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

2

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.

4 5

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

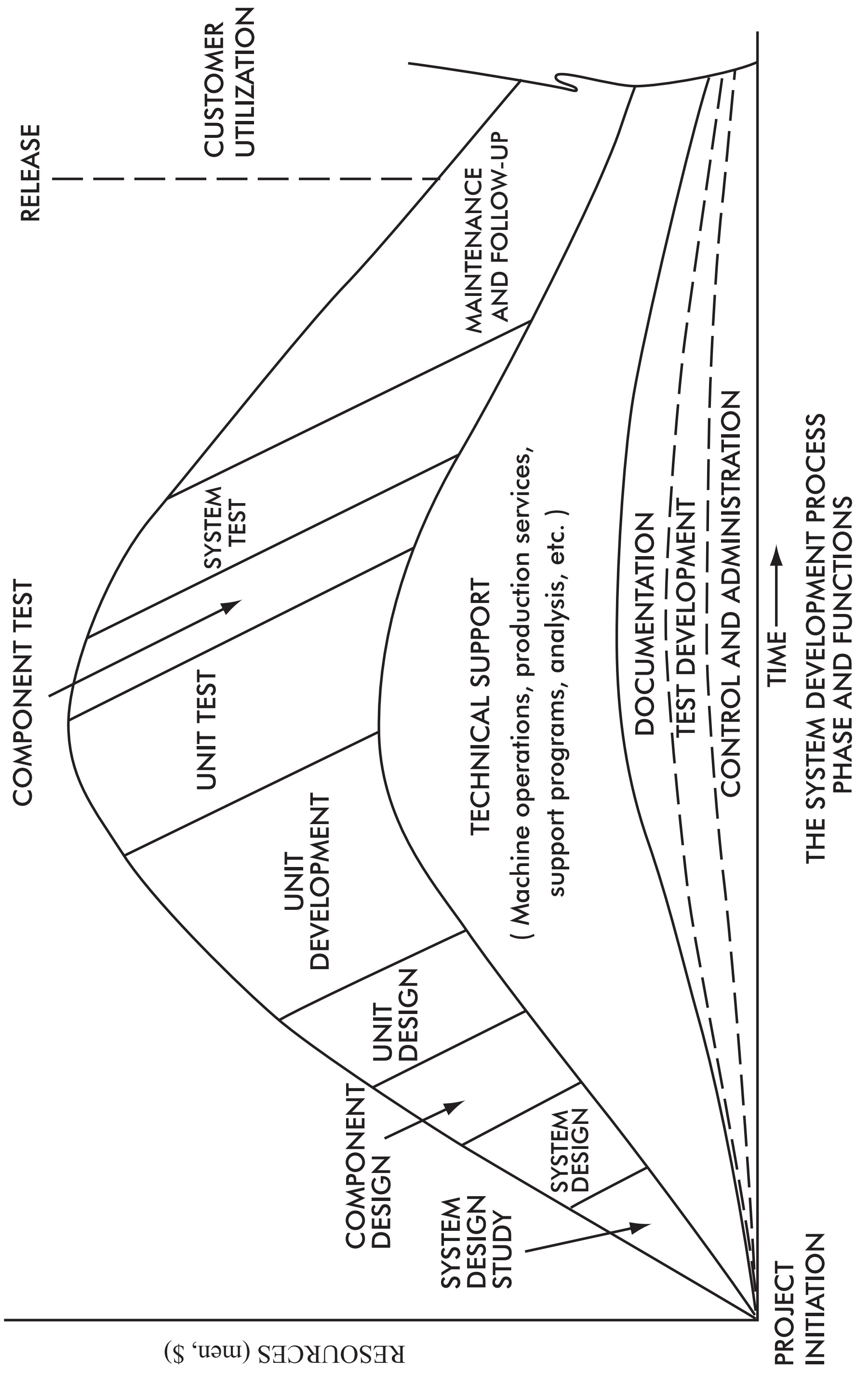

Figure 1. From Nash: Some problems in the production of large-scale software systems.

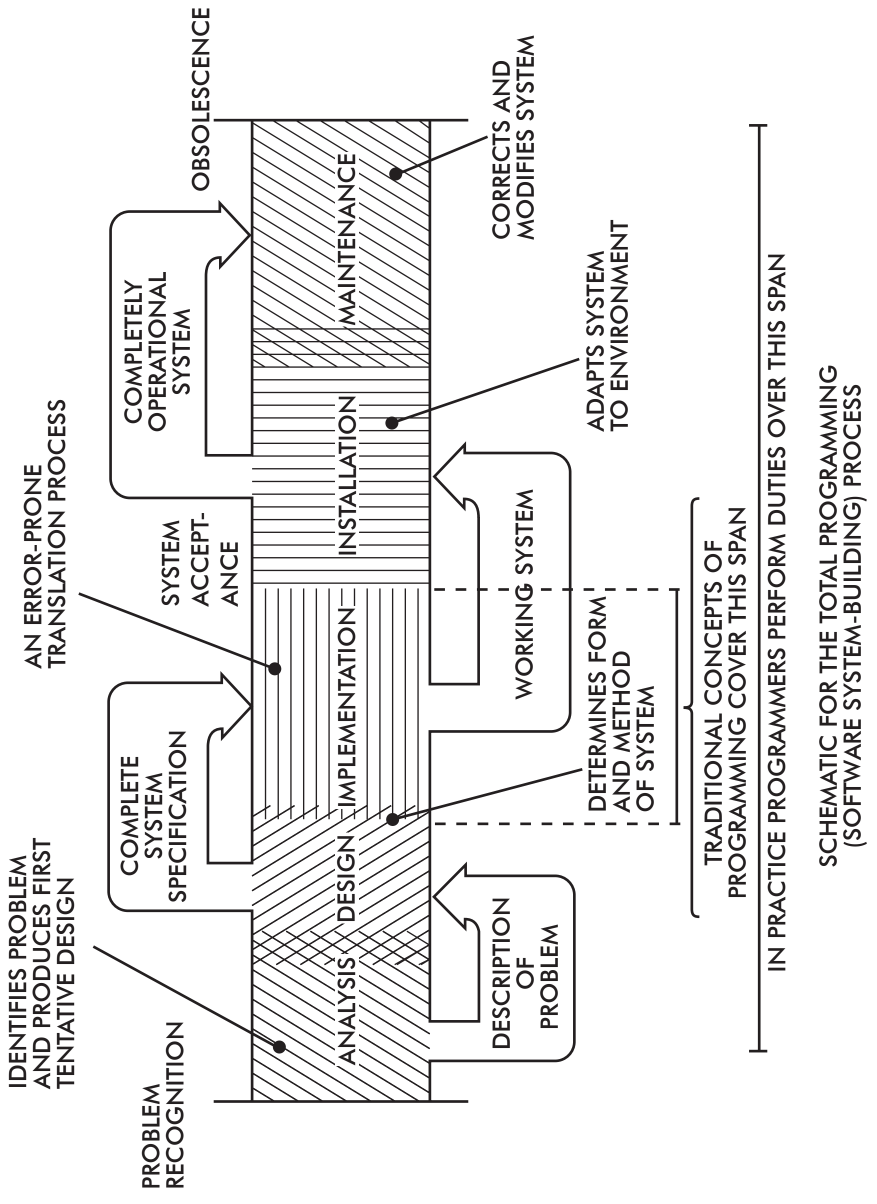
PROBLEM RECOGNITION
IDENTIFIES PROBLEM AND PRODUCES FIRST TENTATIVE DESIGN
ANALYSIS
DESIGN
DESCRIPTION OF PROBLEM
COMPLETE SYSTEM SPECIFICATION
IMPLEMENTATION
DETERMINES FORM AND METHOD OF SYSTEM
AN ERROR-PRONE TRANSLATION PROCESS
WORKING SYSTEM
SYSTEM ACCEPT-ANCE
INSTALLATION
COMPLETELY OPERATIONAL 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.

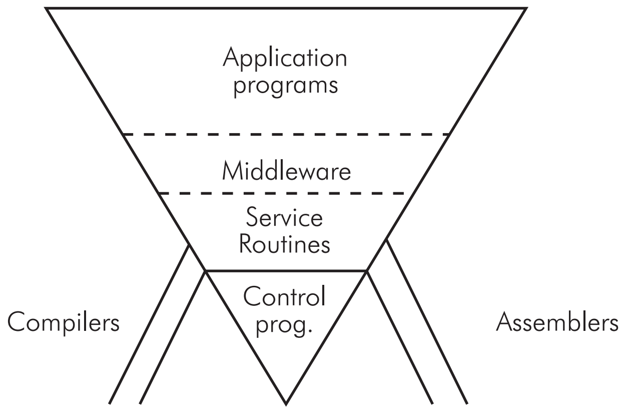

Figure 3. d'Agapeyeff's Inverted Pyramid

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.

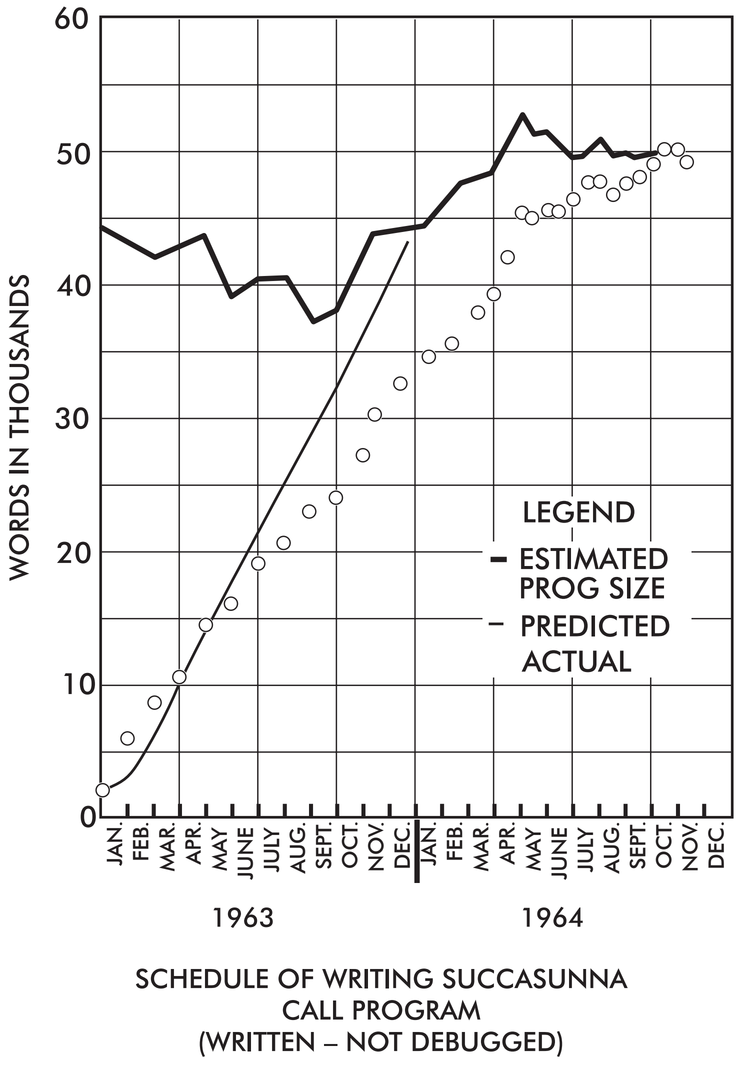

Figure 4. From Harr: The design and production of real-time software for Electronic Switching Systems.

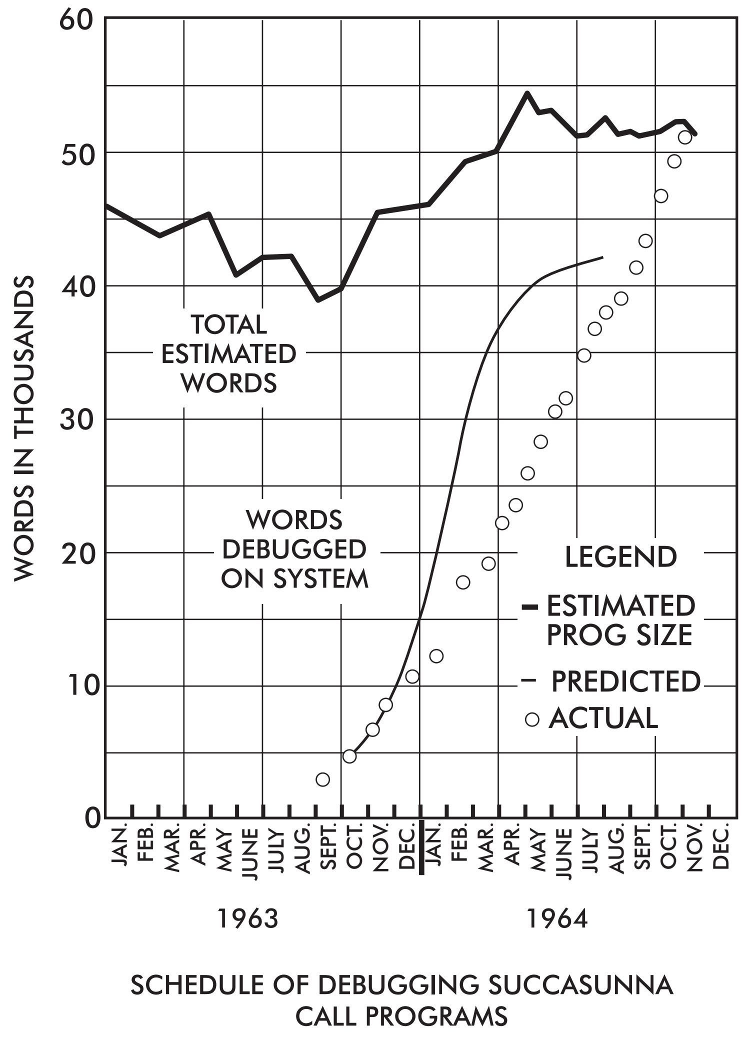

Figure 5. From Harr: The design and production of real-time software for Electronic Switching Systems.

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

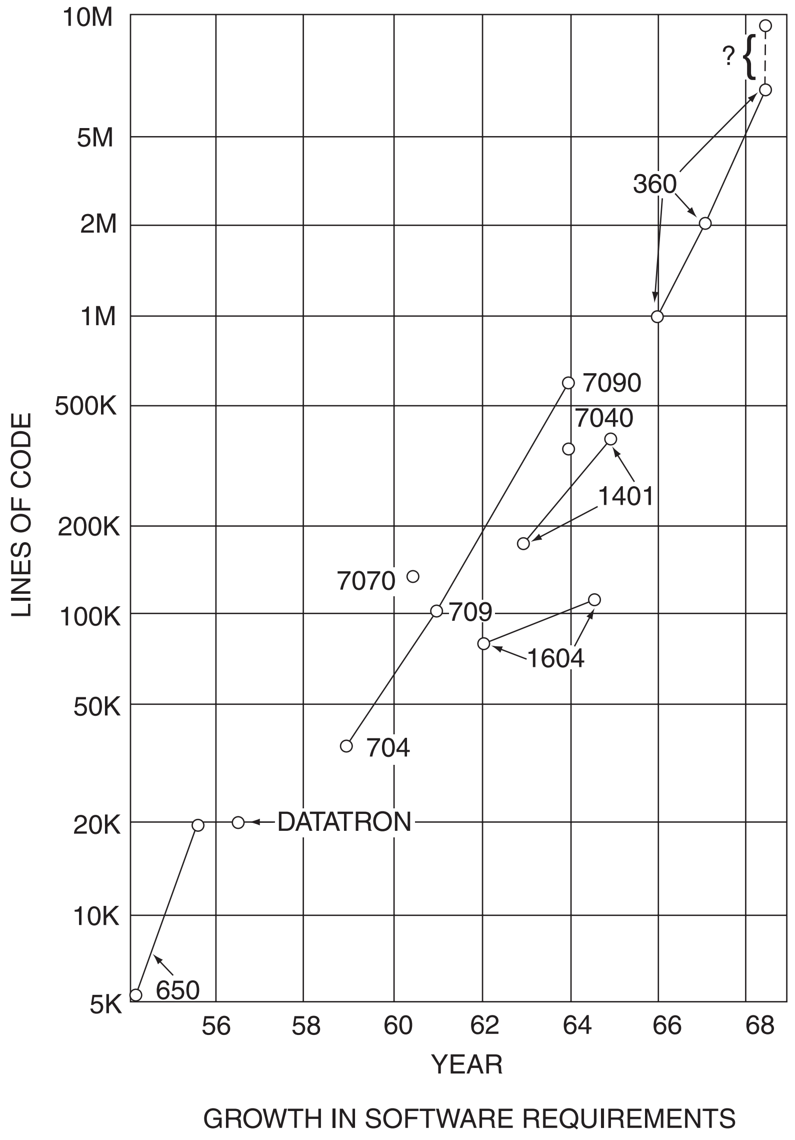

Figure 6. Provided by McClure

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.«

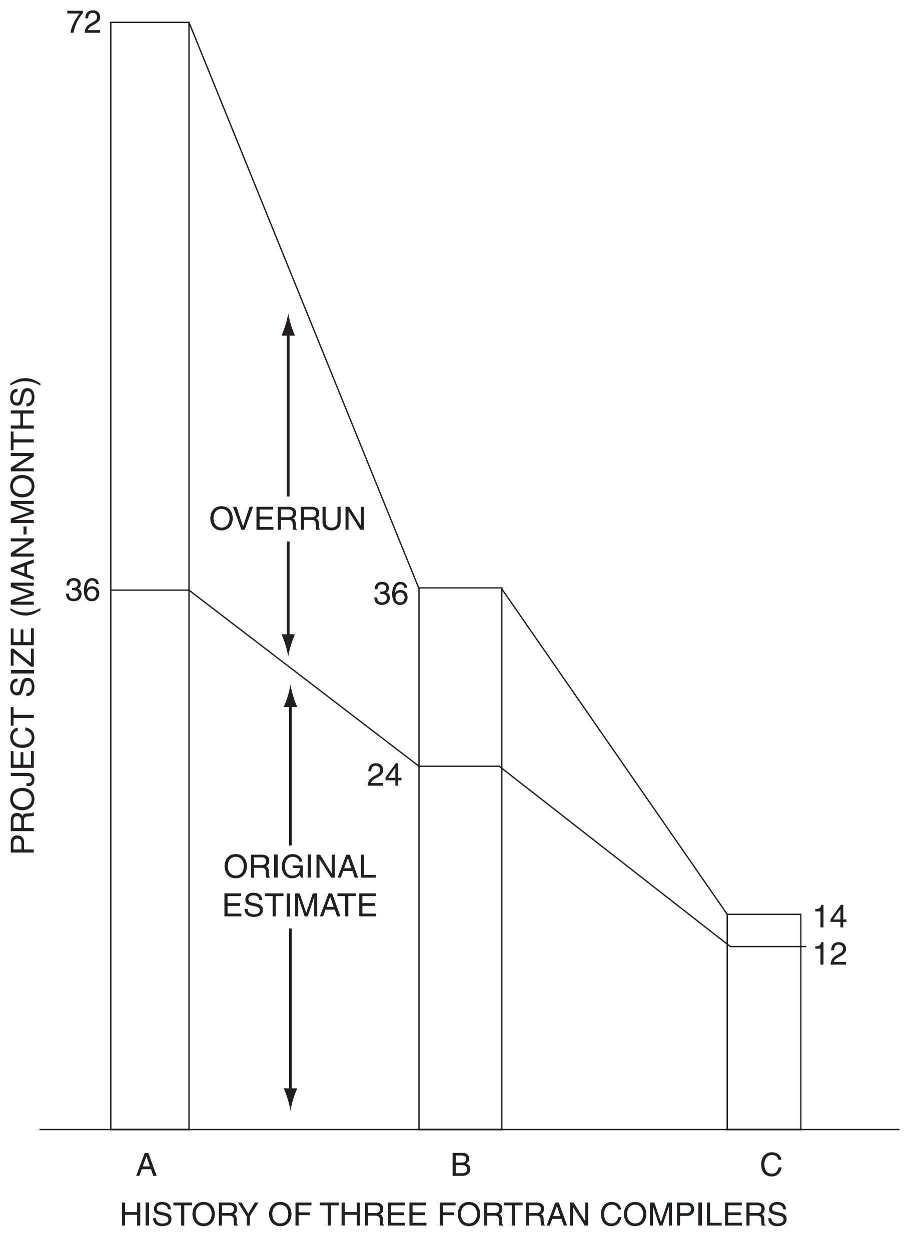

Figure 7. Provided by McClure

77

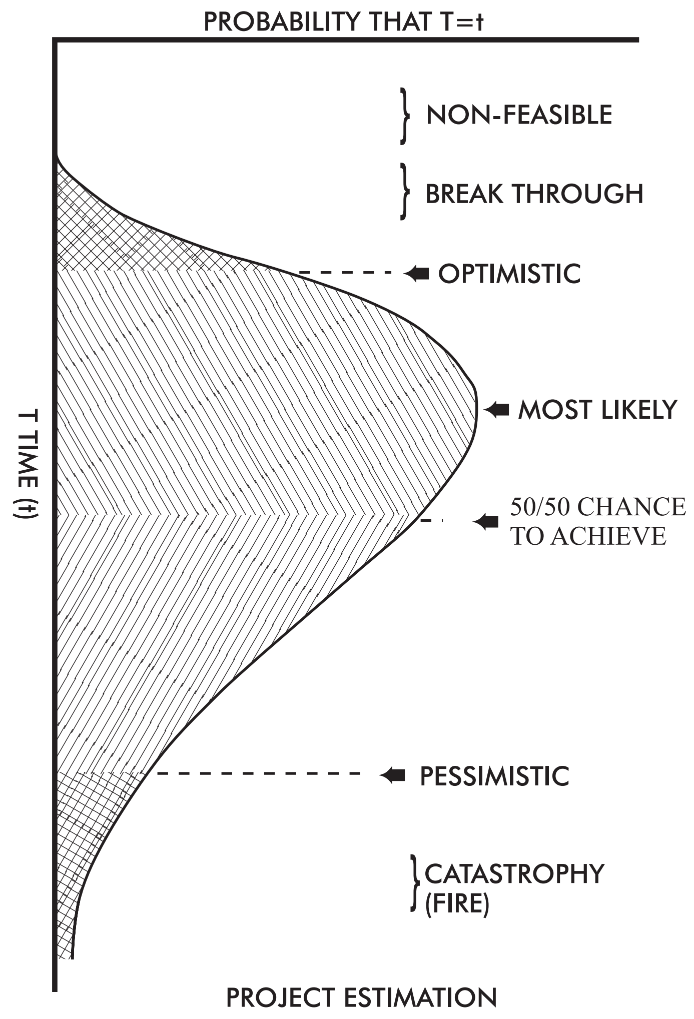

Figure 8. From Nash: Some problems of management in the production of large-scale software systems.

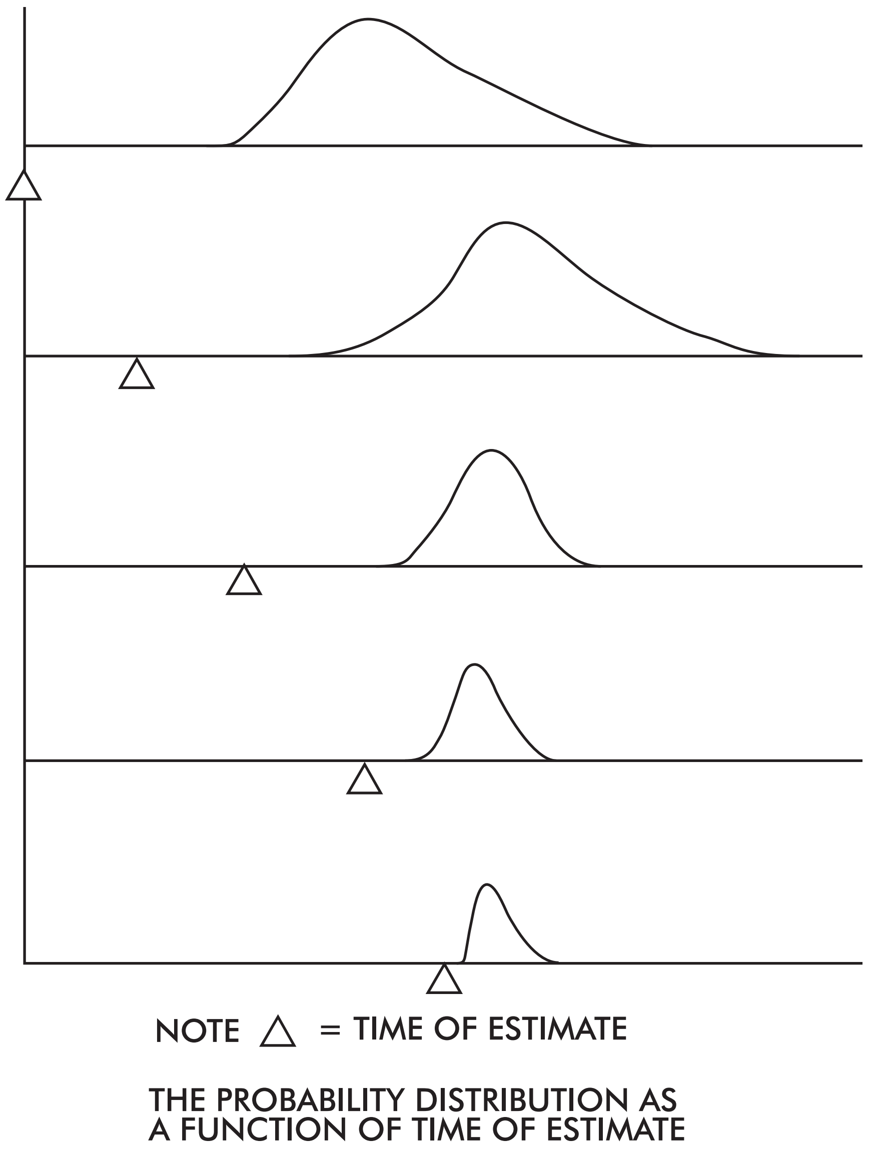

Figure 9. From Nash: Some problems of management in the production of large-scale software systems.

79

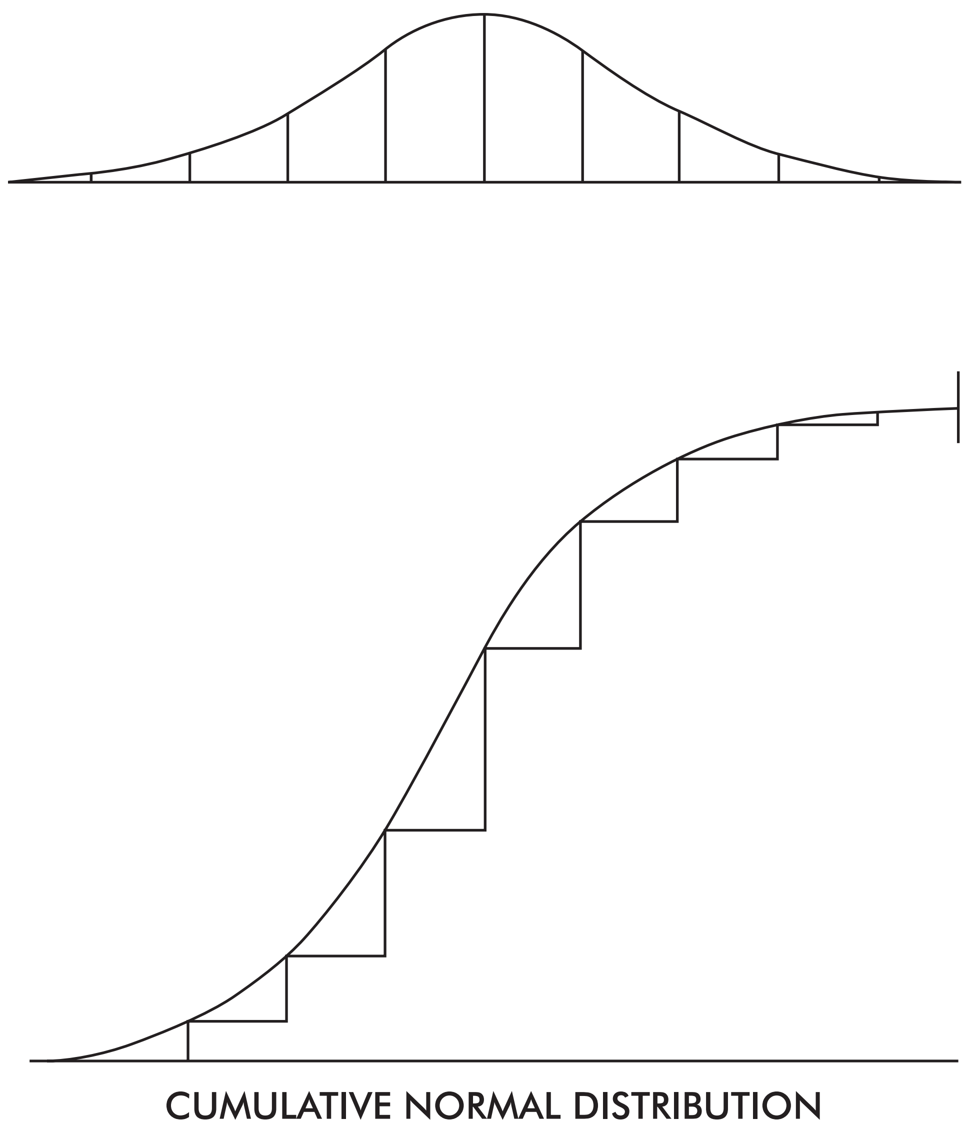

Figure 10. From Nash: Some problems of management in the production of large-scale software systems.

81

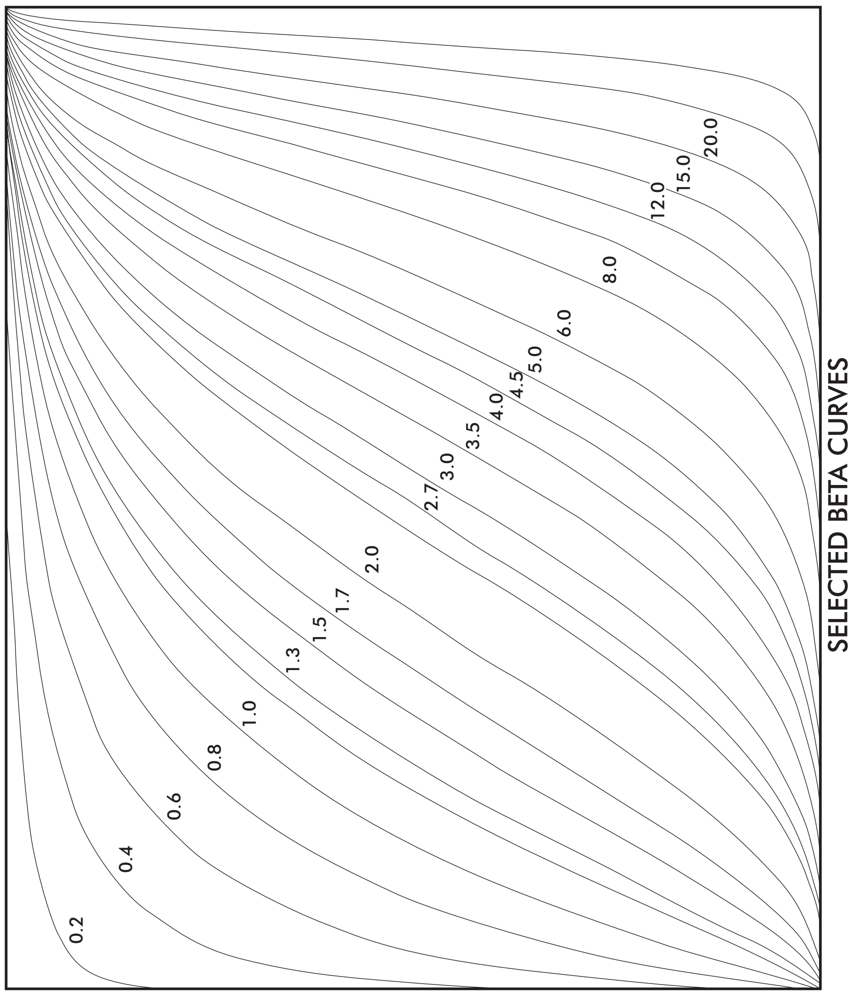

Figure 11. From Nash: Some problems of management in the production of large-scale software systems.

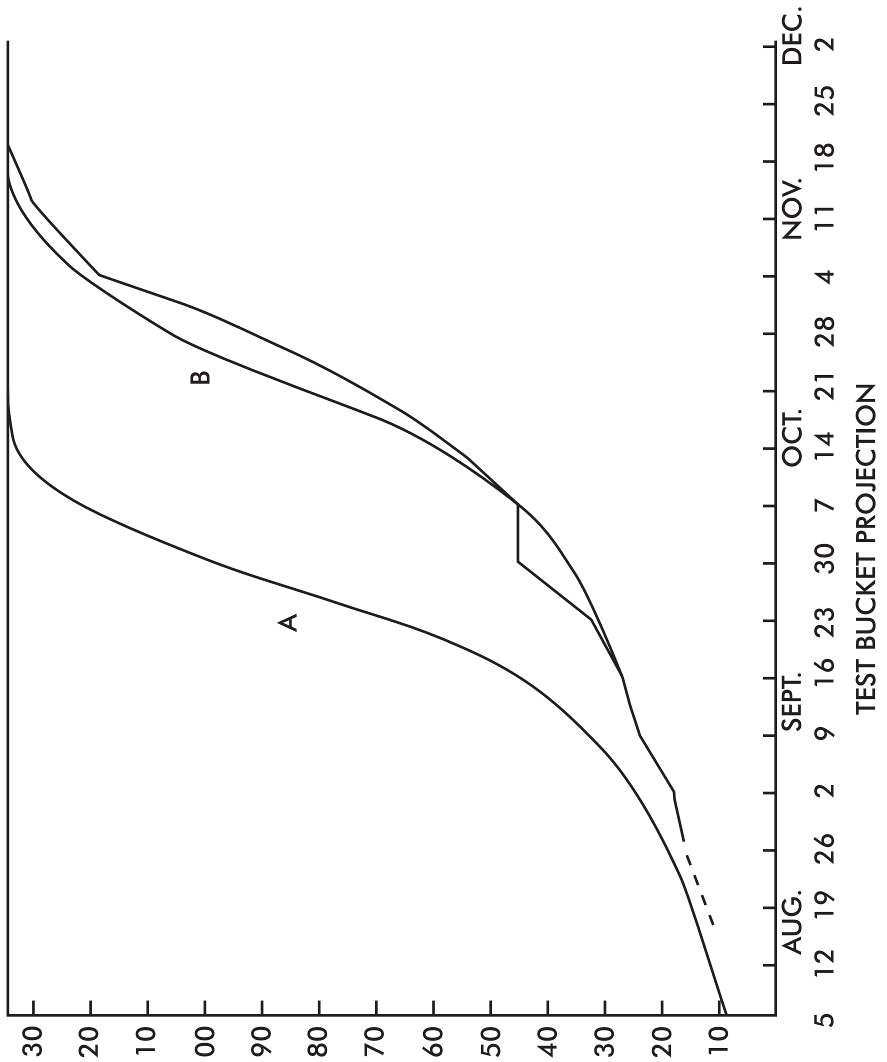

Figure 12. From Nash: Some problems of management in the production of large-scale software systems.

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.«

119

7. SPECIAL TOPICS

7.1. SOFTWARE: THE STATE OF THE ART

7.1.1. INTRODUCTION

Quite early in the conference statements of concern were made by several members about the tendency for there to be a gap, sometimes a rather large gap, between what was hoped for from a complex software system, and what was typically achieved. This topic was therefore discussed both at a special session and during the final plenary session of the conference. The essence of these discussions is given below, for the large part, as usual, in the form of more-or-less verbatim quotations.

One statement made by Buxton, given in reply to the worries of several members that the debate was unbalanced because too much attention was being paid to past and possible future software failures, is worth bringing out of context as an introduction for the reader.

Buxton: In a conference of this kind, when those present are technically competent, one has a tendency to speed up the communication by failing to state the obvious. Of course 99 percent of computers work tolerably satisfactorily; that is the obvious. There are thousands of respectable Fortran-oriented installations using many different machines and lots of good data processing applications running quite steadily; we all know that! The matter that concerns us is the sensitive edge, which is socially desperately significant.

120

7.1.2. PROBLEM AREAS

There was a considerable amount of debate on what some members chose to call the 'software crisis' or the 'software gap'. As will be seen from the quotations below, the conference members had widely differing views on the seriousness, or otherwise, of the situation, and on the extent of the problem areas.

David and Fraser: (from their *Position paper*)

»There is a widening gap between ambitions and achievements in software engineering. This gap appears in several dimensions: between promises to users and performance achieved by software, between what seems to be ultimately possible and what is achievable now and between estimates of software costs and expenditures. The gap is arising at a time when the consequences of software failure in all its aspects are becoming increasingly serious. Particularly alarming is the seemingly unavoidable fallibility of large software, since a malfunction in an advanced hardware-software system can be a matter of life and death, not only for individuals, but also for vehicles carrying hundreds of people and ultimately for nations as well.«

Hastings: I am very disturbed that an aura of gloom has fallen over this assembly. I work in an environment of many 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. Perhaps their systems do not meet everybody's need, they don't meet the time sharing people's demands for example, but I don't think software engineering should be confused with time sharing system engineering. Areas like traffic control, hospital patient monitoring, etc., are very explosive, but are very distinct from general purpose computing.

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.

Randell: There are of course many good systems, but are any of these good enough to have human life tied on-line to them, in the sense that if they fail for more than a few seconds, there is a fair chance of one or more people being killed?

121

Graham: I do not believe that the problems are related solely to on-line systems. It is my understanding that an uncritical belief in the validity of computer-produced results (from a batch-processing computer) was at least a contributory cause of a faulty aircraft design that lead to several serious air crashes.

Perlis: Many of us would agree that Multics and TSS/360 have taken a lot longer to develop than we would have wished, and that OS/360 is disappointing. However, perhaps we are exaggerating the importance of these facts. Is bad software that important to society? Are we too worried that society will lose its confidence in us?

Randell: Most of my concern stems from a perhaps over-pessimistic view of what might happen directly as a result of failure in an automated air traffic control system, for example. I am worried that our abilities as software designers and producers have been oversold.

Opler: As someone who flies in airplanes and banks in a bank I'm concerned personally about the possibility of a calamity, but I'm more concerned about the effects of software fiascos on the overall health of the industry.

Kolence: I do not like the use of the word 'crisis'. It's a very emotional word. 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.

Ross: It makes no difference if my legs, arms, brain and digestive tract are in fine working condition if I am at the moment suffering from a heart attack. I am still very much in a crisis.

Fraser: We are making great progress, but nevertheless the demands in the industry as a whole seem to be going ahead a good deal faster than our progress. We must admit this, even though such an admission is difficult.

Dijkstra: The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement.

122

7.1.3. THE UNDERLYING CAUSES

Several basic causes for what many believed were serious problem areas were suggested.

Kinslow: In my view both OS/360 and TSS/360 were straight-through, start-to-finish, no-test-development, revolutions. I have never seen an engineer build a bridge of unprecedented span, with brand new materials, for a kind of traffic never seen before — but that's exactly what has happened on OS/360 and TSS/360. At the time TSS/360 was committed for delivery within eighteen months it was drawn from two things:

1. Some hardware proposed, but not yet operational, at M.I.T.
2. Some hardware, not quite operational, at the IBM Research Center.

Buxton: A possibly fairly fundamental cause of the gap between the specifications of a large software system and what one gets in practice is a deep confusion between producing a software system for research and producing one for practical use. Instead of trying to write a system which is just like last year's, only better implemented, one invariably tries to write an entirely new and more sophisticated system. Therefore you are in fact continually embarking on research, yet your salesmen disguise this to the customer as being just a production job.

David and Fraser: (from their *Position paper*)

»The causes of this 'software gap' are many, but a basic one lies in the unfortunate telescoping of research, development and production of an operational version within a single project effort. This practice leads to slipped schedules, extensive rewriting, much lost effort, large numbers of bugs, and an inflexible and unwieldy product. It is unlikely that such a product can ever be brought to a satisfactory state of reliability or that it can be maintained and modified. Though this mixing of research, development, and production is a root cause of the ‘software gap’, there are many other contributory factors, from the lack of management talents to the employment of unqualified programmers and sheer incompetence in software design.«

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

»It seems almost automatic that software is never produced on time, never meets specification, and always exceeds its estimated cost. This conference is in fact predicated on this alarming situation. However, on 123 closer inspection the situation does not appear quite so alarming, nor unexplainable, nor incorrigible. The situation is quite analogous to that pertaining in any research and development shop in any line of business whatsoever. The ability to estimate time and cost of production comes only with product maturity and stability, with the directly applicable experience of the people involved and with a business-like approach to project control. The problem with software stems specifically from the refusal of industry to re-engineer last year’s model, from the inability of industry to allow personnel to accumulate applicable experience, and from emotional management.

…

One recent situation is worthy of note. The users of the IBM 7090 used a system called the Fortran Monitor System (FMS) quite satisfactorily for a number of years. Although its facilities were limited, it generally performed as it was supposed to. Very recently, the SDS Sigma 7 was delivered to the accompaniment of howls of anguish because it initially came equipped with only a basic operating system substantially superior to the old EMS. The root problem was that the manufacturer had promised far more and could not deliver on his promises. Did this failure lie in the inability of the software people to produce or in the ability of the sales office to overpromise?«

Gill: (from his *Position paper*)

»Software is as vital as hardware, and in many cases much more complex, but it is much less well understood. It is a new branch of engineering, in which research, development and production are not clearly distinguished, and its vital role is often overlooked. There have been many notable successes, but recent advances in hardware, together with economic pressures to meet urgent demands, have sometimes resulted in this young and immature technology of software being stretched beyond its present limit.«

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

»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. «

124

Hastings: Some of the problems are caused by users who like to buy ‘futures’ in software systems, and then ignore the problems inherent in this.

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.

7.1.4. POSSIBLE SOLUTIONS

There were several comments on possible partial solutions, and on the difficulty of finding a simple overall solution to the problems of producing large systems.

Opler: Either of the following two courses of action would be preferable to the present method of announcing a system:

1. Do all development without revealing it, and do not announce the product until it is working, and working well.
2. Announce what you are trying to do at the start of the development, specify which areas are particularly uncertain, and promise first delivery for four or five years hence.

Buxton: As long as one has good reason to believe that the research content of a system is low, one can avoid either of these extremes.

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

David and Fraser: (from their *Position paper*)

»The 'software gap' may not be immutable, but closing it will require metamorphosis in the practice of software production and its handmaiden, software design.«

Gill: (from his *Position paper*)

»We can see no swift and sure way to improve the technology, and would view any claims to achieve this with extreme caution. We believe that the only way ahead lies through the steady development of the best existing techniques.«

Ross: My main worry is in fact that somebody in a position of power will 125 recognise this crisis — it is a crisis right now, and has been for some years, and it's good that we are getting around to recognising the fact and believe someone who claims to have a breakthrough, an easy solution. The problem will take a lot of hard work to solve. There is no worse word than 'breakthrough' in discussing possible solutions.

Perlis: There are many good, albeit somewhat limited systems in the field now. I believe the best hope for a solution to our problems is evolution from these systems. Solutions are not likely to come out of designing a new SABRE system from scratch; that system should have been a warning to us five years ago.

7.1.5. SUMMARY

Rather than attempt a direct summary of the set of sometimes conflicting points of view given above, it is perhaps better to finish with just one last quotation.

Gill: (from his *Position paper*)

»It is of the utmost importance that all those responsible for large projects involving computers should take care to avoid making demands on software that go far beyond the present state of the technology, unless the very considerable risks involved can be tolerated.«

7.2. EDUCATION

Most of the remarks quoted below were made during a special discussion devoted to software engineering education.

Perlis: It is a fact that there are software engineers around today who are quite competent. There are systems in many places which are quite stable and which provide magnificent service. It is also the case that there are large numbers of efforts, containing large numbers of programs and programmers, which have no software engineers on them, that is that people function as though they did not know how to build software. I have a number of questions:

1. Is it possible to have software engineers in the numbers in which we need them, without formal software engineering education?

126

2. Is software engineering the same as Computer Science?
3. Is software engineering best provided by baccalaureate programs in universities? Or by adult education courses? Or by two year courses following the standard grade school education?
4. Do the people educated in these programs have a growing and future role in our society?
5. Will they be useful enough in a firm or government or university, and is their value such that they can distribute their talents in other activities, or must they always remain programmers?

6. What curriculum do we have in mind for software engineers, regardless of what level we choose to educate them?
7. Is software engineering really different from what we now call systems engineering?

We should answer these questions before we start giving recommendations, e.g., in the US to the National Science Foundation, that large sums be spent on such education programs.

David: May I add another question:

8. What does software engineering and computing engineering have in common with engineering education as it is defined in the United States today, or in Western Europe?

It does seem that computing engineering and software engineering, as they exist, are outside of the classical engineering education area.

Perlis: There is in the United Stated a committee called COSINE, Computer Science in Engineering Education. Their view of education of engineers in computers is primarily the view of users of computers, but not that there should be a branch of engineering having as its goal the training of a new class of engineers.

David: However, there is nothing in what they have said that would preclude a branch of engineering education concerned particularly with computing as such. We should ask ourselves seriously whether that would not be a good thing. Certainly Richard Hamming has stated that the essence of computing today is an engineering viewpoint. It certainly is not mathematics in the classical sense. In order to find colleagues who have a philosophy which may contribute to our own enterprises, engineering is a much more fruitful area than would be one of the sciences or mathematics, at least in my opinion. Incidentally, I think that a lot of engineering education in the United States is stuck in the mud.

127

Software engineering and computing engineering have an extremely important and nice aspect to them, namely that people want to work on things that meet other people's needs. They are not interested in working on abstractions entirely, they want to have an impact on the world. This is the real strength of computing today, and it is the essence of engineering.

Ross: I agree very strongly that our field is in the engineering domain, for the reason that our main purpose is to do something for somebody. To Perlis: my answers to your questions are: no; no; BA; yes; yes; question mark; yes.

Randell: I am worried about the term 'software engineering'. I would prefer a name indicating a wider scope, for instance 'data systems engineering'.

Dijkstra: We, in the Netherlands, have the title Mathematical Engineer. Software engineering seems to be the activity for the Mathematical Engineer par excellence. This seems to fit perfectly. On the one hand, we have all the aspects of an engineering activity, in that you are making something and want to see that it really works. On the other hand, our basic tools are mathematical in nature.

I want to add another question or remark to your list. You are right in saying that lots of systems really work, these are our glimmer of hope. But there is a profound difference between observing that apparently some people are able to do something, and being able to teach that ability.

Wodon: There are many places in Europe where there is no education either in hardware or in software. This conference should drive home the point that this is ridiculous.

Hume: In University of Toronto we have a graduate department of Computer Science. We also have some bachelor degrees in Computer Science, one of which is the engineering stream in a course called Engineering Science, presumably something like mathematical engineering. In this stream there is the opportunity to choose graduate work, even in the department of electrical engineering, which has set itself up as a specialist in software. The people in this department have written compilers themselves.

What really worries me about software engineering is, do universities have to engage in large software projects in order to remain experts in 128 the field of software engineering? Do they have to hire people who have had such experience with large projects, or do they have to have their professors go out and as consultants experience large software engineering projects? Then, when they come to exercise the students, do they have to have laboratories of some considerable consequence in software engineering exercises? Otherwise, who is to formulate principles of software engineering that can be used to train software engineers?

Dijkstra: To the question of how one can get experience when working in a university I have two answers: (1) If you undertake something at a university it has to be one of your main concerns to organize your activity in such a way that you get exactly the experience you need. This again must be the main concern in the choice of projects. (2) We have a Dutch proverb: 'One learns from experience', suggesting that it happens automatically. Well, this is a lie. Otherwise everyone would be very, very wise. Consequently in a university with limited resources, from the experience one has got one should try, consciously, to learn as much as possible.

David: The problem of how the software engineers will get their practice in precisely the same as in other fields of engineering, and is insoluble. This is recognized in industry, where one makes sure that the young engineers coming in will get the proper kind of experience in time. Really it seems that the problem is less serious in software engineering than in other fields.

Berghuis: We need students better trained in standards, standards of communication, of documentation, of set-up, and of use of software.

Fraser: I was impressed by Douglas Ross's Session [section 5.3.2] and I am convinced that there is a future in software science or technology. Nevertheless, I am convinced that much of the game in which we are involved is one of making the best of the world around us, understanding what the world wants and matching what science can offer. This, to my mind, is truly engineering. What worries me about the courses I have been associated with, is that they have been courses in mathematics, rather than courses in engineering. What is lacking is an awareness of the requirements of the world. One indication of this is the complaint that the graduates know nothing about standards and discipline.

McIlroy: With Fraser I am concerned about the connection between software 129 engineering and the real world. There is a difference between writing programs and designing bridges. A program may be written with the sole purpose to help write better programs, and many of us here have spent our life writing programs from this pure software attitude. More than any other engineering field, software engineering in universities must consciously strive to give its students contact beyond its boundaries.

Galler: I would like to include under education the continuing education of professional people in the field, stressing an awareness of what others have done. I am appalled at the lack of attempts to educate people in what others are doing that we see throughout this industry.

Perlis: Most of the Computer Science programs in the United States, at least at the graduate level, are producing faculty for other Computer Science departments. This is appropriate, because we must first staff these departments. But it is also the case that almost all the Computer Science departments are turning out PhD's who do not do computer software engineering under any stretch of that term's meaning. You have to look hard to find anything that is dedicated to utility as a goal. Under no stretch of the imagination can one say that Computer Science, at least in the United States, is fostering software engineering. In the United States National Academy of Sciences Research Board one education committee being formed is precisely to study software engineering as a possible engineering education activity. NATO would probably not be making a mistake in holding another conference within the near future concerning software engineering education.

7.3. SOFTWARE PRICING

7.3.1 INTRODUCTION

A special session on the issue of software pricing was arranged in response to the generally expressed feeling of the importance of this topic in relation to the whole future of software engineering. During the session it became clear that one of the major causes of divergent views on whether software should be priced separately 130 *from hardware was the fact that people had differing aims and also differing estimates of the possible effects of separate pricing. The session is reported below by summarising the arguments put forward by various people for and against separate pricing, and about the desirability of preventing* *hardware manufacturers from providing software. The discussion lasted over three hours, ending after midnight, yet was well-attended throughout. At the end of the discussion the opinion of those present was tested. It was clear that a large majority were personally in favour of separate pricing of software.*

7.3.2. ARGUMENTS IN FAVOUR OF SEPARATE PRICING

1. Software is of obvious importance and yet is treated as though it were of no financial value. If software had a value defined in terms of money, users could express their opinion of the worth of a system by deciding to accept or reject it at a given price.

2. Systems organisations, which buy hardware, and either buy or produce software in order to sell complete systems, would flourish.

3. We are presently in a transition stage — when it ends, by far the largest proportion of people influenced by or using computers will be application oriented. Application software will become independent, in a sound practical way, of the underlying hardware and operating systems, and will be separately priced. However, the knowledge built into application programs will have to be protected by some means such as patenting.

4. From the viewpoint of the hardware manufacturer software is currently a sales aid. Most manufacturers are really in the business of selling computer time, not computers, and hence have no primary interest in making its computers run faster than the minimum speed required to sell them.

5. What the user needs is better software — he does not care too much where it comes from. The increased competition ensuing from separate pricing would cause an increase in quality of software produced 131 by hardware manufacturers as well as increasing the number of sources of software available to the user.

6. Until software is separately priced it is difficult for the software talents of the smaller hardware manufacturers, the software houses and the universities to be effectively utilised. It is these sources and not IBM which have produced the majority of good systems and languages, such as BASIC, JOSS, SNOBOL, LISP, MTS.

7. Separate pricing would benefit hardware manufacturers, and particularly IBM, in controlling the production of software, and enabling cost/performance figures to be calculated, by making normal cost accounting practices immediately applicable.

8. Even if hardware prices decrease only slightly as a result of separate pricing, this is a small gain to the user and anyway is not the main point at issue, which is the improved quality and service that would result.

9. Manufacturers will have a considerable lead over other software producers in designing software for new machines prior to their announcement; but this is unimportant. Most computers have two or three different software systems associated with them during their lifetime, and not just the one that the manufacturer devised before announcing the hardware. Furthermore, the lead time is not always used — IBM had barely started to plan OS/360 at the time System 360 was announced.

7.3.3. ARGUMENTS AGAINST SEPARATE PRICING

1. There will not be enough decrease in hardware prices to have any noticeable financial effect.

2. Users are worried about the service they get from the total system. Separate pricing would widen the gap between hardware and software design.

3. Purchase of software from multiple vendors will create a tower of Babel. At the time of the IBM 704, even two competing assemblers caused much dissension.

4. Separate pricing would enable the software produced by manufacturers to find its true price in a free economy. It would very 132 likely be priced at a level such that the total system price would be significantly increased, despite the decrease in hardware prices.

5. Hardware manufacturers have such an advantage in the knowledge of future systems and devices, and in the availability of hardware prototypes, that independent software producers could not compete.
6. The lack of separate pricing of software has not prevented the growth of independent service companies such as UCC.
7. Separate pricing may bring IBM, which currently owns the largest software house in the world (namely their Federal Systems Division) in more direct competition with the independent software houses. Thus, contrary to prevalent opinion, the independents may have more to lose than to gain by separate pricing.
8. Some people undoubtedly argue in favour of separate pricing because of their worries about the concentration of power in the hands of a single manufacturer. However, separate pricing may well be of most benefit to IBM
9. Software belongs to the world of ideas, like music and mathematics, and should be treated accordingly.

7.3.4. SPLITTING SOFTWARE AND HARDWARE PRODUCTION

7.3.4.1. THE ARGUMENT IN FAVOUR

A user's dependence on his computing system is such that he should not have to rely on a single manufacturer for all aspects of it. The dangers inherent in an organization with sufficient capital resources producing comprehensive software for any industry, educational activity, research organisation or government agency are considerable and far outweigh, for instance, those of a national or international data bank. A hardware manufacturer who also produces the software on which business and industry depend has the reins to almost unlimited power. Preventing hardware manufacturers from producing any software for sale or gift would be a great encouragement to competition essential in an area which is so broad that it knows no boundaries at all.

133

7.3.4.2. THE ARGUMENT AGAINST

The above argument is persuasive in only a superficial way. It is the 'there oughta be a law' type of reaction to a worrying situation. The computing industry is still embryonic, and a law such as this would cause an inflexibility that we would later regret. We should build on strength and not on weakness — when we begin to fear competence, if it happens to be correlated with bigness, we are trying to build on weakness and not on strength.

134 135

8. ADDRESSES

8.1. KEYNOTE SPEECH, BY A.J. PERLIS

Why has this meeting been scheduled?

Why have we agreed to participate?

I believe it is because we recognize that a practical problem of considerable difficulty and importance has arisen: The successful design, production and maintenance of useful software systems. The importance is obvious and the more so since we see only greater growth in demands and requirements in the future. The consequences of poor performance, poor design, instability and mismatching of promise and performance are not going to be limited to the computing fraternity, or even their nearest neighbors, but will affect considerable sections of our society whose ability to forgive is inversely proportional to their ignorance of the difficulties we face. The source of difficulty is distributed through the whole problem, easy to identify, and yet its cure is hard to pinpoint so that systematic improvement can be gotten.

Our problem has arisen from a change of scale which we do not yet know how to reduce to alphabetic proportions. Furthermore we must assume that additional magnification of goal will take place without necessarily being preceded by the emergence of a satisfactory theory or an organized production of tools that will permit work and costs to fall on growth curves which lie significantly below those which now exist. For example, we can see coming the need for systems which permit cooperation, e.g., between engineering and management information. Not only must we know how to build special purpose systems but how to combine them into larger ones.

We work with software knowing that the design of a software system always seems to make some complex functions available with 136 ease and others, seemingly little different, available only in a cock-eyed way. Such shortcomings in design are probably inevitable even in the very best systems and are simply consequences of the inevitable disparity between the degree of connectivity of human thought processes and those of a programmed system. It is also true that every system creates, through the very pattern of its usage, a set of anticipated bottlenecks. To avoid these bottlenecks users of systems learn to accommodate. Every software system thus imposes an etiquette on users, as every system which is created is itself a recognition of an existing etiquette.

Software is intimately tied to language even though it obviously involves more. Computer languages are processed by software and language is used to command the processing. This causes a problem and suggests a cure. The problem arises out of the relative ease with which out of our involvement with language we propose innovation. The cure is that the same ease of innovation can be focused on improvements which will help in the creation of systems. Programming tools are created — or innovated — to harness the power of an already existing hardware system. The specialization of these tools is of major importance in making the computer available to a wide range of genius for application to a wide range of purpose. A major function of these tools is the utilization of equivalences, i.e., a program which is easily written is shown equivalent to one which is easily executed by the computer. Another major function of the tools is the management of data with their appropriate operations and language inventions. Here the essential innovations are those which force sequencing to be handled implicitly and selectively by explicit command, e.g., as in the use of patterns, keys, generators, etc. the establishment of equivalences and the management of data are often accomplished by the invention of virtual computers which themselves are tools existing only as expressions in language.

To my mind the natural way to explore and manage the software problem is through the design of virtual machines — and all that the concept of machines implies — the establishment of relevant states, their transformations, the design of communication channels, the nature of and magnitude of storages, the natural 137 sets of operations, the I/O problem, etc. We have before us many examples which have, in more restricted areas of programming, worked amazingly well. The symbol manipulating languages — IPL and LISP to name two — have been so organized that the problems coded in them have yielded codes organized as complexes of machines, each one of which is much like those from which it was constructed. Indeed certain operations in the primitive machine carry over to all levels in these machine cascades because they are regarded as indigenous to the class of tasks handled.

Consider the case of operating systems, these are certainly hierarchical. Are there not primitive operations in these systems which are indigenous to All levels? Can we not design a background machine — and hence a set of such machines — for this hierarchy of systems? If one were so designing, would not a point of departure be a set of primitives for handling interprocess communication and, in particular, for handling interrupts? For example, is it not so that the attempt to interrupt invokes in the interruptor an interrupt which will surely be processed? Is it not the case that interrupts plant ultimate interrupts for restoring the status which held prior to the given ones?

We are going to concern ourselves here with the design, production, and servicing of objects which are complex and automatic, mechanical and symbolic, whose performance and decay, breakdown and efficiency depend in only very weak ways on the laws of physics. Their structure and rigidity depend as much on the social laws governing their usage as on their internal constraints. It should be clear that when we speak of production we do not speak so much of mass production as of specialty production and mass distribution. The problems of mass producing software are clearly less important than that of producing software at all.

This is the first conference ever held on software engineering and it behooves us to take this conference quite seriously since it will likely set the tone of future work in this field in much the same way that Algol did. We should take quite seriously both the scientific and engineering components of software, but our concentration must be on the latter.

138

Our problems arise from demands, appetites, and our exuberant optimism. They are magnified by the unevenly trained personnel with which we work. In our deliberations we will not, I believe, be able to avoid the education problem — software engineering does not exist without software engineers. Stability in our goals, products and performances can only be achieved when accompanied by a sufficient supply of workers who are properly trained, motivated, and interchangeable. Their production should be the subject of another meeting. Our goal this week is the conversion of mushyware to firmware, to transmute our products from Jello to crystals.

8.2. MASS PRODUCED SOFTWARE COMPONENTS, BY M.D. MCILROY

ABSTRACT

Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and timespace performance. Existing sources of components — manufacturers, software houses, users' groups and algorithm collections — lack the breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors. The talk will examine the kinds of variability necessary in software components, ways of producing useful inventories, types of components that are ripe for such standardization, and methods of instituting pilot production.

The Software Industry is Not Industrialized.

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. 139 Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software.

In the phrase 'mass production techniques,' my emphasis is on 'techniques' and not on mass production plain. Of course mass production, in the sense of limitless replication of a prototype, is trivial for software. But certain ideas from industrial technique I claim are relevant. The idea of subassemblies carries over directly and is well exploited. The idea of interchangeable parts corresponds roughly to our term 'modularity,' and is fitfully respected. The idea of machine tools has an analogue in assembly programs and compilers. Yet this fragile analogy is belied when we seek for analogues of other tangible symbols of mass production. There do not exist manufacturers of standard parts, much less catalogues of standard parts. One may not order parts to individual specifications of size, ruggedness, speed, capacity, precision or character set.

The pinnacle of software is systems — systems to the exclusion of almost all other considerations. Components, dignified as a hardware field, is unknown as a legitimate branch of software. When we undertake to write a compiler, we begin by saying ‘What table mechanism shall we build?’ Not, ‘What mechanism shall we **use**? but ‘What mechanism shall we **build**? I claim we have done enough of this to start taking such things off the shelf.

Software Components

My thesis is that the software industry is weakly founded, and that one aspect of this weakness is the absence of a software components subindustry. We have enough experience to perceive the outline of such a subindustry. I intend to elaborate this outline a little, but I suspect that the very name ‘software components’ has probably already conjured up for you an idea of how the industry could operate. I shall also argue that a components industry could 140 be immensely useful, and suggest why it hasn’t materialized. Finally I shall raise the question of starting up a ‘pilot plant’ for software components.

The most important characteristic of a software components industry is that it will offer families of routines for any given job. No user of a particular member of a family should pay a penalty, in unwanted generality, for the fact that he is employing a standard model routine. In other words, the purchaser of a component from a family will choose one tailored to his exact needs. He will consult a catalogue offering routines in varying degrees of precision, robustness, time-space performance, and generality. He will be confident that each routine in the family is of high quality — reliable and efficient. He will expect the routine to be intelligible, doubtless expressed in a higher level language appropriate to the purpose of the component, though not necessarily instantly compilable in any processor he has for his machine. He will expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, he should be able safely to regard components as black boxes.

Thus the builder of an assembler will be able to say ‘I **will use** a String Associates A4 symbol table, in size 500x8,’ and therewith consider it done. As a bonus he may later experiment with alternatives to this choice, without incurring extreme costs.

A Familiar Example

Consider the lowly sine routine. How many should a standard catalogue offer? Offhand one thinks of several dimensions along which we wish to have variability:

Precision, for which perhaps ten different approximating functions might suffice

Floating-vs-fixed computation

Argument ranges 0-$\pi/2$, 0-2π, also -$\pi/2$ to $\pi/2$, -π to π, -big to +big

Robustness — ranging from no argument validation through signaling of complete loss of significance, to signaling of specified range violations

141

We have here 10 precisions, 2 scalings, 5 ranges and 3 robustnesses. The last range option and the last robustness option are actually arbitrary parameters specifiable by the user. This gives us a basic inventory of 300 sine routines. In addition one might expect a complete catalogue to include a measurement-standard sine routine, which would deliver (at a price) a result of any accuracy specified at run time. Another dimension of variability, which is perhaps difficult to implement, as it caters for very detailed needs is

Time-space tradeoff by table lookup, adjustable in several ‘subdimensions’:

(a) Table size

(b) Quantization of inputs (e.g., the inputs are known to be integral numbers of degrees)

Another possibility is

(c) Taking advantage of known properties of expected input sequences, for example profiting from the occurrence of successive calls for sine and cosine of the same argument.

A company setting out to write 300 sine routines one at a time and hoping to recoup on volume sales would certainly go broke. I can’t imagine some of their catalogue items ever being ordered. Fortunately the cost of offering such an 'inventory' need not be nearly 300 times the cost of keeping one routine. Automated techniques exist for generating approximations of different degrees of precision. Various editing and binding techniques are possible for inserting or deleting code pertinent to each degree of robustness. Perhaps only the floating-vs-fixed dichotomy would actually necessitate fundamentally different routines. Thus it seems that the basic inventory would not be hard to create.

The example of the sine routine re-emphasizes an interesting fact about this business. It is safe to assert that almost all sines are computed in floating point these days, yet that would not justify discarding the fixed point option, for that could well throw away a large part of the business in distinct tailor-made routines for myriads of small process-control and other real-time 142 applications on all sorts of different hardware. 'Mass production' of software means multiplicity of what manufacturing industry would call 'models,' or 'sizes' rather than multiplicity of replicates of each.

Parameterized Families of Components

One phrase contains much of the secret of making families of software components: 'binding time.' This is an 'in' phrase this year, but it is more popular in theory than in the field. Just about the only applications of multiple binding times I can think of are sort generators and the so-called 'Sysgen' types of application: filling in parameters at the time routines are compiled to control table sizes, and to some extent to control choice among several bodies of code. The best known of these, IBM's OS/360 Sysgen is indeed elaborate — software houses have set themselves up as experts on this job. Sysgen differs, though, in a couple of ways from what I have in mind as the way a software components industry might operate.

First, Sysgen creates systems not by construction, but rather by excision, from an intentionally fat model. The types of adjustment in Sysgen are fairly limited. For example it can allocate differing amounts of space to a compiler, but it can't adjust the width of list link fields in proportion to the size of the list space. A components industry on the other hand, not producing components for application to one specific system, would have to be flexible in more dimensions, and would have to provide routines whose niches in a system were less clearly delineated.

Second, Sysgen is not intended to reduce object code or running time. Typically Sysgen provides for the presetting of defaults, such as whether object code listings are or are not standard output from a compiler. The entire run-time apparatus for interrogating and executing options is still there, even though a customer might guarantee he'd never use it were it indeed profitable to refrain. Going back to the sine routine, this is somewhat like building a low precision routine by computing in high precision and then carefully throwing away the less significant bits.

143

Having shown that sysgen isn't the exact pattern for a components industry, I hasten to add that in spirit it is almost the only way a successful components industry could operate. To purvey a rational spectrum of high quality components a fabricator would have to systemize his production. One could not stock 300 sine routines unless they were all in some sense instances of just a few models, highly parameterized, in which all but a few parameters were intended to be permanently bound before run time. One might call these early-bound parameters 'sale time' parameters.

Many of the parameters of a basic software component will be qualitatively different from the parameters of routines we know today. There will be at least

Choice of Precision. Taken in a generalized sense precision includes things like width of characters, and size of address or pointer fields.

Choice of Robustness. The exact tradeoff between reliability and compactness in space and time can strongly affect the performance of a system. This aspect of parameterization and the next will probably rank first in importance to customers.

Choice of Generality. The degree to which parameters are left adjustable at run time.

Choice of Time-space behavior.

Choice of Algorithm. In numerical routines, as exemplified by those in the CACM, this choice is quite well catered for already. For nonnumerical routines, however, this choice must usually be decided on the basis of folklore. As some nonnumerical algorithms are often spectacularly unsuitable for particular hardware, a wide choice is perhaps even more imperative for them.

Choice of Interfaces. Routines that use several inputs and yield several outputs should come in a variety of interface styles. For example, these different styles of communicating error outputs should be available:

a. Alternate returns

b. Error code return

c. Call an error handler

d. Signal (in the sense of PL/1)

144

Another example of interface variability is that the dimensions of matrix parameters should be receivable in ways characteristic of several major programming languages.

Choice of Accessing method. Different storage accessing disciplines should be supported, so that a customer could choose that best fitting his requirements in speed and space, the addressing capabilities of his hardware, or his taste in programming style.

Choice of Data structures. Already touched upon under the topic of interfaces, this delicate matter requires careful planning so that algorithms be as insensitive to changes of data structure as possible. When radically different structures are useful for similar problems (e.g., incidence matrix and list representations for graphs), several algorithms may be required.

Application Areas

We have to begin thinking small. Despite advertisements to the effect that whole compilers are available on a 'virtually off-the-shelf basis, I don't think we are ready to make software subassemblles of that size on a production basis. More promising components to begin with are these:

Numerical approximation routines. These are very well understood, and the dimensions of variability for these routines are also quite clear. Certain other numerical processes aren't such good candidates; root finders and differential equation routines, for instance are still matters for research, not mass production. Still other 'numerical' processes) such as matrix inversion routines, are simply logical patterns for sequencing that are almost devoid of variability. These might be sold by a components industry for completeness' sake, but they can be just as well taken from the CACM.

Input-output conversion. The basic pieces here are radix conversion routines, some trivial scanning routines, and format crackers. From a well-designed collection of families it should be possible to fabricate anything from a simple on-line octal package for a small laboratory computer to a Fortran IV conversion package. The variability here, especially in the matter of accuracy and robustness 145 is substantial. Considerable planning will evidently be needed to get sufficient flexibility without having too many basically different routines.

Two and three dimensional geometry. Applications of this sort are going on a very wide class of machines, and today are usually kept proprietary. One can easily list a few dozen fundamental routines for geometry. The sticky dimension of variability here is in data structures. Depending on which aspect of geometrical figures is considered fundamental — points, surfaces, topology, etc. — quite different routines will be required. A complete line ought to cater for different abstract structures, and also be insensitive to concrete structures.

Text processing. Nobody uses anybody else's general parsers or scanners today, partly because a routine general enough to fulfill any particular individual needs probably has so much generality as to be inefficient. The principle of variable binding times could be very fruitfully exploited here. Among the corpus of routines in this area would be dictionary builders and lookup routines, scanners, and output synthesizers, all capable of working on continuous streams, on unit records, and various linked list formats, and under access modes suitable to various hardware.

Storage management. Dynamic storage allocation is a popular topic for publication, about which not enough real knowledge yet exists. Before constructing a product line for this application, one ought to do considerable comparison of known schemes working in practical environments. Nevertheless storage management is so important, especially for text manipulation, that it should be an early candidate.

The Market

Coming from one of the larger sophisticated users of machines, I have ample opportunity to see the tragic waste of current software writing techniques. At Bell Telephone Laboratories we have about 100 general purpose machines from a dozen manufacturers. Even though many are dedicated to special applications, a tremendous 146 amount of similar software must be written for each. All need input-output conversion, sometimes only single alphabetic characters and octal numbers, some full-blown Fortran style I/O. All need assemblers and could use microprocessors, though not necessarily compiling on the same hardware. Many need basic numerical routines or sequence generators. Most want speed at all costs, a few want considerable robustness.

Needless to say much of this support programming is done suboptimally and at a severe scientific penalty of diverting the machine's owners from their central investigations. To construct these systems of high-class componentry we would have to surround each of some 50 machines with a permanent coterie of software specialists. Were it possible quickly and confidently to avail ourselves of the best there is in support algorithms, a team of software consultants would be able to guide scientists towards rapid and improved solutions to the more mundane support problems of their personal systems.

In describing the way Bell Laboratories might use software components, I have intended to described the market in microcosm. Bell Laboratories is not typical of computer users. As a research and development establishment, it must perforce spend more of its time sharpening its tools, and less using them than does a production computing shop. But it is exactly such a systems-oriented market toward which a components industry would be directed.

The market would consist of specialists in system building, who would be able to use tried parts for all the more commonplace parts of their systems. The biggest customers of all would be the manufacturers. (Were they not it would be a sure sign that the offered products weren't good enough.) The ultimate consumer of systems based on components ought to see considerably improved reliability and performance, as it would become possible to expend proportionally more effort on critical parts of systems, and also to avoid the now prevalent failings of the more mundane parts of systems, which have been specified by experts, and have then been written by hacks.

147

Present Day Suppliers

You may ask, well don't we have exactly what I've been calling for already in several places? What about the CACM collected algorithms? What about users groups? What about software houses? And what about manufacturers' enormous software packages?

None of these sources caters exactly for the purpose I have in mind, nor do I think it likely that any of them will actually evolve to fill the need.

The CACM algorithms, in a limited field, perhaps come closer to being a generally available off-the-shelf product than do the commercial products, but they suffer some strong deficiencies. First they are an ingathering of personal contributions, often stylistically varied. They fit into no plan, for the editor can only publish that which the authors volunteer. Second, by being effectively bound to a single compilable language, they achieve refereeability but must perforce completely avoid algorithms for which Algol is unsuited or else use circumlocutions so abominable that the product can only be regarded as a toy. Third, as an adjunct of a learned society, the CACM algorithms section can not deal in large numbers of variants of the same algorithm; variability can only be provided by expensive run time parameters.

User's groups I think can be dismissed summarily and I will spare you a harangue on their deficiencies.

Software houses generally do not have the resources to develop their own product lines; their work must be financed, and large financing can usually only be obtained for large products. So we see the software houses purveying systems, or very big programs, such as Fortran compilers, linear programming packages or flowcharters. I do not expect to see any software house advertising a family of Bessel functions or symbol tabling routines in the predictable future.

The manufacturers produce unbelievable amounts of software. Generally, as this is the stuff that gets used most heavily it is all pretty reliable, a good conservative grey, that doesn't include the best routine for anything, but that is better than the average 148 programmer is likely to make. As we heard yesterday manufacturers tend to be rather pragmatic in their choice of methods. They strike largely reasonable balances between generality and specificity and seldom use absolutely inappropriate approaches in any individual software component. But the profit motive wherefrom springs these virtues also begets their prime hangup — systems now. The system comes first; components are merely annoying incidentals. Out of these treadmills I don't expect to see high class components of general utility appear.

A Components Factory

Having shown that it is unlikely to be born among the traditional suppliers of software I turn now to the question of just how a components industry might get started.

There is some critical size to which the industry must attain before it becomes useful. Our purveyor of 300 sine routines would probably go broke waiting for customers if that's all he offered, just as an electronics firm selling circuit modules for only one purpose would have trouble in the market.

It will take some time to develop a useful inventory, and during that time money and talent will be needed. The first source of support that comes to mind is governmental, perhaps channeled through semi-independent research corporations. It seems that the fact that government is the biggest user and owner of machines should provide sufficient incentive for such an undertaking that has promise for making an across-the-board improvement in systems development.

Even before founding a pilot plant, one would be wise to have demonstrated techniques for creating a parameterized family of routines for a couple of familiar purposes, say a sine routine and a Fortran I/O module. These routines should be shown to be useable as replacements in a number of radically different environments. This demonstration could be undertaken by a governmental agency, a research contractor, or by a big user, but certainly without expectation of immediate payoff.

149

The industrial orientation of a pilot plant must be constantly borne in mind. I think that the whole project is an improbable one for university research. Research-calibre talent will be needed to do the job with satisfactory economy and reliability, but the guiding spirit of the undertaking must be production oriented. The ability to produce members of a family is not enough. Distribution, cataloguing, and rational planning of the mix of product families will in the long run be more important to the success of the venture than will be the purely technical achievement.

The personnel of a pilot plant should look like the personnel on many big software projects, with the masses of coders removed. Very good planning, and strongly product-minded supervision will be needed. There will be perhaps more research flavor included than might be on an ordinary software project, because the level of programming here will be more abstract: Much of the work will be in creating generators of routines rather than in making the routines themselves.

Testing will have to be done in several ways. Each member of a family will doubtless be tested against some very general model to assure that sale-time binding causes no degradation over runtime binding. Product test will involve transliterating the routines to fit in representative hardware. By monitoring the ease with which fairly junior people do product test, managers could estimate the clarity of the product, which is important in predicting customer acceptance.

Distribution will be a ticklish problem. Quick delivery may well be a components purveyor's most valuable sales stimulant. One instantly thinks of distribution by communication link. Then even very small components might be profitably marketed. The catalogue will be equally important. A comprehensive and physically condensed document like the Sears-Roebuck catalogue is what I would like to have for my own were I purchasing components.

Once a corpus of product lines became established and profit potential demonstrated, I would expect software houses to take over the industry. Indeed, were outside support long needed, I would say the venture had failed (and try to forget I had ever proposed it).

150

Touching on Standards

I don't think a components industry can be standardized into existence. As is usual with standards, it would be rash to standardize before we have the models. Language standards, provided they are loose enough not to prevent useful modes of computation, will of course be helpful. Quite soon one would expect a components industry to converge on a few standard types of interface. Experience will doubtless reveal other standards to be helpful, for example popular word sizes and character sets, but again unless the standards encompass the bulk of software systems (as distinguished from users), the components industry will die for lack of market.

Summary

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

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

151

8.2.1. DISCUSSION

Ross: What McIlroy has been talking about are things we have been playing with. For example, in the AED system we have the so-called feature-feature. This enables us to get round the problem of loaders. We can always embed our system in whatever loader system is available. The problem of binding is very much interlocked there, so we are at the mercy of the environment. An example is a generalized alarm reporting system in which you can either report things on the fly, or put out all kinds of dynamic information. The same system gives 14 different versions of the alarm handling. Macro-expansion seems to me to be the starting place for some of the technical problems that have to be solved in order to put these very important ideas into practice.

McIlroy: It seems that you have automated some of types variability that I thought were more speculative.

Opler: The TOOL system produced six years ago for Honeywell was complementary to the one McIlroy described. It has facilities for putting thing together, but it did not provide the components. The difficulty we had was that we produced rudimentary components to see how the system would work, but the people for whom we developed the system did not understand that they were to provide their own components, so they just complained that the system was not good. But I am very enthusiastic about what you suggest.

Perlis: The GP system of the first Univac was a system for developing personalized software as long as you stayed on that machine. The authors of this system asked me: how would one generalize this to other computers? They did not know how to do it at the time, and I suppose it has not been done. I have a question for McIlroy. I did not hear you mention what to me is the most obvious parametrizations, namely to build generalized business data file handling systems. I understand that Informatics has one out which everybody says is OK, but — . This seems to be a typical attitude to parameterized systems.

McIlroy: My reason for leaving that out is that this is an area that I don't know about.

Perlis: Probably it would be one of the easiest areas, and one with the most customers. Before d'Agapeyeff talks I have another comment. [*Laughter*] 152 Specialists in every part of software have a curious vision of the world: All parts of software but his are simple and easily parametrized; his is totally variable.

d'Agapeyeff: There is no package which has received more attention from manufacturers than file handling. Yet there is hardly a major system that I know of that is relying solely on the standard system produced by the manufacturer. It is extremely difficult to construct this software in a way that is efficient, reliable, and convenient for all systems and where the nature of the package does not impose itself upon the user. The reason is that you cannot atomize it. Where work has been successful it tends to be concerned with packages that have some structure. When you get down to small units it is not economic to make them applicable to a large set of users, using different machines with different languages, and to do all the binding work, such that it doesn't take twice as long to find out how to load it. The problems with Sysgen are not to be dispensed with, they are inherent. But why do we need to take atoms down from the shelf? What you want is a description which you can understand, because the time taken to code it into your own system is really very small. In that way you can insert your own nuances. The first step in your direction should be better descriptions.

Endres: Two notes of caution: You discarded the algorithms in the Comm. ACM in part because they are written in high-level language, so I understand that you refer to routines written in a more machine oriented language. I think you oversimplify the problem of transliteration. Or do you assume a de facto machine standard? Second question: You refer to the problems of Sysgen, where you cut out pieces from a large collection. If instead you want to put together systems, I think the problems of Sysgen become a dimension larger. Who will bear this cost, and maintain the system?

McIlroy: The algorithms in the *Communications of the ACM* effectively use one language, which is suitable for a particular class of applications. This may not be the right one for things like input/output packages. On the second question: I am convinced, with you, that at first it will be harder to build systems by accretion, rather than by excision. The people who build components will have to be skilled systems builders, not run of the mill users.

153

Kjeldaas: I strongly favor this idea. I think the examples mentioned are within the state of the art. However, later we will want macros needing parameters having more intricate relations, for instance if you want some functional relationship between the parameters. We will need some language for describing the parameters. Another point: documentation can also be included in this. When you have given the parameters to the program, you can give the same parameters to the documentation, and the documentation for the particular use can be produced automatically. Catering for different machines will raise big problems, needing research.

Kolence: May I stress one point: McIlroy stated that the industrialization is concerned with the design, not the replication process. We are concerned with a mass design problem. In talking about the implementation of software components, the whole concept of how one designs software is ignored. Yet this is a key thing.

Naur: What I like about this is the stress on basic building principles, and on the fact that big systems are made from smaller components. This has a strong bearing on education. What we want in education, particularly at the more elementary level, is to start indoctrinating the knowledge of the components of our systems. A comparison with our hardware colleagues is relevant. Why are they so much more successful than we are? I believe that one strong reason is that there is a well established field of electronic engineering, that the young people start learning about Ohm's Law at the age of fourteen or thereabouts, and that resistors and the like are known components with characteristics which have been expounded at length at the early level of education. The component principles of our systems must be sorted out in such a form that they can be put into elementary education.

Gill: Two points: first on the catalogue question. I hope we can do better than the Sears-Roebuck catalogue. Surely what we want is a computerized conversational catalogue. Second point: what is it that you actually sell when you sell a piece of software, what exactly does a software contract look like?

Barton: McIlroy's talk was so well done that it took me about three minutes to realize what is wrong this idea. Another compliment: If I were running Intergalactic Software, I would hire McIlroy for a manager. Now the serious point: Over the last few years I have taught the ACM Course 154 *Information Structures* and used the game not to let anyone code or write anything in any programming language at all. We have just thought about data representations. If in this way you get people over the habit of writing code right away, of thinking procedurally, then some very different views on information representations come to view. In McIlroy's talk about standard components having to do with data structures I have the feeling that this is not a problem to take out of the universities yet. Now a heretical view: I don't think we have softened up enough things in machines yet. I don't think we will get anywhere trying to quantify the space-time trade-off unless we discard fixed word sizes, fixed character sizes, fixed numerical representations, altogether in machines. Without these, the thing proposed by McIlroy will prove to be just not quite practical.

Fraser: I wish to take issue with d'Agapeyeff. I think it will be possible to parameterize data representation and file management. From a particular file system experience I learned two lessons: first, there are a large number of parameters, to be selected in a non-mutually-exclusive manner. The selection of the parameters is so complicated that it is appropriate to put a compiler on the front end of the software distribution mechanism. Perhaps we are talking more about compilers than we realize. Concerning catalogues: in England a catalogue of building materials is a very ad hoc catalogue, you have left hand flanges to go with left hand gates, etc. I think the catalogue is likely to be ad hoc in that nature, rather than like an electronics catalogue where the components are more interchangeable.

The second issue is the question of writing this compiler. Our file management generator effectively would generate a large number of different file management systems, very considerably in excess of the 300 that McIlroy mentioned. There was no question of testing all of these. We produced an ad hoc solution to this problem, but until more research is done on this problem I don't think McIlroy's suggestion is realistic.

Graham: I will speak of an adjunct to this idea. In Multics we used a subset of PL/1, although PL/1 is quite inadequate, in that the primitive operations of the language are not really suited for system design. In Multics you do a lot of directory management, simple operations like adding and deleting entries, but in a complicated directory. With a higher 155 level language with these operations as primitives one could easily write a new system. By simulating the primitives one could test the performance of the system before actually building it. If one had McIlroy's catalogue stored in the system, with the timings of a lot of routines, then the simulation backing up this higher-level language could in fact refer to the catalogue and use the actual timings for a new machine that this company offered and get realistic timings. Another point, I wish to rebut is McIlroy's suggestion that this is not for universities; I think it is, There are very difficult problems in this area, such as parametrizing more sophisticated routines, in particular those in the compiler area. These are fit for universities.

Bemer: I agree that the catalogue method is not a suitable one. We don't have the descriptors to go searching. There is nothing so poorly described as data formats, there are no standards, and no sign that they are being developed. Before we have these we won't have the components.

McIlroy: It is for that reason that I suggest the Sears-Roebuck type now. On-line searching may not be the right answer yet.

156 157

9. WORKING PAPERS

Below we give first the list of working papers of the conference. Quotations from most of these papers are given throughout the report. The list is followed by a few papers which it was found desirable, for one reason or another, to reproduce in full. In the list these papers are marked with the page number where they appear in the report.

d'Agapeyeff: Reducing the cost of software.

Babcock: Variations on software available to the user.

Babcock: Reprogramming.

Bemer, Fraser, Glennie, Opler, and Wiehle: Classification of subject matter [pages 160–164].

Bemer: Checklist for planning software system production [pages 165–180].

Bemer: Machine-controlled production environment — tools for technical control of production.

Berghuis: The establishment of standard programming and management techniques throughout the development and production of software and their enforcement.

Berghuis: Systems Management.

Cress and J.W. Graham: Production of software in the university environment.

Dadda: Designing a program library for electronic engineering computer aided design.

Dahl, Myhrhaug and Nygaard: Some uses of the external class concept in SIMULA 67.

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

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

David and Fraser: Position Paper.

Dijkstra: On useful structuring.

158

Dijkstra: Complexity controlled by hierarchical ordering of function and variability [pages 181–185].

Donner: Modification of existing software.

Enlart: Program distribution and maintenance.

Enlart: Service level evaluation in software.

Enlart: Services provided to EPL users.

Ercoli: On the economics of software production.

Fraser: I. Classification of software production methods.

Fraser: II. The nature of progress in software production.

Gill: Thoughts on the sequence of writing software [page 186-188].

Gill: Position paper.

Gillette: Aids in the production of maintainable software.

Gillette: Comments on service group statements.

Harr: The design and production of real time software for electronic switching system.

Hastings: Software usage and modifications for the industrial user.

Hume: Design as controlled by external function.

Kjeldaas: The use of large computers to reduce the cost of software production for small computers.

Köhler: Maintenance and distribution of programs.

Kolence: On the interactions between software design techniques and software management problems.

Letellier: The adequate testing and design of software packages.

Llewelyn and Wickens: Software testing [pages 189–199].

McClure: Projection versus performance in software production.

Nash: Some problems of management in production of large-scale software systems.

Naur: The profiles of software designers and producers.

Opler: Acceptance testing of large programming systems — abstract.

Opler: Measurement and analysis of software in production.

Pinkerton: Performance monitoring and systems evaluation [pages 200–203].

van der Poel: A simple macro compiler for educational purposes in LISP.

Randell: Towards a methodology of computer systems design [pages 204–208].

Reenskaug: Adapting to object machines.

Ross: Example of discovering the essence of a problem.

Ross: Example of mechanical transformation from beads to arrays.

159

Ross: Problem modeling.

Selig: Documentation standards [pages 209–211].

Selig: On external and internal design.

Wodon: Influence of data structures on system design.

160

CLASSIFICATION OF SUBJECT MATTER

by

R.W. Bemer, A.G. Fraser, A.E. Glennie, A. Opler, and H.R. Wiehle

Introduction

At this early stage in the discussion on software production methods it is desirable to find an appropriate way of classifying the subject matter. A working party was given the task of finding an appropriate classification method and its conclusions are given here.

As a by-product of its deliberations, the working party produced a list of production cost elements and this is appended.

Classification Method

It is convenient to identify two major divisions of the management function. These are identified here as **Production Management** and **Technical Direction**. The former is subdivided into the procedures that constitute the production process whereas the latter is conveniently identified by the technical components involved in the production task.

In the lists which follow, aspects of the subject which working group P consider to be especially important are marked with an asterisk.

Production Management

1. PREPARING A PRODUCTION CAPABILITY — HUMAN

    1.1 Training — technology and methods

    1.2 Indoctrination in conventions for coding, testing, documentation

161

    1.3 Organization structure

        Quality, types and background of programmers used

        Project or product oriented

        Group size

        * Internal communications

    1.4 Productivity

        Evaluation

        Methods of output increase

    1.5 Support Services — Documentors, clerical, operators, etc.

2. PREPARING A PRODUCTION CAPABILITY — NONHUMAN

    2.1 Support equipment and availability (real or paper)

        Local/remote access

    2.2 Physical facilities — offices, file storage, data preparation

    * 2.3 Machine-controlled production environment

        Software tools — job specific (simulators, etc.)

        Software tools — general (high level languages, flow charters, text editors, indexers, etc.)

Software tools — management (accounting, control, progress)

3. PRODUCTION PLANNING

3.1 Requirements and specifications

3.2 Choice of standards and conventions for the project

3.3 Estimating

Magnitude of code and documentation

Costs (standards of comparison)

3.4 Budget

3.5 Workforce allocation, phase-in

3.6 Structural breakdown of system, interfaces

3.7 Configuration management (relative to hardware configurations)

3.8 Identification of software units, and their maintenance categories

3.9 Work schedules

3.10 Negotiation of changes, inquiries

3.11 Implementation plan for user documentation

* 3.12 Reliability, maintainability plans

162

3.13 Test plan

3.14 Special problems of complex, large systems (over 100 man-years, new varieties such as interactive, control and deadline)

4. ADMINISTRATION AND METERING OF THE PRODUCTION PROCESS (SOFTWARE AND TEST SOFTWARE)

4.1 Cost accounting

4.2 Size control (elimination of duplication through multiple usage)

* 4.3 Progress reporting and supervision

* 4.4 Measuring production performance

4. 5 Schedule adjustment

4.6 Change control

4.7 Control of innovation and reinvention

4.8 Standards control

4.9 Technical reviews

* 4.10 Instrumentation and analysis of software

4.11 Feedback to design

* 4.12 Internal communication and documentation

4.13 User documentation

4.14 Quality control and component tests

4.15 System assembly

4.16 Final project report (Innovations, pitfalls, recording major decisions and justification, etc.)

5. FINAL PROCESSING

5.1 System generation

* 5.2 Quality assurance and measurement of performance

* 5.3 Quality assurance of documentation

5.4 Release approval

5.5 Delivery methods — shipping and installation

5.6 Customizing and subletting

5.7 Maintenance and installation tools

5.8 Transfer to service for maintenance, enhancement and replacement versions

163

Technical Direction

1. TERMINOLOGY AND TECHNOLOGY OF SOURCE TASK

2. PRODUCTION MACHINE AND ITS SOFTWARE

2.1 Hardware configuration to be used in production

* 2.2 Operating system (on-line or batch)

2.3 Software for information handling

* 2.4 Programming languages

2.5 Aids to documentation

2.6 Debugging aids

2.7 Environment construction for testing

2.8 Simulator for parts of object machine (or bootstrap)

* 2.9 Tools for technical control of production

2.10 Aids to assembly and coordination of testing

3. OBJECT MACHINE AND ITS SOFTWARE

3.1 Hardware configuration to be used

* 3.2 Method of adapting to object machine

3.3 Static structure of object program (binding, linking, etc.)

3.4 Dynamic structure of object program (re-entrance, relocation, etc.)

3.5 Run-time interface with machine and operating system

3.6 Run-time interface with other software

3.7 Facilities available at assembly or compile time

3.8 Tools for performance measurement

4. THE PRODUCT

4.1 Object program

4.2 Matching documentation (various audiences)

4.3 Set of test cases and results

* 4.4 Procedure for generating, maintaining and modifying the system

4.5 Specification of installation procedure

5. STANDARDS AND CONVENTIONS

5.1 Applicable industry and international standards

5.2 Standards and conventions to be used in house

164

ADDENDUM — SOFTWARE COST ELEMENTS

NORMAL

Training
Personnel turnover
Design
Functional description
Review — Conceptual
Coding
Testing — Quality control
Review — Implementation
Machine time — Test
Technical documentation
User documentation
Supplies — Paper, cards, tapes
Input Data Preparation
Rework and false starts
Failure — Hardware
— Basic Software
— Interaction
Testing — Quality assurance
Terminal and line costs

SEMINORMAL

Duplication
Waiting for Test time
Waiting for return
Programmer fatigue (irregular schedules, lack of direction)
Communications misunderstanding
Deterioration
Design inflexibility for change

165

CHECKLIST FOR PLANNING SOFTWARE SYSTEM PRODUCTION

prepared in August 1966 by

R.W. Bemer

1. TOOLING

Is the first prototype or pilot of each hardware unit assigned (in sufficient quantity) to software production for a sufficient and continuing period of time?

Are these hardware units periodically updated and rotated to customers so that software production has up-to-date and complete equipment?

Is this hardware under the firm control of software production so as to be free of interference due to hardware test modification, confiscation for customers, etc.?

Is this hardware operated in a customer-like environment to reflect and anticipate customer needs before distribution to the field?

Is there sales representative service and field engineering service for the software systems equivalent to that for any other customer?

Is sufficient research undertaken in software production methods, such as construction languages and bootstrap methods?

Are the software production methods of other manufacturers studied if known, and sought out if unknown?

Is the manufacturer's most powerful computer used for controlled production, distribution and maintenance of software systems for all of his machine systems?

Is there a computerized system for software field reporting?

Is there a computerized system for-software production control?

Is there a computerized system for automated software production?

Is there a computerized system for a customer roster, together with their hardware and software configurations?

166

Is this roster integral with the corresponding roster for manufacturing, projects and field engineering?

Does this roster contain the hardware field change level which could affect software or system performance?

Is there a software facility for file maintenance of source programs?

Is a general assembler provided which can assemble, using the tool machine, for any new machine?

Is there a tool for proving the identity of the general assembler to the specific assembler for the specific machine?

Are simulators provided for the tool machine which can simulate any new machine for which software is being constructed?

Are these simulators in a form suitable for any period of scarcity, such as early testing by customers?

Does the software production system have provision for

a) Updating the test library with additional quality assurance programs and suitable excerpts from the field reports.

b) Updating the roster.

c) Modifying itself.

d) Producing provisional systems for temporary programming usage and testing.

e) Producing modifiers to update customer's system and documentation for distribution.

f) Producing original manuals and updatings.

g) Producing the field report summary and statistics.

h) Producing records of all these processes for the manufacturer.

i) Acting as well or better than a human supervisor in the control of production (i.e., not accept software systems modifications submitted by the programmer unless they meet certain standards)?

2. TRAINING AND ORGANIZATION

Are programmers given formal training in production methods and techniques?

Do the programmers provide training in system operation to field marketing and field engineering personnel so that they can pass such training on to customers?

Is there an excess of programming personnel to allow flexibility of redistribution, emergency and unbudgeted projects, meanwhile utilizing them in being trained, assisting in the field, and experimentation and research?

167

Are programmers periodically rotated to machine operation or to the field to obtain current hands-on experience?

Is a skills inventory maintained for the programming staff?

Is there a method to upgrade operators and other personnel to programming?

Are personnel operating the software production computers given the opportunity to review operating system design for proper man-machine characteristics?

Are basic manuals on software production available?

Is there an adequate library?

Is good usage made of the library by programmers?

Does an adequate percentage of programming personnel maintain membership in professional societies?

Is publication encouraged?

Is the software staff organized to participate in advanced projects (as opposed to the 85 percent of repetitive software production), particularly when hardware and software must be developed concurrently?

Does software production undergo continuous mechanization to minimize perturbations from management change?

Does the software organization provide comprehension courses for their management to ensure that its relative importance is properly reflected in their planning?

3. DESIGN

Is the software planned for categorization by the relative amount of effort expended to produce, maintain and distribute it?

Are the criteria for this categorization published and distributed to field personnel and customers?

Are gross functional specifications for software provided by a joint effort of Product Planning and Field Marketing, with considerations and options for hardware tradeoffs?

Do these gross functional specifications include data on allowable configurations for both hardware and software, with suggested categories for quality and service?

168

Is there a formal procedure for planning basic software simultaneously with hardware, with an effective systems control for allocating tradeoffs?

Is there a formal review procedure for software specifications for

a) Marketing to ensure adequacy and efficacy of market coverage

b) Product Planning to ensure conformity to planning goals

c) Hardware Engineering to ensure accuracy and consistency

d) Internal computer operations to ensure user suitability and proper human engineering?

Is there a formal procedure to amend specifications as a result of such feedback, with provision for alternate methods when review schedules are not met?

Is there a mechanism for interchange (or at least crediting) of budgeted monies to facilitate maximum benefit from hardware–software tradeoffs?

Is the tradeoff principle extended to field maintainability, with a mechanism for interchange of budgeted monies?

Does Product Planning provide intelligence from competition and advanced development areas in the form of surveys and comparative evaluation materials, for due influence upon software design?

Is a substantial portion of the design staff familiar with current field operating methods, so as not to be insulated from changing requirements?

Is each software unit identified and numbered for direct correspondence to

a) Its corresponding quality assurance program?

b) Its actual source program instructions, listed?

c) Its actual source program instructions in entry form.

d) Its corresponding elements of user documentation.

e) Its corresponding elements of technical documentation.

f) The minimum hardware configuration required to run it.

Is each software unit equipped with a document detailing the following:

a) Its purpose.

b) Its inputs and their forms.

c) Its outputs and their forms.

d) The processes applied to the inputs to yield the outputs.

e) The inventory of tools (other units, utility routines, usable store, executive controls) available to it.

f) The constraints of time and interaction with other software units.

g) Its operational design goals and characteristics.

h) The characteristics of its interfaces with other software units.

i) Its precision and accuracy, where relevant.

169

Are these questions complete before any flowcharting, programming or coding commences?

Are they matched against the other similar specifications to detect conflict, duplication or system imbalance?

Is there a periodically updated software design checklist available to all software designers?

Does it specifically include material on the following:

a) International, national, industry and company standards.

b) Design for offline vs. online operation.

c) Design for closed vs. open shop usage.

d) Flexibility for various customer job mixes.

e) Alternate software with different performance but same functional characteristics.

f) Diagnostics under the executive system.

g) Modelling for design decisions (and simulation).

h) Initialization, bootstrap and restart procedures.

i) Optimizing human operating characteristics.

j) Special diagnostic aids for users.

k) Making the executive system forgiving and helpful in case of hardware faults.

l) Providing for special, nonstandard hardware.

m) Accounting and measuring procedures.

n) Modular construction.

o) Alternative modes of system operation, with a provision to indicate those selected.

p) User priorities, special accounting and custom software units supplied by the user.

q) Machine room operational procedures.

4. SCHEDULING AND COSTING

Do plant accounting procedures provide all necessary information on costs by project, rather than by cost center?

If not, does software production incorporate this in its own production control system?

Is the gathered data processed, not merely accumulated?

Is there a software production control system to fabricate to predicted schedules for predicted costs?

Are costs of previous system construction utilized in predicting new costs?

170

Are all sales commitments for software, and software delivery clauses in contracts, concurred in by software production for incorporation in the production control system?

Are there responsible software project managers who utilize the mechanized production control system to ensure that all elements are available on scheduled delivery dates?

Are there production documents for each phase of production of each software unit, representing a scheduling commitment upon the part of the unit manager?

Do these then yield gross schedules of expected completion which are published internally and incorporated in the production control system?

Are these expected schedules then modified by maximum slippage factors to produce schedules which can be incorporated into contracts without specific approval each time, particularly for non-delivery and non-performance penalty provisions?

Is it guaranteed that all contracts with schedules prior to these dates are subject to approval by software management?

Are all Software commitments for items not in the price book, or variations of these, quoted at an additional cost to the customer, subject to the existence of adequate personnel and facilities for production?

If any software is furnished free as a sales concession, is there a signed acknowledgement attached to the final copy of the contract which states the distribution of cost against individual sales commission, regional sales or manufacturer's market penetration?

5. PRODUCTION CONTROL

Is the quantity of software to be produced of a manageable size by virtue of control of the number and character of hardware and software configurations which are permitted to be sold?

Given variation possible in store size, number and variety of peripherals functional operations, etc., are all combinatorially possible hardware configurations generated by computer and submitted to

a) Marketing — for reduction of variety with respect to saleability?

b) Engineering — to ensure completeness of interface and interconnections?

c) Software — to ensure software operability for each combination?

d) Product Planning — for performance control (i.e., a salesman might claim that a low price system is saleable but it may not be of sufficient performance with only one tape unit as secondary store)?

171

Is this set systematically reduced to a reasonable size (i.e., no more combinations than number of machines planned to sell)?

Is this set then further reduced by matching with a software configurator?

Does this software configurator distinguish between software which is vital to operate the hardware and that which is optional for better system performance or added facility to the user?

Does general management then sign off that the deleted combinations are not to be sold (except with special permission, where software will be sold rather than furnished free)

Does this information now form the basis for the price book?

Are the remaining (official) hardware/software configurations numbered so that a single number in the customer roster will indicate the exact hardware and software with which he is supplied?

Is software designed to minimize the automatic adjustment to variance in configuration?

Does software production have the right of concurrence for software announcements and publicity, with respect to accuracy and feasibility?

Given the possible combinations of hardware units the manufacturer is willing to supply, and the software units to be furnished free for each of these, is the customer told clearly that hardware or software outside of this group is supplied only at special cost?

Is such a policy enforced by a contract review board?

Does this review board also approve contractual commitments and penalties for software?

Is a request for any additional or special software accompanied by a dollar value which is either a) the amount the customer is willing to pay additionally or b) the amount by which salability is enhanced and which Marketing is willing to fund?

Are these categories open to query by salesman to determine the best performing hardware/software configuration to do the customer's job at minimum cost?

Does Marketing provide an assessment of marketing strategy which may influence the amount or nature of software provided, or rescheduling of production and delivery dates?

Is there a computerized system for manipulating the data on each software unit, such as

172

a) total size

b) maximum residency

c) set membership

d) the other units which may call it

e) the other units which it may call

f) the interfaces with these surrounding units, together with the entry and exit points

in order to determine

a) that the hardware configuration is not exceeded

b) that enough hardware exists to run the software

c) a mechanical diagram of the entire system for completeness and consistency for quality control, diagnostic and maintenance purposes?

Does the production control system recognize and allow for the relative invisibility of software?

Given a specific software unit of a minimum size feasible for individual control, do the responsible supervisors estimate total elapsed time and costs for man and machine hours as a function of their resources? Is internal competitive bidding allowable for cost reduction?

Is the data for each stage (functional specifications, flowcharts, implicit quality test, coding, checkout in vacuo, checkout in processor, checkout in system, documentation, explicit quality test, release) of development of that software unit given by the supervisor on a signed document?

Are labor distribution reports developed from the timecards of programmers participating on each unit?

Does the individual programmer periodically give an estimate of the percentage of completion of each unit, for correlation to labor distribution and schedules, and perhaps PERT charts?

Are full records kept on original estimates, revised estimates and actual completion dates for

a) Recalibration of supervision

b) Improvement of future estimates

c) Deriving production standards

d) Possible rebalance of staff or redesign

e) Notification to marketing in case of slippage?

Are these viewed as official company records, so that detected falsifiers are subject to discharge?

Does effort expended on unapproved or bootleg projects put the controlling supervisor subject to discharge?

173

Is the production control keyed to a roster containing the list of official software units supplied without charge, keyed to the documentation units, and for each customer the

1) User's name, address and representative

2) Branch office name, address and representative

3) Contact pattern between user, branch and programming

4) Machine type, serial, installation date, on-rent date

5) Hardware configuration, operational dates of units

6) Channel assignments, other determinations of logical options

7) Field change orders affecting software and whether installed or not

8) Software options for:

    a. Required units

    b. Characteristics of their storage

    c. Characteristics of their usage

    d. Maximum store allotted for processing and usage

    e. Hardware restrictions affecting software operation, such as reserved elements or lock-outs

    f. Delivery form of software unit (symbolic, relocatable, absolute, FORTRAN, etc.)

    g. Special software supplementing or replacing standard units, by whom supplied, data descriptions and linkages

9) Number of last system delivered. Updating pattern and requested frequency (6 month maximum interval for archivage limitation):

    a. Every system

    b. Every nth system

    c. Upon specific request

    d. First new system after elapsed time interval

    e. Only on change to specified software units

    f. Combinations of these

10) Requirements for backup system on another machine

11) Special commitments by sales or programming personnel

12) List of customer's field reports by number?

6. FABRICATION

Is software fabricated for utility in the international market?

Is there an active standards unit in the software production group policing compliance with national end international standards available?

Are there internal local standards for production consistency?

Does programmer terminology conform to the standard IFIP/ICC vocabulary?

174

Are sufficient personnel provided to participate in advanced standards making work?

Is all system planning done with standard flowchart templates, where applicable?

Are logic equations an allowable alternative to flowcharting?

Are good records kept in any stage of development?

Are systems kept on tape or disc periodically copied on cards or tape for recovery?

Are there finite points in time during development and system updating when the system is reassembled to a clean form with updated listings?

Are hand coding sheets destroyed and each reassembly listing used as the legal representation?

Is periodic recoding recommended when a routine has lost a clean structural form?

Is the system oriented to reassemble with at least the ease of patching?

Are all system tools utilized as provided?

Is there periodic exchange of fabrication information and progress among the various programmer groups constructing an integrated system?

Are original programs and changes controlled so that they cannot be introduced without meeting certain minimum criteria (e.g., a minimum length of comments associating with the statement, or preventing entry or modification of a software unit unless the appropriate links and interfaces in both directions are given)?

Is adequate attention given to modularity in construction, but without overemphasis which could destroy operating efficiency?

Since it is cheaper to be prepared for a malfunction than not, are program units created in three forms for testing:

a) As a self-contained unit, complete with synthetic input and output, created perhaps by a generator

b) In a form suitable for usage within its own major program

c) In a form suitable for use within the overall system?

Can the extra instructions required for (a) and (b) above be removed mechanically for the final stage?

Are statistics kept on the type of malfunction incurred, which may be:

175

a) a hardware malfunction

b) a malfunction in the programming system used

c) an operating mistake

d) data errors, such as unexpected type, outside of expected range, physical errors in preparation or reading, etc.

e) programmer's mistake, such as a misunderstanding or disregard for the rules of syntax, grammar, construction, file layout, system configuration, flow process for solution, etc.?

Is considerable desk checking performed to

a) Check conformity to rules, such as those for justification

b) See that enough restart points exist for long programs

c) Compare actual program logic for match with intended logic as given by a flowchart or equation

d) Examine live input for peculiar characteristics which could cause erroneous branching, such as bad data, blank records, etc.

e) Inspect the list of identifiers produced and assigned by the processor, looking for conflicts, insufficient definition, completeness and spelling

f) Check permissible spellings of reserved words, allowable usage of spacing, hyphens and commas, and juxtaposition of illegal word or operation pairs?

Do the programmers plan for maximum machine utility per run in checkout by:

a) Submitting multiple related runs

b) Getting multiple service per run by modifying his program to read in correct conditions periodically to nullify any mistakes so the next section of program can also be checked

c) Avoiding extensive store dumps

d) Programming flow indicators which print to show the paths of decision in actual execution

e) Providing a full range of controlled data input

f) Designing testing on a ‘design of experiment basis’ to achieve maximum progress for each run?

7. USER DOCUMENTATION

Is the user documentation given top priority and consideration for the sale and successful operation of the system?

Does the documentation conform to standard models?

Is the documentation uniform across product lines so that the user may expect to find similar information in corresponding places and form for every system?

176

Is user documentation consistent with itself and with national and international standards?

Do all flowcharting symbols and methods conform to international standards?

Does all terminology conform to the IFIP/ICC vocabulary?

Do software processors (Fortran, Cobol, Algol, etc.) conform to standard language specifications?

Are single manuals produced which are valid across machine lines (Fortran, File Structure, Labeling, etc.)?

Are variable software system characteristics excluded, but enclosed separately in machine line manuals with cross indexing?

Is there a formal, consistent document numbering system?

In consideration of the various audiences addressed (for the customers — purchasers, utilizers, programmers, operators; for the manufacturer salesmen, customer technical assistants, basic software programmers, field engineers and maintenance programmers), are user documents looseleaf rather than bound?

Can pages have multiple set membership (i.e. present in several different manuals)?

Does the education staff provide other documents which are in effect ‘road maps’ through these manuals to accelerate the learning process:

Is there a computerized system for writing, editing and boiler-plating this user documentation?

Are programmers forced to prepare documentation very early in the production cycle?

Are tentative manuals, with missing decisions identified, published in preference to having no manuals at all?

Is there a formal review procedure for manuals for

a) Marketing — to ensure adequacy and efficacy of market coverage

b) Product Planning — to ensure conformity to planning goals

c) Hardware Engineering — to ensure adequacy and consistency?

Is there a formal procedure to amend manuals as a result of such feedback, with provision for alternate methods when review schedules are not met?

Does hardware engineering provide complete operational specifications early in the software production cycle?

177

Are hardware manuals forbidden to exist separately for users, so that the system is described in terms of the software system?

When software is produced by a non-English speaking group, is a full set of English documentation provided?

Is this simultaneous with the non-English documentation, or at least within a 2 month lag?

Does the software allow only for writing programs in English, although the natural language of the user may be used for variables?

Is the software itself always written entirely in English?

Is the volume of manuals produced carefully controlled and charged to the field organization, so that these expensive documents will not be used for sales handouts?

Is the user documentation publishing a function of software production, including an art group?

Is there a computerized distribution system to control inventory and reorder points?

Is the decision for internal or external printing controlled by bids, or by delivery schedules in case of equivalent bids?

Is there a computerized system to ensure that user documentation is always correct and matches the current software system in the field?

Is responsibility for documentation allocated to individuals by software unit or groups of units, with a required sign-off on a checklist before distribution of changed software systems?

Is there a standard interior printing area to accommodate the difference between U.S. and ISO paper size?

8. TECHNICAL DOCUMENTATION

Is the software system fully supported by correct flowcharts, descriptions of design algorithms, listings, performance specifications and structure descriptions for AE's and Field Engineers?

Do listings carry a maximum of technical information in annotation?

Is such documentation susceptible to limitations on how much may be furnished users in proprietary cases?

Are there mechanical methods of producing technical documentation, such as flowcharters and structure decoders?

178

Is there a control mechanism to ensure that technical documentation is revised to correspond with actual program operation?

9. QUALITY ASSURANCE

Is the quality assurance function recognized to be different from implicit and continuous quality control during fabrication, in that it is discrete, explicit following production, and ignores the sequence or nature of the fabrication steps or processes?

Is software quality assurance done by an independently reporting agency representing the interests of the eventual user?

Is the product tested to ensure that it is the most useful for the customer in addition to matching functional specifications?

Do software quality assurance test programs undergo the same production cycle and method (except Q/A) as the software they test?

Are they defined and constructed concurrently with the software?

Is at least one person engaged in software quality assurance for every ten engaged in its fabrication?

Whenever tests are specified as a part of national or international standards, are these utilized?

Are there tests for overall system performance as well as for components (i.e., road-testing, French: 'rodage')?

Are software quality assurance tests a part of the general hardware acceptance test on the customer's machine before it leaves the factory'?

Can software field release be held up if these tests are not passed?

Do the tests include a system logic exerciser?

Are tests provided to ensure matching of computational results with those of other equipment?

Is there a growing test library for each software system, including

a) a test for roster consistency and permissibility of hardware and software configurations requested

b) a program acceptance filter

c) specifically designed quality assurance tests for components

d) accumulated field reports?

Is this test library applied upon issuance of each modification of the software system?

179

Is each customer's system tape tested on the software production machine for a sufficient period of time, where feasible?

Are Q/A personnel employed part time on maintenance of older systems, for efficiency and competence in judging?

10. FIELD INSTALLATION

After delivery and putting hardware in service, is the software similarly delivered in person and verified to be operable on the customer's machine at his site?

Are a selected subset of the customer's programs, which previously ran upon his machine at the factory test line, then run in order to have him sign an acceptance form for rental payment or purchase price?

Does the field installation programmer remain at the site until programs execute correctly?

Is there a follow-up plan to ensure that systems do not stay off rental?

Is this service performed by Field Engineering personnel?

11. DISTRIBUTION AND UPDATING

Is there a centralized library and distribution operation?

Is it responsible for maintaining records on users and their equipment insofar as it is necessary to distribute

a) software systems (cards, tapes, etc.)

b) manuals and other documentation

c) supporting material such as coding forms, code cards, CAD interchange forms, housing devices for supplies, flowchart templates, listing binders and training aids, both filmed and programmed

d) lists of various software materials available to sales and support personnel, with order prices

e) lists and abstracts of basic software and interchange programs available for distribution, grouped by category of software maintenance, by machine, by industry and by application — with schedules of availability

f) updated and corrected materials as produced

g) actual user programs for interchange?

Does this group maintain the standard software library, including systems from other software components?

180

Are updatings distributed in loose-leaf form, and remaining stock updated simultaneously?

Are all updates adequately supported by cover letters containing instructions and notification of obsolescence, with periodic summaries of available material on each system?

Does the library group utilize a computer for this service, for mailing lists, customer interest and software profiles, controlling and monitoring distribution?

Is a customer able to incorporate his own software units and special requirements without interference or malfunction as each system update is made?

Is the customer's software system modified to become the new system rather than replace his entire system?

12. FIELD REPORTING AND MAINTENANCE

Is there an official, supported system for field reporting on

a) Software malfunction

b) Software mismatch to documentation

c) Inferior software performance

d) Requests for change to software design

e) Nonconformity to standards?

Is this reporting procedure enforced by written instructions to

a) Software Production

b) Sales and A.E. personnel

c) our customers?

Is turnaround service provided in the shortest time by means of high-speed communications systems?

Are field reports processed by formal procedures so the A.E. or customer can know the status at any time?

Are current lists of outstanding field reports furnished to A.E's so they may protect the customer intelligently against these dangers?

Do all field reports carry identification of software units used and actual level of field changes installed in hardware?

Is the customer supplied with manufacturer's recommended good practices in diagnostic methods, operations and use of processors?

Is the user clearly instructed that it is his responsibility to develop the simplest and smallest program which demonstrates the malfunction?

181

COMPLEXITY CONTROLLED BY HIERARCHICAL ORDERING OF FUNCTION AND VARIABILITY

by

Edsger W. Dijkstra

Reviewing recent experiences gained during the design and construction of a multiprogramming system I find myself torn between two apparently conflicting conclusions. Confining myself to the difficulties more or less mastered I feel that such a job is (or at least should be) rather easy; turning my attention to the remaining problems such a job strikes me as cruelly difficult. The difficulties that have been overcome reasonably well are related to the reliability and the producibility of the system, the unsolved problems are related to the sequencing of the decisions in the design process itself.

I shall mainly describe where we feel that we have been successful. This choice has not been motivated by reasons of advertisement for one's own achievements; it is more that a good knowledge of what — and what little! — we can do successfully, seems a safe starting point for further efforts, safer at least than starting with a long list of requirements without a careful analysis whether these requirements are compatible with each other.

Basic software such as an operating system is regarded as an integral part of the machine, in other words: it is its function to transform a (for its user or for its manager) less attractive machine (or class of machines) into a more attractive one. If this transformation is a trivial one, the problem is solved; if not, I see only one way out of it, viz. 'Divide and Rule', i.e. effectuate the transformation of the given machine into the desired one in a modest number of steps, each of them (hopefully!) trivial. As far as the applicability of this dissection technique is concerned the construction of an operating system is not very much different from any other large programming effort.

182

The situation shows resemblance to the organization of a subroutine library in which each subroutine can be considered as being of a certain 'height', given according to the following rule: a subroutine that does not call any other subroutine is of height 0, a subroutine calling one or more other subroutines is of height one higher than that of the highest height among the ones called by it. Such a rule divides a library into a hierarchical set of layers. The similarity is given by the consideration that loading the subroutines of layer 0 can be regarded as a transformation of the given machine into one that is more attractive for the formulation of the subroutines of layer 1.

Similarly the software of our multiprogramming system can be regarded as structured in layers. We conceive an ordered sequence of machines: A[0], A[1], ... A[n], where A[0] is the given hardware machine and where the software of layer i transforms machine A[i] into A[i+1]. The software of layer i is defined in terms of machine A[i], it is to be executed by machine A[i], the software of layer i uses machine A[i] to make machine A[i+1].

Compared with the library organization there are some marked differences. In the system the 'Units of dissection' are no longer restricted to subroutines, but this is a minor difference compared with the next one. Adding a subroutine of height 0 to the library is often regarded as an extension of the primitive repertoire which from then onwards is at the programmer's disposal. The fact that, when the subroutine is used, storage space and processor time have been traded for the new primitive can often be ignored, viz. as long as the store is large enough and the machine is fast enough. Consequently the new library subroutine is regarded as a pure extension. One of the main functions of an operating system, however, happens to be resource allocation, i.e. the software of layer i will use some of the resources of machine A[i] to provide resources for machine A[i+1]: in machine A[i+1] and higher these used resources of machine A[i] must be regarded as no longer there! The explicit introduction (and functional description!) of the intermediate machines A[1] through A[n-1] has been more than more word-play: it has safeguarded us against much confusion as is usually generated by a set of tacit assumptions. Phrasing the structure of our total task as the design of an ordered sequence of machines provided us with a useful framework in marking the successive stages of design and production of the system.

183

But a framework is not very useful unless one has at least a guiding principle as how to fill it. Given a hardware machine A[0] and the broad characteristics of the final machine A[n] (the value of ‘n’ as yet being undecided) the decisions we had to take fell into two different classes:

1) we had to dissect the total task of the system into a number of subtasks

2) we had to decide how the software taking care of those various subtasks should be layered.

It is only then that the intermediate machines (and the ordinal number ‘n’ of the final machine) are defined.

Roughly speaking the decisions of the first class (the dissection) have been taken on account of an analysis of the total task of transforming A[0] into A[n], while the decisions of the second class (the ordering) have been much more hardware bound.

The total task of creating machine A[n] has been regarded as the implementation of an abstraction from the physical reality as provided by machine A[0] and in the dissection process this total abstraction has been split up in a number of independent abstractions. Specific properties of A[0], the reality from which we wanted to implement, were:

1) the presence of a single central processor (we wanted to provide for multiprogramming)
2) the presence of a two level store, i.e. core and drum (we wanted to offer each user some sort of homogenous store)
3) the actual number, speed and identity (not the type) of the physically available pieces of I/O equipment (readers, punches, plotters, printers, etc.)

The subsequent ordering in layers has been guided by convenience and was therefore, as said, more hardware bound. It was recognized that the provision of virtual processors for each user program could conveniently be used to provide also one virtual processor for each of the sequential processes to be performed in relatively clone synchronism with each of the (mutually asynchronous) pieces of I/O equipment. The software describing these processes was thereby placed in layers above the one in which the abstraction from our single processor had to be implemented.

The abstraction from the given two level store implied automatic 184 transports between these two levels. A careful analysis of, on the one hand, the way in which the drum channel signalled completion of a transfer and, on the other hand, the resulting actions to be taken on account of such a completion signal, revealed the need for a separate sequential process — and therefore the existence of a virtual processor — to be performed in synchronism with the drum channel activity. It was only then that we had arguments to place the software abstracting from the single processor below the software abstracting from the two level store. In actual fact they came in layer 0 and 1 respectively. To place the software abstracting from the two level store in layer 1 was decided when it was discovered that the remaining software could make good use of the quasi homogeneous store, etc. It was in this stage of the design that the intermediate machines A[1], A[2], ... got defined (in this order).

At face value our approach has much to recommend itself. For instance, a fair amount of modularity is catered for as far as changes in the configuration are concerned. The software of layer 0 takes care of processor allocation; if our configuration would be extended with a second central processor in the same core memory then only the software of layer 0 would need adaption. If our backing store were extended with a second drum only the software of layer 1, taking care of storage allocation, would need adaptation, etc.

But this modularity (although I am willing to stress it for selling purposes) is only a consequence of the dissection and is rather independent of the chosen hierarchical ordering in the various layers, and whether I can sell this, remains to be seen. The ordering has been motivated by ‘convenience’....

The point is that what is put in layer 0 penetrates the whole of the design on top of it and the decision what to put there has far reaching consequences. Prior to the design of this multiprogramming system I had designed, together with C.S. Scholten, a set of sequencing primitives for the mutual synchronization of a number of independent processors and I knew in the mean time a systematic way to use these primitives for the regulation of the harmonious co-operation between a number of sequential machines (virtual or not). These primitives have been implemented at layer 0 and are an essential asset of the intermediate machine A[1]. I have still the feeling that the decision to put processor allocation in 185 layer 0 has been a lucky one: among other things it has reduced the number of states to be considered when an interrupt arrives to such a small number that we could try them all and that I am convinced that in this respect the system is without lurking bugs. Fine, but how am I to judge the influence of my bias due to the fact that I happened to know by experience that machine A[1], with these primitives included) was a logically sound foundation?

186

THOUGHTS ON THE SEQUENCE OF WRITING SOFTWARE

by

Stanley Gill

In other papers presented at this conference, Dijkstra stresses the layer structure of software, and Randell refers to the alternative possibilities of constructing the layers from the bottom up (i.e. starting with the primitives) or from the top down (i.e. starting with the target system). Clearly the top-down approach is appropriate when the target system is already closely defined but the hardware or low-level language is initially in doubt. Conversely the bottom-up approach is appropriate when the hardware is given but the target system is only defined in a general way.

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. Thus in bottom-up programming, peculiarities of the hardware tend to be reflected in the higher levels (as for example in some high-level languages) when they would be better out of the way, while top-down programming may leave the programmer at the lower levels struggling to implement some feature inherent in the target system, which should have been dealt with higher up. The success of either approach depends upon the designer's ability to anticipate such problems, and to generate and to recognize solutions that avoid them.

In practice neither approach is ever adopted completely; design proceeds from both top and bottom, to meet somewhere in between, though the height of the meeting point varies with circumstances.

Each program module constitutes a definition, in terms of the primitives available in that layer, of a facility that constitutes a primitive (or primitives) of the layer above. It is thus a 'downward-facing' definition. Associated with it is an 'upward-facing' definition, which is referred to when the facility provided by the module is used in a higher 187 layer. Downward-facing definitions are formal, and they are of course the means by which the system is implemented physically. Upward-facing definitions are used by the designer, and are usually informal.

In bottom-up programming the upward-facing definitions, though informal, are complete (or intended to be). Thus for example when a routine has been written in a defined language, the function performed by the routine should be unambiguously defined. In top-down programming this need not be so. In choosing the primitives one layer further down, out of which to construct the facilities required in a given layer, one has only to define them sufficiently closely to determine their role in this construction; the details may be left undefined. A classic example of this is the first step in breaking down a problem, as taught to all student programmers: drawing a flow chart. The blocks in the flow chart are by no means fully defined; their functions are only indicated sufficiently clearly to enable them to be put together correctly. As the lower layers are designed, they gradually complete the definitions of the top layers, and hence the outward functions of the whole system.

Thus in top-down programming one is often working with incompletely defined components. This demands more care and discipline; for example one must ensure that, in completing a definition, one does not introduce side effects that were not foreseen in the upper layers where the definition has been applied, and which could be harmful.

Top-down programming does however have an advantage in that it allows the designer to see which operations are called for frequently in the upper layers, and should therefore be given special treatment below. In bottom-up programming one must try to guess which primitives will prove most useful higher up. This is not easy; almost every programming language is littered with features that are hardly ever used.

In practice, a large software project is rarely a matter of implementing one single target system in terms of one given primitive system. The target is often modified in the course of the project, and the software may later be extended to meet several other targets also. It may be required to implement the whole system again on different hardware. The expectation of such later developments may influence the choice of programming method. Thus, for example, if later implementation on other hardware is likely to be needed, it is unwise to use the bottom-up approach 188 except from a layer at which one can provide useful common facilities in terms of any of the hardware alternatives. Similarly if (as is often the case) later variations of the target system are likely, it is less attractive to use top-down programming except from a layer at which one can define a useful set of primitives applicable to all the target systems.

Thus although, with single primitive and target systems, one would nearly always program up from the bottom and down from the top to meet in the middle, in practice the picture is often more complicated. Multiple targets could result in an inversion of this arrangement in the upper layers, and multiple hardware could have the same effect in the lower layers. Thus for example a common strategy in such situations is to define a programming language at some intermediate level, intended for a range of target problems or for several different hardware systems. Having defined the language, one then proceeds to program both upwards and downwards from it.

It is in constructing such an intermediate system, before it has been formally related to either the lower or higher levels, that the greatest demands are made on a programmer's intuition. Programming is practically never a deductive process, but the depth of induction required can vary greatly according to the situation. The least demanding situation is that in which there is a single primitive system available, and a single clearly defined target system to be implemented, with only a small conceptual gap (i.e. number of software layers) between them. In the early days of programming such situations were common; now they only occur as small components of bigger tasks.

Within the last fifteen years the range of targets spanned by single software projects has increased greatly, and on the whole successfully. The range of hardware has also increased, but to a lesser extent, with greater difficulty and less success. One of the most fruitful areas for improving software productivity lies in devising an effective universal primitive system (such as Iliffe's 'Basic Language').

189

THE TESTING OF COMPUTER SOFTWARE

by

A.I. Llewelyn and R.F. Wickens

Contents

1. Introduction

Computer software has changed rapidly from being merely an adjunct to the hardware to becoming an extremely important part of an installation and one that profoundly affects the overall performance. It is now widely accepted that computer hardware should be given some form of acceptance test by the customer and such tests have formed part of the United Kingdom government's computer contracts for about eight years. However, the weak area of most installations is now that of software.

Users complain that the software they received was badly designed and produced, was not delivered on time and when finally delivered, contained serious errors and restrictions and was also inefficient. Unfortunately, 190 these criticisms are often only too true although, happily, it is unusual for all the criticisms to apply simultaneously. The consequences of inadequate software can be disastrous, both in financial terms as a result of delays and wasted effort in getting a system operational and also in terms of the morale of an installation's staff.

It is my opinion that software should now be subjected to a testing discipline just like any other product. The user should be provided with a specification of the software he is to receive and be able to have confidence that the software he gets will perform in accordance with this specification. By specification is meant a document that describes the product's operation and performance in some detail. Certainly, this specification should be more detailed than the typical user manuals provided to-day. To obtain the user's confidence, the software will have to be rigorously tested for conformity with the specification. This testing has two aspects: that carried out by the producer of the software to satisfy himself that it is fit for release to his customers and that carried out on behalf of the customer in order to give him sufficient confidence in the software to justify his paying for it. This paper is primarily concerned with the second aspect of testing. However, both aspects are very closely related and, of course, if the in house testing of the supplier is adequately carried out over a long enough period of time to gain customer confidence, the second aspect becomes largely unnecessary. It is certainly true that what might be described as the acceptance testing of software must not become a substitute for proper in house quality assurance.

2. The aims and problems of acceptance testing software

The aim of any testing scheme is to ensure that the customer gets substantially the software that he ordered and it must provide the customer with convincing evidence that this is so. However, any attempt to test software comes up against several basic problems.

One of the main problems is that of specifying unambiguously the facilities and performance to be provided by an item of software. This must be done in such a way as to allow the manufacturer to introduce improvements as time goes by and yet be capable of being written into a contract. A problem closely associated with this is that of defining the acceptance criteria to be used. These must be capable of unambiguous application and leave the supplier in no doubt as to the performance required.

191

Software can only function through the action of the hardware; fundamentally, only the combined effects of the software and hardware can influence events outside the computer, software alone is useless. Any testing method can only test the validity of the whole software/hardware edifice. Further, any testing procedure must take account of the fact that software is often capable of operating in several different environments representing the many possible configurations and patterns of system usage.

A problem under failure conditions is to determine what has failed; is it the software, the hardware, or a subtle combination of both? As systems become more complex, this latter condition is more likely. Modern software is called upon to interact in real time with its environment. The external stimuli can occur almost at random. Any attempt to test exhaustively such a system is going to be very difficult. It is, perhaps, fair to say that it is going to be as intellectually challenging to test the complex software systems of the future as it will be to design them initially.

3. The requirements of a software testing procedure

A test procedure should fulfil the following requirements:

(a) Enable a user to check that he gets the product that he ordered. Any discrepancies should be detected and clearly defined so that meaningful discussions can take place between the user and the supplier.

(b) Be reasonably easy to apply and provide an effective measure of the acceptability of the software.

(c) Provide an incentive for the suppliers to improve their own Quality Assurance procedures for software.

(d) Check the ‘efficiency’ of the software.

(e) Must be applicable from the first deliveries of a system onwards. Before a machine is ordered, a customer should have carried out a detailed appraisal of the competing systems. The tests carried out prior to final acceptance are to confirm that the information on which the chosen system was selected and finally ordered were correct. Thus, the software testing procedure may be regarded as the culmination of this appraisal. However, the depth of any such appraisal will depend upon the circumstances under which a system was selected and the organization doing 192 the selection; thus, any test procedure should not rely upon any previous appraisal for its success.

4. Present software testing procedures

Most customers accept the software without any form of test; indeed, their contracts with the suppliers would often not allow them to do so even if they wished to. Those attempts that are made are usually limited to providing demonstrations of the more important features and the user writing special programmes that he believes will test areas important to him.

The United Kingdom government’s standard computer contract allows for demonstrations of items of software to be called for during the acceptance trials. However, because of the limited time available, it is realized that these demonstrations do not represent a thorough test of the software: merely a check on the availability and a very limited test of certain critical items. When these demonstrations are based on a prior knowledge of the state of the software they can be arranged to highlight its failings and thus allow some action to be taken under the contract but without such knowledge these demonstrations are of little value. The United Kingdom government’s Computer Advisory Service will, in general, have appraised the software of a computer before it is bought for use in the government service and will subsequently keep it under review. It is this fact that enables the present procedure to work reasonably well with the currently available software. However, it would be unrealistic to suppose that such a procedure will provide the user much protection against faults in the very complex systems that are likely to become available in the next few years.

The greatest benefits are obtained from a testing procedure in the early life of a new range of computers. Once a usable version of the required software is available, the basic needs of the user are satisfied. If subsequent issues are in error, it is inconvenient but not disastrous. However, this does not mean that testing standards can be relaxed for new issues of current packages, although it may make it less urgent. Seemingly, quite minor changes may have a profound effect upon the operation of a package. It is certainly necessary to check that a new issue of a package is compatible with its predecessor. It must not be expected that a 193 test procedure will guarantee error-free software but it should tell the user what is the state of the software he is being offered so that he may decide whether it is capable of doing his work effectively.

5. Possible test strategies

There are, fundamentally, two different methods of determining whether a product meets its specification. One can analyse the product in great detail and from this determine if it is in accordance with its specification, or one can measure its performance experimentally and see if the results are in accord with the specification; the number and sophistication of the experiments can be varied to provide the degree of confidence required of the results.

Our current software appraisals are biased towards the analytic approach but the limited time and effort available and the often limited access to detailed information prevent the appraisal doing away with the need for an experimental approach to software testing.

6. A proposed test procedure

The proposed test procedure attempts to remove, as far as possible, software testing from the hardware acceptance trials. The aim of the procedure is to test software as soon as it becomes available and merely to check that each installation has a good copy of the required items in its library. It is intended for use by large organizations rather than by individual small users.

It is proposed that software should be rigorously tested outside the acceptance trials. It requires that items of software should be given 'type' approval on a number of prescribed configurations as soon as possible after company release. During an acceptance trial, it should merely be proved that a correct copy of the software exists at the installation for that particular configuration.

Various levels of type approval would be required but only the more important packages need be subjected to all levels of the testing envisaged. The suggested levels of approval are:

(a) Documentation. This would check that the user manuals are available, are in accordance with the specifications issued by the supplier, that they are of an acceptable standard of presentation and literacy and 194 are self-consistent. Basically, the question that should be answered is 'is it likely that the potential user will be able to use the documentation to write effective programs for his installations'. It would also be necessary to cheek that an acceptable up-dating system is provided so that a user can be sure that he has up-to-date manuals. The greater part of the work involved in this level of approval could be done as part of the normal appraisal of software carried out before a machine is ordered.

(b) Availability. This would check that an item of software had been released and was in general conformity with the documentation. This would not be a rigorous check but merely a cheek of a package's existence in a working form on a number of prescribed configurations. This check should be carried out as soon as possible after the contractor makes the software available. These tests would be very similar to the current demonstrations of software during acceptance trials.

(e) Detailed verification of facilities. This stage would consist of testing the specified facilities of each package in detail and checking the package's interaction with others with which it can be associated. Such tests should include the running of special test programs to check on facilities, plus 'soak' or random tests using a wide variety of typical user type programs from a wide variety of sources.

(d) Performance assessment. This test would check on the performance of a package. It would determine the ‘efficiency’ of a package in terms of its core store requirements, peripheral requirements and running times under various conditions.

Approval at each level would be given as soon as those carrying out the tests had established, with an acceptable degree of confidence, that a package would provide satisfactory service in field use. We cannot call for perfections merely a reasonably high probability that the ‘average’ user will find the package usable. In practice it may be necessary to provide partial approval to a package, for example approved for use on machines over a certain size only.

7. Application of the proposed procedure

It is envisaged that each ‘level’ of testing should be carried out in the following manner:

(i) Documentation check. This would be carried out very largely during 195 the normal appraisal of the software before the machine was ordered, or at least as soon as the documentation became available. The work would take about three man-months for a major package. Every effort should be made to complete this work at least three months before a package’s proposed release date. This phase of the work would be done by attempting to use the documentation to write test programs for use in subsequent phases of the testing procedure. However, as the manuals are often produced in parallel with the package itself, it is to be expected that the programs produced will have to be amended in the light of subsequent amendments to these manuals.

(ii) Availability check. This should be carried out as soon as possible after the release of a package. Only a few test programs would be required and these would be written during the documentation check. The final debugging and running of these programs would form the availability check. It should be possible in most cases to accept the supplier’s own validation procedures for this check if, after investigation by those responsible for operating the ‘type’ approval scheme, it is found to be effective. It would probably still be necessary to carry out random checks to ensure that the company’s standards are being maintained. This phase should be concluded within a month of a package’s release. The effort required is likely to be less than one man-month per major package tested.

(iii) Detailed check. This level of approval is the most onerous part of the test procedure. To be of real value to the user, this check should be completed as soon as possible after the release of a package, certainly within three months unless there is no requirement by the user for the package in the near future. For such packages for which there is no immediate requirement, evidence of extensive satisfactory use by other customers could be accepted in lieu of a detailed check. Such a procedure might also be adopted for the lesser used packages that are unlikely to be vital to the functioning of the approval organization.

The detailed testing will be made up of two types of work: the testing of packages limited to one manufacturer, or even one machine, such as operating systems, and testing those items that tend to be machine independent, such as high level languages. The effort required for machine dependent items is likely to be large and to be a continuing load for the ‘approving’ body. However, the work involved on the standard languages, 196 such as Cobol, Algol and Fortran, can very largely be done ‘once and for all’ with only minor adaptations having to be made to suit each new system.

The effort required is likely to be about nine man-months to test an operating system and about three man-months to design and write test programs for the first test of the implementation of a well known language such as Fortran and about one man-month for each subsequent test of that language.

(iv) Performance check. Some idea of the performance of a package will be gained during the ‘availability’ and ‘detailed’ checks but it is desirable to have separate tests aimed at measuring performance or ‘efficiency’. These performance tests should be conducted after a few months of field use to allow operating experience to be gained and any residual failings to be removed. Such tests might be conducted three to six months after the release of a package. It might be easier to measure performance comparatively rather than absolutely, in that an absolute test of, say, a compiler’s efficiency would be very difficult and time-consuming to carry out but for a comparative test, a variety of benchmark programs might suffice. However, for machine dependent packages, such as operating systems, absolute measurements will have to be made.

The likely effort required would be:

(a) For computers using pre-written benchmark programs, about one man-month per language.

(b) For operating systems, up to six man-months per system.

8. Criteria of acceptance

The supplier would have to have a criterion to aim at for each level or the approval procedure and it is here that the greatest difficulty arises. Software is a highly versatile product whose performance depends very much on the precise use made of it and thus useful generalised measures of software performance are not easily found. The value of a piece of software is still very much a matter of opinion but the testing procedure is aimed at finding facts and highlighting failures so that reasonable discussion should be possible between the parties.

Approval at Level 1 could initially be given as soon as the appropriate set of manuals become available for each package and had been used 197 to write a few simple programs. Criticisms would be fed back to the supplier but it is not realistic to expect manuals to be re-cast at the ‘approving’ authority’s request, although a long-term influence might be expected. Of course, it could be expected that errors would be corrected as soon as possible. Approval at Level 1 might be interpreted as indicating that usable manuals are generally available to customers.

Approval of Level 2 could be given as soon as demonstrations of a package had been witnessed by the approving body. The demonstration programs would, of course, have to be acceptable to the approving body. If the supplier’s own validation procedures were acceptable, then documentary evidence of their completion might be acceptable instead of a demonstration.

Approval at Level 3 would be given when all the facilities offered or called for in the contract were shown to be working. However, we cannot expect perfection and it might be necessary to give partial approval in some cases. An alternative might be to divide the facilities into ‘required’ and ‘desired’ and call for 100 per cent implementation of the required facilities and, say, 70 per cent of the ‘desired’ ones, together with, say, three months’ grace to complete the implementation.

The requirements for approval at Level 4 can often be well defined; for example, ‘shall occupy not more than 8,000 words of core store’, ‘shall compile at 1,000 statements per minute for a program of 200 statements’, etc.

9, Effort and cost estimates for the approval scheme

The total cost of implementing the previously described Type Approval scheme can be expected to vary greatly from system to system. The following figures are for testing a typical currently available operating system in accordance with the above proposals.

198

Software Item	Effort in Man-months				Machine time in hours			
	Level 1	2	3	4	Level 1	2	3	4
Languages								
Algol	2	1	2	1	0	1	2	1
Cobol	2	1	2	1	0	1	2	1
Fortran	2	1	2	1	0	1	2	1
R.P.G.	2	1	1	1	0	1	2	1
Assembly code	3	1	2	1	0	1	3	1
Operating system								
Supervision	3	1	3	2	0	1	3	1
File Control packages	3	1	2	2	0	1	3	1
Peripheral handling routines	3	1	2	1	0	1	3	1
Program testing aids	3	1	2	1	0	1	3	1
Application packages								
Sort routines	2	1	1	1	0	1	1	1
Miscellaneous	3	2	3	1	0	1	1	1
Totals:	28	12	22	13	0	11	25	11

Thus, the total effort required is 75 man-months, together with the use of 47 hours of machine time. The total cost of the exercise would therefore be approximately £25,000 spread over a period of a year. This expenditure is not entirely over and above existing expenditure because, in effect, Levels 1 and 2 of the proposed procedure are already carried out by the present procedure. Thus, the new expenditure required is about £13,000, i.e. a doubling of current expenditure.

10. Consequences of approval

For the procedure to be effective, there must be some financial consequences of the software not obtaining approval. A possible approach is to attach 10 per cent of the purchase price or rental to each level of approval. This would allow up to 40 per cent of the price to be retained under late delivery conditions. It would also be necessary to require at least a minimum set of software being available before any money was paid. A possible approach would be for the contract to list the required items of software and against each item give the minimum level of approval 199 required for acceptance of the installation. It will also be necessary to have associated with the contract the specifications of the required software. These would detail the facilities and performance of the software. However, this will require a change of attitude on the part of computer manufacturers who are often very reluctant to provide any detailed performance information until after the software in question is working. This attitude requires the customer to purchase his software almost as an 'act of faith' in the supplier and is surely not something that can be allowed to continue.

11. Conclusions and discussion

This paper has attempted to present a possible method by which large organisations, such as the United Kingdom government, might test the software supplied by computer manufacturers to its installations. Its main aim may be described as an attempt to ensure that new computer installations can rapidly take on the work for which they were purchased by ensuring that installations have the best available information on which to plan their work. It is a procedure aimed at elucidating the facts about software; what works and what does not and how well does it work? The cost of doing this sort of exercise is not trivial but with the number of computer systems involved, the cost per installation is quite small and if the procedure can speed up the take-on of work by allowing the confident use of proved software, rather than the hesitant use of the unknown, it will prove worthwhile. Hopefully, such a procedure would encourage the computer manufacturers to improve their own quality assurance procedures, perhaps even to the point where use of testing of software becomes unnecessary.

200

PERFORMANCE MONITORING AND SYSTEMS EVALUATION

by

Tad B. Pinkerton

I. Performance Monitoring

There are many specific motives for evaluating the performance of a large-scale computing system. Today's systems are more complicated and more expensive than their predecessors; hence their operation is more difficult to comprehend and more costly to misjudge. At the same time, a utility-grade service of increasingly high quality is being demanded. These circumstances require the availability of performance data for use in (a) system design (b) system acquisition (c) changes in configuration (d) software production (e) system checkout (i) normal operation (g) and advanced research.

Raw data obtained from a computing system usually consist of descriptions of instantaneous events: the time at which each such event occurs together with information peculiar to its nature. The data reduction process under which such data become meaningful is often complicated, requiring considerable cross-referencing and consolidation of individual event descriptions. For example, the question of system behavior is bound up with the consideration of system load. Hence operational information must be combined with workload data in order to interpret the former. And at a more basic level, an analysis program must 'invert' the operation of a job scheduler in order to extract resource allocation data from scheduling event descriptions. Many useful items are found in counts or averages produced by tabulating data over intervals of time.

201

At least four different techniques have been successfully used for the collection of system performance data:

(1) hardware measurement

Special purpose devices such as TS/SPAR for the IBM System 360 Model 67 and the UCLA 'snuper computer' have been built to 'plug in' to a machine and directly monitor signals in interesting places. The principal advantage of this procedure is that the measurement is non-interfering: no distortion of the data occurs as a result of its collection, which can take place during normal system operation. The extremely high resolution obtained with this technique is occasionally useful, but it is frequently a disadvantage in that problems are created with the storage and reduction of vast quantities of data. A more serious problem with hardware measurement is that the required engineering expertise and cost of special-purpose hardware place it out of the reach of most installations.

(2) hardware simulation

It is not at all uncommon to simulate one machine with a software package on another, or on the machine itself. A current example is the System 360 simulator at Princeton University. A simulation package provides data at a resolution nearly as high as that obtained by hardware measurement, under program control and in program-oriented terms. In addition, it is readily modifiable to produce information of different kinds. Unfortunately, the simulated system must run at such a small fraction of the rate of the actual system that it can only be run for short periods of time, and hence not under normal operating conditions. Most timing data is at least suspect, for it is difficult for the simulator to maintain comparative operating speeds of simulated components, to say nothing of unsimulated peripherals.

(3) software sampling

Data can be obtained from a system in normal operation by adding instructions to the system itself. Sampling, e.g. by periodically interrupting processes to record their status, is usually not hard to implement and provides good control over the data collection overhead. The primary disadvantages of this technique are that certain kinds of information may be difficult or impossible to obtain, and considerable attention must be given to questions of sample size and validity.

202

(4) continuous software monitoring

Provided that care is taken to control the overhead and that hardware-level resolution is not required, the most desirable data collection procedure allows continuous monitoring of events under program control. Such a facility can be turned on and off during normal operation, easily changed to adjust the quantity and kinds of data, and it provides a complete description of the monitored phenomena in a given period. The author's experience in implementing this kind of facility in the Michigan Terminal System (MTS) has shown that it can be done with surprisingly good resolution and low overhead, to produce invaluable data for the purposes mentioned above.

The following general observations may be useful in the implementation of a data collection facility (DCF);

a. A DCF should be included in the design of a system, for part of it will be embedded in the supervisor at a low level, and careful planning is necessary to ensure a clean implementation.

b. A DCF is very useful for the development of the system itself, and therefore should be produced early in that development.

c. For reasons of both effectiveness and generality, as much of the analysis of collected data as possible should be deferred for independent processing.

d. Both the collection and analysis phases of operation require a variety of data selection modes, e.g. by type of datum, by task, by recording time. Good selection capability reduces both the collection overhead and data reduction time.

II. Systems Evaluation

Once measures have been taken to provide accurate operating system data, it becomes reasonable to consider the problem of evaluating system performance. The performance criteria one uses clearly depend on one's motives for evaluation. Within a single installation, one usually focusses on criteria of utilization of both hardware and software components, and on standards of the quality of service provided. Discussions of both these factors require a basis for characterizing the workload under which given levels of utilization and response have been obtained. Indices chosen to describe these three operational factors are biased by the types of performance data available and the way in which they are obtained.

203

Because distinct installations rely on different performance data and emphasize different criteria for evaluation, it is difficult to compare their performance. Thus there is an additional need to define standards for the descriptions of workload, utilization and response so that performance in different environments can be meaningfully compared. It would be particularly useful to provide such standards for general purpose multi-access systems (where the need is greatest!) in such a way that they also apply to simpler multi-programming and single-thread systems as special cases.

204

TOWARDS A METHODOLOGY OF COMPUTING SYSTEM DESIGN

by

B. Randell

Introduction

Three independent, but related, projects concerned with the methodology of computing system design have recently been reported. The work described by Dijkstra [1] concerns the design of a multiprogramming system for the X8 computer, at the Technological University, Eindhoven. Parnas and Darringer [2] have described their work on a language SODAS and demonstrated its use in the design of a fairly complex hardware system. Finally F. Zurcher and the present author [3] have given a short discussion of 'iterative multilevel modelling', the methodology being investigated to aid the design of an experimental multi-processing system [4] at the T. J. Watson Research Center. The purpose of the present note is to compare and contrast these three projects very briefly, and to attempt an identification of major problems still to be faced in achieving an effective methodology of computer system design.

Structuring the Design Process

The author's belief is that the most important aspect of all three projects is the stress they put on achieving a structuring of the design process, and on making the system being designed reflect this structure.

Computing systems are undeniably extremely complex. They are notoriously difficult to design successfully, and when complete are difficult to understand or to modify. A system can be thought of as being the embodiment of a set of solutions to a set of distinct although related problems. All three projects lay stress on a careful consideration of the order in which the various problems should be tackled, and of the consequences of each design decision, both on those decisions which have already been taken, and on those problem areas which remain to be addressed. 205 Most important however, they make the system that they are designing contain explicit traces of the design process — in other words, the system is structured to demonstrate the way in which it was designed, and the designers' views as to how the various problem areas are related to each other.

The papers by Dijkstra and by Zurcher and Randell both use the term 'level of abstraction'. Their systems are constructed as a sequence of levels of abstraction, each consisting of a set of co-operating sequential processes [5]. In general, the primitives used to construct the programs which define processes on one level are provided by the processes of the immediately lower level. Each level therefore is in essence a set of solutions, specified directly in terms of appropriate quantities, to a set of problem areas which the designers have chosen to regard as being closely related. Less related problem areas are dealt with on other levels. For example, the lowest of Dijkstra's levels is one which contains solutions to problems caused by timing constraints, and the fact that there is only a single processor in his computing system. The levels above this can ignore these problems, and assume, for example, the existence of a multiplicity of processors. By such means Dijkstra and his colleagues have retained control of the complexity of their system to such a degree that they could convince themselves, a priori, as to its logical correctness.

The Ordering of Design Decisions

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. The design structuring described above is an attempt to mitigate the effects of such occurrences.

There are two rather 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 206 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.

Of the three projects under discussion, those by Parnas and Darringer, and by Zurcher and Randell lay most stress on the top-down approach. Zurcher and Randell in fact state that their aim is to defer decisions as to whether a given component be constructed out of software or of hardware until a late stage in the design process, when cost performance analyses can be made. In contrast, Dijkstra is concerned with the design of a multiprogramming system to run on an existing hardware system, so has at least presented the results of his design effort as if produced by a bottom-up approach.

The Place of Simulation in the Design Process

Since the computer profession insists on building systems which are more complicated than it can analyze mathematically, many designers have made extensive use of simulation. Probably the most successful uses have been for investigation of isolated problem areas such as storage interference or I/O buffering, and for detailed modelling of completed designs (see for example Nielsen [6]). Two of the three projects under discussion, namely SODAS and the Iterative Multi-level Modelling technique, involve the use of simulation. However in these projects simulation is regarded not as an adjunct to, but rather as an integral part of, the entire design effort.

207

In many design efforts it is difficult to identify exactly what is in the design at any given stage. Usually the partially completed design will consist of inaccurate and out-of-date documentation, unwritten 'understandings' between groups of designers, and in late stages of the design, partially constructed components (hardware or software). The intent of the above two projects is to keep the design so formal that at each stage in the design process it is capable of objective evaluation — being machine-executable is a very practical means of achieving this goal. The machine-executable partial design is, since it can obviously not be doing the complete job required of the system, therefore just a 'simulation' of the complete system. As design work progresses this simulation will gradually evolve into the real system.

With this view of simulation as part of the design process, there is no room for arguments that the design has not been simulated faithfully, that the simulation results are hopelessly pessimistic and in any case only apply to last month's design, etc. The simulation is the design.

Future Problems

It would be easy to talk about the problems directly corrected with providing an effective tool for use by a large group of designers in simulating their system as the design progresses. However it is fairly clear that we have barely started to tackle the problem of finding an effective methodological approach to computing system design.

We are sorely in need of techniques for representing our partial designs so that at each stage the spectrum of possible future design decisions is made clearer, and the consequences of a particular choice can be more easily evaluated. (For example in programming the choice as to whether to make multiple copies of a given set of information, or to maintain just a single set and access it from different places by varying levels of indirection, is often very arbitrary). Similarly, if we could find some way of guiding designers as to what might be a good order in which to take decisions, this might have a considerable effect on the overall quality of our system designs.

Acknowledgements

The author's initial work in this area was carried out jointly with Frank Zurcher, with much valuable advice and criticism from Carl Kushner, 208 Meir Lehman and Hans Schlaeppi. The viewpoints, both on existing approaches

to methodologies of system design and on future problems, expressed herein have greatly benefitted from extensive discussions with Edsger Dijkstra at the Technological University, Eindhoven.

References

1. Dijkstra E.W., *The Structure of the 'T.H.E.' Multiprogramming System*. Comm. ACM 11, 5 (1968) pp. 341–346.

2. Parnas, D.L., and Darringer, J.A., *SODAS and a Methodology for System Design*. AFIPS Conference Proceedings Vol. 31, 1967 Fall Joint Computer Conference. Thompson Books, Washington D.C. (1967) pp. 449–474.

3. Zurcher, F.W. and Randell, B., *Iterative Multilevel Modelling, A Methodology for Computer System Design*. IFIP Congress 68, Edinburgh, August 5–10, 1968.

4. Lehman, M., *A Survey of Problems and Preliminary Results Concerning Parallel Processing and Parallel Processors*. Proc. IEEE 54, 12 (1966) pp. 1889–1901.

5. Dijkstra E.W., *Cooperating Sequential Processes*. Technological University, Eindhoven (September 1965) .

6. Nielsen, N.R., *The Analysis of General Purpose Computer Time-Sharing Systems*. Doctoral Dissertation, Stanford University, California, (December 1966).

209

DOCUMENTATION STANDARDS

excerpted from

DOCUMENTATION FOR SERVICE AND USERS

by Franz Selig

Documentation standards are set in many individual centers using forms easy to fill out which typically contain the following entries:

TITLE PAGE

a. Project name and number
b. Computer models on which it can be run
c. Program language(s)
d. Client department and number
e. Client contact
f. Frequency of program use (e.g. annual, monthly, continual)
g. Program user(s) (i.e. individual, department, general)
h. Date issued
1. Brief abstract
J. Issued by: (Programmer's name)

MODEL DESCRIPTION

a. Problem statement
    1. Complete description with history
    2. Purpose and value
    3. Urgency
    4. Area of intended use
b. Technical writeups and diagrams
c. Basic formulae (with definitions of symbols and terminology)
d. Restrictions concerning physical limitations and bounds
e. List of references

210

USER'S GUIDE

a. Description of how to use the program
    1. How to submit a job
    2. Where the program resides
    3. Sample job card

b. Input cards description

1. Sample deck layout of JCL Cards and indication of where to insert data
2. Data input
    - i) Order
    - ii) Layout
    - iii) Sample input forms (filled in with data)

c. Output

1. Description of information
2. Sample output from case data given in 2) iii

d. Estimation of computer run times

e. Table of error messages and recommended courses of action

f. Complete list of options, with a brief description of how the programming affects these variations

g. A list of related programs:

1. Is there communication of data between this program and others under separate cover? If so, describe in detail.
2. Are there other programs which could complement this program? What are the basic differences?

h. Possible Extensions

PROGRAMMER'S GUIDE

a. Overall logic flow diagram, including interaction of data sets

b. If overlay, show the tree structure

e. The map listing from the sample case shown in the User's Guide

d. List of definitions of all variables in common

e. Description of flow sheet of routines (for main routine and each subroutine)

f. Description of data sets

h. Off-line devices (plotter, etc.)

1. Description
2. Sample diagrams

211

OPERATOR'S GUIDE

a. Job flow on computer

Schematic drawing identifying all data sets and devices

b. Input

Exact layout of input streams, with detail of JCL (data cards, tape files, disc files)

c. Operation

1. Normal — instructions and description (with sample console sheet)

2. Abnormal — list possible troubles, with recommended courses of action

d. Output instructions

E.g., tapes to be listed or plotted, no. of copies

SOURCE DECKS

a. All source decks would begin with comment cards, identifying:

1. Project name and number

2. Client dept. name and number

3. Programmer’s name

4. Completion date

b. Additional comment cards throughout program to:

1. Define nomenclature

2. Identify areas of calculation

c. Each source deck should have the project no. in columns 73 to 76

d. All Fortran source decks would have sequence numbers in columns 77 to 80 (e.g. 0010, 0020, 0030 — to a low for additions). COBOL would have sequence numbers in columns 01 to 06).

212 213

APPENDICES

A1. Participants

Chairman:

Prof.Dr. F.L. Bauer,
Mathematisches Institut der Technischen Hochschule,
D-8 München 2,
Arcisstraße 21, Germany

Co-Chairmen:

Dr. H.J. Helms,
Director,
Northern Europe University,
Computing Center,
Technical University of Denmark,
DK-2800 Lyngby, Denmark

Professor L. Bolliet,
Laboratoire de Calcul,
Université de Grenoble,
Boite Postale 7,
F-38 Saint-Martin-D'Hères
France

Group Leaders

Design

Professor A.J. Perlis, (Chairman), Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, USA.

Dr. M. Paul, Leibniz-Rechenzentrum, D-8 München 2, Richard Wagnerstr. 18, Germany.

Mr. B. Randell, IBM Corporation, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598, USA.

214

Production

Dr. P. Naur (Chairman), A/S Regnecentralen, Falkoneralle 1, 2000 Copenhagen F. Denmark.

Dr. H.R. Wiehle, AEG-TELEFUNKEN, D-775 Konstanz, Postfach 154, Germany.

Professor J.N. Buxton, University of Warwick, Coventry CV4 7AL, Warwick, England.

Service

Prof.Dr. K. Samelson (Chairman), Mathematisches Institut der Technischen Hochschule, D-8 München 2, Arcisstraße 21, Germany.

Professor B. Galler, The University of Michigan Computing Center, North University Building, Ann Arbor, Michigan, USA.

Dr. D. Gries, Department of Computer Science, Stanford University, Stanford, California 94305, USA.

Mr. A. d'Agapeyeff, Computer Analysts and Programmers Ltd., CAP House, 14/15 Great James Street, London, W.C.1., England.

Mr. J.D. Babcock, Allen-Babcock Computing Inc., 866 U.N. Plaza, Suite 554, New York, New York 10017, USA.

Professor R.S. Barton, Consultant in System Design, P.O. Box 303, Avalon, California 90704, USA.

Mr. R. Bemer, GE Information Systems Group, 13430 Black Canyon Highway, C-85, Phoenix, Arizona 85029, USA.

Prof.Dr. J. Berghuis, N.V. Philips' Computer Industrie, Postbus 245, Apeldoorn, The Netherlands.

Mr. P. Cress, Computing Center, University of Waterloo, Waterloo, Ontario, Canada.

Professor L. Dadda, Politecnico, Piazza L. Da Vinci, I-2O133J Milano, Italy.

Dr. E.E. David, Jr., Bell Telephone Laboratories Inc., Murray Hill, New Jersey O7971, USA.

Prof.Dr. E.W. Dijkstra, Department of Mathematics, Technological University, Postbox 513, Eindhoven, The Netherlands.

Dr. H. Donner, Siemens Aktiengesellschaft, ZL Lab Gruppe Programmierungsverfahren, D-8 München 25 Hofmannstraße 51, Germany.

215

Mr. A. Endres, IBM Laboratory, Programming Center, D-703, Böblingen, P.O.B. 210, Germany.

Mr. C.P. Enlart, European Program Library IBM, IBM France, 23, Allee Mailasson, F-92 Boulogne Billancourt, France.

Professor P. Ercoli, Istituto Nazionale per le Applicazioni del Calcolo, Piazzale delle Scienze 7, I-00185 Rome, Italy.

Mr. A.G. Fraser, The University Mathematical Laboratory, Corn Exchange Street, Cambridge, England.

Mr. F. Genuys, IBM France, 116 Avenue de Neuilly, F-92 Neuilly, France.

Professor S. Gill, Centre for Computing and Automation, Imperial College, Royal School of Mines Building, Prince Consort Road, London, S.W.7., England.

Mr. H.R. Gillette, Control Data Corporation, 3145 Porter Drive, Palo Alto, California 94304, USA.

Mr. A.E. Glennie, Building E.2., Atomic Weapons Research Establishment, Aldermaston, Berks., England.

Dr. G. Goos, Mathematisches Institut der Technischen Hochschule, D-8 München 2, Arcisstraße 21, Germany.

Professor R.M. Graham, Project MAC, M.I.T., 545 Technology Square, Cambridge, Massachusetts 02139, USA.

Mr. R.C. Hastings, IBM Corporation, 540 East Main Street, Rochester, New York 14604, USA.

Mr. J.A. Harr, Bell Telephone Laboratories Inc., Naperville, Ill. 60540, USA.

Professor J.N.P. Hume, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada.

Mr. H.A. Kinslow, Computer Systems Consultant, 14 Donnelly Drive, Ridgefield, Connecticut 06877, USA.

Mr. P.M. Kjeldaas, Kjeller Computer Installation, P.O. Box 70, N-2007 Kjeller, Norway.

Mr. H. Köhler, AEG-TELEFUNKEN, D-775 Konstanz, Büchlestraße 1-5, Germany.

Mr. K. Kolence, Boole and Babbage Inc., 1121 San Antonio Road, Palo A1to, California 94303, USA.

Dr. G. Letellier, Départment Techniques Nouvelles, Division Informatique, SEMA, 35 boulevard Brune, Paris 14e, France.

216

Mr. A.I. Llewelyn, Ministry of Technology, Abell House, John Islip Street, London, S.W.1., England.

Dr. R.M. McClure, Computer Science Center, Institute of Technology, Southern Methodist University, Dallas, Texas 75222, USA.

Dr. M.D. McIlroy, Bell Telephone Laboratories Inc., Murray Hill, New Jersey 07971, USA.

Mr. J. Nash, IBM UK Laboratories, Hursley Park, Winchester, Hants., England.

Mr. A. Opler, IBM Corporation, Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, New York 10598, USA.

Dr. T.B. Pinkerton, Department of Computer Science, 8 Buccleuch Place, Edinburgh 8, Scotland.

Prof. Dr. W.L. van der Poel, Technische Hogeschool, Julianalaan 132, Delft, The Netherlands.

Mr. G.D. Pratten, Department of Computer Science, 8, Buccleuch Place, Edinburgh 8, Scotland.

Mr. T.M.H. Reenskaug, Central Institute for Industrial Research, Forskningevn. 1, Blindern – Oslo 3, Norway.

Mr. D.T. Ross, Electronic Systems Laboratory, M.I.T., Room 402A, 545 Technology Square, Cambridge, Massachusetts 02139, USA.

Mr. F. Sallé, C.I.I., Rue Jean Jaurès, F-78 Clayes-sous-Bois, France.

Dr. F. Selig, Mobil Research and Development Corporation, Field Research Laboratory, P.O. Box 900, Dallas, Texas 75221, USA.

Mr. J.W. Smith, Scientific Data Systems, Station A-102, 555 South Aviation Blvd., El Segundo, California 90245, USA.

Mr. R.F. Wickens, Ministry of Technology, Computer Advisory Service, Technical Support Unit, 207 Old Street, London E.C.1., England.

Dr. P.L. Wodon, M.B.L.E., Research Laboratory, 2, avenue van Becelaere, Brussels 17, Belgium.

Scientific Secretaries

Prof. L.K. Flanigan, Department of Computer and Communication Sciences, The University of Michigan Computing Center, North University Building, Ann Arbor, Michigan, USA.

Mr. I. Hugo, International Computers and Ltd., 93/99 Upper Richmond Rd. London, S.W.15., England.

217

Observers

Dr. H. Haller, Deutsche Forschungsgemeinschaft, D-552 Bad Godesberg, Kennedy-Allee 40, Germany.

Mr. P.H. Kenney, OSCAD, SHAPE, B-7010, Belgium.

Dr. E.G. Kovach, Director, Office of General Scientific Affairs, International Scientific and Technological Affairs, Department of State, Washington, D.C. 20520, USA.

Captain B. Pavlidis, R.H.N., HGNS, D.M.E.O. 14, Pentagon, Athens, Greece.

Major Gr. Tsiftsis, HGNS, D.M.E.O. 14, Pentagon, Athens, Greece.

218

A2. CONFERENCE SCHEDULE

MONDAY 7th OCTOBER

9.00	to	9.30	Introductory remarks by Professor F.L. Bauer, Conference Chairman
9.30	to	10.00	Keynote speech by Professor A.J. Perlis
10.20	to	12.30	Design of software, plenary session
14.00	to	15.00	Planning meetings of the three working groups
15.00	to	16.30	Production of software, plenary session
	and		
16.50	to	19.00	

TUESDAY 8th OCTOBER

9.00	to	10.00	Lecture on 'Software Components' by Dr. M.D. McIlroy
10.20	to	13.00	Service of software — plenary session
14.30	to	16.30	Parallel meetings of the working groups
	and		
17.00	to	18.30	
19.00	to	00.00	Discussion on software pricing

WEDNESDAY 9th OCTOBER

9.00	to	11.00	Design of software, plenary session
11.25			Excursion for the rest of the day to Munich

THURSDAY 10th OCTOBER

9.00	to	10.30	Production of software, plenary session
	and		
10.50	to	13.00	
14.30	to	17.00	Service of software, plenary session
17.30	to	19.00	Discussion on 'Techniques for Software Component Mechanisation'
21.00	to	22.30	Discussions on 'Gaps between Intentions and Performance in Software' and 'Software Engineering Education'

FRIDAY 11th OCTOBER

9.00	to	11.00	Parallel meetings of the working groups
16.00	to	18.30	Summary plenary session

219

A3. ORIGINAL LIST OF TOPICS THAT WERE TO BE DISCUSSED

Working Group D — Design

D1. General Guidelines

D1.1 Design as controlled by external function.

The environments (user, equipment, etc.) guide the design.

Example: User languages and storage requirement guide the design.

D1.2 Design as controlled by internal function.

The functions (parsers, I/O routines, file systems, symbol tables) guide the design.

Example: Sharing of functions among sub-systems which appear different to the user.

D2. General Techniques

D2.1 Deductive or inductive method of design.

Increasing specialization of goal versus increasing complexity of combinations of building-blocks.

D2.2 The search for modularity and establishment of general interfaces.

D2.3 Complexity controlled by hierarchical ordering of function and variability.

D2.4 Design controlled by increasing specialization of description.

The use of simulation to monitor the design process.

D3. Proscriptions

D3.1 Intrinsic features I: Completeness, efficiency, modularity, communicability.

With respect to the goals performs and communicates well the complete set of functions using a minimum of units.

D3.2 Intrinsic features II: self-monitoring and improvement of performance.

Every system represent a compromise under insufficient knowledge of load. Use should lead to improvement of performance.

220

D3.3 Designing incremental systems.

Reaching a target system from a given system.

Adjoining and merging separately designed systems to form a new one.

D3.4 The issue of balance I:

Balancing the cost versus the merit of control, security, training, and convenience.

D3.5 The issue of balance II:

The virtue of limited goals to attain excellent performance.

D4. Relevant Design Problems

D4.1 The influence of data structures on system design.

D4.2 Allocation of fixed resources among competing processes.

Application to time sharing.

D4.3 Co-operation of processes among shared resources.

Time sharing.

D5. Documentation

D5.1 General properties of design documentation: Levels, precision and function (how and what). Incremental documentation.

D5.2 The role of high level language in system design:

D5.2.1 The use of PL/1 to describe Multics.

D5.2.2 The use of an Algol-like language (PL/360) to describe a machine dependent assembly language.

Working Group P — PRODUCTION

P1. The Organization for Producing Software

P1.1 Number and quality of people used;

P1.2 Structure of large groups of programmers;

P1.3 Control and measurement of a programmer's output and possibility of its improvement;

P1.4 Project or product oriented organization;

221

P1.5 Internal communication within a large group of programmers.

P2. Production Techniques for Producing Software

P2.1 Schedules of work;

P2.2 Standards for programming, testing and documentation;

P2.3 Compatibility;

P2.4 Use of simulators, high level languages etc;

P2.5 Redesign of package specification during software production.

P3. Monitoring the Production Process

P3.1 Reporting techniques and control of schedules;

P3.2 Comparison of current costs, current estimated completion date, current estimated technical quality of the final product compared with specification requirements.

P4. The Final Product and its Evaluation

P4.1 Evaluating performance and final quality control of software;

P4.2 Evaluating adequacy of documentation for the user, for maintenance and modification.

Working Group S — SERVICE

S1. Distribution of Software

S1.1 Characterization of distribution media;

S1.2 Levels of language in which distributed software is represented;

S1.3 Acceptance criteria for distributed software validation;

S1.4 Documentation;

S1.5 Adaption to configuration;

S1.6 Parameter identification for adaption.

S2. Maintenance and Extension of Software

S2.1 Detection of errors: responsibility, diagnostic techniques.

S2.2 Error reporting: responsibility, direction of report.

222

S2.3 Method of response to error reports.

S2.4 Distribution of corrections.

S3. Instruction of Users

S3.1 Manuals for different classes of users such as:

a) computing centers

b) large scale permanent users

c) incidental users.

S3.2 Techniques of instruction (including automatic techniques).

S4. Documentation for Service and Users

S4.1 Levels of documentation such as:

a) maintenance

b) instruction;

S4.2 Criteria for documentation requirements;

S4.3 Techniques for document production.

S5. Performance Monitoring and Systems Evaluation

S5.1 Criteria and measures of performance;

S5.2 Methods of measuring

a) by hardware or by software

b) by sampling or continuous

c) micro or macro.

S6. Feedback into Design and Production

S6.1 Mechanisms for feedback (such as users groups);

S6.2 Responsibility;

S6.3 Description of feedback channels and recipients;

S6.4 Consideration of efficiency of feedback;

S6.5 Marketing considerations.

S7. Modifications of Existing Software

S7.1 Reasons for modifications;

S7.2 Improvement of performance;

223

S7.3 Problem modification leading to:

- a) generalization of software
- b) adaption of software, influence of cost estimates and marketing considerations.

S7.4 Responsibility for execution of modified product.

S8. Variations of Software Available to the User

S8.1 Criteria for determination of:

- a) degree of variations needed by the user — tolerated by the user for variation in time, across group of users
- b) techniques for producing variations (incl. automatic).
- c) communications service covering variations.

S9. Reprogramming

S9.1 Problem techniques (incl. automatic).

S9.2 Responsibilities.

S9.3 Alternatives.

S9.4 Cost estimates, etc.

224

A4. ADDRESSES OF WELCOME

Welcome address by Dr. Otto Schedl, Minister of Industry and Transport for Bavaria

Software engineering has become one of the most important and comprehensive aspects of the technology of Information Processing, or informatics. Developments in this area play a key role in the overall progress of technology and economy in our countries.

With a host of possible applications and many immediate necessities for its use in science and industry successful software engineering is today one of the basic conditions for competitive computer production. In Germany this aspect of computer science will be one of the main activities of the special research group Informatics which has been established at the Technische Hochschule München, following a recommendation by the German Science Council. This group at the Technische Hochschule München collaborates with the University of München and is supported by the Leibniz-Rechenzentrum of the Bavarian Academy of Sciences.

The international Conference on Software Engineering in Garmisch is, therefore, an event which is hoped to stimulate the work on computer science in Munich, and the conference may very well, as the first of its kind, become a milestone in an increasing collaboration between scientists engaged in software engineering, since the explosive development in data processing techniques in all areas of modern society throughout the world constitutes a challenge to international collaboration.

It is a particular pleasure to me, as Minister of Industry and Transport for Bavaria, to welcome the participants to this international conference. The Science Committee of NATO has found with Garmisch not only a meeting place which is preferred in Germany for its scenery but it has also chosen a German state where the development of data processing techniques has always been followed with interest and given all the support possible within the available economic framework.

Allow me to take a certain pride in the fact that by 1952 computer science had already found a home in Bavaria at the Technische Hochschule München. It is hoped that the Leibniz-Rechenzentrum of the Bavarian Academy 225 of Sciences, when its construction is completed, will form the hub from which advances in data processing techniques will radiate to science and technology.

I wish the conference every possible success in its work and hope that all participants will have an enjoyable stay in Werdenfels country.

7th October, 1968

Translation of telegram from Mr. Gerhard Stoltenberg, Minister for Scientific Research of the Federal Republic of Germany, 7th October 1968

To Prof.Dr. F.L. Bauer, Chairman of the NATO Conference on Software Engineering.

Dear Professor Bauer,

The productivity in science, industry and public administration is to an ever increasing extent determined by the progress in data processing.

I am therefore particularly pleased that a working conference of such a high scientific level, devoted to problems in software engineering, is being held in the Federal Republic of Germany.

I wish you and all participants every success with the conference.

With best regards

Gerhard Stoltenberg
Bundesminister für Wissenschaftliche Forschung.

CONTRIBUTOR INDEX

The index gives the page number of named quotations. Within square brackets is given the page number where the full name and address of the contributor is given.

SUBJECT INDEX

The making of a subject index proved to be very difficult, and the result given below is known to be deficient in many respects. It should only be tried when the list of contents on pages 5 to 7 has been consulted in vain