impossible to keep all of the users happy. You must identify and then concentrate on the requirements common to a majority of users, even if this means driving a few users with special requirements away. Particularly in a university environment you take certain liberties with people’s freedom in order to have the majority happy.

Babcock: In our experience the users are very brilliant people, especially if they are your customers and depend on you for their livelihood. We find that every design phase we go through we base strictly on the users’ reactions to the previous system. The users are the people who do our design, once we get started.

Berghuis: Users are interested in systems requirements and buy systems in that way. But that implies that they are able to say what they want. Most of the users aren’t able to. One of the greatest difficulties will be out of our field as soon as the users realize what kind of problems they have.

Smith: Many of the people who design software refer to users as ‘they’, ‘them’. They are some odd breed of cats living there in the outer world, knowing nothing, to whom nothing is owed. Most of the designers of manufacturers’ software are designing, I think, for their own benefit — they are literally playing games. They have no conception of validating their design before sending it out, or even evaluating the design in the light of potential use.

The real problem is training the people to do the design. Most designers of software are damn well incompetent, one way or another.

Paul: The customer often does not know what he needs, and is sometimes cut off from knowing what is or what might be available.

Perlis: Almost all users require much less from a large operating system than is provided.

41

The handling of user requests within a design group caused some discussion.

Goos: A main question: Who has to filter the recommendations coming from outside? The man or group who filters user requests to see what fits together is the most important aspect of design guided by user requirements.

Hume: Experience is the best filter.

Randell: Experience can be misused as a filter. It is very easy to claim that one's own experience denies the validity of some of the needs expressed by users. For example, this was one of the causes for IBM's slowness in responding to users' expressed needs for time sharing facilities.

Galler: We should have feedback from users early in the design process.

Randell: Be careful that the design team will not have to spend all its time fending off users.

Goos: One must have the 'filter' inside, not outside, the design group.

Letellier: (from *The adequate testing and design of software packages*)

»… any limitations, such as a number of variables or a maximum dimension, must be discussed with the user; new concepts have to be understood by the designer who must refrain from sticking to his own programmer's concepts; this is not as simple as it looks.

As soon as a draft of the external specifications is drawn up, the designer must go to the users and ask them to describe typical uses of the product to make sure that operational flexibility will be convenient.«

The effect a software system can have on the users also gave rise to comment.

Hume: (from *Design as controlled by external function*)

»It must be noted that the software can control or alter the way that users behave and this in turn generates demands for equipment. For example, software that offers a variety of languages in an interactive mode requires a different hierarchical arrangement of storage than that required if only one language is to be available.«

Ross: On this point of the system affecting the user, one little example is that recently a new command was put into the MAC CTSS system called BUY TIME, by which the individual user is able to purchase time within 42 his own allotment from his supervisor, instead of having to go and ask the supervisor for more time or tracks. When this system was put into effect, our own group began to use significantly more resources and my group leaders assure me that there is more work being done at the same time. It is interesting that just taking out of their loop the finding of the man who could assign them more time really had an impact on their work.

Perlis: I think everybody who works with a big computer system becomes aware that the system influences the way people react to it. In a study we made we observed, what should have been obvious, but had not been, that if one depends on a system over which one has no control, the one thing you wish from it is that the system match your expectations of it. So anyone who runs a system and gives the service he promises, however bad that may be, is giving something much better than someone who promises a great deal and then does not deliver it, except perhaps on a random basis.

Quite specific user wishes were also discussed. The views are summarized below.

Harr: (from *Design as controlled by external function*)

»… Library programs should be designed to take as little as possible of the user's time and offer him the greatest confidence in the results. When this requirement is satisfied the methods employed should be as economical of machine time as possible.

Because the various modes of operating (interactively and with fast or regular compilers) all have their special advantages, it is most useful to have the same language processed by all three methods. …

Operating systems should have scheduling algorithms that maintain the quality of service under changing user demands. In many instances as a system gets more heavily loaded, service to users deteriorates. The control of users' demands by bad service is to be deplored.

Operating systems should be simple for the user. The default options, when leaving information unspecified in a control card, for instance, should always be the most commonly wanted ones. In time-sharing or multiprogrammed environments each user's information should be assiduously protected. Nothing is more disconcerting than the possibility of 43 interference from some other user. This is perhaps a greater concern than privacy of files.

An important part of any operating system is the accounting scheme. Assignment of charges to various parts of the computer resource-memory, processor time, disk space, etc., can influence the use made of these facilities. Any rationing schemes or assessing of prices should ensure that good user practice is encouraged. Since this costing is internal to the system it should aim to produce external minimal costs for the user-equipment system as a whole.«

d'Agapeyeff: (from *Reducing the cost of software*)

»Is one operating system per machine enough? Can an operating system within a single framework (although with many options) satisfactorily meet the needs of all classes of users (e.g. bureaus, universities, business batch processing, commercial online installations) or if not, how many different systems are likely to be necessary?«

Only a few remarks were directly concerned with the problems of quite specific designs, possibly because these problems are so numerous.

Kolence: (from *On the interaction between software design techniques and software management problems*)

»The definition of certain types of external characteristics may effectively dictate a basic internal design structure which results in unsatisfactory operational characteristics. Two opposing examples are useful to illustrate this point. In commercial data processing, the functions associated with a given run in a batch processing system may be so numerous that extremely tight coding is necessary to fit the program into core. In this instance, the evolutionary power of the program, (that is, the ability to modify the system,) will be severely limited. On the other hand, requirements to permit additional features or use of various input/output formats may be so extensive that a table driven design is dictated. This design type usually has slow operational speeds.«

44

4.2.3. RELIABILITY AND DESIGN

Reliability is one of the several issues that goes across the design-production distinction. Reference should therefore also be made to section 5.1.2. One remark specially mentioning design is given below.

Fraser: I just want to make the point that reliability really is a design issue, in the sense that unless you are conscious of the need for reliability throughout the design, you might as well give up.

4.2.4. LOGICAL COMPLETENESS

Logical completeness was put forward as an important design criterion. This gave rise to the following remarks.

Perlis: Logical completeness means that the system must be capable of performing at least a 'basic' set of operations before it can be classified as a system of certain kind. We are not interested in modest systems which do only half the job that one would ordinarily expect; that is too modest. I think that all of us agree that we could list a set of processes that we feel must be included in any language translator, without which we would not call it such. Designing perfect, but incomplete, systems is probably worse than designing somewhat unreliable, but complete, systems.

Genuys: I would like to know what exactly you mean by logical completeness.

Randell: We certainly don't have a complete answer. It is just that you will sometimes see a system that is obviously not logically complete; that can, for instance, produce tapes that it cannot read back.

Ross: The idea is the mathematicians' concept of **closure**, of a group for example, where for every operation you have an inverse operation. It is that idea iterated many times in all parts of the system.

Perlis: Another example: building an assembly system in which the routines that are assembled cannot be entered into the library, except by a totally separate set of actions and tasks, and possibly recoding and so forth.

45

Genuys: I think I would just prefer another term because this one has a certain logical flavor, and I am not certain that …

Perlis: [Interrupting] The word 'logical' has a meaning outside the realm of logic, just as the word 'complete' does. I refuse to abrogate to the specialist in mathematics the word 'completeness' and in logic, the word 'logical'.

Bauer: The concept seems to be clear by now. It has been defined several times by examples of what it is not.

4.3. DESIGN STRATEGIES AND TECHNIQUES

4.3.1. SEQUENCING THE DESIGN PROCESS

The problem of the proper order in which to do things during design is currently a subject for research in software engineering. This was reflected in the extensive discussions during the conference.

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 an 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 really the essential thing, what should you decide first. The approach suggested by Christopher Alexander in his book: *Notes on the Synthesis of Form*, is to make a tree structure of the decisions, so that you start by considering together those decisions that hang most closely together, and develop components that are sub-systems of your final design. Then you move up one step and combine them into larger units, always based on insight, of some kind, as to which design decisions are related to one another and which ones are not strongly related. I would consider this a very promising approach.

David: (from *Some thoughts about the production of large software systems (2)*)

»Begin with skeletal coding: Rather than aiming at finished code, the 46 first coding steps should be aimed at exploring interfaces, sizes of critical modules, complexity, and adequacy of the modules […]. Some critical items should be checked out, preferably on the hardware if it is available. If it is not, simulation is an alternative. The contributions of this step should be insight and experience, with the aim of exploring feasibility.«

Kolence: (from *On the interaction between software design techniques and management problems*)

»Consider how the understanding of the types of structures and relationships that exist in a software product affect the manager's decision on how to begin the development of a design, by establishing the initial design specifications. The most important concepts which must be available to him are those of external product characteristics and internal program design. A set of specifications must be developed on the external features of a product but at the same time, these features must be properly related to the internal design structure of the program to be produced. Unless this relationship is understood, and made explicit, the manager runs the risk of becoming committed to the development of minor external features which may be disproportionately expensive to implement by the desired internal design.

…

The last element of a design methodology to which we will relate the problems of software management is that of defining an overall standard process. The existence of a standard set of steps through which any given software design proceeds to implementation is something which exists in an installation whether or not it has been formally observed and described. Many sets of standard steps may exist, one set for each programmer in the installation. The program manager must determine for each design process what requirements for interfacing exist within his group and with other groups in the installation. If a formalized description of the general process does not exist, then the programming manager is required to re-establish it with each job assignment he makes. «

The specific sequencing principles 'top-down' and 'bottom-up' were introduced as follows.

47

Randell: (from *Towards a methodology of computer systems design*)

»There is probably no single 'correct' order in which to take a series of design decisions, though some orderings can usually be agreed to be better than others. Almost invariably some early decisions, thought at the time to have been clearly correct, will turn out to have been premature.

There are two distinct approaches to the problem of deciding in what order to make design decisions. The 'top-down' approach involves starting at the outside limits of the proposed system, and gradually working down, at each stage attempting to define what a given component should do, before getting involved in decisions as to how the component should provide this function. Conversely the 'bottom-up' approach proceeds by a gradually increasing complexity of combinations of buildingblocks. The top-down approach is for the designer who has faith in his ability to estimate the feasibility of constructing a component to match a set of specifications. The opposite approach is for the designer who prefers to estimate the utility of the component that he has decided he can construct.

Clearly the blind application of just one of these approaches would be quite foolish. This is shown all too frequently in the case of designers who perhaps without realizing it are using an extreme 'bottom-up' approach, and are surprised when their collection of individually optimized components result in a far from optimum system. The 'top-down' philosophy can be viewed mainly as an attempt to redress the balance. In fact a designer claiming to follow the top-down approach, and specifying what a particular-component is to do before he designs the component, can hardly avoid using his previous experience and intuition as to what is feasible.«

In the following passage the 'top-down' approach seems to be taken for granted.

Kolence: (from *On the interaction between software design techniques and software management problems*)

»However, perhaps a more important notational need is for one which permits an initial description of the internal design of software to be broken apart into successively more detailed levels of design, ultimately 48 ending up with the level of code to be used. Current flowcharting practices do not exhibit this property, and so each time a portion of the design is detailed, it no longer fits naturally into the initial design description. In particular, it may be that the entire flow sequence of an area of the design is radically altered when it is re-expressed in more detailed terms.«

The 'top-down' and 'bottom-up' approaches were discussed in a working paper by Gill, reproduced as a whole in section 9. Two remarks are particularly pertinent.

Gill: (from *Thoughts on the Sequence of Writing Software*)

»The obvious danger in either approach is that certain features will be propagated through the layers and will finally cause trouble by proving undesirable and difficult to remove, when they should have been eliminated in the middle layers. … In practice neither approach is ever adopted completely; design proceeds from top and bottom, to meet somewhere in between, though the height of the meeting point varies with circumstances.«

The whole question caused considerable discussion.

Barton: I think at this point in the field we are almost forced to start in the middle, if we are concerned with the problem of general software construction and the organization of the machines to go with these programs. To give an example of how to determine where the middle is: we have seen an enormous variety of programming languages develop, but if you want to get all the ideas in programming you could probably select somewhere between six and twelve languages and see all the ideas fairly well represented in this composite language. Now if one starts with a middle language, which sort of extracts the ideas of programming, and ruthlessly discards all questions of detail in style, such as precedence of operator conventions, then you can do two kinds of things with it. First, this forms the specification of a machine, though at this point we don't know how much of that machine is going to be soft and how much is going to be hard. But some people can go away and work on it. But you can also say to people who are concerned with producing particular processors, where human preferences will get into the language: this is the 49 target language you will work into; go away and implement in this language. Now if you are successful at this in between point you may have achieved the specification of a machine language that would be widely acceptable. If you haven't been quite that successful, you will at least give the software designers who work with the initial machine a kind of minimum task to perform. They can consider this language as a tool they would use in the development of further languages. If you are successful in picking an in between point you will avoid the disadvantages of the propagation of undesirable features up or down.

Gill: I would like to add to my paper by pointing out that one of the problems is that many designers are aware of the danger of propagating features too far through their work and they overcorrect for this by deliberately suppressing hardware features which they think the user ought not to have to worry about. The user then finds that he has absolutely no way of controlling these aspects of the hardware.

Fraser: In the designs I have been involved in, and which have not involved too many people, I have not been able to identify whether these have been 'top-down' or 'bottom-up'. They seem to me to be more like frame stressing, where one is trying to stress a structure with welded joints. You fix all the joints but one, and see what happens to the one, then fix that joint and free another and see what happens to that. It's a sort of iterative process which follows an arbitrary pattern through the structure. Perhaps this only holds for small designs, with few people and good communications. About large designs, I don't know.

Perlis: Fundamentally, the procedure you mention, which is 'fit-and-try', will work very well with three or four people. If you have a hundred people the 'fit-and-try' process diverges because of lack of control.

McIlroy: Talking about where one starts is, I think, perhaps a slight lie. One starts in fact with the grand conception, including the top and the bottom. One can see this in a conception like that of Algol 68. This starts with the top, which is the program, and the bottom, which is the word, the machine word. These are the two fundamental premises, everything else is fitted somehow in between. This is the way a designer starts on Day One. However, he has to do a very large job, and therefore he must structure it in some way. So now we come on the day after Day One to the issues of how to put on a structure.

50

Barton: In the beginning was the word, all right — [*general laughter*] but it wasn't a fixed number of bits!

4.3.2. STRUCTURING THE DESIGN

The structure of the software resulting from the design process was discussed extensively. The background ideas are reported on partly in working papers reproduced in section 9: *E.W. Dijkstra: Complexity controlled by hierarchical ordering of function and variability* and B. Randell: *Towards a methodology of computing systems design*. Other background passages are given below.

Kolence: (from *On the interaction between software design techniques and software management problems*)

»A design methodology, above all, should be coherent. A design expressed in one notation should permit the various functions required to realize a design to be defined efficiently in terms of that notational description of the design. Software design notation, for example, should decompose naturally from the highest level of design description down to design documents which suffice for maintenance of the final software.

…

Other types of relationships should also be considered in making the choice of how to break a design apart and what potential problem areas are implied by any given design decomposition. Currently a software manager is greatly hampered by the use of fuzzy concepts about such things as data structures (particularly files and records) and the properties of the operators over such data structures. In fact, the central concept in all software is that of a program, and a generally satisfactory definition of program is still needed. The most frequently used definition — that a program is a sequence of instructions — forces one to ignore the role of data in the program. A better definition is that a program is a set of transformations and other relationships over sets of data and container structures. At least this definition guides the designer to break up a program design problem into the problems of establishing the various data and container structures required, and defining the operators over them. The definition requires that attention be paid to the properties of the data regardless of the containers (records, words, sectors, etc.), 51 the properties of the containers themselves, and the properties of the data when combined with containers. It also forces the designer to consider how the operators relate to these structures.«

Dijkstra's paper: Complexity controlled by hierarchical ordering of function and variability gave rise to several remarks.

Van der Poel: I agree with the systematics behind the ideas of Dijkstra and his layered structure. In fact when you develop a program systematically in his fashion you have given the proof of its correctness and can dispense with testing altogether. There are, however, a few points to make.

Dijkstra requires that you should be able to verify the correctness of that proof. However, if you insist on having seen every piece of programming yourself, of course you can never step up to a high level of programming. At some stage you will have to believe the correctness of a piece of software which has not been seen or proved by yourself, but by some other party.

Another point is that I think the picture a bit unrealistic on the kind of errors. Errors cut right across the layers, because these are just abstractions. The machine does not know anything about sub-routines, and an error can cut across all these layers in such a way that very illogical results can ensue.

My next point is that I miss an important point in Dijkstra's deductions, and that is he doesn't include the solution of the problem. When he constructs a program then in fact the solution of the problem has been made already. I would like to ask how do we solve the problem. At least I do not know it. When you can visualize a flow diagram or a program in an abstract way, somehow you have solved the problem first; and there is some missing link, some invention, some intuition, some creation process involved which is not easily or not at all symbolised or mechanised. When it's mechanised it is no problem anymore, it's just a mechanical solution of the problem.

Then about errors and error propagation and testing: a program is a piece of information only when it is executed. Before it's really executed as a program in the machine it is handled, carried to the machine in the form of a stack of punch cards, or it is transcribed, whatever is the 52 case, and in all these stages, it is handled not as a program but just as a bunch of data. What happens to errors which occur at that stage, and how can you prove the correctness of all these transmission steps?

Then, as a last question I would like to put the following. The specifications of a problem, when they are formulated precisely enough, are in fact equivalent to the solution of the problem. So when a problem is specified in all detail the formulation can be mapped into the solution; but most problems are incompletely specified. Where do you get the additional information to arrive at the solution which includes more than there was in the first specification of the problem?

Dijkstra: I see in the remarks you made three main elements, one of them the error propagation at a rather mechanical, clerical level. The problem of clerical errors is very serious if it is neglected in practice. On the other hand there are very effective and cheap methods to deal with it by using suitable redundancy.

Next you said that you missed in my description something which you described as, how does one solve a problem. Well, as you, I am engaged in the education business. If I look for someone with a position analogous to the way in which I experience my own position, I can think of the teacher of composition at a school of music. When you have got a class of 30 pupils at a school of music, you cannot turn the crank and produce 30 gifted composers after one year. The best thing you can do is to make them, well, say, sensitive to the pleasing aspects of harmony. What I can do as a teacher is to try to make them sensitive to, well, say, useful aspects of structure as a thinking aid, and the rest they have to do themselves.

With respect to the tackling of an incompletely specified problem, even if your problem is completely specified, the first thing you do is forget about some of the specifications and bring them in later on. It does not in fact make very much difference whether the problem you have to solve is completely specified or not, provided the specifications are not conflicting when the task to be done is subject to alteration. It only means that the things left open will be readily answered in the case of the completely specified one. In a sense, treating the completely specified problem is more difficult because then you have to decide for yourselves which of the aspects of the problem statement you can allow 53 yourselves to forget for the time being. If you have a completely specified problem it means that in the earlier stages of analysis you have to find yourself the useful generalisation of the problem statements. Whereas, if you have an incompletely specified problem you have to solve a class of problems and you start with given class and that's easier.

Randell: Though I have a very great liking for what Dijkstra has done, as usual part of this is because of how he has explained it and part of it is in spite of how he has explained it. There's one particular example I would like to give of this. The word 'proof' causes me to have a sort of mental hiccough each time he uses it. Whenever I try to explain his work to somebody else I say 'satisfy oneself as to the logical correctness of'.

Paul: What is a proof about an algorithm?

McIlroy: A proof is something that convinces other mathematicians.

Perlis: I think that we have gotten switched off the main track, in that Dijkstra's paper has another point besides the idyllic one of proof, and that is that there is also a design process described in the paper. He may have designed that process to make proof easy but I regard that as putting the cart, as it were, before the horse. The design process was organised as a view of constructing a complex system, which is that of building a layering of virtual machines, the organisation being that at one layer something is a variable which at another layer becomes a constant, one layer or many layers below it, and this is the way he chose to design the system. Now that design process is, I think, independently valuable whether you prove anything or not.

4.3.3. FEEDBACK THROUGH MONITORING AND SIMULATION

The use of feedback from a partly designed system to help in design, was discussed at length. The center of interest was the use of simulation during design. This subject was introduced in the working paper: B. Randell: 'Towards a methodology of computer systems design,' reproduced in section 9. This paper gave rise to an extended discussion.

Barton: What Randell is doing is also being done by a small organization in a production environment that can't afford research, and the approach seems to work.

54

Perlis: The critical point is that the simulation **becomes** the system.

McIlroy: I would have much more faith in a manager's statement: 'The system is two weeks from completion' if he had used this technique.

Bauer: But one should be careful not to use wrong parameters during the simulation.

Graham: The important point in Randell's paper is the use of simulation. To-day we tend to go on for years, with tremendous investments, to find that the system, which was not well understood to start with, does not work as anticipated. We work like the Wright brothers built airplanes: build the whole thing, push it off the cliff, let it crash, and start over again. Simulation is a way to do trial and error experiments. If the system is simulated at each level of design, errors can be found and the performance checked at an early stage. To do simulation we should use a high level language and the standard simulation techniques, with the steps:

1) describe functions
2) describe data structures
3) describe as much of the model as you know, guess at the rest; the language will need primitives for description of high-level elements;
4) describe input/output patterns

5) describe variables and functional relations for on-line display.

Galler: Question to Graham: The Multics system of Project MAC was very carefully designed. Was it simulated?

Graham: No, the designers did not believe they could find adequate mathematical models. The decision was not really made consciously.

Haller: There is a special problem in simulating highly parallel processes on a sequential machine.

Wodon: A general point: the amount of simulation is a reflection of the amount of ignorance.

McIlroy: I feel attracted to the simulation approach. But is it not as hard to write the simulation model as the system itself? What about writing the actual system, but starting with dummy modules?

Randell: You have to go pretty far down the design process to be able to use dummy modules. Simulation should come earlier, even at the gross level.

55

Perlis: I'd like to read three sentences to close this issue.

1. A software system can best be designed if the testing is interlaced with the designing instead of being used after the design.
2. A simulation which matches the requirements contains the control which organizes the design of the system.
3. Through successive repetitions of this process of interlaced testing and design the model ultimately becomes the software system itself. I think that it is the key of the approach that has been suggested, that there is no such question as testing things after the fact with simulation models, but that in effect the testing and the replacement of simulations with modules that are deeper and more detailed goes on with the simulation model controlling, as it were, the place and order in which these things are done.

4.3.4. HIGH-LEVEL LANGUAGES

The use of high-level languages in writing software systems was the subject of a debate.

d'Agapeyeff: (from *Reducing the cost of software*)

»In aiming at too many objectives the higher-level languages have, perhaps, proved to be useless to the layman, too complex for the novice and too restricted for the expert.« I maintain that high-level programming languages have, to this extent, failed.

Fraser: Software is generally written in a low-level language. Has anyone written low-level software in a high-level language? Would you do it again?

David: (from *Some thoughts about the production of large software systems* (2))

»Few large systems have been written in high-level languages. This is not surprising since there are inevitable penalties in size and performance of compiled code, and these factors have been paramount in people's minds. In my opinion, this view is no longer appropriate in many instances since techniques are available to overcome these penalties. Secondary memories can take the squeeze out of the size issue, and performance can be brought to a high level with the aid of a traffic analysis of flow of control in the system. Thus, those modules crucial to performance can become 56 the subject of special attention. Indeed, the vast range of programmer performance indicated earlier may mean that it is difficult to obtain better size-performance software using machine code written by an army of programmers of lesser average calibre.

The advantages of coding in a high-level language lie in increased programmer productivity, fewer bugs in the code, fewer programmers required, increased flexibility in the product, 'readability' of the source code ('self-documentation') and some degree (unspecified) of machine independence (better called 'portability'). Many of these advantages have a face validity, which is fortunate since they are difficult to support with hard evidence. There is evidence, however, on the flexibility issue drawn from the Multics experience. Multics is coded almost entirely in a subset of PL/l known as EPL. The early versions of Multics were large and slow. Major improvements were made by improving the EPL compiler (3 times), by capitalizing on experienced programmers' ability to produce EPL code which compiles for fast execution (3-10 times), and by changing strategies in the system to optimize its performance (3-10 times). Important in this process was metering of the system performance. Some idea of the magnitude of the improvements achieved can be obtained from the following: the overall system was at one time well over 1 million words; this was reduced to 300,000 by the combination of measures mentioned above. Further reduction of as much as 100,000 more is thought possible. In certain modules of the system, the improvements to date can be documented by the following approximate figures:

Module	**Size Improvement**	**Performance Improvement**	**Effort**
Page Fault Mechanism	26/1	50/1	3 man-months
Interprocess Communication	20/1	40/1	2 man-months
Segment Management	10/1	20/1	1/2 man-month
Editor	16/1	25/1	1/2 man-month
I/O	4/1	8/1	3 man-months

These figures indicate that major changes can be made in the software without a massive effort. To me, this flexibility is an absolute necessity for software incorporating new concepts, since initial versions must undergo evaluation to reach a satisfactory state. In my opinion, this is 57 true of **any** large software package — it must be coded so as to make evaluation easy.«

Barton: Processors for higher-level languages are written in higher-level languages now. Soon we will have conversational languages for this sort of thing. The thing is that people don't program well. They need the best tools. All questions of efficiency can be handled by improving hardware and generalizing language. At Burroughs we found Algol very successful. Later, influenced by Simula, we thought you have to provide programmers with a still more convenient tool.

Haller: It's not sensible to make software better by making hardware better.

(*d'Agapeyeff:* Agreed.

Graham: Multics is written essentially in a subset of PL/1 except for a very few basic programs. Whether we would do it again: yes. The advantages show up particularly in big projects, with complex tasks. We get increased productivity of programmers. The programs are more easily understood, hence we can move people around easier, or replace them easier. One cannot predict the best techniques in advance, hence there is a need to rewrite parts of the system on the fly, which is easier with a high-level language. The machine code produced is not as good as that of good bit twiddlers, but probably as good as that of the average programmer.

Ross: The AED system is written in itself. I am for using higher-level languages. In this way we really capture what the system is all about and carry it over to the machine. We can build into a higher-level language and related packages a higher level of knowledge than the average programmer possesses, hence the possibility of giving the programmer better programs than he would write in assembly code himself.

Perlis: We tried a formula manipulator based on Algol, using the compiler, but found debugging easier with low-level language. Compilers often do not interface well with machines. It's unfair to say that you wrote a system in a high-level language when in fact you are using only very few features, whereas many things that you need are not there at all.

Randell: In debates like this most people agree on the benefits of high-level languages, but back in the field very few large projects use such languages. I believe the reason is that project managers, if left to make the decision, will decide on the basis of their own small world. They will seek to optimise the easily measurable figures on which they expect to be 58 judged, and will aim to minimise core store used and maximise speed. They will ignore advantages such as portability, ease of recoding, etc., even though in the long term these factors may well be of paramount importance. If such decisions were made at a higher level of the organisation, the outcome would be very different.

This point was also made by Barton, in stating that part of system design is the design of a style which will permeate down through the organisation, quoted in section 4.4.

McClure: I want to defend the Fortran H compiler. It is perhaps the most complex Fortran compiler that has ever been written. It is unfair to compare it with compilers written in machine language, that do not do the same functions. It is indeed doubtful whether it could have been written in anything but a high-level language. I know of other successful uses of high-level languages for writing assemblers and compilers. The point of using high-level languages is to suppress unnecessary detail, which is vital in complex problems.

David: The answer is: use high-level for research production, low-level for commercial production.

Barton: Efficiency depends on the integrated design of the original system, including both its hardware and software.

Kjeldaas: We have made software products using high-level as well as low level languages, and our experience is that the high-level language projects were the most successful ones.

Bauer: People are praising or condemning specific products — what was the cause of the success or failure in these cases?

Kinslow: TSS/360 was done in assembly language, and I would do it again that way, at least at monitor level. Reason: need for efficiency; I am afraid a high-level language wouldn't provide the bit manipulation needed. I don't believe in the suggestion to start designing in, say PL/1, and then shift to assembly code, because PL/1 will affect the design approach. In a monitor one cannot accept any control level between designer and machine.

Smith: There are critical matters, such as basic software for a new machine. Here the software designer must be fully aware of the checked features of the machine, he must discuss all facets of the new machine with 59 the hardware designers and must know and avoid the problematic areas of the machine.

Galler: Let us not forget that macro languages are really high-level languages and are used widely.

Kolence: (from *On the interaction between software design techniques and software management problems*)

» An understanding of the relationships between the types of languages used, and the types of internal designs provided, would be of great value to the software manager. It is simply not true that all internal program designs can be implemented in a higher level language. Many installations settle on a given higher level language by which to implement all systems, and in so doing restrict themselves to implementing a limited structural class of programs. Unknowingly they may be eliminating certain classes of internal designs which can perform their system requirements in a much more effective manner than otherwise possible.«

4.4. COMMUNICATION AND MANAGEMENT IN DESIGN

Several remarks were made on the need for special notations or languages for communication during the design process.

Kolence: (from *On the interactions between software design techniques and software management problems*)

»Another area of difficulty which faces the designer of a program is how to relate the descriptions of the external and internal designs. The notation of external characteristics tends to be narrative and tabular, whereas internal design notation normally consists of flow charts and statements about key internal containers and tables. An external set of characteristics which is appropriately cross-referenced to the internal design, and which clearly illustrates the impact of features or sets of features on the internal design would be of great value to both the designer and the manager.

. . .

The general difficulties are further compounded by the lack of a notation by which the internal design explicitly show the data interfacing requirements 60 between flowcharts at a detailed level. Thus the manager cannot easily determine if various portions of a design mesh properly.

Still a third type of notational need exists. It is well understood that many portions of any given internal design are common to a great number of different programs. An attempt has been made to capitalize on this knowledge by providing system macros at a relatively high level of language statement. However, these macros are not common across machine lines, and indeed are not even common across languages in general. A notation by which to describe these general relationships in a concise, machine and language independent form would be of great value to the software industry by reducing design costs.

Lastly a requirement on notational form exists to permit the design description of a program to be used to define the checkout requirements of a system. Actually, this requirement is on both the external characteristic and the internal structure design notational forms. The problem of programming management, once the system is in the checkout states, are well known. A standard joke in the industry is that a program typically remains 90% debugged for about 25% of the total implementation time. A notational form, in conjunction with a greater understanding of the properties and structures of a program, is required to permit the program manager to properly monitor the progress of checkout.«

David: (from *Some thoughts about the production of large software systems* (2))

»Design Before Coding: The basic structure of the software, including modular divisions and interfaces, should be determined and documented before a coding begins. Specifications for modules and interfaces can be described in English, carefully phrased to avoid ambiguities.«

Gillette: One type of error we have to contend with is inconsistency of specifications. I think it is probably impossible to specify a system completely free of ambiguities, certainly so if we use a natural language, such as English. If we had decent specification languages, which were non-ambiguous, perhaps this source of error could be avoided.

Barton: Putting the integration of programming in machine organizations into the hands of one, or at most a few, people, gives an interesting opportunity 61 to impose styles upon the workers under them. The old question of how do you implement programming systems, is it wise to use higher-level languages and so on, is often debated. I like to note that you can eliminate any need for debate in a working organization by imposing such a style through the initial system design. I have observed that people who have such a style imposed on them spend very little time objecting, it's too late to object. Part of system design is the design of a style which will permeate down through the organization, however large. You can't have anarchy down there, at least you have to restrict the area of anarchy.

Dijkstra: I have a point with respect to the fact that people are willing to write programs and fail to make the documentation afterwards. I had a student who was orally examined to show that he could program. He had to program a loop, and programmed the body of it and had to fill in the Boolean condition used to stop the repetition. I did not say a thing, and actually saw him, reading, following line by line with his finger, five times the whole interior part of his coding. Only then did he decide to fill in the Boolean condition — and made it wrong. Apparently the poor boy spent ten minutes to discover what was meant by what he had written down. I then covered up the whole thing and asked him, what was it supposed to do, and forced out of him a sentence describing what it had to do, regardless of how it had been worked out. When this formulation had been given, then one line of reasoning was sufficient to fill in the condition. The conclusion is that making the predocumentation at the proper moment, and using it, will improve the efficiency with which you construct your whole thing incredibly. One may wonder, if this is so obvious, why doesn't it happen? I would suggest that the reason why many programmers experience the making of predocumentation as an additional burden, instead of a tool, is that whatever predocumentation he produces can never be used mechanically. Only if we provide him with more profitable means, preferably mechanical, for using predocumentation, only then will the spiritual barrier be crossed.

Perlis: The point that Dijkstra just made is an extremely important one, and will probably be one of the major advantages of conversational languages over non-conversational ones. However, there is another reason why 62 people don't do predocumentation: They don't have a good language for it since we have no way of writing predicates describing the state of a computation.

The use of the computer itself to help in the documentation of the design was suggested many times, see particularly section 4.3.3 and 5.3.1. One other remark is given below.

Gillette: In the large, automation has not been exploited very well to aid in the communication process. Some experiments have been made, however. In developing the documentation for OS/360 the programmers had consoles in their offices and they could edit the texts with the aid of the computer. I have read some of the OS/360 documents, and I am not sure the experiment was successful. At Control Data we have used text editors, but there is a great bottleneck, and that is getting the original text into an editable form.

Many problems of software engineering were recognized to be of a general managerial nature, as in the following remark.

David: We must distinguish two kinds of competence, or incompetence, one is in the substance of the subject matter, the other comes in when a person is promoted to coordinate activities. There is a principle, a kind of corollary to Parkinson's Law, called the Peter Principle, named after a high school principal in Long Island. It goes like this: 'In the real world people are eventually promoted to their final level of incompetence'. That is, if a person is extremely competent at the particular level he happens to be working at, he is immediately promoted. This brings upon him additional responsibility. If he does well at that job he is eventually promoted again, and again, until he reaches the level where he no longer performs satisfactorily, and he is never promoted again. So people are left in a state of incompetence. This, in part, is the problem of any big project area.

Samelson: By far the majority of problems raised here are quite unspecific to software engineering, but are simply management problems. I wonder whether all other branches of large scale engineering have the same problems, or whether our troubles simply indicate that programmers are unfit 63 to direct large efforts. Perhaps programmers should learn management before undertaking large scale jobs.

Randell: I have a question on the huge range of variability of programmer performance. Are similar ranges found in other engineering areas?

David: The variation range in programming is in fact greater than in other fields. Certain remarks reflected on the relation between the group structure and the structure of the systems produced.

Endres: It is important that the structure of the system and the structure of the organization are parallel. This helps very much in communication. Communication should follow the same lines along which decisions are made.

Pinkerton: The reason that small groups have succeeded in the past, and that large groups have failed, is that there is a need for a certain structure of communication, and a structure of decision making in the development of software. This succeeds with small groups, because it can all be done intuitively by one person serving as most of the network, but it has failed with the large groups. For large groups to succeed, (and we do need large groups), we just have to face organizational structure for communications and decisions. Second, this does induce a structure on the system. We ought to consider how a system designed with a group with a certain structure might have a reflecting structure.

Randell: As was pointed out by Conway (*How do committees invent?* — Datamation, April 1968) the system being produced will tend to have a structure which mirrors the structure of the group that is producing it, whether or not this was intended. One should take advantage of this fact, and then deliberately design the group structure so as to achieve the desired system structure.

64 65

5. PRODUCTION

5.1. INTRODUCTION

Section 3 of this report contains much material on the place of Production in the total task of designing implementing, delivering, maintaining, etc., a software system. During the conference a working party produced a report, *Classification of subject matter*, which attempted to list and classify the procedures that constitute the production process and the technical components involved in the production task. This report is reproduced in its entirety in Section 9. The conference covered only a small part of the subject matter listed by the working party, so the organization of this section of the conference report does not exactly follow the classification proposed by the working party.

5.1.1. THE PROBLEMS OF SCALE

The rate of growth of the size of software systems is dramatically represented by figure 6 (prepared by McClure). This shows, on a logarithmic scale, the amount of code (compiled instructions) provided as standard programming support (Type 1 programs in IBM terminology) for a variety of computers. The reader is invited to use this chart in order to make his own predictions as to the amount of programming support that will be provided with systems delivered in 1975, and to speculate on the number of systems programmers that might then be required.

Additional data regarding the size of some current systems was given by Nash, in his paper 'Some problems in the production of large scale software systems'. The data, which concerns the number and size of modules in an early version of OS/360 (released in the first quarter of 1966) is reproduced below.

67

Component	**Number of Modules**	**Number of Statements**
Data Management	195	58.6 K
Scheduler	222	45.0
Supervisor	76	26.0
Utilities	86	53.0
Linkage Editor	24	12.3
Testran	74	20.4
Sys. Gen.	32	4.4
Subtotal	709	219.7
Assembly E	32	43.0
Cobol E	77	50.6
Fortran E	50	28.7
Sort	175	56.5
Subtotal	334	178.8
TOTAL	1,043	398.5 K.

The situation that has been brought about by even the present level of programming support was described by one conference member as follows.

David: (from *Some thoughts about the production of large software systems* (2))

»Production of large software has become a scare item for management. By reputation it is often an unprofitable morass, costly and unending. This reputation is perhaps deserved. No less a person than T. J. Watson said that OS/360 cost IBM over 50 million dollars a year during its preparation, and at least 5000 man-years' investment. TSS/360 is said to be in the 1000 man-year category. It has been said, too, that development costs for software equal the development costs for hardware in establishing a new machine line. The commitment to many

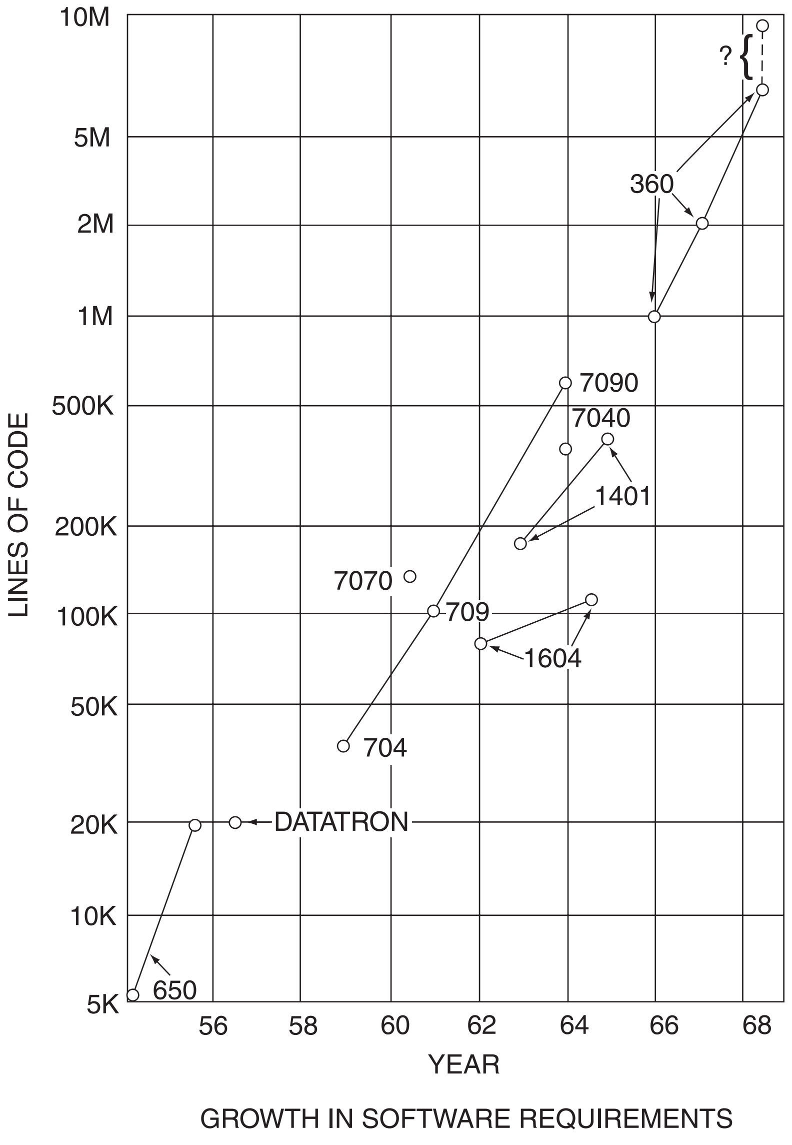

Figure 6. Provided by McClure

software projects has been withdrawn. This is indeed a frightening picture.«

68

As was pointed out by d'Agapeyeff, Europe is not lagging far behind:

d'Agapeyeff: (from *Reducing the cost of software*)

»In 1958 a European general purpose computer manufacturer often had less than 50 software programmers, now they probably number 1,000-2,000 people; what will be needed in 1978?«

The question arose as to whether it was necessary to have large teams on a single project.

Buxton: The good systems that are presently working were written by small groups. More than twenty programmers working on a project is usually disastrous.

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 how to build software systems with hundreds, possibly thousands of people. It is not going to be easy, but it is quite clear that when one deals with a system beyond a certain level of complexity, e.g. IBM's TSS/360, regardless of whether well designed or poorly designed, its size grows, and the sequence of changes that one wishes to make on it can be implemented in any reasonable way only by a large body of people, each of whom does a mole's job.

The term 'the problems of scale' was used to describe the problems of large software systems.

David: (from *Some thoughts about the production of large software systems* (2))

»Regardless of how brave or cowardly the system planners happen to be, they do face difficulties in undertaking large software projects. These have been called 'problems of scale', and the uninitiated sometimes assume that the word 'scale' refers entirely to the size of code; for example, any project of more than 50,000 source statements. This dimension is indeed a contributory factor to the magnitude of the problems, but there are others. One of increasing importance is the number of different, non-identical situations which the software must fit. Such demands 69 complicate the tasks of software design and implementation, since an individually programmed system for each case is impractical.

A related dimension involves the number of different hardware configurations which the software must accommodate, and another is the range of input error conditions which it must handle gracefully. I'm sure you can think of many more. So, the problems of scale grow with many factors in addition to sheer code size. «

Opler: A logarithmic scale in units of man-years is a useful means of characterizing the scale of a software project. People discussing techniques and possible solutions to the various problems associated with the production of software systems should make clear what scale of system they are considering — 1, 10, 100, 1000 ... man-years.

The reasons that scale brought problems in its train were discussed by David and Harr.

David: (from *Some thoughts about the production of large software systems* (2))

»The problems of scale would not be so frightening if we could at least place limits beforehand on the effort and cost required to complete a software task. Experience indicates, however, that in the past (and probably in the foreseeable future) estimates of the effort (man-years) to complete tasks involving new software concepts are likely to be low by factors of 2.5 to 4. Similar factors, perhaps not as great) are common for code size and performance. When one considers that a change of 20-50% in any one of these items can mean the difference between economic and deficit operation, one can indeed sympathize with the person who must commit his company or himself to such a task.

Many factors contribute to this situation. There is no theory which enables us to calculate limits on the size, performance, or complexity of software. There is, in many instances, no way even to specify in a logically tight way what the software product is supposed to do or how it is supposed to do it. We can wish that we had the equivalent of Shannon's information theorems, which tell how much information can be transmitted over a channel of given bandwidth and given signal-to-noise ratio, or Winograd's theorem specifying the minimum addition time, given the switching 70 and delay times in the basic circuitry, but we don't have such existence limits for software.«

Harr: (from *The design and production of real-time software for Electronic Switching Systems*)

»Now why do we often fail to complete systems with large programs on schedule? Most likely some or all of the following events occur during the design process which cause the schedules to be missed.

1. Inability to make realistic program design schedules and meet them. For the following reasons:
   a. Underestimation of time to gather requirements and define system functions.
   b. Underestimation of time to produce a workable (cost and timewise) program design.
   c. Underestimation of time to test individual programs.
   d. Underestimation of time to integrate complete program into the system and complete acceptance tests.
   e. Underestimation of time and effort needed to correct and retest program changes.
   f. Failure to provide time for restructuring program due to changes in requirements.
   g. Failure to keep documentation up-to-date.
2. Underestimation of system time required to perform complex functions.
3. Underestimation of program and data memory requirements.
4. Tendency to set end date for job completion and then to try to meet the schedule by attempting to bring more manpower to the job by splitting job into program design blocks in advance of having defined the overall system plan well enough to define the individual program blocks and their appropriate interfaces. «

5.1.2. THE PROBLEMS OF RELIABILITY

Users are making ever more heavy demands on system reliability, as was indicated by Harr for example.

Harr: A design requirement for our Electronic Switching System was that it should not have more than two hours system downtime (both software and hardware) in 40 years.

71

The subject of the consequences of producing systems which are inadequate with respect to the demands for reliability that certain users place on them was debated at length. This debate is reported in Section 7.1. However, as Smith pointed out, it is possible to over-estimate the user's needs for total reliability.

Smith: I will tell you about an experiment, which was triggered by an observation that most people seem to work under remarkably adverse conditions: even when everything is falling apart they work. It was a little trick on the JOSS system. I had noticed that the consoles we had provided, beautiful things, were hard to maintain, and that people used them even when they were at an apparently sub-useful level. So I wandered down into the computer center at peak time and began to interject, at my discretion, bits into the system, by pressing a button. I did this periodically, once or twice an hour over several hours. Sometimes they caused the system to go down, leading to automatic recoveries, and messages being sent out to the users. But the interesting thing was that, though there are channels for complaints, nobody complained. This was not because this was the normal state of things. What you may conclude seems to be that, in a remote terminal system, if the users are convinced that if catastrophes occur the system will come up again shortly, and if the responses of the system are quick enough to allow them to recover from random errors quickly, then they are fairly comfortable with what is essentially an unreliable system.

Other quotations on the subject of total system reliability included:

d'Agapeyeff: (from *Reducing the cost of software*)

»Engineering advances and back-up faculties through duplex equipment have reduced the danger of hardware faults and increased user expectancy of reliability (especially in on-line real time installations) but programming has grown more complex and unreliable while software has not provided analogous back-up facilities.«

Harr: Time-shared systems need a level of reliability far beyond that which we have been getting from our batch-processing systems. This can be achieved only by a careful matching of hardware and software. The essence of the approach that we used to achieve high reliability in our Electronic 72 Switching System was to plan on the basis that even when the system went live it would still have errors in it. Sure enough, it did.

d'Agapeyeff: I agree with this approach. You must design and implement your system so that it will continue to give service in the presence of both hardware and software errors.

Harr: It requires a very careful program design to ensure that your crucial system tables will not be mutilated except in the very rarest of types of error situation.

Bemer: One interesting question is how much the software people are prepared to pay for extra hardware to assist in maintaining data integrity. My guess would be perhaps 15-20 % of the normal hardware cost.

Harr: We definitely need better techniques for changing a program in the field while continuing to provide service.

d'Agapeyeff: In large on-line systems the cost of testing a change is almost 100 times as much as the cost of producing the change. We cannot afford to go through this process too often.

One final pair of quotations are given below, to indicate the need for quantifying our notions of reliability.

Opler: How many errors should we be prepared to accept in a system containing one million instructions?

Perlis: Seven [*Laughter*].

5.2. PRODUCTION — MANAGEMENT ASPECTS

5.2.1. PRODUCTION PLANNING

There was much discussion on the difficulties of planning the production of a system that involved a high research content.

Genuys: If the job has been done before, estimates are fairly easy. If the research content is high, estimates are difficult. The trouble is that it is not always possible to tell beforehand which jobs are which.

McClure: It's research if it's different. Even if it is just higher performance, it is still research.

73

An illustration of how the original estimate of the work involved in producing a particular piece of software gets more reliable as the group of people doing the work gains experience with that particular problem was provided by McClure (figure 7). This shows, for three Fortran compilers produced by the same group, both the original estimate of the work required and the work actually needed.

David: (from *Some thoughts about production of large software systems* (2))

»Computing has one property, unique I think, that seriously aggravates the uncertainties associated with software efforts. In computing, the research, development, and production phases are often telescoped into one process. In the competitive rush to make available the latest techniques, such as on-line consoles served by time-shared computers, we strive to take great forward leaps across gulfs of unknown width and depth. In the cold light of day, we know that a step-by-step approach separating research and development from production is less risky and more likely to be successful. Experience indeed indicates that for software tasks similar to previous ones, estimates are accurate to within 10–30% in many cases. This situation is familiar in all fields lacking a firm theoretical base.

Thus, there are good reasons why software tasks that include novel concepts involve not only uncalculated but also uncalculable risks.«

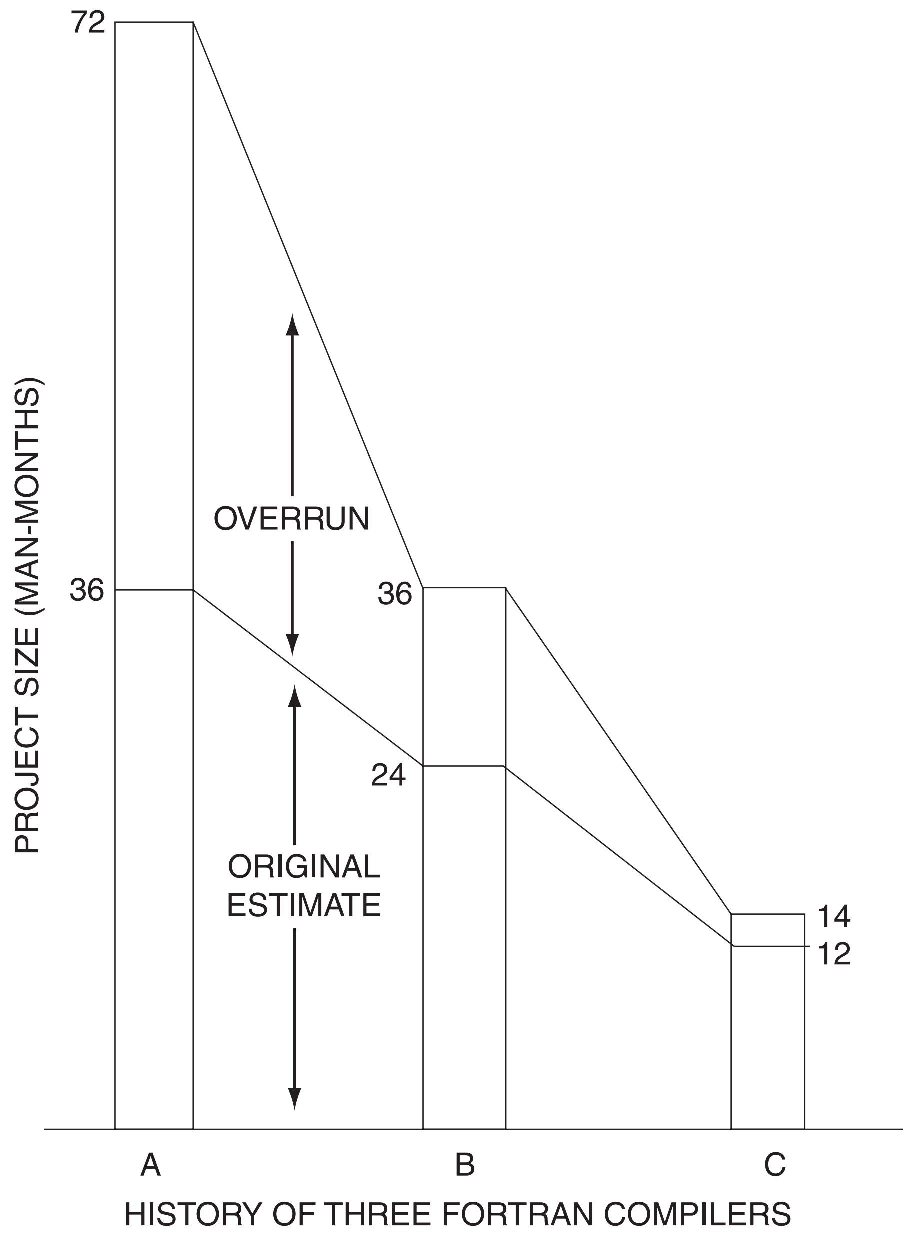

Figure 7. Provided by McClure

77

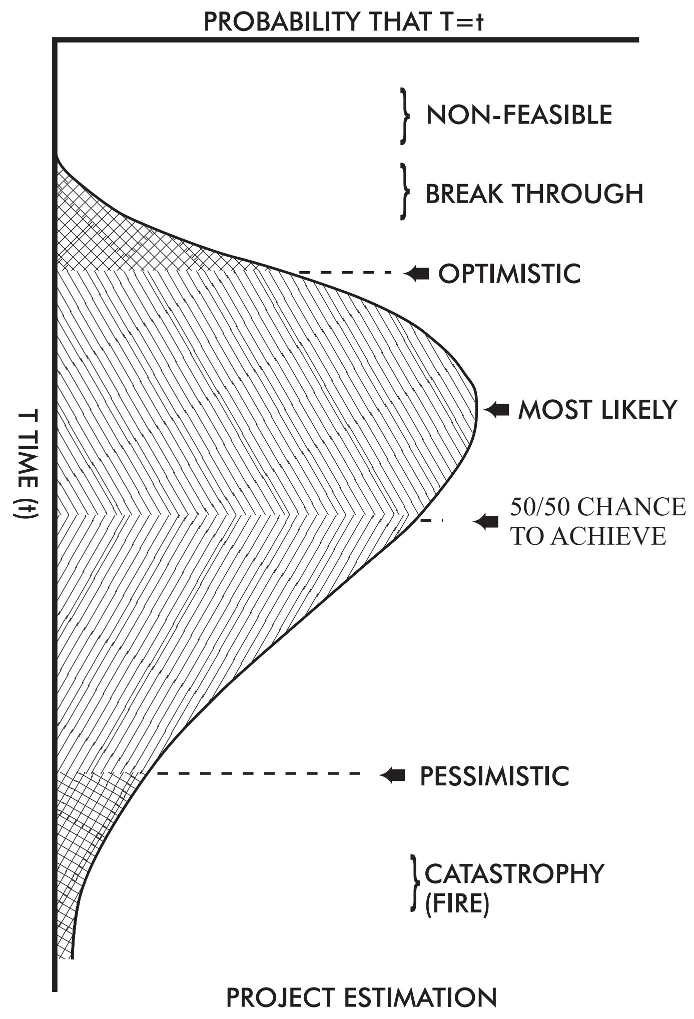

Figure 8. From Nash: Some problems of management in the production of large-scale software systems.

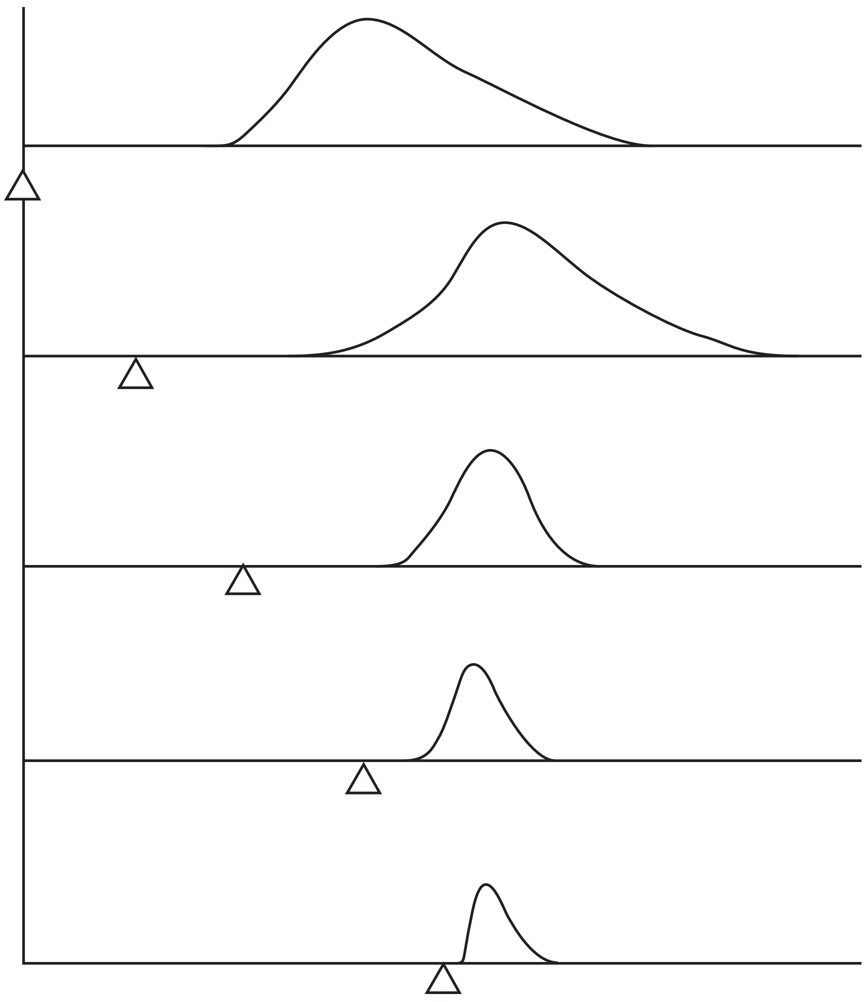

Figure 9. From Nash: Some problems of management in the production of large-scale software systems.

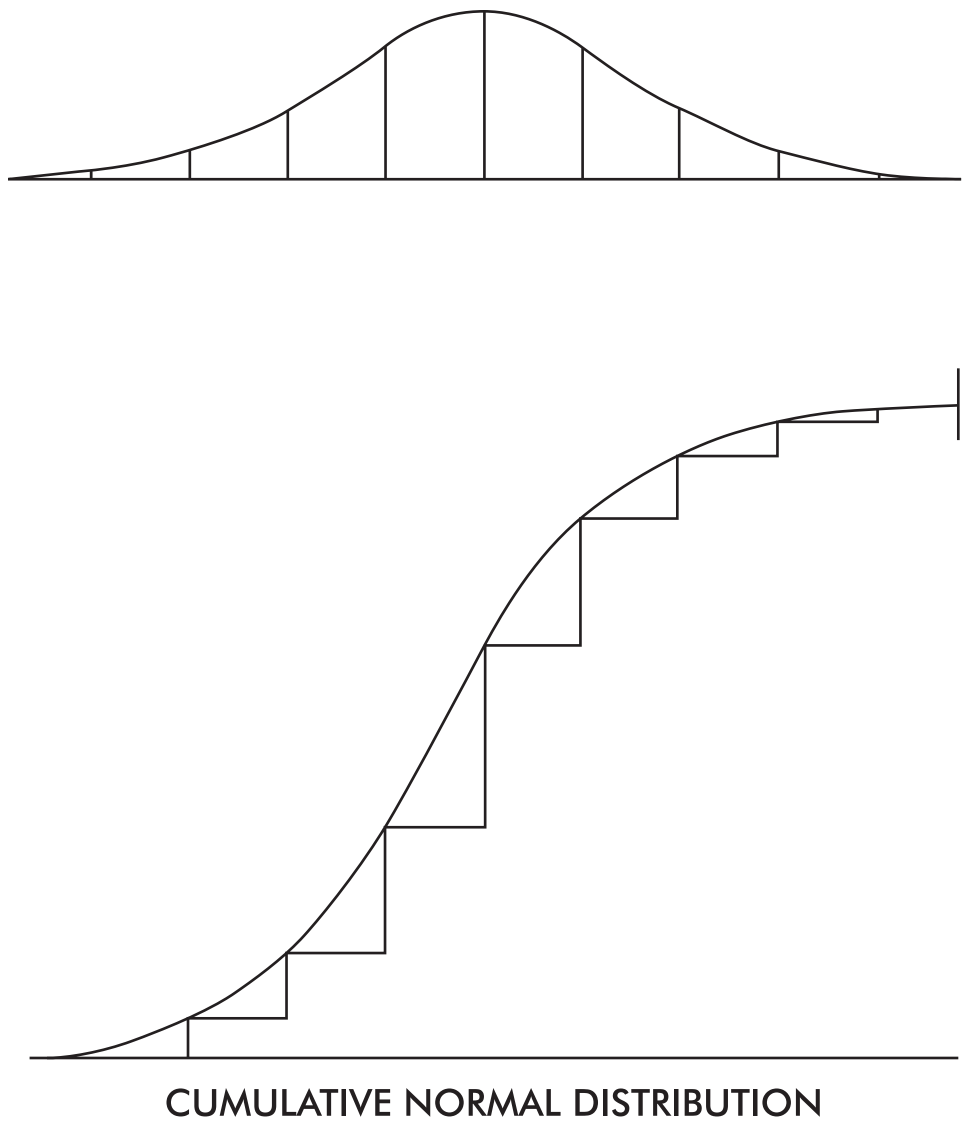

Figure 10. From Nash: Some problems of management in the production of large-scale software systems.

81

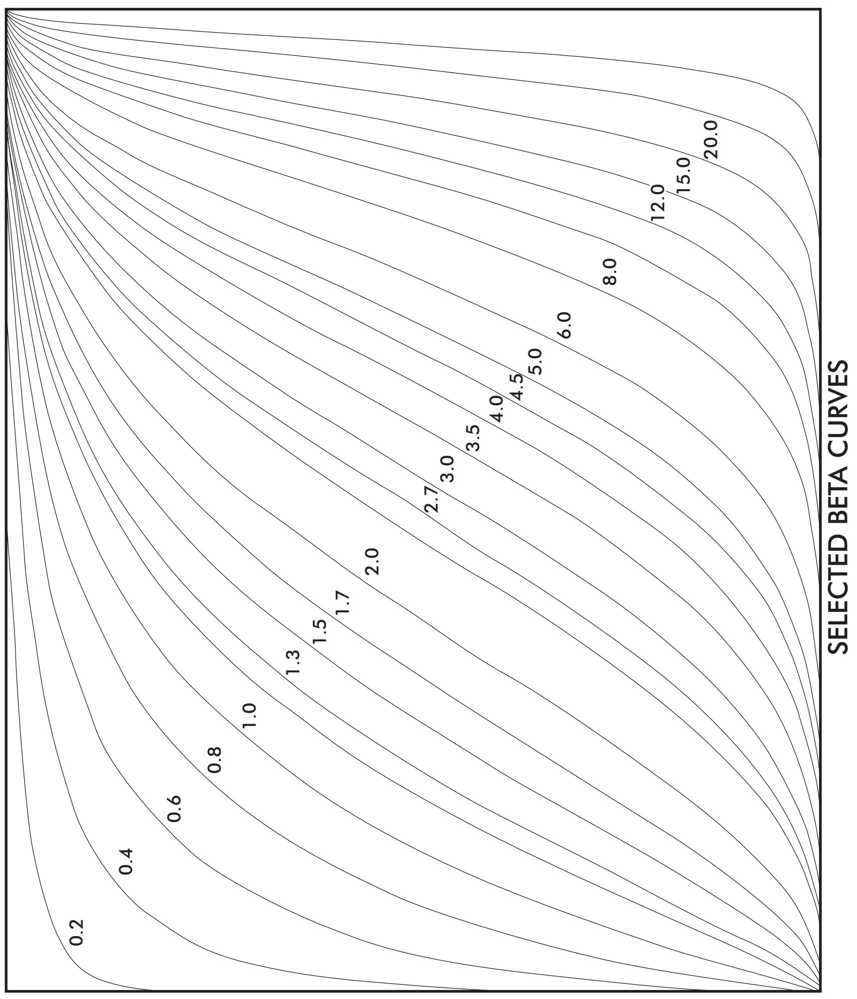

Figure 11. From Nash: Some problems of management in the production of large-scale software systems.

81

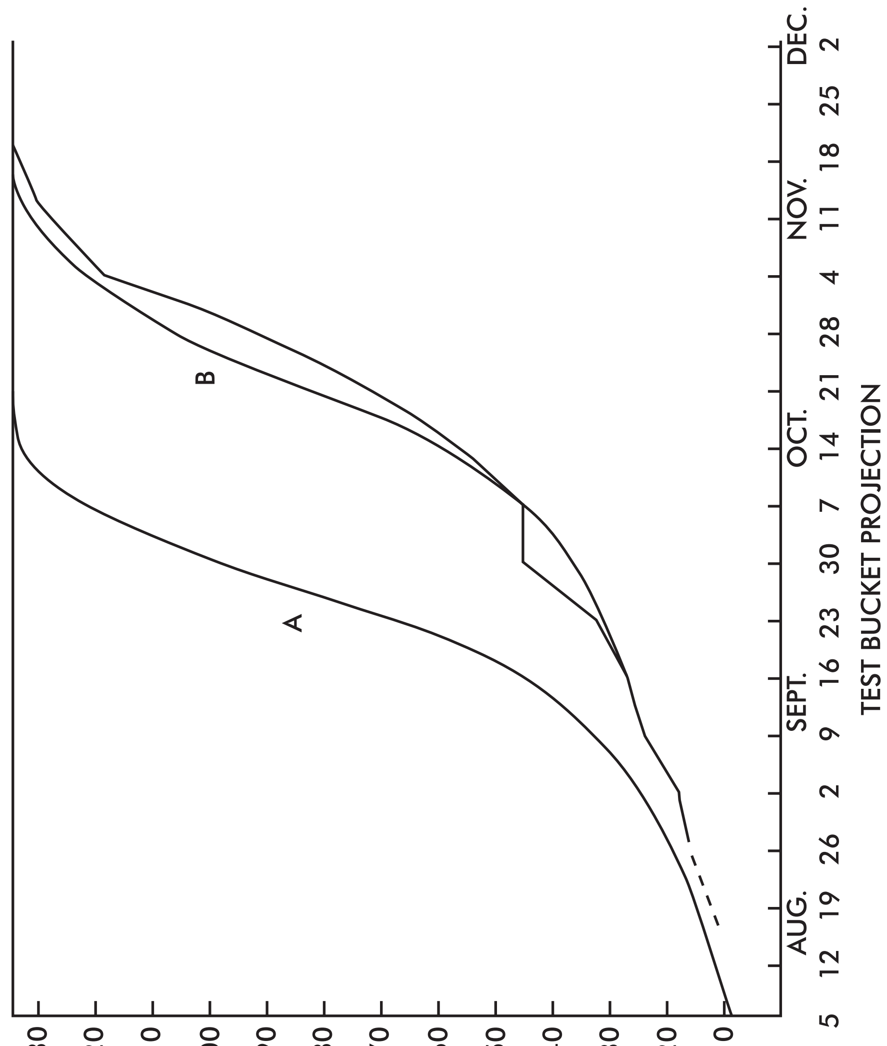

Figure 12. From Nash: Some problems of management in the production of large-scale software systems.

It was however pointed out by McClure that research content was not the only problem in preparing estimates:

McClure: (from *Projection versus performance in software production*)

»Before undertaking a project to produce a new software system or component, read the following statements, checking all that apply:

1. The new system will be substantially superior to its predecessor and to competitive systems.
2. The new system corrects a basic philosophical defect in the previous system.
3. The specification is not yet complete, but it will be finished before any important programming decisions are made

75

4. The specification is absolutely firm, unless, of course, the Acme Company signs a big order and requests some slight changes.
5. The programming team will be made up by selecting only the best programmers from other projects.
6. Because of expansion, a fresh team of programmers with applicable experience will be hired.
7. The new computer is a great machine; the programmers will love it as soon as they can get their manuals.
8. The programmers will, of course, have to share the machine with the hardware team checking out the new peripherals and the diagnostic package.
9. Interfacing this system to the rest of the software is trivial and can be easily worked out later.
10. Although the assembler (compiler, loader, file system, etc.) is not completely checked out, it will be ready long before coding is complete.
11. The debug package isn't done but this system can easily be checked out at the console.
12. The budget is only preliminary but it's obviously conservative.
13. The project manager may have missed his budget on his last project, but he has learned his lesson and won't miss this time.

For each statement checked on the preceding list add ten percent to the estimated cost and one month to the estimated time. If statement six is checked, add thirty per cent and six months. The result should come much closer to the final result than the original estimate under the assumption that the original estimate was honestly made.«

The major contribution on techniques for estimating the amount of time it will take to complete a system was by Nash.

Nash: (from *Some problems of management in the production of large-scale software systems*)

»Consider one of the most difficult problems facing producers of large software systems, namely, estimating the time required for completion. Suppose we could establish 100 equal teams to implement the same system, in as nearly equal conditions as could be achieved. Then the times actually required by the various teams would follow some distribution, 76 which might be similar to that in figure 8. Even if we had such data to draw on, it is still no simple task to estimate a completion date for the same project thereafter. A sound approach would be to use the most likely time at the outset, but to take a progressively more conservative view as the development proceeded. The spread of the distribution curve would reduce as the end-point was approached as in figure 9, and at the point where a commitment should be made, a 90% probability would be more reasonable.

However in practice no such data exists and we have to rely on experience and judgement to assess the likely nature of such distributions. How, then, can commitments be made with any degree of certainty? There is one valuable tool available in the later parts of a development cycle, namely Test Status. In any large-scale software production, extensive testing is necessary, which normally takes the form of a large number of carefully designed test-cases to run successfully before the system can be released. The test progress consists of a number of discrete events, i.e. test-case successes, which tend to occur in some pattern, and which pattern can be predicted before-hand. If the pattern of successes followed a normal distribution, the test history curve would follow the Cumulative Normal Distribution curve, figure 10. Other distributions produce other cumulative curves, of which a selection is shown in figure 11. From studying test histories of previous developments, a judgement can be made about the nature of the curve applicable to a particular project. For most new projects curves around the 2.5 value are found to be applicable. For improvements to an existing product, successes tend to occur early on, and curve 1.0 can be used. In the case of an existing product being adopted for new environment, for example converting a language processor to run under a different operating system, successes are slow to start but rise quickly once under way, as typified by the higher valued curves.

As an example of the use of this technique, figure 12 shows the actual case history of a project, which was a major extension of a large component of OS/360.

The original test prediction, curve A, was made at the beginning of August. By mid-September it was clear that actual progress, shown by the irregular line, was lagging about 2 weeks behind the plan. The end-date 82 was therefore postponed by four weeks, to allow for the cumulative effect of the lag, and a new prediction made, curve B. In October, it was again evident that the plan was not being met, but by working a substantial amount of overtime, this was corrected to reach the required objective.

There are several important conditions that must be observed if this technique is to be of any value. First, the system build plan must be such that simple test-cases can be run as early as possible, so that testing stretches over a maximum period. This allows the measurement of progress against prediction to give warnings of trouble at an early stage. Second, the set of test-cases must be constructed so that simple tests can be used in the early stages, building up progressively to more complex cases. Third, for the method to work smoothly, careful measures must be taken to avoid changes to the build system causing serious regression, i.e. the familiar event where a correction introduces further errors. With these provisos, test planning and status control can be a valuable tool in managing the later part of a development cycle.«

There was some discussion of the problems of estimating the costs of software projects:

Ercoli: I maintain that if proper management and accounting methods are used so that a software job can be considered as composed by well defined and normalized parts then the cost per instruction in implementing each part is roughly $5 per debugged instructions. This is also the average cost of software for known applications and for efforts below the limit of about 15 man-years. If the general cost of software production is considered then there are wide variations: as low as $1 per instruction using semiautomatic means of software production (such as compiler generators) or as high as $20 per instruction in cases of very large operating systems. I would have liked to have replaced these figures by a rough graph on semilogarithmic paper indicating cost per instruction against size of project but it was lost with my luggage.

Endres: I believe that cost/instruction can vary by a factor of 50.

Salle: It is very difficult to use cost per instruction in estimates because there is no general agreement on what the average cost per instruction is, and also because the ratio varies according to which phase of the production process your are in. The number of tables, number of interfaces, 83 etc., are at least as important for costing as the number of instructions. If these figures are to be used for planning they should be correlated with a number of other important parameters such as range of hardware configurations, number of customers, research content, etc. In fact I think cost/instruction should be the last thing one uses for planning purposes.

Barton: I've just realized that the cost/instruction measure has to be meaningless. Now I don't know the right measure but I suggest as a first approximation to one a weighting which would take into account the number of times the instruction had been used.

5.2.2. PERSONNEL FACTORS

Many of the problems of producing large software systems involve personnel factors:

David: (from *Some thoughts about production of large computer systems* (2))

»Also, system programmers vary widely in their productivity. Figures from an experiment by SDC indicates the following range of performance by 12 programmers with 2 to 11 years' experience in completing the solution to a specific logic problem (Comm. ACM 11 (1968), 6):

Performance on	**Worst/Best**
Debug Time Used	26/1
Computer Time Used	11/1
Coding Time	25/1
Code Size	5/1
Running Time	13/1

These figures confirm my own informal observations. Of course the importance of individual talent to projects generally is widely appreciated and is traditionally reflected in promotions and dismissals. Yet, in software projects, talent is often so scarce that marginal people are welcomed. More to the point, the range of productivity indicated above seems quite large compared to what one might expect or tolerate on a hardware project, increasing the difficulty of accurate estimation.«

Fraser: I would never dare to quote on a project unless I knew the people who were to be involved.

84

Opler: In managing software projects the real problem is 'spot the lemon and spot the lemon early'.

This started a somewhat surprising line of conversation.

David: I have heard rumours that the Japanese are extremely good at producing software.

Perlis: There is apparently some truth in it. I believe it to be due to three factors.

1. A man stays at one job for his entire working life — there is no concept of mobility.
2. They have a tremendous amount of self-discipline.
3. The complexities of their language give them good preparation for the complexities of software systems.

Opler: I agree — yet by American standards they work long hours for low pay.

There were several comments on the personnel factors involved in very large projects:

David: (from *Some thoughts about production of large computer systems*)

»Among the many possible strategies for producing large software systems, only one has been widely used. It might be labeled 'the human wave' approach, for typically hundreds of people become involved over a several year period. This technique is less than satisfactory. It is expensive, slow, inefficient, and the product is often larger in size and slower in execution than need be. The experience with this technique has led some people to opine that any software system that cannot be completed by some four or five people within a year can never be completed; that is, reach a satisfactory steady-state.«

d'Agapeyeff: There is the problem of morale. For example I have no idea how you keep your people when you try the 'human wave' approach to the production of large systems. In my experience programmers are rather difficult to shove around like platoons in the army.

The benefits of the small team, in those situations which it is sufficient for, were discussed by Babcock and by Cress and Graham.

85

Babcock: (from *Variations on software available to the user*)

»Before I leave the software area, let me discuss briefly the staff. We selected a few very highly talented, creative people who 1. designed the system, 2. implemented the system, 3. continually developed and researched new capabilities for the system.

The staff comprised over a century of programming experience and I feel this tight control over the entire production is one of the major factors in the success of Rush. No minor programmer ever wrote a bug fix, no minor programmer ever maintained the system. From the beginning, it has been a group of experts who followed every development to the finish.«

Cress and Graham: (from *Production of software in a university environment*)

»Much of the design and programming is done by students. They are eager to learn, full of energy and add a vitality to the project which keeps it alive. But they must be supervised!

We have never experienced personnel turn-over during the development of software. This is probably because of the desire to be successful, coupled with the relatively short duration of the projects. We try not to hire people who want a job; instead we try to hire people who want to do the job under consideration.

A common property of all our project groups has been that they have been small, and faced with a relatively well-defined task. We find that, with the objective always in sight, the spirit of the group can be maintained.«

Finally, we include comments by Perlis on how the realities of personnel factors might be faced in structuring a large group of people:

Perlis: A man can communicate with about five colleagues on a software project without too much difficulty. Likewise he can supervise about five people and know pretty well what they are doing. One would structure 120 people in three levels, in which no man is talking to more than about eight people, both across his level and up and down — which is well within our capabilities I think, since most of the communication will go across rather than down. We have confused the simplicities of administration with 86 the aims of production. When someone comes into a structure we naturally assume that he starts at the bottom and works his way up, whereas the bottom is the place where the information is most difficult to get at in order to find out what you should be doing. So a new person should start perhaps one level down from the top. Another interesting concept we might apply is that used in the Air Force, to fly a number of hours each month, in order to retain one's 'wings'. So, in a system which will take a long time to complete, for example a year, nobody should be allowed to function permanently at one level, but should percolate. In a situation where code actually has to be produced, nobody should be allowed in the system who doesn't write some given number of lines of code per month. I think that one of the major problems with the very large programming projects has been the lack of competence in programming which one observes as soon as one goes above the very bottom level. People begin to talk in vague general terms using words like 'module', and very rarely ever get down to the detail of what a module actually is. They use phrases like 'communicate across modules by going up and then down' — all of the vague administratese which in a sense must cover up a native and total incompetence.

5.2.3. PRODUCTION CONTROL

One of the big problems of controlling the production of a large system is obtaining information which will reveal how much progress has been made.

Fraser: (from *The nature of progress in software production*)

»One of the problems that is central to the software production process is to identify the nature of progress and to find some way of measuring it. Only one thing seems to be clear just now. It is that program construction is not always a simple progression in which each act of assembly represents a distinct forward step and that the final product can be described simply as the sum of many sub-assemblies.

…

Various attempts were made at measuring progress without much success. As the work advanced we evolved a simple method of accounting for the components that went to form or support the final product. But this approach was no better than the various 'seat of the pants' methods which 87 are still much in evidence. We experienced one of the classic problems of metering: that of interference between the measuring device and the process that is being measured. At one time we kept totals of the number of subroutines that were classed as 'tested' and compared this number with the total number of subroutines in the final product. But, once a programmer's performance is formalised in this way it changes his attitude to his work. The effect was to produce an apparent increase in productivity but this was accompanied by a drop in quality. Of course, the latter did not come to light until very much later.

Towards the end of the production task, when most of the required techniques had been developed, it became very much easier to measure performance. At this stage we were better able to assess the performance of the programmers individually and the methodology had become quite standard throughout the group. The only confidence that I can gather from this experience is that of estimating the production rate for known men performing a known task that relies only on a known technology.«

Kolence: The main interest of management is in knowing what progress has been made towards reaching the final goal of the project. The difficulty is to identify observable events which mark this progress.

David: Psychology is a central point in the problem of measuring progress. Instead of looking for nice quantifiable measures, why not just ask people to give their own estimate of the amount of progress that has been made. In a controlled environment you will soon have enough evidence on which to calculate how the various individuals' estimates should be weighted.

Fraser: (from *The nature of progress in software production*)

»Perhaps the most effective way of assessing the progress of a team project is to study the interface documentation. In the early stages particularly, a programming task resembles frame stressing by relaxation. The progress of the relaxation process can be assessed by studying the magnitude and pattern of successive adjustments to the structure. Similarly, the adequacy and stability of interface details provide one good measure of project status. One might even suggest, a little dangerously perhaps, that rapid change to interface descriptions is a sign of good progress and that the work content of a development project could be measured by the number of changes that must occur during the process of design development.«

88

Harr: (from *The design and production of real-time software for Electronic Switching System*)

»Each program designer's work should be scheduled and bench marks established along the way so that the progress of both of his documentation and programs can be monitored. (Here we need a yardstick for measuring a programmer's progress other than just program words written.) The yardstick should measure both what has been designed and how, from the standpoint of meeting the design requirements. Programmers should be required to flowchart and describe their programs as they are developed, in a standard way. The bench marks for gauging the progress of the work should be a measure of both the documents and program produced in a given amount of time.

A standard for program documentation when programs are written in symbolic machine language should be set and each program should include this standard documentation in the 'remarks field' of the symbolic program.

This documentation should include sufficient information and in such a way that a computer program can flow trace any program according to specified conditions.

Then a computer program can be used to assist in evaluating and checking the progress of the program design. Computer studies of the realtime used under varying conditions can be made and studies to see that memory interfaces are satisfied can be made.«

McClure: I know of one organisation that attempts to apply time and motion standards to the output of programmers. They judge a programmer by the amount of code he produces. This is guaranteed to produce insipid code — code which does the right thing but which is twice as long as necessary.

The difficulties of measuring real as opposed to apparent progress were made clear by Smith:

Smith: 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, 89 etc., etc. This activity is represented by a PERT chart with many nodes. 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.

The analogy with engineering drawings, however, raised the question of measuring the quality of work produced, as well as progress towards the project goal.

McIlroy: I think we should consider patterning our management methods after those used in the preparation of engineering drawings. A drawing in a large organisation is usually signed by the draughtsman, and then after that by a draughting supervisor when he agrees that it looks nice. In programming efforts you generally do not see that second signature — nor even the first, for that matter. Clarity and style seem to count for nothing — the only thing that counts is whether the program works when put in place. It seems to me that it is important that we should impose these types of aesthetic standards.

McClure: I know of very few programming establishments where supervisors actually bother to read the code produced by their staff and make some effort to understand it. I believe that this is absolutely essential.

Buxton: I know of very few programming establishments in which the supervisor is capable of reading code — some present company excepted!

5.2.4. INTERNAL COMMUNICATION

There was considerable discussion on the means by which programmers working on a project might communicate with each other.

Buxton: We could use more and better and faster communication in a software group as a partial substitute for a science of software production. We can't define the interfaces, we don't really know what we're doing, so we must get in a position where we can talk across the interfaces.

Dijkstra: I believe that both the total density of information flow necessary between groups, and the percentage of irrelevant information that a given group gets, can be greatly reduced by effectively structuring the object to be constructed and ensuring that this structure is reflected in the structure of the organisation making the product.

90

Gillette: An attack on the problem of communication is crucial for successful production. We are not using automation (remote consoles, textediting, etc.) as much as we should.

Buxton: I know myself that if I'm setting up a software group to carry out a project I'm extremely careful that all the people working on it are close personal friends, because then they will talk together frequently, and there will be strong lines of communication in all directions. One in fact uses personal relationships to support technical communication.

Nash: There are dangers in uncontrolled mass-communication. You can get into trouble if people start taking advantage of information that they gain by chatting that they should not know (and which may well lose its validity in a day or so).

A detailed discussion of the importance of careful documentation was given by Naur:

Naur: (from *The profiles of software designers and producers*)

»In order to characterise the work of the production programmer I would first of all stress the importance of documentation during production. Both the need for documentation of software systems, and the difficulties in filling these needs, are well known items in software work. It is my experience that the most constructive way to solve this problem is to insist that the production of essential parts of the documentation is a natural part of the production of the software program, and that the two proceed in parallel.

In discussing documentation one should keep in mind that this is aimed at the human reader, and should be developed along the principles of report writing set forth in several different texts on the subject. Of particular significance is the insistence, among competent report writers, that reports be structured hierarchically and written from the top of the hierarchy, i.e. starting with a brief synopsis. I feel strongly that in software production this principle should be followed carefully, at all levels of the work. If this is done, the first thing to be done by the software producer about to start writing a piece of software, is the writing of a synopsis of what his piece of program is supposed to be doing. The next level may consist of a description of a few pages describing the essential data structures and the major processes to which they will be 91 subjected. This description should include carefully selected examples, to illustrate the functions and their most important variations (these will be useful later as test cases). The lowest level of the hierarchy is the program text itself.

This way of work not only has the advantage that important parts of the documentation are actually produced. It also leads to better programs. In fact, when working out the higher level descriptions in written form, the software producer inevitably will be forced to think out his data and program structures more carefully than otherwise. This regularly leads to programs with clearer structure, higher efficiency, and fewer bugs.

This way of developing the software and its documentation also allows for mutual review, check, and criticism within small groups of software programmers. This should take place frequently while the work is in progress and can very well be done within groups of two people who look into another's work. In my experience this is a highly effective way of organizing the software work.«

David: One has to avoid flooding people with so much information that they ignore it all. Selective dissemination of information, by means of a system such as Mercury (see W.S. Brown, J.R. Pierce, J.F. Traub: The Future of Scientific Journals, Science, December 7967) at Bell Laboratories should be tried in a large software project.

Randell: It is relatively easy to set up a communication system, manual or automatic, which will let me find information that I already realise I need to know. It is more difficult to make sure I also get information which I need, but of whose very existence I am ignorant.

Finally, several people gave interesting case histories of the methods used for internal communication within a production team.

Opler: I think I know how to organise reasonably successful communication for projects of between 10 and 50 people. I am quite sure I don't know how to do it with projects of much greater size. The method we used was as follows. From the moment the project is created every member of the staff receives a three-ring binder and perhaps half-a-dozen pages stating the very first decisions and ground rules for the project, including an 92 index. As the project proceeds everybody contributes sheets, which must be countersigned by their management. As the project grows so does the notebook. Hopefully the document classification system remains adequate. We had at most a one-day delay between origination of a document and distribution to all members of the group. This had interesting side-effects. I noticed that one part of the book was not filling in very fast — this led to early discovery of a worker who was lagging behind, and who eventually had to be dismissed.

Fraser: The question of what methods should be used for organising information flow between members of a production team depends largely on the size of the team. I was associated with a 30-man project, producing a commercial compiler (NEBULA) for the I.C.T. ORION Computer. We had three, or rather four, forms of information flow. The first was based on the fact that the compiler was written in a high-level language and hence provided, in part, its own documentation. The second form of information flow was based on documentation kept in a random access device which was regularly accessed by every member of the team. This was a steel filing cabinet kept in my office. It contained files consisting of scruffy bits of paper with various notes and messages from one member of the team to another. This was probably the most important form of communication we had. Its merits were that there was only one set of authoritative information, and that the indexing scheme, albeit crude, was sufficient to allow one to find, in most cases, the relevant information when you needed to make a decision. I don't see any advantage in automating such a system for a group of 30 men. A filing cabinet is easy to use, and there were not many occasions when there were long queues outside my office.

The other filing system we had was an automated text-handling system, in which we kept the official project documentation. This was not much use for day-to-day communication, but invaluable for program maintenance and also program revision in the light of further developments. There was a fourth communications mechanism which every project has, and which perhaps doesn't get encouraged as much as it should be. There are certain people in any organization who are remarkably effective at passing gossip. Many of the potential troubles in a system can be brought into the open, or even solved, by encouraging a bit of gossip.

93

Nash: I would like to report on documentation of the F-level PL/1 compiler, where we had a team of about two dozen people on the actual development. We did not have any private memos, or notes, although we had a considerable amount of verbal communication. What we did establish was a book which described in complete detail every part of the compiler. All members of the team were obliged to describe their parts of the compiler by means of flow-diagrams and English prose in complete detail at design time. The book grew very large — eventually to about 4000 pages. It was a lot of work to maintain it — perhaps 10–20% of our total effort — but it was worth it.

d'Agapeyeff: We make great use of seminars as an aid in preventing disasters, and in determining what to do when a disaster occurs. To participate in such seminars it is necessary to communicate fully and to a greater degree than has been possible here. For example, if I were suddenly to recruit you lot and form a rather good software house it would be excellent publicity, but it would not actually work. It certainly wouldn't work at first, because you do not have a sufficient level of communication. One way to obtain this is by a commonality of experience. This is a major difficulty because it leads exactly to the point made by Buxton. It encourages you to work with your friends. But you have to remember that those who are incompetent find each other's company congenial.

5.3. PRODUCTION — TECHNICAL ASPECTS

5.3.1. TOOLS

As a partial solution to the problems of making a success of the 'human-wave' approach to producing large Systems, David suggested:

David: (from *Some thoughts about production of large software systems* (1))

»A reliable, working system incorporating advanced programming and debugging tools must be available from the beginning. One requirement is an accessible file system for storing system modules and prototypes. An adequate file system should act as a common work space for system programmers 94 and should have back-up facilities to insure against loss of valuable records.«

The most ambitious plans for a set of tools to aid in the production of large systems that were presented at the conference were those contained in a working paper by Bemer.

Bemer: (from *Machine-controlled production environment*)

»Tools for Technical Control of Production

1. Goals
   - a. Maximizing programmer effectiveness and personnel resources.
   - b. Minimizing time and costs for original production, changes and checkout.
   - c. Maintaining the best-conditioned system from a quality viewpoint.
2. Attainment

   By utilizing the machine-controlled production environment, or software factory. Program construction, checkout and usage are done entirely within this environment using the tools it contains. Ideally it should be impossible to produce programs exterior to this environment. This environment should reside on the computing system intended for use, or in the case of manufacture of a new system, on the most powerful previous system available.
3. Functions Provided
   - a. Service
     - (i) Computing power and environment
     - (ii) A file system
     - (iii) Compilation
     - (iv) Building test systems
     - (v) Building final systems and distribution

(vi) Information during the process
- Listings/automatically produced flowcharts/indexing
- Index and bibliography of software units
- Directed graph of system linkages
- Current specifications
- User documentation, text editing
- Classification of mistake types
- Production records to predict future production

(vii) Diagnostic aids

(viii) Source language program convertors

(ix) File convertors

95

b. Control

(i) Access by programmer

(ii) Code volume

(iii) Documentation matching to program

(iv) Software and hardware configurations, and matching

(v) Customizing

(vi) Replication and distribution

(vii) Quality Control

(viii) Instrumentation

(ix) Labor distribution

(x) Scheduling and costing«

Buxton: I would be interested to know how much of the system described by Bemer is actually working.

Bemer: We have about a quarter of it presently built and working. It is a very large project. Many improvements are already seen to be necessary — such as in terminal equipment.

Opler: IBM is also developing such a system. The cost is enormous, and a vast amount of hardware is needed.

Fraser: I welcome Bemer's system as a long term project, but I think pieces should be implemented first to see how they work. There is one point that worries me: human monitoring of production is very adaptive — the automated system may disguise some of what is happening.

Bemer: We are starting gradually, and building up. My motto is 'do something small, useful, now.'

McIlroy: It would be immoral for programmers to automate everybody but themselves. The equivalent to what Bemer is discussing is done by all big manufacturers to assist the process of hardware design. However, in addition to the storage of information provided voluntarily by the programmer, one should take advantage in such a system of the chance to accumulate additional information without bothering the programmer.

Harr: One has to be very careful in designing such a system to ensure that one does not end up slowing down the progress of software production, and/or adding significantly to the programmer's burden by increasing the amount of information that he has to provide.

Ross: If you don't know what you're doing in producing software, then automating the system can be dangerous. However, I am in principle in favour of such program production tools.

David: We have had some experience of using an on-line system for program 96 development, in fact to aid in the production of a large military system. The on-line system, called TSS/635, was developed specifically for this task, and provided means of accessing a large data base, and facilities for on-line program development. It worked reasonably well, but as the project has evolved, people have used the on-line system less and less, and are now starting to switch to a batch system running on the machine for which the military system is being developed. There could be many reasons for this — unreliability of the on-line system was certainly one. However, I believe that another was that people preferred to spend their money developing software which would be useful for development, for the computer for which they were developing the military system.

Cress and Graham: (from *Production of software in the university environment*)

»The danger of writing a special piece of software to expedite production is that we have a project within a project. Often the inner project is more interesting and it is certainly more volatile in design. Thus it is never really complete, as its effectiveness is not known until it is used. This makes it extremely difficult to make deadlines and stick to them.«

5.3.2. CONCEPTS

The above title has been chosen, perhaps somewhat arbitrarily, for a report on a discussion about the basic techniques or ways of thinking, that software engineers should be trained in.

It is perhaps indicative of the present state of software production that this topic was one of the most difficult to report on.

Ross: I would like to present some ideas of mine merely as a starting point for this discussion. The hope is that we will trade ideas on the actual techniques whereby we might work toward the evolution of a true software engineering discipline and technology. The three main features of my view of what software engineering and technology are all about are:

1. The underlying theory or philosophy, about what I call the ‘plex’ concept.

97

2. How this translates into the ideas put forward by McIlroy on software components — for which I use the term ‘integrated package of routines’.
3. How you systematically compose large constructs out of these smaller constructs — for which I use the term ‘automated software technology’.

A ‘plex’ has three parts: Data, Structure, and Algorithm (i.e. behaviour). You need all three aspects if you are going to have a complete model of something — it is not sufficient to just talk about data structures, though this is often what people do. The structure shows the inter-relationships of pieces of data, and the algorithm shows how this structured data is to be interpreted. (For example a data structure could stand for two different things if you didn’t know how to interpret it.) The key thing about the plex concept is that you are trying to capture the totality of meaning, or understanding, of some problem of concern. We want to do this in some way that will map into different mechanical forms, not only on different hardware, but also using different software implementations. An ‘idealised plex’ is one in which the mechanical representation has been thrown away. This is done by degenerating the data and structure aspects of a plex — not throwing them away — but, putting the entire discussion into the algorithm domain. Thus for example one avoids talking about data, by using **read** and **store** procedures which do whatever handling of the data is necessary. Such a pair of procedures is called an ‘idealised component’, which represents a data item (and the ‘box’ which contains it) that is accessed by the procedures. An ‘idealised element’ is composed of idealised components glued together, so that they can’t be taken apart, and is represented by a **create-destroy** procedure pair. Finally an ‘idealised plex’ is a set of ‘idealised elements’ which have been hooked together into a whole structure, with many inter-relationships. So one needs further procedures for adding elements to the plex (a **growth** function) and what we call a ‘**mouse**’ function, which is one which can be used to trace relationships, and to apply a procedure to the elements visited during the tracing. These idealised facilities allow me to talk abstractly about any object I wish to model. To actually use these ideas one needs mechanizations of the facilities. For example, a read-store procedure pair could be handled as subroutine calls, or could be compiled, if the compiler had been given declarations of the data structures being used.

98

van der Poel: You are using, without real definition, many terms which I just don’t understand.

Perlis: The entire description could be phrased entirely in LISP, in which the ‘plex’ is a ‘function’, the ‘data’ is a ‘function’, the ‘structure’ is a ‘function’, and the ‘algorithm’ is another ‘function’. A ‘component’ is another pair of functions, which operates on complexes of list structures which are built up out of primitives. My interpretation is that when Ross uses terms like ‘ideal’ and ‘model’ and so forth, he is really talking about a specific mechanization of the process of representing complex storages in terms of elementary ones, complex algorithms in terms of elementary ones, and so forth.

In van der Poel’s paper there is a five or six line program describing an assembler. Now the question that we, as people interested in software engineering should ask ourselves is this: Is that five-line program the guts of our business or is it not? I personally feel that it is not — it is just part of it. The issue of building an assembly program goes far beyond that five line description. The description is an essential beginning, and is in a certain sense no different from what Ross has talked about so far.

Ross then went on to give a detailed description of a data structuring package built using his concepts illustrating techniques for selecting alternate mechanizations of ordered relationships in an automatic fashion.

Fraser: There is a practical problem here. It seems to me that in order to realise anything like this you have to do all your binding at run time. Is this true or have I missed a neat dodge somewhere in your AED system?

Ross: The neat dodge is some four to five more years of work on the compiler. Many things that now involve run-time action can be resolved beforehand with a more fully developed system.

Ross: Let me turn to software technology, and take the problems of banking as an example. In such a situation you would make a set of integrated packages, or as I call them ‘semantic packages’ each covering a major area of your business, and in terms of which you could describe, albeit abstractly, the entire process of banking.

99

Each integrated package is a sublanguage — the question is how to process it in a systematic way. My model of the communication process consists of:

1. Perform lexical processing of the input string (for example, recognising items from character strings)
2. Perform syntactic and semantic parsing of the input string
3. Build a model of the implications of the information transmitted by the input string, i.e. understand the information
4. Act on the information (for example, reply to the message). One therefore makes an idealised plex for each of these four phases for each semantic package, and then provides a mechanization of the plexes, interlocked to provide one cohesive and comprehensive language and system for banking problems.

Perlis: I wouldn’t build a banking system this way. The model that you are using is too sequential. I would start with Simula and build a simulation model of an abstract ‘banking machine’, simulate information flow on it, and gradually make it more detailed. I would choose Simula because it has facilities which enable me to construct and meter processes all at the same time.

Ross: Fine. But the only thing that is sequential is the communication process itself for each of the sublanguages. Simula would be just a different example to use instead of banking for my discussion. I think Simula’s success stems largely from the fact that it already incorporates many of these ideas.

5.3.3. PERFORMANCE MONITORING

The provision and the use of facilities for monitoring system performance is of importance in all stages of system manufacture. In fact both Section 4 (Design) and 6 (Service) also deal with the subject.

A concise description of performance monitoring in production was provided by Opler, and is reproduced here in its entirety.

Opler: (from *Measurement and analysis of software in production*)

»1. **Who, When, Why**?

100

a. Guidance Measurement by Production Group, starting at the earliest existence of code in measurable form, for purposes of analyzing: 1) Conformity to design requirements; 2) conformity to internal conventions; 3) identifications of erroneous or deficient areas; 4) Identification of areas subject to optimizing by tuning. As separate modules are combined, measurements are repeated.

b. Completion measurement by Production Group, immediately prior to delivery, for quality assurance of the final product.

c. Formal measurement by control group to determine if quality of final product is acceptable.

2. **What is measured**?

a. **Performance**: Space, speed, throughput, turn around.

b. **Language**: compliance with requirements, accuracy of object system. External Function: error isolation, configuration modularity/clear documentation, availability, installation ease, modification ease.

d. **Internals**: serviceability, reliability (freedom from mistakes), conformity to standards.

3. **How are measurements made**?

a. **Gross (external) measurements**: typical programs, mixes, streams; data files; special test programs for language conformity; for mathematical accuracy; for standards conformity.

b. **Fine (internal) measurements**: by special hardware monitors; by special software packages; by built-in measurement schemes.

c. **By use of product operation**: serviceability; configuration modularity; installation ease«

Opler: It is important to emphasize the necessity for providing sufficient resources during production of a software system to design and conduct performance measurements, and to feed the results back to those concerned, in both the design and the production groups. One warning is in order. It is fatally easy to concern oneself with only those quantities which are easy to measure, and to ignore other, possibly more important, quantities whose measurement is more difficult. Monitoring aids must be an integral part of the system, so that every attempt to use the system is potentially a source of useful data.

101

Kinslow: There is a whole class of what production management tends to think of as ‘auxiliary’ functions, such as system generation, system start-up, system shut-down, and monitoring. If the budget gets tight, monitoring is the first to go, because of a lack of appreciation for its importance.

Fraser: We found it useful to monitor a system, feed the results into a simulator, and then experiment with the simulator, as being much easier to fiddle with than the actual system.

Pinkerton provided a survey of the various techniques that have been used for monitoring systems. Because of its length this survey is reproduced in Section 9, rather than here.

Finally, a description was given by Gillette of a method of automating both the testing and performance monitoring of a system.

Gillette: (from *Aids in the production of maintainable software*)

»System testing should be automated as well. A collection of executable programs should be produced and maintained to exercise all parts of the system. The set should be open ended and maintenance utilities should be included. A result (i.e. output) from a system source text update should be a set of necessary control statements to enable a selective test of all modules modified and referenced entities. A control sequencer utility should exist which would selectively execute all programs to exercise specified modules. Test codes should exercise modules separately and in combination. A total system stability and performance test should be included in such a scheme. Driving the system from this source should be recoverable in event of an error so that a thorough test can be made in one pass through the test collection. Errors should be diagnosed and reported, tracing the chain of events which resulted in the error, and should attempt to isolate the causal factor to a module or referenced element. The test set should be maintained in source text language by the same facility discussed with reference to system source text. Clearly there is a need to be able to include the system module and change directories as a part of the test source environment and to be able to cross reference to these entries from within non-generative code in the source text of the test programs. Tests should be dated to permit exclusion of their usage when they become obsolete. As an output of a test validation 102 run, each test should list the modules it has exercised, and as well, should list the interfaces and tables it has tested. It is important to document success, as well as failure.«

103

6. Service

6.1. INTRODUCTION

6.1.1. THE VIRTUE OF REALISTIC GOALS

Much discussion centered on the conditions necessary for acceptable service to be obtained from a software system. The first point covered was the virtue of realistic design and production goals,

Opler: I am concerned about the current growth of systems, and what I expect is probably an exponential growth of errors. Should we have systems of this size and complexity? Is it the manufacturer's fault for producing them or the user's for demanding them? One shouldn't ask for large systems and then complain about their largeness.

Dijkstra: It is not clear that the people who manufacture software are to blame. I shirk manufacturers deserve better, more understanding users.

Llewelyn: Lots of time and money are lost in planning on what turns out to be false data provided by manufacturers. We need more realistic delivery estimates from manufacturers.

Randell: The users should demand contractual safeguards.

6.1.2. INITIAL SYSTEM RELEASE

The second point was concerned with the quality of initial releases of software systems.

Babcock: The initial release of a software system should **work well** (albeit with limited facilities) and must contain the basic system philosophies that ensure orderly growth.

Genuys: We need pre-release versions of systems, whether they work well or not, for training our staff.

104

Galler: Manufacturers should not deliver a system unless it is working well, although it need not be entirely free of bugs.

David: Define a subset of the system which is small enough to be manageable, then build on that system. This strategy requires that the system be designed-in modules which can be realised, tested and modified independently, apart from conventions for intermodule communication. It also implies that the system can be a tool vital in its own development.

Kolence: Large systems must evolve, and cannot be produced all at one time. You must have an initial small core system that works really well.

Randell: The users are as much to blame for premature acceptance of systems as the manufacturers for premature release.

Samelson: The real problem is the user, since he needs software and takes it whether or not it is correct.

Glennie: Software manufacturers should desist from using customers as their means of testing systems.

6.1.3. FREQUENCY OF RELEASES

The subject of frequency of system releases, and its effect on the level of service that could be expected from a system gave rise to the following comments.

Babcock: Fewer releases, containing major functional improvements (other than corrections) that **work well** are more desirable than frequent releases of versions containing only minor improvements.

Gillette: CDC recently undertook an extensive update of one of its software systems in order to increase its performance. Users were given the chance to wait for development to be complete or to receive incremental updates that would not have been fully integrated and tested. All users elected to receive monthly system updates. Our field analysts explained that they could cope more easily with small incremental changes.

Opler: The latest release of OS/360 was intended to introduce 16 changes (many complex), and to correct 1074 errors. The current policy is to have releases at 90 day intervals. This indicates a rate of over 11 corrections per day between versions. It is obviously better to batch improvements to a system together. On the other hand, because customers need errors to be corrected the release frequency cannot be too low. Once a year would be much too infrequent.

105

Hastings: Many of these 1000 errors are of course quite trivial, perhaps being just documentation errors.

d'Agapeyeff: (from *Reducing the cost of software*)

»The growth of complexity in software has led, understandably enough, to many issues and versions of the same system. But no way has been found of making past issues sub-sets of new issues. This is causing great disturbance to users and is also causing incompatibilities (e.g. on machine A, operating system level X is unlikely to be compatible with level Y to either programmers or operators).«

Babcock: One way to have a successful system is to **never** make a change in it. However, this is obviously impractical. Systems such as our Rush system never remain static, and are never 100 percent checked out. So system managers should go easy on installing new versions of on-line systems, and check them out in a dynamic environment.

Pinkerton: With less frequent releases there would be increased stability, and more opportunity for users to generate responsible feedback on system performance to the manufacturers.

Galler: OS/360 has had 16 releases in two and a half years. We should have frequent updates for corrections, but decrease the frequency of traumatic upheavals.

Gillette: (from *Comments on service group statements*)

»[Babcock's use of the phrase 'works well'] bothers me as it is largely qualitative. For an engineering group I think a metric description would be more palatable and more meaningful.

To be specific, below I have written a copy of one of the paragraphs which has been put into a Product Objectives document. We struggled a great deal to define measurable objectives in the document and this is an example. The numbers used do have relevance, historically, and that is all that need be said about them. Finally our objectives may not have been high enough in this particular area — we tried to push our luck while at the same time being realistic.

The total number of unique bugs reported for all releases in one year on ECS SCOPE will not be greater than the number given by the following formula: Number of bugs $\leq 500 - 45/(I + 10)$ where I is the number of installations using ECS SCOPE. 85 percent of the 106 reported PSRs (see Notes below) will be corrected within 30 days and 50 percent of these will be corrected within 15 days. All PSRs will be corrected within 60 days.

Notes:

1. A PSR is the reporting form which a customer uses to report a bug.
2. A bug is effectively equivalent to a PSR — it may be a mistake in the code or a misunderstanding on the part of the customer.
3. Correcting the bug may result in modified code or clarifying a definition with correction to the reference manual.
4. The limiting function is bounded and increases with the number of users.«

6.1.4. RESPONSIBILITY FOR MODIFIED SYSTEMS

This discussion led to the subject of assigning responsibility for user-modified systems.

Babcock: In those areas that have not been affected by user modifications, software manufacturers should be responsible for all maintenance and improvements.

Paul: One cannot easily define the borderline between affected and unaffected parts.

Galler: I am concerned that manufacturers use this difficulty as an excuse for absolving themselves of any responsibility for the system after any change the user makes.

Naur: If you want to modify a system, you had better choose a manufacturer and a system that allow this. With most products the standard case is that a warranty is voided if the customer fools about with the product.

Babcock: Often there is no choice for such a system open to the user.

Bemer: This shows the need to improve the means of specifying program interfaces.

Berghuis: A paper in the Communications of the ACM in 1963 or 1964 gives the manufacturer's viewpoint. This paper, written by the ECMA, the European Computer Manufacturer Association, states that no manufacturer will take responsibility for any user modifications.

107

6.2. REPLICATION, DISTRIBUTION AND MAINTENANCE

6.2.1. REPLICATION

As discussed in Section 3.3, the replication of multiple copies of a software system is the phase of software manufacture which corresponds to the production phase in other areas of engineering. It is accomplished by simple copying operations, and constitutes only a minute fraction of the cost of software manufacture.

Randell: If software replication costs were commensurate with hardware replication costs, there would be a great incentive for manufacturers to improve the quality of initial software releases.

The main contribution on the subject of software replication dealt with the problems associated with mass production of copies of software.

Enlart: (from *Program distribution and maintenance*)

»We do not consider in this paper the sophisticated headache of generalized operating system testing. We assume that the development programmers did a good job and tested their product carefully by means of sample problems, selected potential users' cooperation, bench mark application, etc The development programmers will issue an information medium loaded with the programming system, together with its literature. The literature can be mass produced and stored to meet the forecasted requirements of the potential users. Unfortunately, mass production of information media is not feasible to date, for lack of standardization in machine configurations and input devices.

Another reason is the very low efficiency of the available DP material to mass produce information media. The performances of the fastest input-output devices available now are limited for technological and physical considerations, and, in spite of their high efficiency in terms of data processing, they are definitely not mass-production tools. The last point is the number of interfaces between program authors and users which results in the number of times a master should be duplicated to supply the Program Distribution Centers with their own master copies, enabling them to disseminate copies of the program.

108

Basically, every bit of information is vital, hence the requirement for the 100 percent checking of every copy. As performance checks cannot be resumed after every copy operation, it is necessary to provide the successive functions which should duplicate a program with tools and means of control, to achieve the highest ratio of reliability.

This quality control problem is worsened by the sensitivity of information media to a variety of mishandling, in or outside of the machine room: there is no 'acceptable percentage of variations' or 'plus or minus acceptable tolerance' in software: a bit of information recorded on a tape is true information or false information.

If a bit is false, it will spoil the whole product, and, if reported as a bug to the authors, it will cause bewilderment and useless attempts at a corrective solution. In any case, it will ruin confidence (if any) in relations between both parties.

In a normal data processing operation, a deficient card reader, for example, will sooner or later be discovered because it introduces inconsistencies in results.

The case is different in a program library: the deficient card reader may introduce an undetected error in hundreds of copies of a program, even if a control run to match the resulting tape with the card deck has been made. The error will be discovered occasionally in a remote location and perhaps months later. The case is worsened if one of the faulty tapes becomes the master of a sublibrary.

Program distribution and maintenance requires quite a few skills in a great variety of specialized fields: programming and machine operation, file organization, wrapping and packing, shipping procedures, customs and postage regulations, reproducing and printing, together with cost evaluation and financial foresight.

Among the conditions required to smooth this thorny road, standardization (media) and formalization (documentation) are the keys to success.

Unfortunately, programmers are intellectually not prepared to recognize the problem.

Awareness of their responsibilities in the field of distribution should be a part of their basic educational training.

109

Software distribution is a challenge for the data processing community, and its response to it will result in either loss of energy or continuous growth.«

Galler: One simple means of checking the correct functioning of the replication and distribution process would be ‘echo-checking’. In other words, the group that produces the system that is handed to the Program Library for replication and distribution should itself be a customer of the library.

6.2.2. DISTRIBUTION

Software distribution per se was only touched on briefly in the conference.

Köhler: (from *Maintenance and distribution of programs*)

»Program distribution as a whole is a problem of organization only. In a well organized and efficiently working distribution center each request should be handled within 5 to 8 hours after it has been received. However, it is a recognized fact that a distribution center, besides receiving requests for programs, also has to deal with queries. In order to answer such enquiries properly the availability of qualified personnel must be guaranteed. Unfortunately documentation is not always perfect, even if there are the best of intentions. The user will appreciate it if he can always get his advice over the telephone.

Such service, however, can generally be rendered only by computer manufacturers, big software groups and perhaps some larger user groups.

As mentioned above, the distribution center is well advised to ask for the preparation of special forms or punched cards when clients wish to request programs, etc. In doing so it educates the user to make his request complete and, furthermore, creates the basis for automation in the distribution system.

When preparing order facilities for the user by way of special forms and prepunched cards, provision should also be made for a users request on the method of delivery. Upon such delivery requests may depend whether a card-pack will arrive at its destination within hours or within days.«

Nash: The delay between the release of a system and its arrival at a user’s installation is a serious problem. Replication and shipping cause 110 part of the delay, but the time taken to perform system integration is also a problem. We are shipping systems, not components, and the time to perform system integration depends on the number of interactions among components, which multiplies very fast as the number of components increases.

Randell: I wonder how the problems of software distribution are divided up among the three categories ‘initial release’, ‘corrections to errors’, and ‘extensions’. I would suspect that the second category has the most, and that efforts to reduce the problems of program distribution should concentrate on this area.

Dijkstra: The dissemination of knowledge is of obvious value — the massive dissemination of error-loaded software is frightening.

6.2.3. MAINTENANCE

The main contributions that were concerned directly with software maintenance were those of Gillette and Köhler.

Köhler: (from *Maintenance and distribution of programs*)

»Maintenance and distribution are strongly interdependent: upon being written each program requires a certain amount of testing, the so-called field test. The field test is considered to be successful when the programs that are being tested have been allowed a reasonable number of fault-free machine runs and when it is thus indicated that they will effect the full spectrum of their intended applications. The duration of the field test depends upon quite a number of different factors, such as the amount of machine time allocated to a given program, the frequency of machine runs required, the complexity of the program, etc. Consequently the actual times for the duration of field tests should always be determined by the people actually responsible for the maintenance of a respective program; also great care should be taken to make sure that a given number of runs has been achieved during the test and that any faults recognized by the user have been properly recorded and reported to the interested parties. No one should hesitate to prolong the field test period if — during its course — it should become apparent that the number of existing program faults shows no, or only small, diminishing tendencies. On the other hand, it should not be expected that a complex program, 111 after being given an appropriate field test, is completely free of errors. Owing to the complexity of certain programs, each user of electronic data processing equipment has at times to be prepared to deal with program failures, especially when handling more sophisticated applications.

Thus each maintenance depends upon the proper recording of programming errors by the user and upon the quality of such records. In those cases where the maintenance-center and the distribution-center constitute a single organisational unit, maintenance can operate with great effectiveness and, when distributing their programs to users, can influence all users as regards proper error reporting.«

Gillette: (from *Aids in the production of maintainable software*)

»The economics of software development are such that the cost of maintenance frequently exceeds that of the original development. Consider, for example, the standard software that many manufacturers provide and deliver with their hardware. This product can include a basic operating system, a machine language macro assembler, an Algol, Fortran, and Cobol compiler, a sort/merge package, a file management facility, and so on. In scope this represents something in the order of more than 250 thousand lines of generated code that must be released to customers whose configurations and requirements vary a good deal, encompassing the spectrum from batch oriented data processing shops, to hybrid time-critical, time sharing and scientific shops. Producing such systems currently requires about a two to three year effort involving perhaps as many as 50 personnel. Maintenance of such systems is an unending process which lasts for the life of the machine; as much perhaps as eight years.

…

Maintenance of a system is required in order to satisfy three basic problems. First, in an effort of the magnitude of that described there will be bugs in the system. These can originate from ambiguous specification and reference documentation, because of design error, or because of programmer and system checkout error. Second, design decisions and code generation cannot always result in 'good' performance. In a basic operating system, for example, a code module may get executed on the average of once per 10 milliseconds while the system is operational; it is desirable to make the code as fast as possible. In the hurry to deliver an 112 operable system it is seldom that code can be truly optimized; the emphasis is on correct execution rather than speed. Much effort is expended to improve system performance in order to remain competitive. Third, and finally, in a span of several years, new hardware is developed which must be supported and new customer needs develop which must be met. To support this a system must be extended to include capabilities beyond those that the original designer conceived. In summary, then, the maintenance process involves corrective code, improvement code, and extensive code. «

Babcock: I am concerned about the division of responsibility for maintenance between user and manufacturer. As a user, I think it a manufacturer's responsibility to generate systems to fit a particular user's need, but I haven't been able to convince my account representative of that fact.

On the subject of pre-release checking, there is the question of how a manufacturer can ensure that a system he distributes will have been adequately checked for use in met environment. It seems to me that the only way to solve such problems is to have the manufacturer simulate my environment, or even use my environment directly, via communication lines.

Kolence: Users should expect to have to test software that is supplied to them, to ensure that it works in their environment. A large manufacturer cannot test out his software on all the environments in which it will operate, It is for this reason that manufacturers typically provide an on-site representative to help the user adapt a general system to his particular environment.

6.3 SYSTEM EVALUATION

The problem of system evaluation runs through all three areas of Design, Production and Service. This section is based on discussion of those aspects of the subject directly related to the system when in a user's environment.

113

6.3.1 . ACCEPTANCE TESTING

By far the most extensive discussion of the acceptance testing of software systems was that given by Llewelyn and Wickens in the paper The Testing of Computer Software. For convenience this is reproduced in its entirety in Section 9, and just the conclusions are paraphrased below.

Llewelyn and Wickens: We are attempting to design a method by which a large organization could test the software supplied by computer manufacturers to its installations. In particular we wish to ensure that new installations can rapidly take on the work for which they were purchased by ensuring that their planning can be based on the best available information with regard to what software exists, and how well it performs. The present situation is that a customer has to purchase his software almost as an act of faith in the supplier — this surely cannot be allowed to continue.

Kolence: The manufacturers are always under pressure from the users to give them something that works even if it is not complete. The classic trade-off in software production is between features and schedule — not between working well and not working well. It is important therefore for users to receive an accurate set of specifications, ahead of time, of what is currently being produced, rather than of what is ultimately intended (as was the case, I believe, with published specifications for OS/360) .

Opler: (from *Acceptance testing of large programing systems*)

»The proper testing of large programming systems is virtually impossible; but with sufficient resources, enough testing can be performed to allow a good evaluation to be made. Most current systems operate under a wide variety of hardware configurations and with highly varied software component selection. A test plan must be developed considering all elements of the written specification (hardware, programming language, system facilities, documentation, performance, reliability, etc.) and describing steps to validate compliance of the final programming system.

For large systems, enormous resources of computing equipment which can configure all available components are required. Many months of computer testing are often required. A significant, sometimes neglected, area of 114 testing is in checking external documentation (user and operator manuals) for accuracy and clarity and checking internal documentation (flow charts and listings) against the final distributed program.«

Dijkstra: Testing is a very inefficient way of convincing oneself of the correctness of a program.

Llewelyn: Testing is one of the foundations of all scientific enterprise. In fact it would be good to have independent tests of system function and performance published.

Galler: When the hardware doesn't work, the user doesn't pay — why should he have to pay for non-working software?

Babcock: We need software meters analogous to the present hardware meters, so that our rental costs can be adjusted to allow for time lost through software errors as well as hardware errors.

Kolence: If the users expect a supplier's software to work exactly as and when predicted, they should in all fairness apply the same standards to the software that they develop themselves.

6.3.2. PERFORMANCE MONITORING

The major discussions on performance monitoring are reported in the sections of this report dealing with Design and with Production. The comments given below are of most immediate relevance to the question of monitoring the performance of an installed system.

Gillette: We have used performance monitoring principally as a maintenance tool to find bottlenecks and deficiencies.

Perlis: Delivered systems should have facilities for accumulating files of run time performance information, which could be shipped back to the manufacturer for analysis' as well as being of value to the user installation. I do not know of anywhere this is being done as a standard operating procedure.

Kolence: We have experience of performance monitoring in a user environment. In such environments extra care must be taken to avoid interference with the running of the system, and in overloading the system. Information can be monitored in relation to the overall system performance or to particular user programs. Typical data that we produce concern core storage usage and I/O channel and device activity. Information on disk cylinder 115 usage can be very useful in learning how to reorganise data on the disk for improved transfer rates. We have found that almost all programs can be substantially improved (between 10 and 20 percent, say), without design changes, in one or two man-days.

Smith: Performance monitoring when first applied can certainly lead to spectacular improvements. However, this can lead one onto the treadmill of incremental improvements to a system which is basically unsound.

Gries: Optional performance monitors should be built into compilers, so as to be available to users.

Galler: Performance monitors should be part of the system, and not in individual compilers.

6.4. FEEDBACK TO MANUFACTURERS FROM USERS

Comments relevant to this topic occurred in entirely separate contexts, and included attempts to classify the various types of information that a user would want to feedback to a manufacturer and the possible means of so doing. See also Section 4.2.2.

Haller: There is feedback of different entities, on different paths, leading to three separate control loops with different time-lag:

1. On the correctness, or otherwise, of a system. This would go to the maintenance group; time to get a reply might be up to a week.
2. On system performance, to the production group, who might be expected to reply within, say, a month.
3. Requests for extra facilities would go to the design group and, if accepted, a year might be the expected delay before an appropriately modified system was released.

Hastings: A user needs a fast means of obtaining corrections to program bugs. We have used a terminal-oriented on-line system for our field engineers to obtain up-to-date information from the maintenance group. Some such system as this is the only substitute for the common means of unofficial feedback, possible when one knows the right person.

Babcock: (from *Variations on software available to the user*)

»Even with a company such as IBM — which incidentally, I am convinced, is the very best — the best was certainly none too good.

…

116

Time to me was money. Time-sharing is a revenue of the moment, once lost, it will not, nor cannot be captured. It took us approximately six months to learn the then current IBM service was archaic, and something must be done. We started at the bottom but found we had to work all the way to the top to be heard. There I learned there was only one concept that was really meaningful to IBM: BACK YOUR TRUCK UP. The fact of the matter was, IBM was in a new area too, which may be considered a plus factor in our case because, being innovators of the highest order themselves, they recognized that here was a new service problem. There were multitudes of users out of service, rather than just one.«

Buxton: The best method of feedback about unsatisfactory systems is money. If software had a clear monetary value a user could express his belief that the software was bad in clear monetary terms by rejecting it.

Galler: The typical user is not in a position to reject a major system produced by a large manufacturer.

6.5. DOCUMENTATION

There was much discussion of Design and Production aspects of documentation, but very little from the user's point of view. The one working paper by Selig was concerned directly with this area. This paper is quoted below, and is the source of the set of document standards that are reprinted in section 9. That section also contains the paper by Llewelyn and Wickens, which touches on the reviewing of system documentation as part of acceptance testing.

Selig: (from *Document for service and users*)

»With the rapid proliferation of computer languages, subroutines and programs, and the tremendous effort they represent, meticulous documentation is becoming essential, not just to save money but to prevent chaos.

…

Reference manuals for the user, the operator and the programmer can be based on the external and internal specifications [of the system]. Such documentation … must be reader oriented and not subject oriented and good manuals are difficult to write. One common mistake is that authors 117 try to reach too many different groups: managers, experienced technologists and beginners. It is recommended to develop different educational documentation with clearly defined prerequisites for the prospective readers.«

Letellier: (from *The adequate testing and design of software packages*)

»Manuals must be of two types: a general introduction to the solution of the problem by means of the package and a technical manual giving the exact available facilities and operation rules; these should be as attractive as possible. «

Gillette: The maintenance of documentation should be automated.

Selig: (from *Documentation for service and users*)

»It is appropriate to mention techniques where the computer itself is producing the required documentation with considerable sophistication. Especially flow-charting and block-diagramming are readily available. Examples of such programs are: Flow-chart plotting routine (CalComp), Autoflow (Applied Data Research), Com Chart (Compress), etc. A more efficient method of automated program documentation became available in the conversational mode of computer usage. A few software systems are now self-documented, and detailed information and instructions can be displayed at the operator's request. The design of such programs is very similar to self-teaching manuals and this technique has become particularly well accepted in the area of computer graphics.«

Gries: The maintenance and distribution of the OS/360 documentation is automated through the use of a text-editing system. I don't think the maintenance of this documentation would be possible without it.

6.6. REPROGRAMMING

The only material on the subject of reprogramming was the short working paper prepared during the conference by Babcock, which is reprinted below in its entirety. There was no direct discussion on the subject of the users' problems in reprogramming for changed hardware or operating systems, but the sections of the Design area concerned with modularity and interfaces are of relevance.

118

Babcock: (from *Reprogramming*)

»We see this problem divided into three areas of discussion

1. Conversion
2. Re-writing of systems and programs
3. Future trends

Most users have faced the conversion problems before. The cost is usually a function of the severity of the conversion, that is, from one machine to another (usually highest cost), from one operating system to another and from one language to another. The most effective means today (but not the most desired) is by rise of hardware techniques, that is, emulation.

New systems usually exhibit features that are desirable and that handle a broader range of application in a more efficient manner. To utilize these new tools, re-writing is sometimes dictated. This is perhaps the most costly of the three areas but is mandatory in many areas. We see the need for higher levels of languages, such as compiler languages of compilers, in order to make this decision less critical.

In the future, we see the need for both hardware and software facilities specially designed for the overall problem of transporting. The stress will be as great on hardware design as on software engineering because the key to portability can be exhibited in the near future whereas software engineering has not yet produced nor announced a true language of languages for hardware independence. Hardware portability functions can be economic in present day ‘Assembler-oriented’ applications, but we hope for integrated hardware/software facilities to reduce substantially the costs of the reprogramming problem.«