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-π/2, 0-2π, also -π/2 to π/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 cooperation 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, Département 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.

226

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