A Gift from Pandora's Box:

The Software Crisis

María Eloína Peláez Valdez

PhD
University of Edinburgh
1988

Abstract

The thesis is an exploration in the history of software development. Its aim is to understand how and why software has developed the way that it has.

The thesis singles out major themes in the development of software. It begins by analysing the early separation of hardware and software and the development of the first high level languages, focusing particularly on the attempt to establish an international standard language and the conflict that arose between the proponents of ALGOL and the supporters of FORTRAN. The issues and themes which emerge at this stage are traced through an analysis of the development in the 1960s of third generation computers, and particularly the dramatic history of the IBM 360. The problems of software development during the 1960s created an awareness of how important and difficult programming was. Software was recognised not just as an appendage of hardware, but as a force in its own right. This was reflected in the commodification of software and in a growing sense of a "software crisis". This feeling of crisis gave rise to conscious attempts to provide solid theoretical foundations for the development of programming. Two main approaches emerged, structured programming and software engineering, and the tensions between these two approaches can be traced back to the early days of software development.

It is argued that the patterns and tensions to be found throughout the whole development of software are not accidental: they arise not simply from academic controversies but from the very nature of software and from the social circumstances of its production and use. It is concluded that there is no easy solution to the software crisis.

I declare that this thesis has been composed by me and is my own work.

Contents

Acknowledgements

It seems violent to claim this thesis as entirely my own work when I know how much others have contributed. In Donald MacKenzie and Stuart Anderson I have been very fortunate in having the perfect combination of supervisors, and I am extremely grateful to both of them not just for their contributions and commitment to this work, but also for their friendship and support: ¡Muchisimas gracias!

Many thanks too to John Holloway for debugging my English and for the many exciting discussions on the algorithmisation of life and the vitalisation of algorithms; to the people in the Science Studies Unit for providing me with a friendly and supportive environment; and to my family and friends for *everything*.

I wish to express my gratitude to the people I interviewed in the course of this work and to the Program in Science, Technology and Society of the Massachusetts Institute of Technology, who provided me with hospitality during the summer of 1984. This research has been made possible by the financial support of the University of Edinburgh, the Consejo Nacional de Ciencia y Tecnología and the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom.

Introduction

This thesis is an exploration in the history of software development. Its aim is to understand how and why software has developed the way that it has.

It is an exploration in the sense that it attempts to move into what is very largely uncharted territory. There are accounts by computer scientists of particular events or particular innovations in programming, and these accounts are extremely valuable, but they do not attempt to provide any systematic analysis of why things have developed the way that they have. There are occasional comments on the role of commercial interests in shaping the development of programming languages or expressions of concern about implications of software failure in relation to nuclear weapons, but, on the whole, the discussion remains purely internal to the field of computer science.

At the other extreme, in those few cases where an attempt has been made to look at software from a critical social perspective, relatively little attention has been paid to the issues actually involved in software development. These writings, of which the most influential are those by Kraft (1977) and by Greenbaum (1979), focus on the labour process of computer programmers. They argue, following in the tradition of Braverman (1974), that the labour process of computer programmers is subject to the general tendency to deskilling inherent in capitalist society. Developments in programming are seen in the context of a constant struggle between programmers and their managers for control over the labour process of programmers. Developments such as high-level languages or

structured programming and software engineering, for example, are seen as managerial strategies to deskill their workforce.

This approach is unsatisfactory in several respects as an attempt to understand the forces that shape software. Software development is seen as following a linear path, so that new developments in software mean nothing more than further attempts to deskill, routinise and fragment the labour process of programmers. This ignores the intense debates among computer scientists as to the nature of programming and all the conflicting interests that shape the development of software.

Another problem with the labour process approach to an understanding of software is that it reduces the relation between software and society to a focus on one particular labour process, the process of writing computer programs. The obvious objection to that is that software is not so simple: what shapes its development is not the struggle between one particular group of workers and their bosses, but the extremely complex interrelation between conflicts throughout the whole of society. It is surely wrong to see software as being shaped simply by the conflict between programmers and their managers when software development plays such a central role in almost every single labour process (and much else besides) in society today. Software is not an ordinary commodity. It cannot be treated as though it were a car or a table.

The question of the forces shaping software thus runs into the question of: what is software anyway? Is it adequate to think of it as a commodity? Is programming a branch of engineering? Or an aspect of mathematics? Is it an art,

a craft or a science?

This thesis approaches these issues historically. This does not mean that it is an attempt to write a comprehensive account of software development. The aim has been rather to focus on certain themes and certain turning-points that seem to be of particular significance, both in terms of their impact on software development and in terms of what they can tell us about the forces shaping that development.

The first chapter begins by tracing the growing separation of hardware and software, and of producers and users, from the earliest post-war days through to the first high-level languages. A particular study of the features and history of two of those languages, FORTRAN and ALGOL, throws light on a web of conflicting theories, traditions and commercial interests (and, entangled in the web, a devil and a demon). The outcome of these conflicts shaped the way that computers would be programmed and the way that programming problems would be conceptualised for years to come.

The conflicts which can be seen in the history of these two languages reappear in different forms in the events of the 1960s, of which the most momentous was the launching of the IBM 360 family of computers (second chapter). The advent of the 360, and of the third generation computers in general, and the far-reaching consequences for both computer producers and computer users are the theme of the third and fourth chapters. The enormous problems encountered in programming these new machines created an awareness for the first time of the importance of programming, an awareness crystallised in the commodification of software (the theme of the fifth chapter).

That the situation was serious, that software was in crisis, that there was a need to give more solid foundations to software development, was openly recognised for the first time in two conferences sponsored by NATO in 1968 and 1969 (the sixth chapter). However, even in the moment of greatest harmony, when it was unanimously agreed that something had to be done to change the way that computers were programmed, the conflicts reappeared: these conflicts continued to dominate both the theory and practice of programming in the years that followed, and continue to do so still (the seventh chapter).

The examination of these themes and turning-points provides neither a definitive history nor a definitive theory of the forces shaping software development, but - to the extent that it is successful - it traces out the important connecting threads between apparently separate happenings. If the history of software is an uncharted territory, the aim of the thesis is to pick out the rivers and the mountains, the significant contours that can provide a basis for understanding what goes on in this most important territory, for what happens in that territory could, literally, decide the fate of the world.

Chapter 1

A Contract with the Devil

What is software? The answer is not a simple one. The nature of software is changing and elusive; it has a history. In order to understand software now, it is necessary to look at what it was, at its historically changing nature.

The history of software can be seen in terms of a dynamic tension between two poles: hardware development and user demand. Software is in the middle. It is the set of instructions given by computer users to the machine in order to obtain the solution to a particular problem. Inevitably software is shaped at one end by the capacity and structures of the machine and, at the other, by the formulation of the problem to which the user requires a solution.

The history of software has to be located within the context of the development of these two constraints. This does not mean that the development of hardware and the development of user requirements are of equal importance at every stage of software development. On the contrary: the history of software is the history of its separation from the machine, its progressive removal to a higher and higher level, at which the formulation of the problem to be solved becomes more and more the focus of attention.

In the first digital computers of the 1940s, there was no clear distinction between hardware and software. In the earliest machines, attention was devoted

almost exclusively to the construction of the machine itself: it was at first assumed that programming the machine would present little difficulty.

Programming itself was done by physically rewiring the machine each time a new program was to be run. The way in which a calculation was programmed depended very much on the structure of the particular machine. Thus, for example, in the ENIAC, one of the first electronic computers, the decimal structure of the machine led to a system of performing subtractions through addition (Goldstine 1972, 159).

At this stage, there was also no clear division between producer and user: machines were built with a specific purpose in mind and the prospective users were closely involved in the design and construction process. The computers were still one-of-a-kind machines used by government agencies, mainly for military purposes.

With the transition to First Generation computers, usually dated from about 1951/52 to the introduction of transistorised computers in 1958/59, software at first remained closely tied to the hardware. Computers at this stage were very large (having vacuum tubes as their basic component), more or less immobile and very expensive. By present standards, they were very slow, had a very limited memory capacity and broke down very frequently.

Computers were now being produced commercially: the first Remington Rand UNIVAC was delivered to the Bureau of the Census in 1951 (at a purchase price of

about $1 million), and the first installations of the first IBM computer, the IBM 701, were made in the spring of 1953. However, this does not mean that they were being produced in large quantities or for an open market. Around 1950 only a few customers - such as the military, the US Weather Bureau, intelligence agencies, defence contractors (especially aeroplane manufacturers), the Atomic Energy Commission and its subcontractors, and the Bureau of the Census - were considered to have the sort of computational needs which would justify expenditure on a computer. When IBM first announced its 701 machine (initially called the Defense Calculator), it received 30 letters of intent from prospective customers (defence and related agencies and companies), a number subsequently reduced to six when IBM announced a rise in the proposed price from $8,000 to $15,000 per month. The first private firm to acquire a computer (a UNIVAC) for non-government related purposes was General Electric in 1954. (Fisher et al. 1983, 8; Moreau 1984, 48).

In the early years of the First Generation, then, although production was being carried out by commercial companies rather than universities or research institutes, the relation between production and use was still a very close one.

The relation between hardware and software was similarly close. The machines no longer needed to be programmed by rewiring, but the form in which the programs were written was still very closely tied to the structure of the machine.The early machines could be programmed only in machine language, in the ones and zeroes which correspond to the switches of the machine itself. For example, a very simple instruction might take the form:

001110 000000 000000 000000 100000

meaning "Put into the accumulator register the contents of register 32" (Moreau 1984, 155). There was a very close correspondence between the structure of the program and the structure of the machine itself. Consequently, programmers required to know every detail of the structure and working of the machine they were programming and inevitably the focus in programming was on the formulation of the problem to fit the structure of the machine; the logic of the program was totally shaped by the structure of the machine.

The use of machine language also meant that the formulation of even the simplest problem became a task requiring close attention to every detail and that even a momentary distraction could easily lead to a mistake, such as writing a 0 instead of a 1 at some point in a long sequence. Programming was closely tied intellectually and physically to the machine itself. Just as the history of software is the history of the separation of hardware and software, so it is also the history of the physical and intellectual separation of the people programming the machine from the machine being programmed.

The course of the 1950s saw a growing separation between producer and user, between hardware and software, between programmers and the computer.

The development of the magnetic ferrite core memory, first marketed in 1954 and probably the most important advance in hardware technology in the 1950s, meant that the computers of the late 1950s were much more powerful and much more reliable. The increased capacity and reliability, and falling costs, of

the computers made them a more viable commercial commodity. Commercialisation advanced quickly during the 1950's. IBM's second computer, the smaller and cheaper 650, first delivered in 1954, turned out to be a great commercial success and over 1000 models were eventually produced and delivered, as compared with an original planned production of about fifty (Rosen 1969, 19).

During the 1950's IBM gradually built up a position of strength in the computer industry. It was originally very hesitant in making a commitment to the production of the new machines: it has been estimated that in 1951 Remington Rand had a five-year lead over IBM in the field. But with the help of Remington Rand's mistakes and particularly with the winning of the important SAGE contract in 1952/53, IBM strengthened its position considerably. SAGE (Semi-Automatic Ground Environment) was a computer-based air defence system commissioned by the US Air Force and designed to give early warning of Soviet attack. It was an enormous undertaking which considerably strengthened IBM's position on the market. During the 1950's more than half of IBM's domestic computer revenues came from its work on SAGE and a B-52 programme undertaken during the Korean War (Fisher et al. 1983, 30).

The commercialisation of computer production meant an increasing separation between producer and user. Where producer and user had worked together in the design of the earliest computers, this relation was now established more and more through the market. The relation between producer and user was becoming more distant, more anonymous.

At the same time as the separation between production and use developed, there developed also a growing separation between the machine and the programming of the machine. To some extent these two processes were related: as the user became further and further removed from the production and design of the machine, the instructions required to use the machine acquired a distinct identity.

As the commercialisation of computers grew, and as programming came to be seen as a distinct activity, there emerged also a group of people who specialised in programming the new machines and who gradually became known as programmers. Edsger Dijkstra, who was later to exercise an important influence on the development of computer science, recalls that when he got married in 1957, "Dutch marriage rites require you to state your profession and I stated that I was a programmer. But the municipal authorities of the town of Amsterdam did not accept it on the grounds that there was no such profession. And, believe it or not, but under the heading 'profession' my marriage record shows the ridiculous entry 'theoretical physicist'!" (Dijkstra 1978, 10)

The clear separation of programming from the machine meant developing a way of writing programs other than in the 0s and 1s which corresponded to the switches of the particular computer. There were two elements to the problem, two difficulties which had to confronted.

The first of these elements was the definition of an *algorithmic language*, sometimes referred to as an *algebraic language*. Programming requires the

concise formulation of instructions to the machine to perform a very large number of repetitious calculations. There must, for example, be a means of expressing concisely and precisely the instruction that a certain operation should be performed and repeated using different variables until certain conditions are satisfied. In other words, the instruction must describe an algorithmic procedure: "an algorithm is an *iterative* procedure, completed in a finite number of steps, for the purpose of solving a given problem. Note that 'iterative' is defined as 'step-by-step' where each step is dependent upon the preceding step - in other words, where the result of each process is the initial condition for the next" (Hilton 1963, 138). Or, to put it another way: "we say that a procedure for solving a problem is algorithmic when it can be expressed as a sequence of statements of operations to be performed and when no knowledge or intelligence is required beyond what is strictly necessary in order to perform these operations" (Moreau 1984, 5).

Although mathematics had developed algorithms from the days of ancient Mesopotamia, an adequate notation for expressing them concisely had never been developed. Mathematicians had developed powerful notations for static functional relations, but they had never invented a good notation for dynamic processes (Knuth and Trabb Pardo 1980, 200).

However, the definition of the language was not sufficient on its own. There must also be the second element, a means of translating the programming language into machine language, the 0s and 1s which can be processed by the computer. It is necessary to devise a special program, called a *compiler*, to translate the

program, the *source code* written in the programming language, into *object code* which can be processed by the machine.

The separation of programming from the machine did not follow a simple straight line. There were different emphases on the one or the other of the two elements mentioned, different directions taken. These different directions reflected both the force of material circumstances and different ideas about the nature of computers.

The earliest attempt to develop a programming language was the work done in Germany by Konrad Zuse towards the end of the Second World War. Zuse had been building computers in Germany since 1936. When his computers were destroyed by Allied bombs, he fled from the advancing Allied troops with what remained of his machines and installed himself in a small Alpine village near the Austrian border. There he worked on the formulation of a programming language, the Plankalkül. His work was necessarily theoretical in its focus:

> "It was unthinkable to continue practical work on the equipment; my small group of twelve co-workers disbanded. But it was now a satisfactory time to pursue theoretical studies. The Z4 Computer which had been rescued could hardly be made to run, and no especially algorithmic language was really necessary to program it anyway. Thus the PK (Plankalkül) arose purely as a piece of desk-work, without regard to whether or not machines suitable for PK's programs would be available in the foreseeable future" (Zuse 1972, 6, quoted by Knuth and Trabb Pardo 1980, 202).

Zuse also wanted to concentrate on theoretical work for another reason:

"The first computers in Germany were exclusively designed for numerical calculations, and the limited financial basis and short time available for the construction did not allow any special features. Besides

> that, the users of the computer did not see any necessity for a more sophisticated logical design in those days. But on paper there was no limit on further ideas, even during the war" (Zuse 1980, 616).

Zuse's emphasis was on the logical design of the computing process. From the earliest days of his involvement with computers, he had been interested in extending their use beyond numerical calculation to the handling of symbols, or logical values (Bauer 1980, 513).

His emphasis on logical design had led him to reject the idea of a stored program computer (in which instructions and data are kept in the same storage):

> "Since programs are like numbers built from bit sequences, it was obvious to store programs, too. Then one can perform conditional jumps and can calculate addresses...[The] feedback from the result of the calculation to the program flow can be established symbolically by a single wire. I hesitated to do this step" (Zuse 1970, 99, quoted by Bauer 1980, 518).

He rejected the stored program because he saw not only the increased efficiency it would bring, but also the potential for confusion: it "could influence the whole computer development in a very efficient but also very dangerous way" (Zuse 1980, 616). He feared that it would open Pandora's box (Bauer 1980, 518): implementing the stored program computer "could mean making a contract with the devil. Therefore I hesitated to do so, being unable to overlook all the consequences, the good as well as the bad" (Zuse 1980, 616). His "own designs for future machines on paper were much more structured with instructions stored independently and special units for the handling of addresses and subroutines nested in several levels" (Zuse 1980, 616).

The aim of the Plankalkül was, in the opening words of Zuse's description, "to provide a purely formal description of any computational procedure" (quoted in Knuth and Trabb Pardo 1980, 203). The Plankalkül was a highly developed language, with many concepts that were adopted only much later, particularly its notion of hierarchically structured data. However, it was never implemented and was not published in full until 1972. Nevertheless, its influence, direct and indirect did much to establish a distinct European tradition in programming.

Zuse's surviving Z4 computer eventually found its way to the Institute of Applied Mathematics in Zurich, where it was programmed by Heinz Rutishauser. Rutishauser certainly knew about the Plankalkül, and in 1952 he wrote a paper in which he described an algorithmic language, together with complete flow-charts for two compilers for that language. The most important feature of his language was the introduction of a nonsequential control structure which bracketed off a sequence of statements introduced by **für** and ending with **ende.** However, Rutishauser's language was not implemented, and the idea of the **für...ende** bracket did not acquire practical significance until the end of the 1950s. Similar work was also done, althought not in conjuction with Rutishauser by an Italian postgraduate in Zurich, Corrado Böhm, but his language too remained unimplemented.

In the United States the development of programming in the early years followed different lines. While the Europeans, through choice or force of circumstance, concentrated on the theoretical elaboration of algorithmic languages, the Americans tended to focus on the second element, the construction of

a means of translating symbols into machine code.

From the earliest post-war days, the US development took place within the context of the stored program computer, the idea which Zuse had explicitly rejected in Germany. Storing the program in the machine meant that both the instructions and the data which were the object of the instructions were stored together in the computer's "memory". Before the stored program, the instructions to the machine were read directly from perforated paper tape or a deck of punched cards placed in the input unit; if some part of a program had to be repeated in the course of a calculation, the corresponding part of the tape or group of cards had to be reinserted by hand. Storing the instructions in the machine itself speeded up the process of calculation. Transfers between the memory and the calculating unit now involved only the passage of electric impulses rather than the mechanical reading of a card or tape. Moreover, "because the instructions could now be treated as data, it became possible to repeat a part of a program several times, operating each time on data from different parts of the memory, so that there was no longer any need to reinput the program. To indicate the new data to be used, it was only necessary to change the number in each elementary instruction giving the address of the memory location where its data were to be found" (Moreau 1984, 37). A further advantage was that it became possible to skip automatically over specified parts of the program. All of these points made the computing process far more rapid, but, as Zuse had feared, storing instructions and data in the same memory also made the control of the process of execution of a program far more difficult. For the moment, however, attention was focused on the more immediate problem of making it easier to feed instructions into the computer.

The first step in the separation of programming from the machine was the replacement of the 0s and 1s of the machine language by symbols for the convenience of the person programming the machine. Goldstine and von Neumann, two of the leading figures in the early development of computers in the United States, proposed such a symbolic notation as early as 1947. It was their practice to simplify the writing of programs by replacing a statement such as:

00000010101111001010

meaning "clear the accumulator and add number stored in location 10 into it", by a symbolic abbreviation:

10 c A a

These symbols would then have to be translated either by the person writing the program or by someone else into machine language (Goldstine1972, 334).

Another contribution made by Goldstine and von Neumann was the proposal (also in 1947) of a system of representing algorithms pictorially through a series of boxes joined by arrows, called a flow diagram. Goldstine and von Neumann's approach to programming differed significantly from Zuse's, for example in their emphasis on numerical calculation rather than on data structures. Their ideas proved very influential, far more influential than Zuse's, partly because they were obviously in a more powerful position than Zuse had been. Although their work was not published, it "was beautifully 'varityped' and distributed in quantity to the vast majority of people involved with computers at that time" (Knuth and Trabb Pardo 1980, 208). This was of considerable importance at a period when there was very little communication between people working in this area.

The next step in the separation of programming from the machine was to develop the idea of symbolic representation further by providing, from 1951 onwards, for these symbolic instructions to be translated automatically into machine language. This was done by writing the special programs called compilers that would translate the symbols written by the programmer (the source program, written in the symbolic or *assembly* language) into machine language (the object program), which could then be processed by the machine. However, the structure of the instructions remained unchanged: the programmer was still required to produce a sequence of instructions very closely related to the structure of the machine. As a result, the structure of the machine still played a central role in the task of programming. The use of assembly language gained widespread acceptance in the United States and to some extent in Britain, but never became widely established in Continental Europe (Rutishauser 1967, quoted by Naur 1981, 93).

One problem in going beyond assembly language was the question of syntax. It was necessary that computers should be able to make simple syntactic analyses. This can be seen if one takes the example of a very simple expression (Moreau 1984, 159):

A=B+(C*D)

In order to evaluate this, one must first multiply C and D and then add this to B; the sum is then placed in the memory in the location corresponding to A. This can only be done after a syntactic analysis has shown the order in which the operations are to be performed. In the early 1950s, many specialists believed that it would be impossible for a computer to perform even a simple syntactic

analysis at adequate speed, or in other words that the process of translating such an expression would be too slow to be of any use (Moreau 1984, 159).

The first language that could be called a "higher level" language, in the sense that the program did not simply mirror the structure of the machine, to be implemented on a computer was a system devised by Laning and Zierler for the WHIRLWIND computer at MIT in 1953. The principal significance of this language lay in the fact that it showed that it was possible to overcome the problem of syntactic analysis. The immediate practical impact of the language was limited, partly because it was designed for a specific computer, but also because it did not make very efficient use of the machine. Laning and Zierler reported a "reduction of computing speed of the order of ten to one from an efficient computer program" (Knuth and Trabb Pardo 1980, 239). This was important because, as Laning later recalled: "This was in the days when machine time was king and people-time was worthless" (Knuth and Trabb Pardo 1980, 239).

The question of efficiency and the way that efficiency was defined was crucial in shaping the course taken by the separation of programming from hardware. Efficiency, from the early days of computing, when the machines had very restricted capacity, was defined in terms of the most efficient use possible of the machine, in terms of speed and memory capacity. Inevitably, the process of translating a programming language into machine code involved a loss of efficiency as defined in these terms. This was so for two reasons. Firstly, the compiler required to translate the source program into machine language took up space in the computer; and secondly, the object code produced by the compiler was unlikely

to use the computer as efficiently as would a machine-language program written by a good programmer. Consequently, although the use of programming languages facilitated the task of programming, it also resulted in a loss in the speed of the computer.

The breakthrough came with the development of FORTRAN. Late in 1953, John Backus of IBM managed to convince the IBM directors that, using the increased memory capacity and new capabilities of the IBM 704, which was to be announced in May 1954, it would be possible to devise a programming language with the necessary syntactic power to make it attractive to users. Besides pointing to the capabilities of the new machine, which, by building floating point arithmetic, which had previously been done through programming subroutines, into the hardware, created the possibility of simplifying programming considerably, Backus based his argument on the economics of programming:

> "FORTRAN did not really grow out of some brainstorm about the beauty of programming in mathematical notation; instead, it began with recognition of a basic problem of economics: programming and debugging costs already exceeded the cost of running a program, and as computers became cheaper this imbalance would become more and more intolerable. This prosaic economic insight, plus experience with the drudgery of coding, plus an unusually lazy nature led to my continuing interest in making programming easier" (Backus 1980, 131).

The economic need for a system like FORTRAN "was one reason why IBM...provided for our constantly expanding needs over the next five years without ever asking us to project or justify those needs in a formal budget" (Backus 1981, 27).

Although the work of the project group was not subject to direct economic

pressures from IBM, the commercial environment in which they were operating had an influence on the way that they approached the question of developing the new language. Backus stressed the importance of proving the efficiency of the language. "In view of the widespread skepticism about the possibility of producing efficient programs with an automatic programming system..., we were convinced that the kind of system we had in mind would be widely used only if we could demonstrate that it would produce programs almost as efficient as hand coded ones and do so on virtually every job" (Backus 1981, 28). Although the motivation for the language is based on a recognition of the importance of programmer time, and therefore implicitly on a change in the definition of efficiency, it was still essential to show that the language would make efficient use of the machine.

As a result: "This belief caused us to regard the design of the translator as the real challenge, not the simple task of designing the language. Our belief in the simplicity of language design was partly confirmed by the relative ease with which similar languages had been independently developed by Rutishauser (1952), Laning and Zierler (1954), and ourselves; whereas we were alone in seeking to produce really efficient object programs" (Backus 1981, 29).

The group's approach to the design of the language itself was a fairly pragmatic one: "As far as we were aware, we simply made up the language as we went along. We did not regard language design as a difficult problem, merely a simple prelude to the real problem: designing a compiler which could produce efficient programs...In our naive unawareness of language design problems - of course we knew nothing of many issues which were later thought to be

important... - it seemed to us that once one had the notions of the assignment statements, the subscripted variable and the DO statement in hand (and these were among our earliest ideas), then the remaining problems of language design were trivial: either their solution was thrust upon one by the need to provide some machine facility such as reading input, or by some programming task which could not be done with existing structures" (Backus 1981, 30, 32).

The definition of the new language, to be known as FORTRAN (FORmula TRANslator), was completed quickly, by November 1954, but the completion of the compiler took another two-and-a-half years, the first implementation being made available in the spring of 1957.

The whole process was very much related to the production and marketing of the IBM 704, which was announced in May 1954. The language was designed for use with the 704, and independence from the machine was not seen as an important goal (Knuth and Trabb Pardo 1980, 241). After the preliminary definition of the language, Backus and his colleagues went to a number of cities to give talks about the proposed language to groups of IBM customers who had ordered a 704 (Backus 1981, 32), and when the compiler was finally ready, it was delivered to 704 customers. The fact that FORTRAN was produced within IBM meant not just a more implementation-oriented approach from the beginning; it also meant that, when completed, the new language had the full weight of IBM's support behind it.

FORTRAN proved an extremely successful and influential language. By the end

of the 1950s most computer manufacturers had adopted it and offered a FORTRAN compiler with their machines. By making programming easier and less tedious, FORTRAN gave an enormous boost both to IBM and to the computer industry in general.

The success of FORTRAN coincided with the advent of the second generation of computers and the two developments fed off each other in shaping computer development. The second generation is generally associated with the replacement of vacuum tubes by the transistor as the basic component of computer technology. Transistors had in fact already been used in the construction of computers as early as 1950 (in the Standards Eastern Automatic Computer, SEAC, built for the US National Bureau of Standards and used mainly for meteorology (Moreau 1984, 63)), but the second generation is generally taken as dating from 1958/59, when several companies announced their first wholly transistorised machines, and as continuing until 1963/64, when the first machines using integrated circuits made their appearance. When the transistorised machines were announced, the various manufacturers felt compelled to provide FORTRAN compilers with them. Rosen recalls:

> "At the time I was in charge of Programming systems for the new model 2000 computers that Philco was preparing to market. An Algebraic compiler was an absolute necessity, and there was never really any serious doubt that the language had to be FORTRAN. The very first sales contracts for the 2000 specified that the computer had to be equipped with a compiler that would accept 704 FORTRAN source decks essentially without charge. Other manufacturers, Honeywell, Control Data, Bendix, faced with the same problems, came to the same conclusion. Without any formal recognition, in spite of the attitude of the professional committees, FORTRAN became the standard scientific computing language" (Rosen 1968,10).

The improvements in speed and reliability of the transistorised machines, compared with the vacuum tube computers, combined with FORTRAN to provide the basis for a considerable further expansion of computer production and use during this period.

As the production and use of computers expanded, the demand for programmers grew too. It was partly in response to these changes that the role of the universities in relation to computers changed. The education of programmers was recognised as a distinct activity by universities and computer science began to emerge as a discipline. This led to the development of further splits in addition to the ones already mentioned (production/use, hardware/software, machine/ programmers). On the one hand, computer science was acquiring an autonomy from other disciplines like engineering and mathematics, and, on the other, this development led to an increasing separation of theory and practice (or of universities and industry).

The year in which the first commercial transistorised computers emerged also saw the publication of another important programming language, ALGOL.

ALGOL grew out of an initiative by the German GAMM (Gesellschaft für angewandte Mathematik und Mechanik, Society for Applied Mathematics and Mechanics). In October 1955, the GAMM organised an international symposium on automatic programming, at which algorithmic languages and their translation into machine code were discussed. Several speakers at the meeting, including Rutishauser, the author of one of the earliest programming languages, stressed

the need for focusing attention on unification, on creating "one universal, machine-independent algorithmic language to be used by all", rather than devising a number of competing languages (Rutishauser 1967, quoted in Naur 1981, 93).

After the conference, a special Subcommittee for Programming Languages was set up in order to design such a language. In the autumn of 1957, the subcommittee decided that, rather than simply presenting its own language, it should try to achieve worldwide unification: this clearly required United States participation. Consequently, the chairman of the subcommittee, F.L. Bauer, wrote to J.W. Carr, the president of the American ACM (Association for Computing Machinery), and proposed a joint conference to establish a common algorithmic language (Rutishauser 1967, quoted in Naur 1981, 93-94). Bauer's letter to Carr stressed the importance of seeking a common language.

> "We consider it a misfortune that at this time several different languages exist, but none of these languages appears to overshadow the other enough so that this would offer a reason for selecting it. We would like to avoid increasing this bad situation by setting up in Middle Europe one more such language" (Letter of 19th October 1957, reproduced in Bemer, 1969, p.160).

The ACM welcomed the GAMM proposal. Indeed, a similar initiative was already under way within the ACM. In May 1957 a conference in Los Angeles, attended by representatives of the users' organisations, USE, SHARE and DUO, and of the ACM, had declared that a single, universal language would be very desirable and had recommended that the ACM should appoint a committee to study ways of achieving this end (Sammet 1969, 173). Even before receiving the letter from Bauer, the ACM had established a committee to study the matter of creating a universal programming language, under the chairmanship of Alan Perlis of

Carnegie Tech (Perlis 1981, 77). The committee consisted of representatives of the major computer manufacturers and of several universities and research agencies that had done work on compilers. Since Bauer's letter arrived before the first meeting of the committee, it was decided that this committee should coordinate efforts with the Europeans.

The committee met several times, but "without any very great sense of urgency" (Rosen 1967, 9). In the first meeting, in January 1958, the committee agreed that the language should be at the level of FORTRAN, an algebraic language (Perlis 1981, 77). The second meeting was addressed by Bauer, who pressed for a date to be set for the international meeting.

The ACM committee felt that the question of a new language could be tackled at two distinct levels. At one level, there was general agreement on what the basic elements of the language should be; beyond that, however, there was considerable disagreement about how far such a language should go and about how to specify many of the more advanced concepts. It was decided to set up two subcommittees, one of which would deal with the definition of a language which included those features on which there was widespread agreement, and the other of which would work towards some future specification of a language that would incorprate the most advanced thinking of the time. It was the short-range committee which set up the meeting with the representatives of GAMM.

The new language initiative did not have universal support on the American side, however. Although SHARE, the IBM users' organisation (formed by the users

of the 704), had supported the recommendation at the conference in May 1957 for the establishment of a universal programming language, the conference had not specified whether the universal language should be at the machine or at the algebraic language level. When the GAMM proposal specified that the new language should be algorithmic, the response of the SHARE chairman, Francis Wagner, was negative. On 22nd November, he wrote to the SHARE Executive Board:

> "I believe very, very firmly that the establishment of a universal algebraic language for programmers to code in is a relatively trivial project. I do not feel that the existence of several such 'higher order' languages would particularly hurt the computing profession. (In fact, I think it necessary that there be many, each adapted to its own field)" (quoted in Bemer, 1969, p.162).

A couple of weeks later, on 9th December 1957, Wagner wrote to the President of the ACM, accepting the selection of a US delegation to meet the Europeans, but objecting to its composition:

> "We think...you are making a mistake in loading it so heavily with compiler designers and university people" (quoted in Bemer 1969, p. 162).

Wagner expanded the reasons for his opposition to the project in another letter to Carr just eleven days later:

> "It seems to me a shame to waste all this time and effort on just another algebraic higher order language even though it purports to be 'universal'. It seems to me that such an assumption is almost a contradiction in terms...the most useful manner of exploiting the computers of the future will be to encourage every discipline to develop a higher order programmer language which most ideally suits its subject matter. Thus there should be programmer languages for aerodynamicists, petroleum engineers, nuclear physicists, medical diagnosticians, clothing manufacturers, etc. Even if this were not sound,...I maintain that human nature will make it inevitable. Thus an algebraic programmer language can never be universal, for lack of universal acceptance" (quoted in Bemer 1969, 163).

SHARE's lack of enthusiasm for the new language was undoubtedly also connected with a vested interest in the established position of FORTRAN. When Bauer stated in his letter to Carr that none of the existing languages "appears to overshadow the other", this may have been true from an academic perspective, but it was clearly not true from the perspective of commercial programming practice. FORTRAN was already well established among the users of IBM's 704 represented by SHARE: their programs were written and their programmers were trained in FORTRAN. However, although FORTRAN clearly did overshadow the existing languages, it would not have been possible to adopt it as the universal language: "Today FORTRAN is the property of the computing world, but in 1957 it was an IBM creation and closely tied to the use of IBM hardware. For these reasons, FORTRAN was unacceptable as a universal language" (Perlis 1981, 77). Another member of the ACM committee, Rosen, who represented Philco on the committee, makes the same point in his account: "There was a feeling on the part of a number of persons highly placed in the ACM that FORTRAN represented part of the IBM empire, and that any enhancement of the status of FORTRAN by accepting it as the basis of an international standard would also enhance IBM's monopoly in the large scale scientific computer field" (Rosen 1967, 10).

The fact that FORTRAN was closely identified with IBM does not mean that IBM itself was unequivocally opposed to the establishment of a new language: indeed one of the most active members of the ACM committee was John Backus, the IBM representative (Rosen 1967, 9). The fact that FORTRAN had been designed for a specific machine meant that it was not necessarily the best language for the further development of IBM machines. IBM probably had a less clearly defined

interest than the users of IBM computers in maintaining the established position of FORTRAN.

The identification of FORTRAN with IBM was also an issue for the Europeans, although this was never mentioned at the time. Perlis, commenting more than twenty years later on the difference in perspective between the American and the European groups, says:

> "The American view of the promise of ALGOL contained the hope that some new view of a programming language would emerge that none of us ... had thought of. For the Europeans the issue was much more serious and pragmatic. It was necessary for the Europeans that a language be created that the numerical analyst could use, that would run on their computers and enable them to avoid, at least for a small period of time, being inundated by IBM" (Perlis 1981, 141).

McCarthy, another member of the ALGOL group, also saw the ALGOL initiative as having a specific target:

> "I really thought that one of the goals of ALGOL was to knock off FORTRAN. I believe that many people at the time considered that a goal" (Wexelblat 1981, 167).

It was against this background of conflicting interests that the meeting took place in Zurich at the end of May 1958, with the participation of four Europeans (Bauer, Bottenbruch, Rutishauser, Samelson) and four Americans (Backus, Katz, Perlis and Wegstein). Before Zurich there had been relatively little communication between European and American computer scientists: Grace Murray Hopper recalls that in the 1950s "I had absolutely no idea of what Rutishauser, Zuse, or anyone else was doing. That word had not come over. The only other country that we knew of that was doing the work was Wilkes in England.

The other information had not come across. There was little communication, and I think no real communication with Germany until the time of ALGOL, until our first ALGOL group went over there to work with them" (Hopper 1981, 22). The European and American traditions in language design also differed in many respects - Perlis mentions the Europeans' "passion for orderliness" (Perlis 1981, 78) - and the interests on the two sides were not identical. Despite all this, the meeting made rapid progress.

This progress was helped by a decision to distinguish between three different representations of the language. Perlis recalls that "after two days of probing attitudes and suggestions, the meeting came to a complete deadlock with one European member pounding on the table and declaring: 'No! I will never use a period as a decimal point.' Naturally the Americans considered the use of comma as decimal point to be beneath ridicule. That evening Wegstein visited the opposing camps and proposed defining the three levels of language. The proposal was immediately adopted and progress resumed" (Perlis 1981, 80). Wegstein's proposal was that there should be a reference language, which would be an abstract representation of the language, a publication language and a hardware language, and that the meeting should focus attention on the reference language, leaving the publication and the hardware languages to be derived subsequently from the reference language. This allowed the meeting to concentrate on the abstract definition of the language without worrying about differences between different character sets (Perlis 1981, 80; Naur 1981, 95)

The result of the meeting was the publication of a report (prepared by Perlis

and Samelson) describing a new language which they initially named IAL (International Algebraic Language), but which subsequently became known as ALGOL (Algorithmic Language). The name ALGOL was suggested at a meeting in the autumn of 1958 "and the ones among us who had studied astronomy immediately saw the pun" (Bauer 1980, 521). "ALGOL, named by the Arabs, is a fixed star in the constellation Perseus. It was among the first of stars noted for its periodic variation in brightness, due to eclipse by its dark satellite. Its name, in Arabic, signifies 'The Demon'" (Granholm 1963, quoted by Bemer 1969, 206).

The report was received with a lot of interest on both sides of the Atlantic. In the United States, SHARE's attitude changed from its original hostility to one of enthusiastic acceptance once the report was published, especially after hearing the report of the IBM representative at the Zurich meeting, John Backus (Bemer 1969, p 164). As a result, SHARE established a special committee (first called the IAL Committee, later the ALGOL Committee) in September 1958 to promote the adoption of ALGOL as a SHARE standard.

In Europe, Peter Naur of Copenhagen established an ALGOL Bulletin as a forum for discussion of suggested amendments to the language. A number of conferences were held on the topic, of which the most important was the International Conference on Information Processing in Paris in June 1959. At this conference an important paper was given by John Backus on "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference", a paper which was to establish a standard form (BNF) for the syntax of programming languages. The conference also established a special Ad Hoc

Subcommittee (consisting of Dijkstra, Heise, Perlis and Samelson) to consider extensions of the ALGOL language.

All this discussion culminated in the holding of another international conference to define the language, held this time in Paris in January 1960, and with a slightly expanded membership of thirteen: seven Europeans (in addition to three of the Zurich members, Bauer, Rutishauser and Samelson, there were four newcomers: Naur from Denmark, van Wijngaarden from the Netherlands, Vauquois from France and Woodger from Great Britain); on the American side there were six participants (the original four, Backus, Katz, Perlis and Wegstein, together with Green and McCarthy; the seventh American participant, Turanski, was killed in a car accident shortly before the conference). The Paris meeting was successful and led to the definition of ALGOL 60, which differed in many respects from ALGOL 58. The report was edited by Peter Naur and described the new language in BNF.

From its origins, ALGOL was conceived as a universal, formal language, designed in abstraction from any particular machine. Consequently, its concepts were shaped not by machine structures but by the aim of structuring problems in the form most appropriate to their solution. The essence of its approach was the breaking down of complex problems into simpler units which could then be progressively refined. It was hoped that this would also make the language easier for users to read and understand.

The key feature of ALGOL is its block structure. Every ALGOL program is made

up of blocks, framed by the key words **begin** and **end.** Each block contains a declaration of the objects to be found inside the block, followed by the statements or instructions to be executed. Each block may contain other blocks within it, the declarations and statements within each block being valid only for that block. The block structure built upon the **für...ende** construction first introduced by Rutishauser in his language proposal of 1952, but the fully developed idea was an innovation which emerged in the discussions between the Zurich and the Paris meeting:

> "It's interesting to follow ... the tortuous step-by-step process by which blocks finally came into being. That concept, now so well understood by all students almost immediately, began with a set of declarations and ended in an understanding of how long a variable exists in such a block environment and was arrived at over the year 1959 in painfully slow steps, by people that are as bright as or brighter than anyone here... Men of the stature of Dijkstra, Naur, Samelson, Rutishauser, labored long and hard through the year of 1959 to come up with something that we now take so much for granted, and is such an obviously desirable programmatic concept. But it wasn't in those days" (Perlis, 1981, p.144).

The idea of the block structure can be explained with a very simple example suggested by Reynolds (1981, 1). Programming involves the breaking down of complex processes into a series of simple instructions which can be executed by the computer. One can imagine that the aim is to program a typical series of actions on a particular morning. For many people, the early morning passes in a blur of confusion, but the programming of a computer requires precision. On reflection, it might be possible to break down the confused morning rush into an orderly series of activities such as:

```
Eat breakfast;
Put on clothes;
Leave in car.
```

It is obviously possible to break each of these acts down into a sequence of more detailed acts, and to break those acts down into even more detailed acts:

Gross	*Detailed*	*More Detailed*
		Peel orange
	Eat orange	Eat bite
		Eat bite
		Put milk on cereal
		Put sugar on cereal
Eat breakfast	Eat cereal	Eat bite
		Eat bite
		Eat bite
		Eat bite
	Eat toast	Eat bite
		Eat bite
	Put on coat	
Put on clothes	Put on gloves	
	Put on shoes	
	Open garage door	
Leave in car	Start car	
	Get car out of garage	

In this example, the program of morning activities is divided into a series of blocks or compound statements which describe procedures or subroutines. At the most gross level, the program could be written as follows

```
begin
Eat breakfast;
Put on clothes;
Leave in car
end .
```

But these statements are still far too gross: they must be further refined into more detailed actions. This can be done by further dividing each block into a nested series of blocks until the whole problem is broken down into detailed instructions

which can be processed by the machine. In ALGOL each block is enclosed by the delimiters **begin, end.** Thus, defining the program at the next level of detail, we might write:

```
begin
    begin comment Eat breakfast;
    Eat orange;
    Eat cereal;
    Eat toast;
    end;
    begin comment Put on clothes;
    Put on coat;
    Put on gloves;
    Put on shoes;
    end;
    begin comment Leave in car
    Open garage door;
    Start car;
    Get car out of garage;
    end;
end .
```

Each block of statements is self-contained. The instruction "eat toast" is specific to the procedure "eat breakfast" and has no existence outside that procedure: in the world of ALGOL there is no possibility of eating toast while you start your car. An ALGOL program is a series of nested blocks, one enclosed inside another, rather like a set of Russian dolls. The outside block describes an environment (in this case eat breakfast; put on clothes; leave in car), which is progressively refined by each of the internal blocks. Each of the inner blocks is embedded in that environment and is accessible only by opening up the outer "dolls".

The nesting of blocks is reinforced by the technique of recursion. As the more detailed column of statements makes clear, one of the problems of programming a computer is the problem of describing repetitions concisely and defining them

clearly. It would be tiresome to write a program which listed every bite of toast to be taken, and in any case it would not be known in advance how many bites would be necessary to finish the piece of toast. It is necessary to describe the dynamic process of toast-eating in such a way as to ensure that all the toast is eaten.

In ALGOL it is possible to do this by means of recursion. In mathematics, recursion is the name given to the technique of defining a function or process in terms of itself. By a recursive subroutine is meant a subroutine which uses its own specifications in its text. For example, the factorial function of N, written N!, has the value N x (N-1) x (N-2) x ... 2 x 1 when N is a positive integer and 1! = 1; thus 4! = 4 x 3 x 2 x 1. This can be computed by a subroutine of the form IF N > 1 FACTORIAL N = N x FACTORIAL (N-1) where the FACTORIAL subroutine is called repeatedly until the number on which it operates has been reduced to 1. If this concept is applied to the Eat toast example, the program might be written:

```
procedure eat toast (a);
value (a); comment a is an amount of toast in bitefuls;
integer a;
begin
    procedure eat (a); value a; integer a;
    begin procedure bite; begin...end;
          if a > 0 then begin bite; eat (a-1); end
    end
    eat (a)
end
```

Recursion reinforces the concept of block structuring because, by allowing a subroutine to call itself, it facilitates the construction of programs as a series of self-contained blocks.

Recursion was highly controversial. This was partly because of the way in which it was introduced. Peter Naur, editor of the ALGOL 60 report, explained it as follows at a conference on the history of programming languages held nearly twenty years later (Naur 1981, 112-113):

> "One of the proposals of the American representatives to the ALGOL 60 Conference was to add the delimiter **recursive** to the language, to be used in the context **recursive function** or **recursive procedure** at the start of declarations (ALGOL 60 document 6). This proposal was rejected by a narrow margin. Then on about 1960 February 10, while the draft was being studied by the members of the committee, I had a telephone call from A. van Wijngaarden, speaking also for E.W. Dijkstra. They pointed to an important lack of definition in the draft report, namely, the meaning, if any, of an occurrence of a procedure identifier inside the body of the declaration other than in the left part of the assignment. They also made it clear that preventing recursive activations through rules of the description would be complicated because of the possibilities of indirect activations through other procedures and their parameters. They proposed to clarify the matter by adding a sentence to section 5.4.4: 'Any other occurrence of the procedure identifier within the procedure body denotes activation of the procedure'. I got charmed with the boldness and simplicity of this suggestion and decided to follow it in spite of the risk of subsequent trouble over the question."

The strength of the feelings aroused by Naur's decision is evident even twenty years later. Bauer comments:

> "Events described by Peter Naur ... (the Amsterdam plot on introducing recursivity) show clearly that Peter Naur had absorbed the Holy Ghost after the Paris meeting. It should be mentioned, however, that there was not only scepticism among the committee members, but also resignation that there was nothing one could do when the editor did arbitrarily change the outcome of the conference: it was to be swallowed for the sake of loyalty. These feelings, however, have not been sensed by the editor. Otherwise, I think too, he did a magnificent job" (Wexelblat 1981, 130).

The whole question of recursion was a contentious issue: "It has been remarked that 'if computers had existed in the Middle Ages, programmers would have been burned at the stake by other programmers for heresy' (Gill 1960,

180). It is almost certain that one of the main heresies would have been a belief (or disbelief) in recursion" (Barron 1968, 1).

Recursion was contentious, because it meant that ALGOL would be very difficult to implement within the existing computer architectures. This can best be explained by going back to FORTRAN and seeing how FORTRAN programs are structured.

FORTRAN is a non-recursive language and does not have a block structure. A FORTRAN program first describes the subroutines to be used in the program and allocates those subroutines to a fixed storage location in the computer's memory. The subroutines then constitute a sort of library, from which particular subroutines can be called as they are required. Thus, to return to the Eat breakfast example, the FORTRAN program might be as follows:

```
SUBROUTINE bite
...
RETURN
END

SUBROUTINE eat orange
CALL bite
CALL bite
CALL bite
RETURN
END

SUBROUTINE eat cereal
CALL bite
CALL bite
CALL bite
RETURN
END

SUBROUTINE eat toast
CALL bite
```

```
CALL bite
CALL bite
RETURN
END

CALL eat orange
CALL eat cereal
CALL eat toast
STOP
END
```

The program first establishes a library of subroutines (bite; eat orange; eat cereal; eat toast), allocating each to a specific part of the memory, and these can then be called as needed at any point during the execution of the program. The program *jumps* from one location in the memory to another as the different subroutines are required. There is a constant transfer of control during execution from the main program to the subroutines and back again. The execution of the program is shaped by the constant pursuit of efficiency, defined in terms of using the least possible storage. It is known in advance how much memory is required for the program and there is a close correspondence between the textual program and the structure of the memory. This makes FORTRAN relatively easy to compile.

This is not the case with ALGOL. In an ALGOL program it is not known in advance how much space will be taken up by the program. The amount of storage will vary enormously depending on how the program executes: recursion means that the subroutines are defined in terms of themselves, they are generated in the process of execution. There is no fixed correspondence between the textual structure of the program and the structure of the storage in the computer. In ALGOL, the storage allocation is dynamic. As a result, ALGOL is more difficult to compile and requires considerable overhead just for managing the allocation of storage.

FORTRAN and ALGOL can be contrasted by saying that FORTRAN looks towards the machine and is structured by its requirements, while ALGOL looks towards the problem and focuses on structuring the problem in a manner appropriate to its solution. FORTRAN programming can be seen as being rather like building up a stock of components which can then be put together (possibly by different programmers on different projects) to solve different problems. The flexibility this provides is useful in allowing the programmer to use various tricks to maximise the use of the available memory. It is rather like programming by cutting and pasting. In FORTRAN there would be no objection to eating your toast while you start your car: it would be not only be allowed, but strongly encouraged if this made the best use of the available resources. ALGOL is very different. Each problem is thought of separately, according to the nature of the problem itself, so there can be no question of making use of a stock of common components. This means, however, not only that ALGOL is more difficult to compile, but also that it involves a more radical break with the established practices of programming. In FORTRAN there is a clear continuity with machine language and assembly language, while ALGOL, which was much influenced by the ideas of Zuse and Rutishauser, requires a radical rethinking of programming. As a result, the problems of implementing ALGOL appeared more difficult than they actually were.

The question of implementation was central to the subsequent development of ALGOL. FORTRAN had been designed for a particular computer, the IBM 704, and the approach in its design was to focus on the problems of compiling. ALGOL, on the other hand, was a formal language in the sense that it was defined independently of any particular machine, and therefore in abstraction from the

process of translation into machine language. The aim was to produce a language which would be both elegant and have universal application. Perlis describes ALGOL as "an object of stunning beauty" (Perlis 1981, 88).

The emphasis on the formal definition of the language was reflected in the initial absence of practical facilities which would make the language immediately implementable, for example the lack of any input/output provisions and the initial absence of a compiler (Bemer 1969, 155). The absence of these facilities certainly meant that it was possible to concentrate on the structures of the language itself, without worrying about the question of translation but it also affected the acceptance of ALGOL.

The acceptance of ALGOL was of course affected by the reactions of the users' organisations and paricularly by the powerful IBM users' organisation, SHARE. By March 1961 SHARE had reverted to its initial position of hostility towards ALGOL. A report to the XVIth meeting of SHARE on 22nd-24th of March 1961 refers to the views which Wagner had expressed in 1957 and continues:

> "At three years distance these objections appear to have lost none of their basic soundness...The position of the recalcitrants was (and still is) simply that problem oriented language universality is neither possible nor desirable; that there should be individually tailored POLs for engineers, nuclear physicists, cost accountants, global strategists or what have you; and that the *real* problem is the drastic reduction of the manpower and elapsed time required to provide a capability of using a given POL with a given machine. Nevertheless, the Pollyannas had their way and ALGOL was born" (Minutes of SHARE XVI - Introduction to the UNCOL Committee Report, quoted in Bemer, 1969, 180-181).

At the same meeting of SHARE a resolution was moved, expressing SHARE's dissatisfaction with ALGOL and rescinding SHARE's endorsement and support of the

language (cited in Bemer 1969, 181 with the footnote: "The language of the original proposal was strongly intemperate and will not be reproduced here"). Decision on the resolution was postponed till the August meeting of SHARE, but in the meantime it was agreed that SHARE should rescind any request made to IBM to implement any ALGOL processors. When the resolution came to be voted on at SHARE XVII on 23rd August 1961, it was carried, with 65 installations in favour, 43 against and 15 abstaining (Bemer 1969, 189).

Why did SHARE's attitude change so drastically? A central issue was the question of implementation. Since ALGOL was defined independently of any particular machine, its implementation depended on the construction of a compiler or processor to adapt it to particular machines. Dijkstra and Zonneveld produced such a processor for use with their X1 computer in Amsterdam as early as the summer of 1960, but what was crucial for SHARE members was of course IBM's willingness to produce an ALGOL processor. IBM initially welcomed ALGOL and produced an experimental ALGOL compiler for the IBM 709 which was demonstrated at a SHARE IAL Committee meeting in May 1959, but were then reluctant to produce a fully operational processor. This was announced by A.L. Harmon to a SHARE meeting (SHARE XIV) in February 1960:

> "Since the development of the ALGOL language has not reached the point where it seems advisable to expend the manpower required for a full processor that SHARE seems to deserve, based upon the recommendations of the SHARE ALGOL Committee, IBM will not produce an official ALGOL processor at this time. However, IBM will continue to support the ALGOL efforts in the areas of language development, transition techniques and, of course, processor development" (Bemer 1969, 172).

Although IBM gave nominal support to ALGOL, they were not prepared to put very

much effort in to promoting its practical implementation.

Given the real problems of implementing ALGOL, IBM's and SHARE's lack of support was crucial in determining the fact that ALGOL was never widely adopted in commercial computing, particularly in the United States.

Although nobody seems to have suggested that ALGOL was not superior to FORTRAN, IBM had practical reasons for not wanting to give it full support. As a *Datamation* report at the time (December 1961) said:

> "Frankly acknowledged by many IBMers as a far superior processor to FORTRAN, ALGOL development is nevertheless far from practical in the eyes of IBM management. The problem is not one of money but largely the lack of experienced programmers to meet present commitments for over 35 FORTRAN processors as well as numerous other dialects promised to IBM customers. In addition, scrapping their present investment in FORTRAN would involve an enormous risk for IBM with no national or international body providing the needed authority for a definitive explanation of ALGOL.... The current status at IBM: considerable head-scratching" (quoted in Bemer 1969, 194).

IBM's attitude, however, was very much intertwined with that of its users. By 1960, SHARE members already had a substantial investment in FORTRAN: their programs were written in FORTRAN and their programmers were trained in FORTRAN. Not only that, but there was a shortage of programmers and computer use was expanding rapidly:

> "ALGOL came on the scene just when US users were engaged in a struggle to achieve production to justify all that expensive computer equipment they had ordered for purposes of advertising and keeping up with the Joneses. Thus most ALGOL processors were experimental at a time when FORTRAN was well into production" (Bemer 1969, 153).

ALGOL was defeated not by a conspiracy so much as by "the pragmatics of the

situation". As a commentator put it in 1963:

> "ALGOL...faced the *de facto* standard, FORTRAN, and the pragmatics of the situation were and are such that popularity is not in the cards for ALGOL - no computer user who has a large library of FORTRAN programs, or who has acces to the huge collective FORTRAN library, can justify the cost of conversion to a system which most are not even sure is superior..." (Jones 1963, quoted in Bemer 1969, 204).

Whatever the motives behind IBM's and SHARE's lack of support for ALGOL, the result was a situation in which ALGOL, although widely recognised as being superior to FORTRAN, was never widely adopted in the United States, particularly in commercial practice. In Europe (including the USSR and, more hesitantly, the UK), it was very widely adopted, particularly but not only in the universities.

It is clear that the history of ALGOL is extremely important for understanding subsequent software development, not only because its innovations (block structure and recursive procedures) foreshadowed many later developments, but also because it illustrates the growing divide between commercial practice (the widespread use of FORTRAN) and the views of most computer scientists. This tension is well expressed in a question from Aaron Finerman to Alan Perlis at a conference on the history of computer languages:

> "You once described FORTRAN as the language of the streets, graffiti upon the bathroom walls. You also indicate, and correctly so, that ALGOL's contributed much to the language and compiler development. Why then is FORTRAN so widely used in the real world, almost to the exclusion of all other languages except COBOL, while ALGOL is noted only for its contributions?" (Wexelblat 1981, 161).

When Perlis failed to give a very explicit response to the question, this prompted an expression of frustration from another participant, Kristen Nygaard:

"Programming languages are a very important part of a very important commerce, and if you are going to have the real full story, then you must also have the story of the commercial reactions and interests in the various languages. When we designed SIMULA, we felt quite strongly the resistance against ALGOL from IBM. And we felt that it was said that ALGOL was a long-haired language; it was such-and-such; it was a distinction between commercial and scientific, etc., etc., which was in line with certain commercial interests" (Wexelblat 1981, 166).

To have the "real full story", it is necessary, as Nygaard suggests, to move from the long-haired world of ALGOL to the short-haired world of commercial interests and to the greatest commercial project of all.

Chapter 2

The $5,000,000,000 Gamble

The advancing commercialisation of computers brought with it not only a proliferation of programming languages, but a proliferation of computer models as well. Up to the early sixties, the tendency of computer manufacturers had been to build machines for particular types of applications. As a result, computers were generally not compatible even with machines of the same line.

The situation at IBM was no different from the rest of the computer industry. By 1960, the incompability, diversity and complexity of IBM data processing products had become a very severe problem. IBM had fifteen different processors in the market, and about seven different lines of second generation computer systems and, on top of this, input-output equipment developed especially for each processor. All this meant that not just the architecture of the various systems differed but so too did their software. For the users this meant great inflexibility in responding to changes in their requirements. Once they were using a particular system, they were more or less stuck with it.

Within IBM the different divisions were working more or less independently of each other, sometimes competing with each other, each of them trying to present their own solutions to market problems. The Data Processing Group was divided into three product divisions: the General Products Division (Endicott), which had the line of small computers and whose plans were to upgrade those machines; the Data Systems Division (Poughkeepsie), in charge of large systems

and whose plans were to develop a modern product line with small machines to support it; and the Data Processing Division, whose responsibility was sales and services. Apart from those, there was also the World Trade Laboratories in England, which also had its own plans for development.

This situation resulted in a proliferation of products and redundant effort. Bob Evans of the Data Systems Division later recalled: "I often told the story of the customer who called to see an IBM salesman about an IBM machine and had three salesmen stuck in the door all shouting about their particular wares" (SPREAD Discussion 1983, 29).

Worries at IBM about the proliferation of products led in 1959 to the creation of the Group Staff, whose mission was, not to direct the product development of the corporation, but to try to give some coherence to all the developments carried out in the different divisions at IBM. Soon it became clear to people in the Group Staff that in order to have a coherent programme that addressed overall market needs, it was necesary to have close, intense interaction, with key people in the different divisions closeted together for a period of time (SPREAD Discussion 1983).

In the autumn of 1961 Don Spaulding, head of the Group Staff, formed a committee to develop a plan for IBM data processing products. The aim of the plan was to "encompass all stored-program processor developments and to provide development and product direction extending to 1970" (SPREAD Report, 1961). The committee was called SPREAD (acronym for Systems Programming, Research, Engineering, And Development). It was chaired by John W. Haanstra,

director of development of the General Products Division, and vice-chaired by Bob Evans, head of planning and development in the Data Systems Division (SPREAD Report, 1961). As the name indicated, the committee included people from all the different divisions at IBM, people with different skills and different backgrounds, including marketing specialists "to make certain that a bunch of engineers were not designing a Taj Mahal" (Evans in SPREAD Discussion 1983, 33).

The thirteen people who constituted the SPREAD commitee, went to live in a motel and worked together full time for sixty days. The progress was very slow, owing largely to the sharp competition among the different divisions, a competition that was highly encouraged by IBM executives, as part of their general approach to management, sometimes referred to as "management by contention". As a result, it was very difficult to reach technical decisions and to agree on a common view of the future product.

There were a number of points of controversy. The first related to the choice of the basic component of the new line of machines. The basic component technology of future computer development had already been under discussion within IBM. It had already been decided that it was essential to go beyond existing component technology, the SMS (Standard Modular System) technology used in STRETCH and a number of IBM products. SMS consisted in printed circuit cards and improved back panel wiring. The option of improving SMS had been dismissed, on the grounds that SMS technology had already been pushed to its limits in terms of cost, performance and reliability, and that competitors could easily go beyond it (Fisher et al 1983, 106).

Once it had been decided to go beyond the existing technology, there were two possibilities: the monolithic integrated circuit and the hybrid Solid Logic Technology (SLT).

Integrated circuits had been developed in the late 1950's, and soon microcircuit companies were "looking hungrily at the commercial computer business" (Siekman 1966, 122). The idea of a marriage between integrated circuits and computers was very attractive. Computers contained hundreds of thousands of separate electronic components that worked simply on the basis of on/off, and there were clear advantages to be gained from replacing all these components by integrated circuits. Integrated circuits seemed the ideal way of reducing assembly costs. Lowering the number of interconnections also meant reducing the number of possible failure points and thus improving both maintainability and reliability (Abelson and Hammond 1980, 19). Microelectronics promised to increase computer speed:

> "Computer switching speeds, the time required for a single operation, are measured in nanoseconds, a billionth of a second, or roughly the time required for light to travel one foot. Individual components are extremely fast, but delays incurred while signals travel from component to component and circuit to circuit limit over-all speed" (Siekman 1966, 122).

In the early 1960's, however, it was not clear that it was practicable to move to monolithic integrated circuits. They were very expensive and had until then been used only by the military in projects where the other characteristics of integrated circuits, such as size, were felt to be more important than considerations of cost.

The alternative was to base the new computers on Solid Logic Technology. The difference is that, whereas in monolithic integrated circuits all components and connections are created in one continuous production process (several transistors and resistors are fabricated on a single silicon chip and interconnected by fine conducting lines in the chip itself), hybrid integrated circuits are manufactured in several distinct steps, in which a number of silicon chips, each of them with a transistor fabricated on it, are mounted on a 0.5 inch square ceramic module in the surface of which conductors and resistors are silk-screened.

The choice of the computer component provoked a lot of discussion at IBM. Another committee, the Advanced Technology Study Committee, had concluded earlier in 1961 that, although monolithics met the performance requirements, they would be too expensive and their development would take too long. Consequently, that committee had recommended pursuing the development of the hybrid technology, SLT (Fisher et al 1983, 107). This view was supported by the SPREAD committee (SPREAD Discussion 1983, 33)

Several advantages were seen in going for SLT rather than monolithic. Firstly, because the production of hybrid circuits was done in several steps, it was possible to control flaws in the circuit: with monolithic circuits the production was done in one single step, so it was feared that there would be problems in maintaining quality in large-scale production. Secondly, it was felt that, because considerable research had already been done on the hybrid technology, successful completion of SLT was reasonably assured, whereas development of monolithics was highly risky and would require much more time (Pugh 1984, 193).

The fundamental issue which had to be decided by the SPREAD committee concerned the general shape of future computer development at IBM. Basically there were two different views. Evans, the vice-chairperson of the committee (from the Data Systems Division) argued that there should be one compatible line to cover the entire computer market, while Haanstra, the chairperson, who was Director of Development in the General Products Division, argued that the small-computer end of the market would be best served by enhancing the architecture of the very successful 1401 system. Eventually, the conflict among the proponents of the different systems was eased by the promotion of Haanstra away from the SPREAD comittee to be President of General Products Division, so that Evans became the effective chairperson of SPREAD (Fisher et al 1983; Pugh 1984; SPREAD Discussion 1983).

The idea of a single computer line to cover the whole market was very radical indeed. There were two elements to this proposal: that it should be a line of general purpose computers; and that there should be compatibility between the different computers of the line.

The idea of the general purpose computer was that there should be a single line of processors "to meet the needs of the commercial, scientific, communications and control markets" (SPREAD Report, 1961). In order to meet this recommendation there would have to be a "fundamental change in IBM's design emphasis" (Evans, quoted by Fisher et al 1983, 109).

Up to then, computers had been clearly either scientific or commercial in their emphasis. This was the case not just for IBM products but for other

computer manufacturers as well. Most users who wanted to do what traditionally had been considered both "scientific applications" and "business applications" often had to acquire two computers.

> "In those days computer people believed that the scientific user fed in a few long numbers and waited for the machine to perform a large number of complex calculations and provide the answer; he presumably would trade relatively slow input and output for enormous calculating speed. The commercial customer, on the other hand, fed great quantities of information into the computer, which then performed some fairly simple processing operations on each item" (Fishman 1982, 42).

However, by the end of the 1950s, the distinction between the scientific and commercial uses of computers had started to blur:

> "Customers themselves were not observing the lines between scientific and business machines in actual practice...Often the 'scientific side' of a user's operation needed the data-handling capabilities associated with business data processors, and the 'business side' needed the arithmetic and logic capabilities associated with scientific systems" (Evans, quoted by Fisher et al 1983, 109).

Evans argued that it was necessary to respond to users' needs by developing a general purpose computer: such a computer would bring great savings to the users:

> "One of the premises from the beginning was that there would be great savings to the users if we could combine in the single machine the ability to cover the full range of business applications and scientific applications as well. So our concept was a single machine that would be equally able in either of those areas" (Evans, quoted by Fisher et al 1983, p 110).

Users would get benefits from the broad range of applications of a general purpose computer in several ways: firstly it would simplify the difficult process of computer selection and acquisition; secondly it would enable them to gain efficiency by acquiring one large capacity rather than two small capacity

computers; and it would also reduce training and improve the efficiency of their data processing staff.

But it was not only the users who would make great savings. The development of a general purpose computer promised great returns for IBM as well:

> "The combination of business, scientific, and other applications in the same line also helped reduce IBM's costs. It enabled IBM to concentrate on a single machine type with fewer sets of program support and software, and with a single program of training and education for customers and IBM personnel" (Fisher et al 1983, p 111).

There were, however, arguments against the idea of a general purpose computer. The opponents of the idea felt that there should be a continuation of the development of scientific and business products, and that users, especially the military, would not necessarily want a general purpose computer, in which case IBM's competitors would step in and offer specialised systems.

The other aspect of the proposal for a single line of computers was that they should be mutually compatible. Users should be able to adjust their computer system according to their needs, without having to rewrite programs. This would mean developing compatible and modular software as well as hardware and peripheral equipment.

The idea of compatibility was not completely new. Some degree of upward compatibility had been achieved by IBM and other computer producers between a few machines of similar design but different power. However, Evans was proposing far more: the idea of providing compatibility between a whole family of computers and peripheral equipment, with full upward and downward

compatibility over a very wide performance range was, in Evans' words, "just a mile apart from the rest of the world" (quoted by Fisher et al 1983, 112).

The implications of compatibility were revolutionary. Compatibility would open up a whole range of possibilities in terms of communication between machines and the international transmission of data:

> "It [IBM] sees itself playing a critical role in a brand-new kind of international data communication, composed of computers that work and talk with each other. And in such a vision compatibility is a necessary element" (Wise 1966b, 142).

In addition to the potential opened up for communications, compatibility would present tremendous advantages for users in other respects. Compatibility would offer great flexibility in relation to the changing needs of users. It would allow them unlimited growth capability while remaining within the same environment. The promise of interchangeability of operating systems and application software was something that users had been calling for since the 1950s. Compatibility would also improve the efficiency of programming departments and save in the time and money spent on training.

The proposal was revolutionary in suggesting full upward and downward compatibility over a wide performance range. The basic idea was that users switching from a smaller to a larger, but also from a larger to a smaller model in the family should not have to rewrite their programs:

> "Their clients could grow from a smaller system to a larger system, or if the economic situations were such that they wanted to go to a lower system, they could do so without having to reinvest in their software. It also was an advantage if you had a multi-faceted organisation that had large computers and small computers, and some commonality of

applications that they wanted to use on both types of systems. It gave the client the advantage of not having to modify his software to do so" (Rooney, quoted by Fisher et al 1983, 113).

It was recognised by the SPREAD Committee that the success of compatibility would require a "technological change in the way computer systems were built ... in IBM" (Fisher et al 1983, 114). Central to this change were the concepts of microprogramming and the read-only control store.

A computer is said to be microprogrammed when it interprets the user's instructions (add, multiply, branch, etc) as calls on routines (the microprogram routines) stored in a microprogram memory. These routines are themselves sequences of the more elementary operations that are built into the machine hardware (Rosen 1968, 1444), normally in a read-only control store. Thus, the complex control logic of a central processor is replaced by the microprogram held in the read-only control store, which emulates the behaviour of the processor (COSERS 1980, 318).

Control stores would make it possible for computers with very different hardware to appear identical to the user, except in speed and price. Each computer of the series, no matter how its internal paths were structured would be able to respond to the same set of instructions (Pugh 1984, 200). Therefore, control stores were fundamental for the idea of compatibility between machines.

The idea of control stores was not a new one. It existed since the early fifties, but it was only with the development of suitable memory technology (ferrite core memories) and the recognition of a market demand for a line of compatible computers that the commercial use of control stores became practical (Pugh 1984, 202; COSERS 1980, 306).

The idea of compatibility went together with a concept of modularity. One of the design objectives of the new system would be to provide flexibility in order to cater for users' changing needs. This design objective was to be achieved by modularity. The system was to be structured as modules, so that users could adjust their computer systems according to their changing needs simply by adding or removing a module rather than by replacing the whole system. This meant that there would have to be a whole range of peripheral equipment, so that users could build up their own systems to meet their own particular requirements. The system could, for example, be structured to optimise either commercial or scientific applications (Fisher et al 1983, 120).

The concept of modularity was to be implemented through the adoption of a standard interface between the peripherals and the input-output equipment in the new line. This meant that the same peripherals could be attached to all processors in the line in the same way:

> "All the auxiliary machines...had to be designed so that they could feed information into or receive information from the central processing unit; this meant that the equipment had to have timing, voltage and signal levels matching those of the central unit. In computerese, the peripheral equipment was to have 'standard interface'" (Wise 1966a, 120).

The standard interface would help maximise the benefits that customers could derive from the broad range of peripherals offered with the new sysstem and it would also offer substantial benefits for IBM.

The whole system of compatibility and modularity opened up new possibilities for the standardisation and mass production of computer systems. There would be savings for IBM in the design, development, manufacturing and testing of

computers. In particular, it would lead to higher production runs of the peripheral devices since the same peripheral device and the same attachment, or plug-in circuitry would be associated with the interface to any of the models (Fisher et al 1983, 126).

However, it was recognised by the SPREAD committee that compatibility and modularity also involved a certain risk for IBM. It would mean that IBM was a better target for its competitors, in the sense that they could decide to become IBM compatible in processors, peripherals and software, or, even worse, that they could anticipate the new line of computers and react more effectively, whereas if IBM committed itself to compatibility it would be very difficult to change to another approach.

The SPREAD report was completed by the end of 1961. It recommended the end of the proliferation of IBM products and, in order to achieve standardisation within IBM, it proposed the development of a family of compatible processors, peripherals and programs. The system was named the New Product Line and, in line with IBM's business goals, it had to have an annual growth rate of 20 percent. In order to achieve this growth the following objectives were to be met:

- a single family of five compatible Central Processing Units (CPU's) with a performance ratio ranging in size from the smallest computer then being produced at IBM to one slightly more powerful than the largest;
- a line of peripherals compatible with all the different CPU's;
- the New Product Line had to meet the needs for both scientific and commercial applications, by having the capacity to put together any number and combination of CPU's into a single stored-program-controlled system;

- each processor was to be capable of operating correctly all valid machine-language programs of all processors with the same or smaller memory configuration;
- each processor was to be economically competitive at the time of its introduction (SPREAD Report, 1961).

As soon as the report was completed, Haanstra, Evans and some of the other members took it "to the top management of IBM on something like January 3, 1962. Management listened to the report, accepted it, and we were all charged with the implementation plans the study had recommended" (Evans in SPREAD Discussion 1983, 36). The plan for a common line of computers was to be implemented: this in itself was of major significance. As one of the SPREAD members, Fred Brooks, later commented: "it requires an extraordinary degree of determination to hold a highly divisionalised, very diverse business together in a common technical strategy. The most remarkable thing about the SPREAD Report is not what it contains, but that the corporation in fact followed it" (SPREAD Discussion 1983, 42).

A major problem still to be confronted was the question of timing. The new family was not to be compatible with any of the existing systems, so that the timing of its announcement had to be planned very carefully to avoid making the existing systems obsolete before their time. This was particularly a problem at the lower end of the market, where the General Products Division's 1401 was still doing very well. It was agreed that the new series would be announced at different times, taking into consideration sales forecasts and the decline of the different installations. The dates of announcement which SPREAD suggested for the

first and fourth models were 1965, for the second and fifth models the dates were early and late 1964 respectively, and the mid-range models were to be announced in 1966 (Pugh 1984, 197).

In spite of the SPREAD recommendations that the timing of the announcements of the different models of the New Product Line should be between early 1964 and 1966, different things were happening that made IBM change its mind.

IBM's product line was running out of steam. The company was barely reaching its sales goals for that period. This was due partly to the fact that rumours about the new product had been spread, so potential customers were holding back their purchases, waiting for the new product. Another reason was that the needs of users were changing and IBM had limited capacity to meet those changes in demand. Thirdly, IBM suffered a terrible blow: in December 1963, Honeywell announced a new computer that threatened the existence of the 1401, IBM's best selling computer (Wise 1966b).

Honeywell's announcement was a blow to IBM because Honeywell was not just offering a computer that was 30 percent cheaper and more powerful than the IBM 1401, but it was offering to the user of the 1401 the possibility of upgrading without having to rewrite programs. Honeywell was providing a very inexpensive way of converting programs of the IBM 1401 into programs for the H-200. The conversion was achieved with a software simulator called *The Liberator.*

The fact that the H-200 was cheaper, designed to allow a relatively easy conversion, and seen as technically a very superior machine ensured its success.

The first H-200 was delivered in early 1964; by that time, the IBM 1401 was four years old. By 1965, the H-200 was accounting for over 50 percent of the value of Honeywell's installed computers (Brock 1975). Designed to replace the IBM 1401, the H-200 meant the end of the 1401's dominance of the market. IBM badly needed the introduction of a better product in order to maintain its position in the market.

IBM's reaction to Honeywell's threat was to improve and drastically reduce the price of the existing line. Discussions about the timing of the announcement of the New Product Line were reopened.

People from marketing argued strongly for a simultaneous early announcement of all the models. There were several advantages in announcing all the models at the same time. One was that it would have a tremendous public relations impact. The company's image would change radically, and "customers would have a clear picture of where and how they could grow" with a family of computers (Wise 1966b, 205). Customers would then wait for IBM's products.

These were not the only advantages. With an early announcement of all the models, they were hoping to solve various problems. One was the possibility of an antitrust action. Rumours about the new IBM product were not just slowing down the sales of the existing product lines, but they could create problems with the Justice Department. According to IBM's policy, no employee was allowed to tell customers of any product not yet formally announced by IBM. But things were getting hard, and in order to keep customers from switching to the competition, some sales people were hinting at the near announcement of a new product. The

Justice Department might find that IBM was taking away competitors' customers in an unfair way. IBM in fact fired or disciplined several employees for violating this policy.

The main opposition to the suggestion from marketing came from the Product Test Department, the Finance Department and the Endicott Division.

Tom Watson, the President of IBM, had created strict rules for the testing of a new product. Fishman (1981, p 99) quotes him saying that: "No new product will be announced for general availability to all customers, entered into regular production, nor delivered until it has been approved by an IBM testing department". Permission to change this rule could come only from the Corporate Management Committee, a body composed of top executives which acted as chief policy maker for IBM.

In December 1963, a product test report on the small model of the family said that an early announcement of this particular model was very risky, both for hardware and especially for software. They suggested that the announcement of the small model, bearing in mind normal risks, should be November 1964. The report pointed out that, for the rest of the models, to bring forward the date of announcement would be a very risky move. Even three days before the final decision was taken, the head of Product Testing made clear his department's opposition to the announcement, arguing that "a large part of the testing was incomplete, and there were known major problems with the system" (Fishman 1982, 99).

The Finance Department was opposed to the announcement, arguing that with all that rush a proper analysis of prices had not been done and that the prices for the new model were too low.

Very strong opposition came from the General Products Division at Endicott. Its opposition was not just to an early announcement but to a particular model of the family, the model 30. From the beginning, Haanstra, head of Endicott, had been against the idea of a family of computers. But once the commitment to a single family of processors was accepted by IBM executives, Haanstra's opposition was mainly centred in the fact that the model 30 which IBM was hoping to sell to its 1401 users and the 1401 were incompatible, whereas Honeywell's H-200 with its new reprogramming technique was compatible with the 1401. Haanstra believed that the only way to stop Honeywell taking away up to three quarters of the 1401 customers was through an improved and cost reduced version of the 1401, the 1401-S (Wise 1966b).

By the end of 1963, the argument for early simultaneous announcement had gained increasing support. In order to speed the work on the new system, Watson abolished the Corporate Management Committee. He felt that the Committee was a barrier to reaching an agreement on the various problems in the development of the new system. Abolishing the Committee would make it possible to speed up vital decisions and allow an early settlement of the problems.

As for the opposition from the Finance Department, it "fell on deaf ears". Watson allowed just two minutes to the people from that department to present their views (Fishman 1982, 100).

The main opposition was still Haanstra. But thanks to Honeywell's Liberator, the marketing people at IBM had become aware of the possibility of making the model 30 of the new series compatible with the 1401 computer. A group of engineers believed that with the technique of read-only, which involves the storing of permanent electronic instructions in the computer, they could design the model 30 to operate like the 1401 (Pugh 1984).

In order to meet Haanstra's objections to the model 30, Learson (head of marketing at IBM) organised in January 1964 a "shoot-out" between the 1401-S and the model 30. The result was that the model 30, emulating the 1401-S, could work at 80 percent of the speed of the 1401-S. For Learson that result was good enough to ignore Haanstra's opposition (Wise 1966b, 205).

On the 18th and 19th of March 1964, T. Watson, IBM's president Al Williams, and thirty top executives got together for a final risk assessment meeting. The aim was to revise every controversial aspect of the program. Haanstra did not take part in the meeting: he had again been "promoted" less than two months before. This meeting was the last opportunity for anyone to state objections to, or doubts about, any aspect of the new system. At the end of the meeting, Williams, who had been presiding,

> "stood up before the group, asked if there were any last dissents, and then, getting no response, dramatically intoned, 'Going...going...gone!'" (Wise 1966b, 205).

On April 7, 1964, IBM made "the most important product announcement in company history" (Burck 1964, 113). The company staged simultaneous press conferences in sixty-two cities in the United States and in fourteen foreign

countries, and organised a huge gathering at IBM's headquarters in Poughkeepsie, New York, for which it hired a special train to transport members of the press. It was at these conferences that IBM announced the launching of its new line, the family of compatible computers, the 360 Family.

The name 360 was chosen to indicate the full circle of the range of possible applications of the system. For the first time the distinction between commercial and scientific computers was totally blurred. IBM was to provide a series of related and program-compatible processors as well as peripherals to satisfy any user's needs.

The announcement caused a great commotion, partly because IBM "is not a corporation given to making earth-shaking pronouncements casually, and the declaration that it was launching an entire new computer line, the System/360, was headline news" (Wise 1966b, 138). But it was the magnitude of the new line that shook the computer industry:

> "No company had ever introduced, in one swoop, six computer models of totally new design, in a technology never tested in the market place, and with programming abilities of the greatest complexity" (Wise 1966b, 138).

The computers announced were the 30, 40, 50, 60, 62, and 70 models. These models were going to have different capacity, they would be compatible and their memories would be interchangeable, so that it would be possible to have nineteen different combinations. On top of that was the peripheral equipment, which included forty different input and output devices, including printers, optical scanners, and high-speed tape drives. There were over 150 different things

announced in all. The delivery of the computers would start a year later, in April 1965.

In announcing the 360 computers before the planned dates, IBM was making obsolete not only almost all the existing computers in the market, but its own computers as well. In an article published in *Fortune*, Wise points out that IBM's announcement was a challenge to the marketing structure of the whole computer industry:

> "It was roughly as though General Motors had decided to scrap its existing makes and models and offer in their place one new line of cars, covering the entire spectrum of demand, with a radically redesigned engine and an exotic fuel" (Wise 1966a, 119).

The announcement of the 360 represented an enormous gamble for IBM, which put the whole future of the company at stake. Evans half-jokingly, half-seriously says of the 360 project: "We called this project 'You bet your company'" (Wise 1966a, 118). In addition to making its existing computers obsolete, the decision involved the commitment of enormous sums of money:

> "The decision committed IBM to laying out money in sums that read like the federal bubget - some $5 billion over a period of four years" (Wise 1966a, 118).

It was this great gamble, "probably the most important event in the history of the computing industry" (Galler in SPREAD Discussion 1983, 27), which created the context for the software problems that were to come. The marketing people had triumphed over the technical people in the battle over the announcement of the 360 series. But once the announcement was made, it was the technical people who were left with the task of realising a commitment of unprecedented dimensions.

Chapter 3

The Producers

The IBM 360 had been launched. With one stroke, IBM's great announcement had declared not only all its own products obsolete but also virtually all the computers then on the market. Only IBM could have afforded to make such a dramatic announcement. But would the great gamble pay off, would IBM be able to keep its promises?

What were those promises? Essentially, IBM was promising five things: a dramatic increase in cost performance through the joining together of two great technologies of the century, the computer and the integrated circuit; the provision of a general purpose computer that would cover both scientific and commercial applications; compatibility between the different models of the series, which would allow the user unlimited growth capability while remaining within the same environment; an operating system that would both provide compatibility and allow on-line real-time multiprogramming capabilities; and, finally, the simultaneous delivery of all the models in the 360 family.

The attempt to fulfil these promises was to turn not only IBM but the whole computer world upside down.

The commitment to deliver all the different models at the same time "was the boldest and most perilous part of the plan" (Wise 1966a, 120).

Announcing all the models simultaneously and promising to deliver them all at the same time had enormous consequences. It meant the 360 hardware and software were developed under terrific pressure. It made almost all existing products in the market obsolete. It unleashed a war in the computer industry.

The effect of IBM's announcement in making existing computers obsolete was greatly accentuated as a result of the renting practices in the industry. It was the practice of IBM and other computer manufacturers to rent most of their machines rather than sell them. Although, under the terms of a settlement in an anti-trust suit in 1952, users had the option of buying the equipment if they wanted, IBM encouraged customers to rent rather than buy, and it remained the practice of computer manufacturers to use short-term leases in the marketing of their computers (Brock 1975, 155).

For a well established company like IBM, renting was more profitable than selling. Firstly, it facilitated the control of the pace of technological innovation: provided that the machines stayed leased long after they had paid for themselves, manufacturers like IBM introduced new products only when they were forced to, in order to avoid making their own products obsolete before time. Another advantage was what is called account control:

> "When a computer is bought outright, the salesman and the customer shake hands and say goodbye until the customer needs a new machine. If the computer is rented, however, the customer has a continuing relationship with the supplier, and the liaison is the salesman, who returns regularly to see that everything is going right and incidentally to suggest new applications for the equipment. The more applications, the more equipment the customer will need - more memory, additional peripherals and ultimately a larger mainframe" (Fishman 1982, 17).

The advantages that IBM derived from renting went beyond just having a salesperson regularly at customers' premises. Renting gave IBM financial control, "the ability to adjust prices on machines in the field in order to correct previous pricing errors, respond to economic conditions and bring revenues into line with goals" (Fishman 1982, 260).

The practice of renting machines also played an important role in keeping new entrants out of the computer industry. A new company entering the industry had to wait several years until its first products started paying off and at the same time it had to have enough money to support the development of new products. It took a period of over ten years for the new company to break even, "when it ships enough computers to realise economies of scale in manufacturing and product development. By then the rented machines contribute enough gross profit to cover fixed costs...Remington Rand entered the computer business in 1950; it was 1966 by the time its descendant, Sperry Rand, broke even on computer operations" (Fishman 1982, 16).

The leasing of computers was also a way of expanding the market for computers. It made computers available to users who otherwise would not acquire them: the leasing reduced the magnitude of the initial investment necessary to obtain a computer system. Short term leasing also shifted the economic risk of technological obsolescence to the manufacturer:

> "If the customer sees a shinier toy, if his needs change or if he simply doesn't like the product or service he gets from IBM or Univac, he can simply return the machine" (Fishman 1982, 17-18).

One result of short term leasing was that manufacturers were required to

respond rapidly to users' needs and were under constant pressure to keep their users satisfied. This pressure, however, did not mean that users could switch computers or computer brands lightly. Manufacturers protected themselves through software:

> "Each manufacturer has a different way of manipulating the ones and zeros, and software written for an IBM computer is largely incompatible with a Univac machine" (Fishman 1982, 18).

For the users, switching computer meant that they had to rewrite all their programs and this was a very painstaking and time-consuming process.

> "The ones and zeros of binary code are known as *bits*. Bits are assembled into groups of six or eight called *characters* or *bytes*, each of which represents a number or letter, and into larger groups called *words*. The number of bits in a byte or a word is peculiar to the architecture of the machine, and this is one of the factors that make it impossible to run the same program on two different computer models without a little or a lot of rewriting" (Fishman 1982, 10-11).

It is not only that the users would have to rewrite software. Often specific processes within the users' organisation interlock with a particular software structure, so that changing to a new software system may require the reorganisation of other aspects of the business too. Consequently, the decision to change computer brand is not one that is easily taken:

> "The cost of switching causes a high percentage of users to order their follow-on computer from the manufacturer who provided the previous one. A 1962 survey showed that 92% of the users who had used a previous computer ordered their second computer from the same manufacturer as the first" (Brock 1975, 51).

Under a leasing system, therefore, software is the main thing that ties users to a particular manufacturer. The development of simulation and emulation techniques weakens this effect, however.

In spite of the effect of software in tying users to particular manufacturers, the leasing system still made the whole computer industry particularly volatile. Once IBM announced their 360 family, other companies had to change their products in order to try to retain the loyalty of their users. The 360 announcement was the starting point for what is generally referred to as the third generation of computers: often defined in terms of its component technology (integrated circuits), the third generation was shaped above all by the enormous impact of the 360. The 360 announcement started a chain reaction, as other manufacturers reacted to IBM's 360 and IBM counter-reacted to their reactions:

> "The fitful competitive struggle in the computer industry has suddenly burst into a full battle, and the marketing strategists have moved forward to take field command" (*Fortune* Jan 1965).

This series of reactions and counter-reactions was to have an important effect in shaping the nature and scope of the 360, and therefore of IBM, the computer industry and the future development of software.

The original concept of the 360 was modified in a number of important respects in response to the competitive environment. The 360 concept always had its critics even within IBM: John Backus criticised SPREAD, arguing that a full line of compatible processors was too ambitious an undertaking, particularly when the architecture of line was supposed to remain competitive through 1970 (Fishman 1982, 96). On a number of points, the internal criticism combined with external developments to bring about changes.

One such area was component technology. Even after the decision was taken in favour of SLT, there were still some inside and outside IBM who criticised this

decision as being over-cautious and short-sighted in underestimating the potential of monolithic technology (Soma 1975; Rosen 1969). One of the leading critics was John Haanstra, who had been removed from the General Products Division for his opposition to the 360 shortly before the announcement was made.

Only two months after IBM's announcement that the 360 family was going to have hybrid circuits, Fairchild, one of the semiconductor companies "stunned its competitors" (Siekman 1966, 122) by drastically cutting the price of its circuits. By the time the first 360 was delivered in April 1965, Scientific Data Systems was delivering the first commercial integrated circuit computer, and soon other companies were joining the market with their own integrated circuit models:

> "Only a little more than a year ago monolithic integrated circuits moved out of the stratosphere of high-priced military systems and into the commercial computer cost range. Today their use in large-scale business and industrial computers is common design practice" (Richmond 1965, 29).

When the response of the other computer manufacturers to the 360 family took the form of computers with monolithic circuits, there was considerable fear inside IBM that the 360 would soon be obsolete. The position of Haanstra and the other critics of SLT was greatly strengthened. In September 1964, Haanstra wrote a report called "Monolithics and IBM", in which he documented how far behind he believed the company to be in monolithic circuitry. His report played on the fears of corporate managers and urged the immediate development of monolithic technology (Pugh 1984, 252).

Fears about the possible obsolescence of the 360 components were not the

only cause of anxiety. There were problems also in the production of SLT which delayed for several months the scheduling required for the 360 computers. The problem had to do with a particular step in the production of the circuitry, which was accomplished by an evaporation process. IBM engineers had used small-capacity evaporators to test the technique, but when large-capacity evaporators were introduced to meet the mass production requirements, problems emerged, which had to do with metallurgical changes that took place when the large evaporators were used. As a result, production came almost to a standstill. Finally the metallurgical problems were solved by using all the small-capacity evaporators they could find (Wise 1966b). But by then it was clear that IBM would not be able to deliver all the computers at the same time.

In January 1965, management responded to the component problems by carrying out a restructuring inside IBM. The Data Systems, the General Products and the Components divisions were united to form a single division, the Systems Development Division. Haanstra was appointed president of the new division, and the company started to move into monolithic technology.

These rapid changes in the component technology of the computers caused considerable confusion within the company. Pugh gives some sense of the feeling of chaos at the time:

> "The frantic efforts to upgrade the company's presumably inadequate technology resulted in a period of uncertainty and frustration. Development objectives, unrealistically aggressive in retrospect, were rejected for not being aggressive enough. Some development groups became so confused that they did virtually nothing. Others continued their own projects while ignoring the rapidly changing objectives established by higher levels of management" (Pugh 1984, 253).

A second way in which the original 360 concept was modified under competitive pressure was through the erosion of the general purpose idea. Some of the fears of those opposed to the general purpose concept did turn out to be well-founded. Some competitors did successfully offer specialised sytems to cater for the needs of users who wanted something more specialised than the general range of functions offered by the 360. In order to assure its presence in the more specialised markets, IBM had to develop specialised models within the 360 family, such as the 360/91, the 360/67, the 360/44 and the 360/20. These new developments expanded its undertaking even more and modified the original concept of the general purpose family of computers.

One of the markets in which IBM was forced to react to the competition was in the area of very large-scale computers. As a result of the failure of IBM's STRETCH, Control Data had become the leader in the production of large-scale computers with its 3600 machine, first delivered in 1963. In the same year Control Data announced a new model, the 6600, which had been discussed in the trade literature from 1961 onwards, and "which had capabilities far beyond even IBM's discontinued STRETCH" (Brock 1975, 170).

In the original announcement of the 360 family in April 1964, IBM had not included a computer that could be competitive with Control Data's 6600. It seemed to outsiders that IBM had lost interest in the market of large scale computers; but inside IBM "there was a lot of heat at the time, that CDC had stolen a march on them and that they had abandoned the top of the scientific line and they were in danger of losing the leading edge customers" (Fisher interview). Four months later, and just before CDC was due to deliver the first 6600s, IBM announced

officially a new super-scale computer, the 360/90 series. However, IBM had already been discussing a new super-scale computer with potential customers of Control Data's 6600 since 1963. For Control Data, IBM's announcement meant that orders for the 6600 were delayed and that, in order to compete with IBM's announced but unavailable computer, Control Data was forced to cut substantially the price of the 6600.

In the development of the 360/90 computers, however, IBM encountered so many problems that in January 1966 it announced that from then on only purchase orders would be taken. Finally, in early 1967, before any computer had been delivered, IBM decided to discontinue the 360/90 line, but promised to fill all contract orders.

Control Data felt that IBM's actions on the super-scale computer market were unfair competition, partly because of the long delay between the announcement of the 360/90 and its delivery, and partly because of the estimated losses incurred by IBM in its production, and it sued IBM for antitrust violations. IBM finally built seventeen machines and lost $126 millon in the project (Fishman 1982, 123). Control Data, in contrast, sold nearly a hundred of its 6600 computers but, as a result of having to compete with the IBM's non-existent computer, the company suffered big financial problems.

There were also problems at the opposite end of the computer scale. After the 360 announcement IBM realised that the model 30 was not small enough to attract small customers. The small low-cost computers were very important because they helped to expand the computer market. As a result, "there was a lot of

pressure to bring out a very small machine" (Fisher interview). In order to protect its position from Univac, Honeywell and GE, IBM announced the model 20 in November 1964. In order to keep costs down, it was decided that the model 20 should not contain all the features of the 360. As a result, it was not compatible with the 360 family to the extent that other models were, and it did not have the standard interface to plug in to the 360 peripherals.

The model 20 was used not only by small users but by large customers as peripherals of their main system and it proved to be a success:

> "As it turned out, the 360/20 was more than merely a good competitive response; as had the 1401 in its day, the 360/20 became the largest-selling of IBM systems, with more than 7,400 installed in the United States by 1970" (Fisher et al 1983, 153).

Another area in which IBM broke away from the general purpose concept was the intermediate scientific area. In the months following the announcement, the company began to be increasingly concerned with the fact that models were not being accepted by customers in the intermediate scientific area. Concerning the acceptance of models 40 and 50, Learson wrote to Watson:

> "Our position here since announcement in April, 1964 is that we have won 44, lost 44, and we have 172 doubtful situations. CDC and SDS have a total of five machines which out-price, out- perform us by a good margin" (Fisher et al 1983, 158).

In the medium scientific market users were rejecting the IBM machines because they did not optimise their particular applications:

> "Some scientific customers were asking for a lean hard machine. They didn't want all the bells and whistles that came with the 360. There was a demand for a somewhat stripped-down and cheaper version" (Fisher

interview).

In addition, they were not prepared to accept the overhead associated with the complex 360 system software. To meet the problem, IBM announced the model 44 in August 1964. With this model IBM was offering "raw binary speed and high throughput to solve a wide range of scientific problems, including high speed acquisition jobs" (Fisher et al 1983, 159). In order to achieve this aim at a reduced cost, compatibility with the rest of the 360 family was sacrificed. The model was not successful: it was soon surpassed by the competition and also:

> "To at least some in IBM, it appeared that this was because IBM had learned to meet customer needs generally, but had not successfully learned to specialise within that talent" (Fisher et al 1983, 159).

A further breach in the 360 family concept arose in connection with the introduction of a time-sharing capability, where several users share the computer resources simultaneously. After IBM had lost several important contracts to General Electric, whose 600 series offered a time-sharing capability, the company announced in March 1965 another addition to the 360 range: the 360/67. In Fisher's words, there was:

> "A great deal of heat on the question of whether they had missed the right way to do time-sharing, and a crash programme to develop what became the 360/67" (Fisher interview).

This was to prove the most traumatic of IBM's changes to the original concept, but, since the problems had to do mainly with the software, this will be taken up later in the chapter.

In the course of the implementation of the 360, there was therefore an important move away from the concept of the general purpose machine. This shift

came about under the constant competitive pressure that did so much to shape the 360:

> "Throughout the development what you've got is an immense crash programme by the corporation with a clear understanding very early, possibly unique in the computer industry, that they could not stand still, and the farther into the programme they get, the more it becomes obvious that in fact they could not stand still, that the need for putting out this product was getting more and more severe as competition went on" (Fisher interview).

As the 360 family developed, the concept of compatibility was modified. When IBM moved away from the concept of a general purpose computer with the introduction of the new models, it also broke with the idea of a fully compatible family. The 360/20 and the 360/44 were never intended to be compatible with the rest of the 360 computers, and there were also difficulties in achieving full compatibility between the other new models and the rest of the family. Downwards compatibility, the possibility of running programs from the larger models on the smaller models of the family, was never achieved.

Another difficulty in the implementation of the 360 concept arose from the problem of the incompatibility between the 360 family and IBM's older models. It was not originally intended to make the older models compatible with the new machines, but after Honeywell's announcement of its H-200 machine which was cheaper and more powerful than the IBM 1401 and which came with its software simulator, the *Liberator*, IBM realised how vulnerable it was in this area and was forced to provide users with a way of running their old programs on the new models.

The transfer of the old programs to the new models was to be achieved

through emulation based on microprogramming techniques. Although emulation meant a loss of efficiency, it offered users the possibility of making a smoother transition to the new system:

> "The provision of emulators on System/360 afforded users an alternative to conversion. It permitted them to transfer jobs to System/360 and to concentrate on new application areas without immediately having to convert their existing applications. Although programs run in emulation mode generally ran more slowly than they would have if rewritten to run in native mode on the new systems, they could be run effectively enough to permit users to forego reprogramming if they chose to do so" (Fisher et al 1983, 117).

During the course of 1964, IBM announced a whole series of emulators for the 1401, 1410, 1440, 1460, 1620, 709, 7010, 7040, 7044, 7080, 7090, 7094 and 7094 II processors. It has been estimated that the cost of developing the 1401 emulator for the 360/30 was $200,000 and the cost of developing the 7090 emulator for the 360/65 was $500,000 (Fisher et al 1983, 116).

Although emulators created a bridge between models, they represented an inefficient use of the 360 computers, and most applications had to be reprogrammed for the new equipment (Rosen 1969).

In implementing the 360 plan, IBM were forced to modify the original concept in a number of significant ways. The most difficult area of all, however, to implement was the software, and particularly the operating system.

Operating Systems

Erasing the distinction between scientific and commercial computers, and achieving compatibility among the different processors and peripherals were

important aspects of the 360 family. But even more important was the promise to provide an operating system which would schedule the computer without manual intervention in different environments and different configurations. IBM decided to call the new operating system by the generic term - "The Operating System" - but it is more commonly referred to as OS/360.

Operating systems are an important step in the further separation of programming from the machine. An operating system is a master program, a set of programs, the aim of which is to allow a more efficient use of the computer resources and peripheral equipment. An operating system is basically the house keeper of the computer: the various programs that constitute the operating system are in charge of different tasks, like scheduling the users' programs, allocating space in the computer main memory and managing the peripheral equipment. Without an operating system, programmers would have to incorporate in each program they write the necessary intructions for scheduling different tasks and controlling different pieces of equipment.

Operating Systems were born as an attempt to eliminate idle time, the primary source of which was the way in which computers were organised and used. Computers represented such a large investment that it was very important to use them in the most efficient way possible and to try to eliminate any idle time.

One of the main sources of idle time was the way in which computers were used:

"The programmer marched into the machine room with his card decks and listings, preparatory to an extended session of playing with the console keyboard. First, an inordinate amount of time was wasted whenever the machine hung up and the programmer scratched his head, trying to figure out what to do next. Second, nothing at all happened - not even constructive thinking - when a job was done. The departing programmer had to gather all his material and insure that nothing was left undone. The new user had to get himself and his material properly emplaced. It was during this hasty transfer of authority that the worst calamity would occur - a dropped card deck (often unsequenced). If this deck belonged to the oncoming user, idle time mushroomed" (Steel 1964, 26).

The initial step towards a solution to this problem was to create a new profession: machine operators. The function of computer operators was to ensure the continuous running of the computer. They would now do the loading of programs into the computer and in general take care of any unanticipated stoppage. The introduction of the operators had the effect of pushing programmers out of the computer room.

This measure provoked anger and resentment among programmers, who felt that they could not debug their programs unless they manipulated the computer's console. Others argued that this would force programmers to write better programs and to ensure that their programs were correct before being run on the computer (Greenbaum 1979).

However, from the point of view of the users, even computer operators took too much time between removing one job and initiating another. It seemed ridiculous to have a machine that could perform thousands of calculations per second, but where the time that it took between unloading one program and loading another was several minutes or even more. By the time computer operators were widely employed, the pressures on total machine time available were becoming very severe.

The first operating systems were attempts to automate as much as possible the functions of the computer operator, in order to reduce idle time. The basic idea of a primitive operating system was job stacking or batching:

> "Rather than being loaded into the machine independently and immediately executed, a collection of jobs is gathered into an input batch and the programs necessary for each job, together with the relevant input, are *all* loaded onto an input file. There is a program, normally kept in the main store, whose function is to load the next job from the input file, halting only when there are no more jobs to process. All jobs are required, upon completion, to transfer control from this program, and machine operators are instructed to execute the appropriate manual transfer if any thing goes wrong" (Steel 1964, 26).

. General Motors was the first to develop an operating system. This elementary operating system was implemented on the IBM 701. Hart from General Motors describes the situation before the operating system as follows:

> "With the 701 it was necessary to schedule people to the computer one at a time to read in the cards at the card reader, wait for the computation to compute, print out the results, and then log off and let the next person approach the machine to repeat the process" (Fisher et al 1983, 31).

After the introduction of the primitive operating system, the increment in productivity was such that General Motors decided to construct a similar system for its next computer, an IBM 704. North American Aviation had been very impressed with General Motors' achievements and, because it also wanted to obtain a 704, it joined forces with General Motors for the development of an operating system for the 704. Efforts were taken to develop an operating system that would provide "an automatic mechanism via software for executing one job after another without operator intervention" (Fisher et al 1983, 31). According to General Motors, the use of this operating system "quadrupled the throughput of the 704 computer by eliminating several steps of manual handling" (Fisher et al 1983,

31).

The IBM 704 became a very successful computer. Its users formed SHARE, one of the first users' organizations. SHARE was to provide a forum in which IBM's users could get together to join efforts and avoid redundant developments in using the 704 computer. SHARE and other users' organizations became a force in their own right in dealing with computer manufacturers and in shaping the development of software, as was seen already in the discussion of ALGOL.

The success of the 704 became the medium by which the General Motors-North American Aviation operating system concepts were spread among SHARE members. Soon most 704 installations had operating systems of their own. Not long after, the operating system concept "jumped the boundaries of machine type", and the use of operating systems in large computers became a common practice (Steel 1964, 27).

There was some discussion at the time as to whether general purpose operating systems should be designed and supplied by the manufacturers. Some users felt that specially designed and tailored operating systems were required because of the great difference in the way in which computer installations work, and that consequently they should be designed by the users themselves. But as operating systems became more complex, it became more common to see them as uniform systems which should be designed and produced by the manufacturers (Rosen 1967, 17).

It was through pressure from SHARE that IBM undertook developments on

operating systems. In the late 1950's SHARE members began to develop the SHARE Operating System (SOS) and they pressed IBM to take on its further development. It became universally accepted that it was the obligation of the computer manufacturers to provide operating systems and software in general with a computer.

As the concept of the operating system spread, operating systems became more complex. In addition to the idle time generated by the fact that people and computers perform at different rates, there were other problems that operating systems had to deal with. A second form of idle time arose from conflicts due to the independent decisions made by programmers in the selection of system components:

> "A simple example of such a situation is the case of programmer N+1 who wants to use tape unit A as an intermediate scratch tape just after programmer N has used the same unit as an output tape. While operators run about dismounting and mounting tapes - or, at the very least, changing switches to rename tape units - everything in the machine must come to a halt" (Steel 1964, 27).

Then there was the problem of turn-around time, a problem that arose from the introduction of operating systems themselves. Although the machine might operate more efficiently, the amount of time that it took to get results often meant that the programmer's time was wasted. Some people felt that operating systems simply replaced one inefficiency with another because they focused on the problem of saving machine time rather than human time:

> "In many shops it takes far too long for a job to pass through the system, and we begin to see the converse of the problem operating systems were designed to solve. A great deal of man time is now expended to gain a little machine time" (Steel 1964, 28).

As computers became more powerful and sophisticated, another type of idle time became increasingly important. It was no longer just a question of keeping the computer running continuously, but also of using the computer resources efficiently.

Multiprogramming systems were a response to this requirement. Multiprogramming was an important feature of third generation computing. The basic idea of multiprogramming is that the resources of the computer can be used more fully if the machine is able to handle more than one program at a time:

> "'Multiprogramming' is the label given to the concept of a dynamic sharing of the resources of a given computer system among two or more programs" (Steel 1968, 99).

At its simplest, multiprogramming means that while program N is being executed, program N+1 is in the input and program N-1 is in the output.

> "An operating multiprogramming system presents to external observers the appearance of effecting the concurrent execution of several object programs. There may or may not be truly simultaneous operation of more than one program, but it will be the case that a second program begins execution before the first program has run to completion" (Steel 1968, 99).

Multiprogramming therefore poses the question of the coordination of the different programs and of the transfer of control from one to another:

> "There must be an *oscillation* of control among the several programs for multiprogramming to come into play" (Steel 1968, 99).

The coordination of the different programs creates a whole series of challenges for the development of operating systems. The transition from one program having all the resources of the computer to several programs sharing the

computer's resources is something like the transition from directing traffic in a one-street village to the problems of traffic control in a city. The coordination of traffic and the avoidance of collisions is similar to the challenge confronted by multiprogramming.

In multiprogramming, traffic coordination is conceptualised in terms of processes, or programs in execution:

> "The term 'process' was introduced in the early 1960s as an abstraction of a processor's activity, so that the concept of 'program in execution' could be meaningful at any instant in time, regardless of whether or not a processor was actually executing instructions from a specific program at that instant" (Denning 1971, 178).

A multiprogramming operating system may be defined therefore in terms of the supervisory and control functions for the processes created by its users. These include creating and removing processes; controlling the progress of processes so that one process does not block another indefinitely; dealing with exceptional conditions arising during the execution of a process, such as ensuring the priority of a privileged process over a less privileged one; allocating hardware resources among processes; providing access to software resources like compilers, assemblers, subroutine libraries etc; providing protection, access control and security for information; and providing communications between processes where these are required. A multi-programming operating system has the task of coordinating and prioritising these different activities. These functions have to be provided by the operating system because they cannot be adequately provided by the processes themselves (Denning 1971, 175).

Although multiprogramming became a common feature of third generation

operating systems, not all the systems try to achieve the coordination, supervision and control of multiple processes in the same way. There are significant differences in approach which are crucial for the development of software during this period. These differences are important for an understanding of the subsequent problems of software. This can be illustrated by comparing two approaches to multiprogramming: the Master Control Program of the Burroughs B-5000 and IBM's OS/360.

The Burroughs B-5000

The Burroughs B-5000 commercial computer (first delivered in March 1963) was remarkable for the relation established between hardware and software from the very beginning. In terms of the analogy between operating systems and traffic control in a city, it was as if the city and the traffic control system had been planned at the same time. The Burroughs system was the first and for a long time the only computer whose hardware design was based on a number of well-defined software objectives (Rosen 1972, 595). The emphasis on software was very much in the minds of the designers from the very beginning. An article by two of the designers, first published in 1961, opens by making this point strongly:

> "Computing systems have conventionally been designed via the 'hardware' route. Subsequent to design, these systems have been handed over to programming sytems people for the development of a programming package to facilitate the use of the hardware. In contrast to this, the B-5000 system was designed from the start as a total hardware-software system. The assumption was made that higher level programming languages, such as ALGOL, should be used to the virtual exclusion of machine language programming, and the system should largely be used to control its own operation. A hardware-free notation was utilised to design a processor with the desired word and symbol manipulative capabilities. Subsequently this model was translated into hardware specifications at which time cost constraints were considered" (Lonergan and King 1987,

16).

The design of the B-5000 started out from the software, and particularly from the choice of ALGOL as a programming language. The choice of ALGOL was made by the designers of the machine:

> "Burroughs decided that high-level languages were the way to go, that efficient compilers could be written, and then we simply selected the best language around, and that was ALGOL. We didn't consider at all that there was the huge customer base in FORTRAN and the momentum that we'd have to overcome. ALGOL was simply a better language" (Waychoff in B-5000 Discussion 1987, 47).

This choice coincided with Burroughs' view "that ALGOL was to become the standard scientific programming language" (Dahm in B-5000 Discussion 1987, 41).

The problem then was to design a machine suitable for ALGOL implementation. ALGOL was very difficult to implement on existing machine architectures, partly because it did not map neatly on to a static storage system in the way that FORTRAN did. The implementation of recursion demanded a dynamic storage system, a *stack* architecture in which information could be stacked and eliminated as required by the execution of the program. In the B-5000, the implementation of the stack architecture was associated with the development of a *virtual memory* system, a concept already used in the Atlas system constructed at the University of Manchester between 1959 and 1961.

A virtual memory system allows a user to address a sub-program in peripheral storage as though it were in the main memory:

"A design feature called virtual memory could shuffle segments of the

> user's programs between the computer's fast, expensive main memory and the cheaper, slower disc storage, giving him the illusion that the main memory he had to work with was twice its actual size" (Fishman 1982, 140).

In the Atlas, the main memory was a core store, while the auxiliary memory was a rotating drum. In order to implement a virtual memory several times the size of the main memory, the virtual memory was divided into blocks of fixed size (512 words) called "pages", while the real memory was divided also into 512 word blocks called "page frames". Pages were the unit of transfer and storage among the page frames of the two levels of memory. When the real memory was too small, the programmer had to divide his program into segments and insert swapping commands into the program's text.

Like the Atlas, the Burroughs machines had a virtual memory system. Unlike the Atlas, the Burroughs system employed not pages, but blocks of varying size called "segments", the size of which could be determined by the programmer or the logic of the program. Although this made address translation more complex, it had the advantage of supporting the structuring of the program into blocks shaped by the logic of the problem rather than the structure of the machine.

Another feature of the design conept of the B-5000 was its emphasis on *reentrancy*. A reentrant program is a program which does not in any way modify itself during execution. In early programming it was common for programmers to write non-reentrant code, i.e. code which was modified in the course of execution, as a means of saving storage space and increasing speed.

In an environment in which only one program was being executed, the use of

non-reentrant code did not necessarily create difficulties. In a multiprogramming environment, however, the situation was different. Multiprogramming is characterised by the oscillation of control between different processes. As the processor moves from one program to another (e.g. to give one program priority over another), the processes are interrupted. For the smooth operation of multiprogramming, it is necessary to be able to control the interrupts and to control the progress of the process. It is important too for a process, once interrupted, to return to the same state: hence programs should not be modified in the course of their execution, i.e. they should be reentrant. Here the distinction between addresses and locations (or code and data) introduced by the Atlas system and developed by the B-5000 is important: in order to be reentrant, the code must be distinct from the data handled, and it must be read-only (non-modifiable). The important feature about the Burroughs system is that, like the Atlas before it, it established such a distinction and so excluded non-reentrant code. By doing so, it effectively recognised that programming in a multiprogramming environment required a new discipline and a new methodology.

With this structure, Burroughs aimed to make its computer secure, reliable and relatively easy to use. There was a strong emphasis on saving user time rather than machine time: where "other manufacturers were designing for efficient use of memory space and to save machine time, Burroughs designers wanted to save the user's time" (Fishman 1982, 167). This is reflected in the exclusive emphasis given to higher level language programming and also in the approach of the designers to the problem of compiling. The emphasis in designing the compiler (in complete contrast to the approach of the team which designed

FORTRAN, for example) was on the speed of compilation rather than the efficiency of the object code produced:

> "One of the goals was to have fast compilation... Unfortunately, we forgot about the need for object-code execution speed. [Laughter.] Not until users started running programs and comparing them to object-code execution on IBM machines did I realise that somebody had made a terrible omission in decimal arithmetic on that machine. It was just terrible" (Hale in B-5000 Discussion 1987, 67).

The design of the B-5000 system was carried out by a very small team, numbering only twenty-five, in which software specialists and engineers worked closely together. Within Burroughs, the team enjoyed a large degree of autonomy. Whereas the marketing people had been incorporated in the SPREAD discussions at IBM from the beginning, marketing had a much lower status at Burroughs. The general sales manager for EDP systems at the time later described the general atmosphere at Burroughs:

> "It was largely Engineering deciding what they could do and somebody deciding what the market wanted. Then the sales department was told, 'This is what you have. Isn't it great?'" (Ford in B-5000 Discussion 1987, 76).

The design of the B-5000 was very much shaped by considerations of the quality of the machine the design group wanted to produce, rather than by marketing criteria. The total contrast with the position at IBM was brought out very clearly in a discussion on the development of the B-5000. Asked if the adoption of ALGOL was "in response to customer requests or at Burroughs's instigation", one of the participants, Duncan MacDonald, replied:

> "*MacDonald:* Things at that time weren't done in response to external stimuli. [Laughter.] They were internally created in an atmosphere that was highly creative, highly charged, and largely running on its own steam.

Galler: And not necessarily market driven, then?

MacDonald: No, they weren't. Absolutely not. Because, you see, things were so advanced that there weren't any external stimuli. They didn't exist at the time to a large extent.

Galler: How did you justify the effort within Burroughs?

MacDonald: Didn't have to. Nobody else in Burroughs knew what was going on. As long as you stayed away from the administrative restrictions, you did what you pleased. That's oversimplified, but the atmosphere was very much that way. Very unique. I've never seen it since" (B-5000 Discussion 1987, 44).

The emphasis on quality rather than marketability was expressed in the team's reluctance to provide a FORTRAN compiler. It was in fact provided much later, but only under heavy pressure from the sales organisation. The design team felt that FORTRAN would not fit well on the machine and that it would lower the tone of what they were trying to do:

"I never wanted to see it on there, because to me it had implications of trying to sell a machine head-on in FORTRAN shops. I personally thought it would be a terrible mistake and not the kind of thing to be trying to do with this wonderful machine called the 5000, which had a lot of other objectives" (MacKenzie in B-5000 Discussion 1987, 49).

Perhaps it is not surprising that the machine was not a great commmercial success. It is universally agreed that it was a very good machine, far ahead of its time in architecture and programming concepts, but it sold only slowly. Partly this was because the radical nature of its conception made it difficult to understand for users used to static memory sytems: "trying to sell a machine where neither the operator nor the user nor the programmer knew where the program was in memory was very difficult" (Dent in B-5000 Discusion 1987, 49). The adoption of ALGOL also brought the B-5000 into confrontation with the entrenched position of FORTRAN:

> "The only thing that was available was higher-level language, that being ALGOL, and who knew what ALGOL was? 'Maybe Edsger Dijkstra and all those people can tell me how wonderful it is and it's the future of the world, but I've got $5 million invested in FORTRAN programs here, and what am I going to do with them?'" (Pearson in B-5000 Discussion 1987, 85).

In addition to those difficulties, the Burroughs sales force was in crisis at that time, with many of the computer specialists having left shortly before, so that it was difficult to train the salespeople sufficiently for them to be able to present and explain a machine that was such a radical departure in computer design. As a result, "there were a lot more Burroughs machines bought than sold" (Collins in B-5000 Discussion 1987, 84): the purchasers of the Burroughs machine tended to be those bodies where the specialists understood the quality of the B-5000 and went to Burroughs and asked to buy it. As Fishman puts it:

> "A Burroughs user was a buff, with that air of self-congratulation that buffs always have. He was pleased that he had the perspicacity, sophistication and daring to appreciate a really superior piece of equipment and to find Burroughs, because God knew the Burroughs salesmen, who were scarce as hen's teeth anyway, weren't astute enough to find him" (Fishman 1982, 166).

Despite its lack of commercial success, the B-5000 was a very significant development in computing. Variously described as a Porsche (Fishman 1982, 166) or a BMW (Rosin in B-5000 Discussion 1987, 90) among computers, the B-5000 is widely seen as having been well ahead of its time, particularly in its recognition of the importance of software for computer design. On the 100th anniversary of the Burroughs Corporation in 1985, Blumenthal, the chief executive of the company, said:

> "Since our beginning, we have made significant accomplishments, but if I were to cite one major milestone in this century of progress, it was our

bold introduction 25 years ago of the B-5000 mainframe - a decade ahead of its time due to a design decision to develop hardware and software in concert, to use higher-level languages exclusively for user productivity, and to ensure compatibility as we developed new generations of mainframe systems" (quoted by Rosin 1987, 6).

The IBM Operating Systems

IBM tackled the question of multiprogramming in a different way from Burroughs. In particular, the relation between hardware and software was quite different. In the design of the 360 system, the programming department was not brought in from the beginning, as it had been at Burroughs, and even later it continued to be managed separately. As an internal IBM report in 1966 put it: "it started out-of-phase and still is" (quoted in Fishman 1982, 101).

This was not just a question of bad timing, but related to the whole way in which the computer was understood. The computer was seen as a piece of hardware: the software simply had to fit in with the design of the hardware. This perception shaped the notion of efficiency embodied in the design of the OS/360: efficiency was seen as hardware efficiency. The aim of programming was seen as being to eliminate superfluous instructions in order to minimise the storage and run-time requirements of the program. The problem with this approach is that in large, complex systems such as the OS/360, with an enormous number of possible interactions, the whole program becomes extremely difficult to understand and to debug. Thus, the efficiency of the code itself, defined in these terms, comes into conflict with the effectiveness of constructing a working system.

In this approach, the framework for the software is pre-established by the hardware and, since the power of the computer is defined in hardware terms,

measured in terms of the price/performance ratio, the result may be a machine that is extremely difficult to program:

> "The official literature tells us that their price/ performance ratio has been one of the major design objectives. But if you take as 'performance' the duty cycle of the machine's various components, little will prevent you from ending up with a design in which the major part of your performance goal is reached by internal housekeeping activities of doubtful necessity. And if your definition of price is the price to be paid for the hardware, little will prevent you from ending up with a design that is terribly hard to program for... And to a large extent these unpleasant possibilities seem to have become reality (Dijkstra 1978, 12).

The OS/360 was pulled by the hardware on the one side, by the enormous demands implied by IBM's promises on the other. On the hardware side, a stack architecture was tried but rejected at an early date because "that was fine at the higher end of the spectrum, but it didn't work at the lower end of the spectrum" (Brooks in SPREAD Discussion 1983, 31). Consequently, the architecture of the 360 is a classic static memory architecture, still bearing the legacy of first and second generation machines, but far more complex to allow for multiprogramming and the compatibility demanded by the family commitment. This meant that programming took place within a very baroque environment.

On the side of the demands to be met, the scope of the OS/360 was, of course, a reflection of the enormous scale of what IBM had promised to achieve with its 360 family. The objectives of the operating system derive from those promises. In line with IBM's promise of a compatible family with a big variety of peripherals, the basic objective of OS/360 was to "accommodate an environment of diverse applications and operating modes" (Mealy 1966, 3). The aim was that it should be applicable to all (or nearly all) the models of the family, that it should be able

to deal with a variety of input-output equipment and other peripherals, that it should be applicable both in scientific and commercial use, and that it should be a multiprogramming system equally appropriate to batch and real-time applications. Other objectives were said to be increased throughput; lowered response time; increased programmer productivity; adaptability of programs to changing resources; and expandability (Mealy 1966).

The mere description of the objectives of the OS/360 gives some idea of the enormous software task that had been created by the 360 announcement. As Kinslow put it:

> "In my view both OS/360 and TSS/360 were straight-through, start-to-finish, no-test-development revolutions . I have never seen an engineer build a bridge of unprecedented span, with brand new materials, for a kind of traffic never seen before - but that is exactly what happened on OS/360 and TSS/360" (Garmisch Conference Report 1968, 122).

The enormous scope of the operating system resulted from the family concept which was the strength of the whole 360 idea. But this vastness brought complexity with it, and complexity resulted in difficulties. The aim of providing software that would be compatible across a range of machines with standard interfaces made the software much more complicated and difficult to test:

> "In a way, the modularity and standard interface of the System/360, which made hardware testing easier, made software testing harder. It allowed customers great flexibility in the range of configurations they could choose, and that, coupled with the wide variety of ways in which OS/360 could be used, led to a very complex hardware-software system that was literally impossible to test adequately" (Fisher et al 1983, 139).

One example of the problems of complexity is provided by the Job Control Language. The basic function of a job control language is to assist in controlling

the movement of jobs within a multiprogramming system. In the OS/360, however, the Job Control Language became very sophisticated and could be used for a wide variety of purposes. As one of the OS/360 manuals put it:

> "The flexibility of the job control language allows the programmer to specify his requirements for a large variety of facilities when he prepares his control statements. He may specify job priority, set-up information, buffering and block methods, space requirements, etc" (IBM 1967, 629).

The problem with this sophistication was that it tended to encourage clever tricks and led to a system that was over-complex and difficult to use, so much so that JCL became "one of the swear words of third generation systems":

> "The glaring failure of current multiprogramming technology is the complications it has introduced for the programmer and operator. The current Job Control Languages (JCL) required to specify what the system is to do are, by and large, disasters. It takes far too much of a programmer's time to construct the appropriate JCL statements, and an even larger amount of time to debug them, not to mention the effect on morale of aborted runs deriving from trivial JCL errors" (Rosen 1968, 102).

Brooks, the manager of the OS/360 project is no less critical: "JCL is the worst programming language ever devised by anybody, anywhere, for any purpose; and it was done under my management" (SPREAD Discussion 1983, 41).

Another illustration of the problems of complexity is provided by PL/1, the programming language developed for the 360. Parallel to the development of a general purpose computer, it was felt that there was a need for a universal language that could handle both scientific and commercial programs. In order to consider these problems, IBM and SHARE formed a joint committee in October 1963. The committee first proposed to upgrade FORTRAN, but after a lot of

discussion they decided in favour of developing a new language. Pressure was put on the comittee to produce the language as quickly as possible: "we literally nailed the door shut on this study team" (Evans in SPREAD Discussion 1983, 39). The committee was confronted by a "series of very early, but seemingly rigid deadlines" (Radin 1981, 553):

> "In order to be included in the first release of IBM's New Product Line, we were first informed that the language definition would have to be complete (and frozen) by December 1963. In December, we were given until the first week in January 1964, and finally allowed to slip into late February. Thus, not only was the total time for language definition very short, but even this period was punctuated by 'deadlines' which gave a sense of crisis to the activity" (Radin 1981, 553).

The new language was called PL/1 (Programming Language 1): in fact, thinking ahead to the future, IBM patented the names PL/1, PL/2...PL/100.

The new language proved to be a very complex one. It took elements from several existing languages (FORTRAN, ALGOL, COBOL, JOVIAL, etc):

> "To produce a formal description of a language of the magnitude and complexity of PL/1 was a formidable task. This task was undertaken by a group at the IBM research laboratory in Vienna [Austria], and the resulting document was so thick that it was humorously referred to as the Vienna telephone directory" (Rosen 1972, 592).

PL/1 was widely criticised for its complexity (Rosen 1972; Moreau 1984; McGovern 1967):

> "Using PL/1 must be like flying a plane with 7,000 buttons, switches, and handles to manipulate in the cockpit. I absolutely fail to see how can we keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language - our basic tool, mind you! - already escapes our intellectual control... When FORTRAN has been called an infantile disorder, full PL/1, with its growth characteristics of a dangerous tumor, could turn out to be a fatal disease" (Dijkstra 1978, 15).

The complexity that characterised the Job Control Language, PL/1 and the OS/360 as a whole provided a wide range of facilities for the programmer, but left a lot of room for exceptions and clever tricks. Good programming in such an environment is a question of finding one's way through the complexity and applying facilities cleverly: the environment does nothing to impose a discipline or a coherent methodology on the programmer.

Originally, it had been intended that the OS/360 should provide support for all the 360 computers except the very small ones. Smaller models would use as startup systems the Basic Programming Support (BPS) and the Basic Operating System (BOS). However, it soon became clear that OS could only be used in the large 360s, with at least 265K of memory, and that it was necessary to develop other systems for the smaller computers. This led in the early part of 1965 to the development of another operating system, the Disc Operating System (DOS). This particular system was released several months before the initial release of OS.

The pressure on software development became even worse when IBM announced the introduction of the specialised models. Of these the most traumatic experience was that connected with the 360/67, the time-sharing model.

IBM had considered the possibility of including time-sharing capabilities for the 360 family, but had initially decided that it was not feasible, and that the market was not yet ready for time-sharing. Time-sharing is a system that shares computer resources among many users, working at different terminals more or less far from the computer itself, so that each one appears to have complete control over the computer. In fact, this is an illusion: the computer works on each

user's program only for a fraction of a second, moving from program to program and then repeating the cycle. From the programmers' point of view, it meant that they could again have a "true rapport" (Steel 1964, 26) with the computer and that they could control the running of their own programs, without having to "hand over their work, punched on a deck of cards, to the operators in the cool, glass-enclosed computer room" (Fishman 1982, 108). Time-sharing was a convenience for programmers and also greatly increased their productivity by reducing the turn-around time.

However, IBM did not at first realise the importance of time-sharing. In line with the concept of efficiency that dominated the whole 360 project, IBM had the idea that "the balance between computer-power and people-power, or the price of computer-power versus people-power still was heavily enough weighted in the direction of computer-power that they could be concerned about optimising the use of computer resources rather than concerned about optimising the use of human resources", as Norman Rasmussen, one of the participants in the TSS project put it (Rasmussen interview). They misjudged how attractive the notion of time-sharing was, especially for "university type people, who had no concern about that value trade-off, because they were dealing with free computers funded by goverment" (Rasmussen interview). It seemed that IBM misjudged the extent to which these people would push hard the notion of improving the life of the computer user, and their influence in changing the perception of the demand.

However, when General Electric announced that its third generation computers, the 600 series, would have time-sharing capabilities and MIT decided to use a General Electric 635 for its MAC project on time-sharing, IBM had to

rethink its position.

Within the company, time-sharing was a highly controversial subject. Those who felt that IBM should develop time-sharing argued that there was more at stake than just a few prestige contracts; that IBM computers risked becoming obsolete if they did not develop time-sharing:

> "The reason for this...was that the real problem in applying computers was the productivity of the programmers, a factor which time-sharing improved considerably" (Fishman 1982, 112).

Those who opposed the development of a time-sharing capability for the 360 argued in terms of machine efficiency: time-sharing would require a modification to the 360 hardware which would increase costs and slow machine performance (Fishman 1982, 111). To this objection were added the complaints of the software people that they simply could not cope with such an extra workload.

After losing a second important contract to General Electric, for supplying Bell Laboratories, in October 1964, IBM decided to work seriously on the development of time-sharing for the 360 series. It was agreed by IBM management that the OS/360, which was supposed to be "everything for everyone", was not suitable for time-sharing, both for marketing and for technical reasons. Therefore, it was accepted that a special model (the 360/67) with a different operating system, the Time Sharing System (TSS), should be built. However, those working on the 360 software still warned of the dangers of making any public announcement:

> "In May the head of software protested strongly against any imminent official announcement. He observed that marketing had not formally

> specified what it wanted IBM to accomplish in the time-sharing software; that both hardware and sofware specifications were incomplete and many important customers disapproved of them; that field engineering had not declared itself able to maintain the system; that testing plans were not clear; that technical issues relating to compatibility hadn't been resolved; and funding for the project hadn't been defined" (Fishman 1982, 115).

Moreover, even if it were not for these difficulties, the head of the 360 software pointed out that the whole software workload was out of control, that it was more than the programming staff could handle:

> "The IBM workload in programming is not only growing at a faster rate than ever in history, but in a more uncontrolled way than ever before. This can only lead to the abrogation of commitments on a large scale in the years 1966 and 1967" (quoted by Fishman 1982, 115).

Against the advice from the software people, the new model that would offer the time-sharing capability, the 360/67, was announced in August 1965, for delivery a year later. The initial release of TSS was scheduled for June 1967.

From the beginning, TSS "had an awful lot of expectations attached to it and an awful lot of advanced technology incorporated into its specifications" (Rassmussen interview). Potential users were said to be enthusiastically "planning installations in which hundreds of consoles would be on-line simultaneously" (Rosen 1969, 32).

But TSS turned out to be extremely difficult to implement. This was mainly because of the complexity of the operating system required. The task of this operating system was to allocate computer resources among users, and to protect each job from interference by the others.

TSS "suffered a tremendous number of slippages, setbacks, decommitments,

reduction of personnel, reduction of performance expectations, etc." (Rassmussen interview). By July 1966, the estimated number of lines of code required for TSS had doubled: the undertaking was far more complex than had been anticipated (Fisher et al 1983,166). In August 1966, IBM announced that there would be a delay of 45 days in the initial release of the TSS. In the autumn of the same year, IBM made calls on its 360/67 customers to explain the situation "and inform them that certain functions were being decommitted and schedules delayed" (Fisher et al 1983, 166). In January 1967 the company announced a major decommitment of functions: while the company would continue to work on improving the basic TSS software, many of the promised features could not be guaranteed for delivery, ever. After this announcement, well over half of the 170 orders already received for the system were cancelled (Fishman 1982, 116).

Such were the problems in the development of TSS that from first being offered as a major attraction, it eventually became just a "very minor offering which hung around the middle market until about 1968, when other things took its place" (Rassmussen interview).

Because of the excessive expectations attached to TSS, and the lack of appreciation of how much had to be invented in order to fulfill those expectations, TSS became highly controversial. As the realisation of the problems involved started to materialise, so there were "reductions in schedule, everyday reductions of personnel, reductions in computer investment, and the bill kept rising. The bill kept rising at the same time." (Rassmussen interview).

Neither IBM nor General Electric was able to deliver a working time-sharing

operating system until 1967. IBM did eventually succeed in building a lot of what was originally promised. Eight successive versions of TSS were produced, almost all of which included the redesign of major components of the system. It was estimated that over 2,000 man years of work had gone into TSS. Even after all that effort, all that could be said for the system was that "with adequate core memory and relatively few users, TSS was a usable, though expensive time-sharing system" (Rosen 1972, 597). Even as late as 1970, the 360/67, used with standard TSS software, could support fewer than 15 simultaneous users, and responses to trivial requests were reported to take between 10 and 30 seconds (Doherty 1970).

Commercially, TSS was a total failure: just 10 to 15 installations used it. Although several hundred computers of the 67 model were delivered, most of them used software provided either by other time-sharing companies or developed by the users themselves. By 1969, the programming expenses of TSS were forecast at $57.5 millon, with losses of $49 millon on the entire 67 model (Fishman 1982, 117).

Watts Humphrey, head of software at IBM in 1966, sums up the TSS project in the following way:

> "As a technical man I'd say TSS was a success in teaching us a whole series of technical concepts. As a product developer, I'd say it was a very expensive education" (Fishman 1982, p 117).

For General Electric, the experience of developing time-sharing was equally traumatic. Although it became the leader in time-sharing technology, it was at the cost of going out of the computer business. It was estimated that General Electric

losses for 1966 were $100 millon and for 1967 $60 millon. After the unexpected costs in developing hardware and software for time-sharing, General Electric could not afford to develop new products and decided to sell its Computer Division to Honeywell in 1970 (Brock 1975).

Software: The New Force

As the 360 project developed, an awareness of the importance of software forced itself upon IBM. The company came to realise that the problems that would hold up the scheduling of the 360 were not hardware but software: "no part of the whole adventure of launching System/360 has been as tough, as stubborn, or as enduring as the programming" (Wise 1966b, 138).

By 1965 orders were flowing in for the new family of machines, but the success of the 360 was not guaranteed simply by the number of orders. In some respects, the amount of orders for the 360 made the problems in manufacturing and software development worse, without ensuring market success, since orders could be cancelled easily and systems in rental could be returned. Producing the software for the 360 was becoming a huge problem. The reality for IBM was that its entire future was shaking. By the end of 1965, Watson wrote:

> "By the spring of '64 our hand was forced and we had to, with our eyes wide open, announce a complete line - some of the machines 24 months early, and the total line an average of 12 months early. I guess all of us who were thinking about the matter realised that we would have problems when we did this, but I don't think any of us anticipated that the problems would reach the serious proportions that they now have" (Fishman 1982, 100).

The problems continued to grow:

> "In 1966 the problems of designing software for System 360 were most serious. Perhaps the operating system would never achieve the sophistication that IBM had planned for it. If so, the 360 would be highly vulnerable to competition, and might not even live out its allotted span" (Fishman 1982, 259).

An internal investigation in August 1966 into the problems of producing software cited poor planning as the root of the problem, observing that the programming department had not been brought into the planning of the 360 early enough and that programming had continued to be managed separately (Fishman 1982, 101).

The planning and management problems involved in the OS/360 were quite unprecedented. Noone foresaw the enormous difficulty of developing the complex multiprogramming operating systems of the third generation.

Most of the major manufacturers ran into similar difficulties and delays in dealing with the unprecedented problems. "There was too much system code that had to be resident in the central memory and too much computer time used in system overhead functions" (Rosen 1972, 594), so that the amount of space available for the users' jobs was reduced. The obvious solution was to have larger and larger central memories, but core memory was very expensive: incorporating the amount of core memory necessary to run the operating system would mean that the price of the computers would become non-competitive. Systems had to be debugged and improved, but the process of change often introduced new bugs and new problems. The problem faced by the computer manufacturers was that nobody really knew how to go about producing and debugging these new systems. They all misjudged the complexity of developing the new operating systems. As Theodore Climas, the corporate head of programming for IBM, put it: "we had to do some of

our planning with the entrails of chickens" (*Business Week*, November 5, 1966, 132).

One of the first problems was that the programmers that were desperately needed to develop the software for the 360 during the whole of 1963 and 1964 were still engaged in improving the programs for IBM's older computers.

As IBM became aware of the complexity of the undertaking and as delays accumulated, more and more programmers were added to the project. By mid-1965, the company had become aware of the magnitude of the problem and the number of programmers working on the 360 software expanded in an unprecedented manner:

> "During the peak effort, some 1000 programmers at 12 locations in the US and five other countries were at one time working on the control programs for System 360, plus another 1000 on application programs" (*Business Week*, November 5 1966, 132).

As the number of programmers working on the 360 mushroomed, the costs mushroomed too. Watson gave some impression of the way in which programming costs were expanding when he spoke to a meeting of IBM customers early in 1966:

> "We are investing nearly as much in System/360 programming as we are in the entire development of System/360 hardware. A few months ago the bill was going to be $40 million. I asked Vin Learson last night before I left what he thought it would be for 1966 and he said $50 million. Twenty-four hours later I met Watts Humphrey, who is in charge of programming production, in the hall here and said, 'Is this figure right? Can I use it?' He said it's going to be $60 million. You can see that if I keep asking questions we won't pay a dividend this year" (Wise 1966b, 139).

There are no official figures on the final costs of the OS/360. Estimates range

from $200 million upwards:

> "One IBM software man estimates that in all the company spent some $500 million to develop, maintain, repair and enhance the 360 systems software, including the programs that controlled the computer's operations and those that translated higher-level languages into binary code. Enhancements to that software which enabled it to run on the next product line probably cost another $500 million. Neither of these figures includes the numerous applications programs which IBM prepared to help the customer get out the payroll or keep track of inventory on the 360" (Fishman 1982, 101-102).

This escalation of costs took place in spite of IBM's decision early in 1966 to "decommit" thirty-one technical capabilities of the operating system. Although this helped to reduce the difficulties, it "represented only a minor gain in the total software campaign" (Wise 1966b, 211).

This mushrooming of costs and programmers led also to a mushrooming of documentation:

> "The documentation for the 360 operating systems is voluminous, almost to the point of being overwhelming. There are literally thousands of documents, and techniques of computer assisted text preparation and publication make it possible to produce new documents and new versions of old documents at an alarming rate" (Rosen 1972, 596).

The complexity of the documentation made the operating system much more difficult to use, whereas one of its aims had been to make programming easier.

The enormous number of programmers and money dedicated to the OS/360 did not prevent it being late. As the delays accumulated, the problems increased. Planning and scheduling was a major problem in the whole project. It was very difficult to estimate the amount of time and the number of people required to develop the software for a whole family of compatible computers. Delays had

consequences not only for the rest of the 360 project, but also for the quality of the software itself, since the whole development took place within a corporate atmosphere of "it must be done". Delays had especially serious consequences for the testing of the software:

> "Failure to allow enough time for system test, in particular, is peculiarly disastrous. Since the delay comes at the end of the schedule, no one is aware of schedule trouble until almost the delivery date. Bad news, late and without warning, is unsettling to customers and to managers. Furthermore, delay at this point has unusually severe financial, as well as psychological, repercussions" (Brooks 1974, 20).

Delays in an environment of urgent deadlines can have disastrous consequences for the quality of the product. As Brooks says in discussing the conclusions to be drawn from the 360 experience:

> "An omelette, promised in two minutes, may appear to be progressing nicely. But when it has not set in two minutes, the customer has two choices - wait or eat it raw. Software customers have had the same choices. The cook has another choice; he can turn up the heat. The result is often an omelette nothing can save - burned in one part, raw in another" (Brooks 1974, 21).

In spite of the heat being turned up, the OS was still late: although delivery of the 360 computers had originally been promised for April 1965 and although the first deliveries of the hardware were actually made in that year, it was 1967 before the systems software was delivered. Even then, the operating system was found to have many errors, so that a whole series of releases was required before the system was running at all well. The complexity of the system made it prone to error. It was calculated that each new release of OS/360, designed to remove existing errors from the system, contained roughly a thousand new software errors (Boehm 1973, 57). The errors continued:

> "Release 16 of OS/360, issued in July 1968, contained nearly 4000 modules, comprising almost one million instructions...Its FORTRAN H compiler alone was stated to contain 2000 distinct faults" (Randell 1979, 6).

The releases continued: release 21 contained 6,300 modules (Belady and Lehman 1979, 140). Even then, IBM never achieved the total compatibility that had been at the core of the family concept. The problems of OS/360 were partly the reflection of the basic concept behind the 360 family. Trying to be everything for everybody led to a failure to optimise in any particular area (McGovern 1967, 19).

Despite all its problems, it was nevertheless the OS/360, and not the Burroughs system, generally considered to have been superior, that set the standards for the development of third generation operating systems:

> "The most important software systems of the period 1965-1970 were the systems developed by IBM for its system 360. This is true not because of the intrinsic merit of the systems themselves, but rather because of IBM's position of dominance in the computer industry" (Rosen 1972, 596).

The effort involved in the 360 project was enormous. As the 360 family grew, and as the problems of realisation grew, IBM also grew. Just as IBM shaped the 360, the 360 came to shape IBM. By 1966, IBM had spent over half a billion dollars in research and development associated with the 360. The programme:

> "involved a tremendous hunt for talent: by the end of this year (1966), one third of IBM's 190,000 employees will have been hired since the new program was announced. Between that time, April 7, 1964, and the end of 1967, the company will have opened five new plants here and abroad and budgeted a total of $4.5 billion for rental machines, plant and equipment. Not even the Manhattan Project, which produced the atomic bomb in World War II, cost so much (The goverment costs up to Hiroshima are reckoned at $2 billion), nor, probably, has any other privately financed commercial project in history" (Wise 1966a, 120).

Customer demand for the 360 was greater than anticipated. The 360 proved to be a turning point in the commercialisation of computers: third generation computers are often defined in terms of the basic component, the integrated circuit; much more important, however, were the traumas of the 360 and the breakthrough to the mass production and use of computers that they brought with them. Within weeks of the announcemment, thousands of orders were filed for the 360. To keep up with the demand many new divisions and plants were created and existing ones expanded:

> "By October 1965, IBM announced that it was 'completing more than three millon square feet of new manufacturing space' to meet requirements for system 360 - including plants in Boulder, Colorado; Raleigh, North Carolina; Montpellier, France; and Vimercate, Italy; and expansion of existing facilities in Owego, East Fishkill, and Endicott, New York; Burlington, Vermont; and San Jose, California. New plants were later added in Boca Raton, Florida, and Brooklyn, New York" (Fisher et al 1983, 140).

Thousands of people were hired:

> "Between the end of 1964 and the end of 1967, IBM increased its work force by approximately 50 percent - adding more than 70,000 new employees" (Fisher et al 1983, 140).

Whole areas of the company were reorganised:

> "The new family of computers cut across all the old lines of authority and upset all the old divisions. The system/360 concepts plunged IBM into an organizational upheaval" (Wise 1966b, 143).

At the peak of the problems:

> "Four technical executives were temporarily relieved of their normal responsibilities in order to devote full time to the task of identifying and solving all problems contributing to delays in manufacturing and shipping" (Pugh 1984, 248).

Careers were made and unmade:

> "Managers of software development projects were replaced and a number of individuals were dismissed for failing to meet schedules. Manufacturing managers who did not meet the ever accelerating schedules were replaced by others who found the problems to be no more tractable" (Pugh 1984, 251).

The whole of IBM was thrown into upheaval:

> "To launch the 360, IBM has been forced into sweeping organizational changes, with executives rising and falling with the changing tides of the battle" (Wise 1966a, 118).

The whole production, delivery and installation of the 360 "required a massive effort on IBM's part and placed a severe strain on the corporation. It was a task that some in IBM likened to trying to swallow an elephant" (Fisher et al 1983, 141).

The whole experience of the 360 placed IBM under financial stress. Although IBM always recognised the risks inherent in the 360 program, in the summer of 1964 it was confident that it had taken the right decision. The whole development of the 360 took such proportions that an investment of $1.1 billion was required to finance it in 1965, and a further $1.6 billion in 1966. IBM found itself without sufficient cash in 1966 and was forced to raise it urgently through an issue of shares, through bank loans and through reducing the price at which its machines were sold while at the same time increasing rental charges. This last measure left IBM very vulnerable at a time when competitors were reducing drastically the rental on their machines (Fisher et al 1983, 141; Wise 1966b, 206).

For some time it was not clear whether the great gamble - "IBM's $5,000,000,000 Gamble", as the title of one article put it (Wise 1966b) - would succeed. In his meeting with the SHARE members in March 1965, Watson even conceded that the announcement of the entire 360 package in April 1964 may have been "ill advised" (Wise 1966b, 211). And John Opel, a future president of IBM, wrote in a memorandum in 1966: "Henceforth, we will instruct our people to avoid like sin a repetition of such a broad announcement and any announcement before its time" (quoted in Fishman 1982, 103).

However, it was not simply the announcement that caused the difficulties. The announcement was just one expression of the general "it must be done" atmosphere of the corporation. The "it must be done" was the rule of the market. It had to be done for marketing rather than technical reasons. At the core of IBM's problems in trying to carry off its great gamble was a tension between technology and the market, personalised in conflict between the technical people and the marketing people. This is a conflict fundamental to the development of computers and of computer software; but in IBM, and most clearly of all in the experience of the IBM 360 it took a particular form. In contrast to the environment within which the Burroughs 5000 series was developed, the IBM 360 was characterised by conflict between the advice of the marketing people and the advice of the technical experts from the very beginning. And in this conflict there was no doubt that marketing set the pace. There is a programmers' saying in IBM that "Marketing is king, engineering is prime minister, and software is the court jester" (Fishman 1982, 104). Perhaps that is the secret of IBM's success.

In spite of all the problems, IBM's multibillion-dollar investment paid off. It

has been estimated that beforè the 360 announcement, IBM had only about 11,000 computer systems installed in the USA, by 1970 it had installed about 35,000 systems (Fisher et al 1983, 141). The 360 program was a financial success, "perhaps the greatest in the history of American industry" . At the end of 1965, before massive shipment of the 360 began, IBM's annual worldwide revenues were $ 3,600,000,000 by the end of the 360 period, revenues had increased to $7,500,000,000:

> "Observers have characterized the 360 decision as perhaps the biggest, in its impact on a company, ever made in American Industry - far bigger even than Boeing's decision to go into jets, bigger than Ford's decision to build several million Mustangs" (Fisher et al 1983, 142).

However, although the 360 was a success, the massive problems associated with the whole venture, and the retreats from many of the original goals, did much to change the image of IBM: "the mystique is probably gone for good - although the successes may just go on becoming greater and greater" (Wise 1966, 212). The achievements of the 360 were enormous, yet the whole experience was double-edged:

> "The great gamble had paid off, but in ways that couldn't have been anticipated when the machines were first planned. What Tom Watson had in mind was a family of technologically advanced computers that would provide IBM with a lead in technology, expand its markets, ward off competitors, and unite his company. He had success in most of these areas. But at the same time he had reshaped the industry; in the history of computers, everything is either pre-360 or post-360. Without meaning to do so, Watson had opened his own Pandora's box" (Sobel 1984, 232)

Pandora's box was well and truly opened, not only for IBM and the rest of the computer industry, but for the whole world.

Chapter 4

The Users

When Pandora's box was opened, the gifts fell upon the computer users. Software development during this period was no less double-edged for users than it was for producers.

The problems of software cannot be understood simply in terms of the difficulties faced within the computer industry itself. Software develops in the tension between the demands of the machine and the requirements of the users. Computers are produced as commodities to be sold on the market, to purchasers who intend to use the computer in some way. The manufacturing companies have to respond, or at least appear to respond, to the requirements of the users. The problems of third generation software reflected not only an "internal" gap between the promises of the manufacturing companies and the ability of their programmers to deliver, and between the capacity of the hardware and the ability of the software to exploit that capacity; there was also an "external" gap between the demands of the users and the performance of the producers.

The third generation marked a turning point in the mass production and use of computers. There was a very rapid increase in the production and use of computers during the 1960s. Since the largest manufacturer, IBM, does not publish figures on the number of computers it installs, there are no precise figures on the rise in computer installations, but it has been estimated that the total number of computers installed leapt from 16,000 in 1964 to 60,000 in

1969 (Brandon 1970, 28). Another estimate (*Computers and Automation* June 1967, 77) gives a figure of 52,460 for world computer installations in 1967, of which well over half (32,500) in the United States, with a figure over 25% higher the following year: 67,200, of which IBM machines accounted for over half - an estimated 39,600 (*Computers and Automation*, June 1968, 132).

Government was still the most important single computer user. A survey published in 1965 reported that about 10% of the total number of computers installed in the United States were used by the Federal Government (*EDP Weekly*, Sept 6, 1965, 14-15, quoted by Armer 1970, 123); a further 2.5% of the total number were being used by state and local governments (*Automatic Data Processing Newsletter*, Vol IX, No. 25, May 10, 1965, quoted by Armer 1970, 123).

Within government the use of computers for military purposes predominated. The 1965 report stated that of the 10% of total computer installations accounted for by the Federal Government:

> "about 7% are in the Department of Defense, with NASA and the AEC accounting for another 2%. Thus, nondefense and nonspace related activities of the Federal Government comprise only about 1% of the total computers installed" (Armer 1970, 123).

However, the most striking feature of this period was the increasing use of computers by private business. Some idea of the growing importance of computers in industry is given in a report by McKinsey & Co. in 1969:

> "In 1963, computer manufacturers shipped hardware worth $1.3 billion to their US customers. By 1967, the value of computer shipments had risen to $3.9 billion, an increase of no less than 200% in four years. Of

every $1 million that business laid out on new plant and equipment in 1963, $33,000 went for computers and computer-associated hardware. By 1967, the computer's share had risen to $63,000, and each dollar was buying at least half as much again in capacity. Computer spending, both absolutely and as a proportion of all plant and equipment outlays is still rising" (McKinsey 1969, 26-27).

The continuation of this trend was confirmed by Hertz, who says that by 1969, of every $1 million laid out in plant and equipment, "approximately $100,000 was going for computers and associated hardware" (Hertz 1969, 181).

At the beginning of the third generation computer, use by business was limited to large companies. One survey of 33 manufacturing companies in 1966 (in *Business Automation*, Oct. 1966, 53) found that:

"- the median company spent nearly $1.2 million yearly for its entire computer operation, with about one third of that amount allocated to equipment, mainly machine rental;
- total computer costs ranged from $128,000 to $50 million per year;
- the median company spent $5.40 per thousand dollars of sales for its computer operation" (Kleiman 1969, 48).

Kleiman points out the implications of these figures:

"If we hypothesize a computer installation renting at $5000 per month and each hardware dollar must be matched by $2 for backup support - a conservative assumption - the total yearly bill runs to $180,000. To support this expense, yearly sales must approximate $33 million" (Kleiman 1969, 48).

By the later 1960s, however, this pattern was beginning to change. Not only the fall in the price of computers - the average purchase price fell from $3,000,000 in 1953 to $410,000 in 1969 (Brandon 1970, 28) - but also the introduction of the first time-sharing systems opened up the possibility of computer use to medium and even small firms. A 1967 article praising the

virtues of time-sharing suggested that now "a typical subscriber can get good use out of the system for under $200 a month, including terminal, data transmission costs and program storage" (O'Rourke 1967, 50). By early 1969, time-sharing companies already operated nearly one hundred computers and their number was growing rapidly (Hertz 1969, 174).

The users had been promised great things. The third generation of computers had been announced with a great fanfare of publicity. The new systems were sold with the claim that they offered many dramatic improvements which would be of direct benefit to the user. First of all, the users would benefit from a considerable increase in cost-performance, resulting from technical advances: faster internal speeds, higher peripheral data transfer rates, larger storage with direct access capability, and the promise of sophisticated software that was to optimise system performance by extensive use of multiprogramming techniques. Secondly, the operating system would allow the user to control the system without requiring an intimate knowledge of the intricacies of the hardware. Thirdly, the family concept would allow the user unlimited growth capability while remaining within the same environment. Other innovations would open up a whole new range of applications: time-sharing for multiple users, rapid access from multiple locations, on-line processing of random transactions would all create the basis for total management information systems. And it was promised too that there would be little problem in converting Second Generation programs to the new system: new software translation techniques would allow the user to bridge the compatibility gap between Second and Third Generation systems (Bouvard 1970).

"Thoroughly dazzled by these rosy prospects, EDP users were in for a rude

awakening when confronted with the practical realities of the third generation" (Bouvard 1970, 120). The actual experience of the users after they had bought or rented the new computers was very different from what they had been promised. By the end of the 1960s, much of the optimism associated with the expansion of computer use had faded.

From the point of view of business, the basic problem was that computers were not doing what they were supposed to do: they were not increasing the companies' profitability. In many cases, the large investments made by companies were simply not paying off:

> "Studies conducted by consulting firms have indicated that many computers are costing organisations more than they are returning as part of that investment" (Trocchi 1969, 29).

The earlier managerial "euphoria" (Alexander 1969) began to fade in the late 1960s as more and more companies realised that the benefits obtained from their investment in computers were at best dubious (Callahan 1967; Alexander 1969; Laver 1970). There was a feeling that business had been tricked, that computers had been oversold. From the point of view of the users, the "traumatic experience of the Third Generation" (Bouvard 1970, 123) was one of disappointment, of an enormous gap between the promises of the producers and the reality of their own experience.

The gap between the producers' promises and the users' experience had two sides: the performance of the producers on the one side, and the users' own development of the software and their expectations on the other. It was not simply that the users had bought a "defective product"; the problems encountered by the

users arose both from the nature of what was supplied to them by the manufacturers, particularly the software, and also from the way in which the users developed and understood that software.

The most obvious source of problems for the users was simply that it was they who suffered the consequences of the manufacturers' failure to meet their commitments. Operating systems were delivered late; when delivered they were full of bugs; as a result, there were frequent releases of the same system; emulators were not as good as had been promised, so that the performance of converted second generation software was often below the standard achieved on second generation machines; compatibility between machines was not achieved to the degree that had been promised, etc.

Beyond the immediate question of delivery dates and programming bugs, however, lay much deeper problems. These sprang from the nature of software and from the way it was understood: from the complexity of third generation software, on the one hand, and from the ways in which users tried to apply it, on the other.

The complexity of the third generation machines led to a great mushrooming of programming problems that often swallowed up the benefits of the increased power of the hardware. One of the promises of the producers, for example, was that the new machines would radically increase the cost/performance ratio. Some authors, however, suggest that, despite the increase in the internal speed of the machines, the users often did not actually receive the benefit of any improvement in cost/performance. Feeney (1981) speaks of the "cost/performance myth":

"A myriad of graphs and charts have been produced showing the reduction in cost per character of storage or per instruction performed. No less impressive was the plethora of new and enhanced peripherals giving an explosion of capacity to the DP department. However, it is the author's contention that for many users the costs per character or per instruction were at best static and in many cases increased over much of the last two decades. The manufacturers' graphs and charts were usually honest. So how is this contradiction to be resolved? Black holes! Performance increases were absorbed by a whole series of black holes which kept the benefit well away from the user" (Feeney 1981, 265).

All of Feeney's "black holes" relate to software problems. One major problem for the users was the complexity of the new operating systems:

> "The operating system concept was supposed to simplify programming and operating problems by shielding the user from the internal complexities of hardware. Too often, however, the problems were displaced rather than solved as complex user-hardware interfaces were traded for even more complex and just as unnatural user-software interfaces" (Bouvard 1970, 120).

Not only were the operating systems full of complexity and difficult to use, but also because of this complexity they used up a lot of the capacity of the new machines: "they took most of the core and many of the cycles just to work out what they were supposed to be doing" (Feeney 1981,265).

The complexity and scale of the new software also gave rise to a problem that had not existed to the same degree previously: maintenance. In a large and complex system involving many programmers, maintaining the program became an important task in its own right. For Feeney (1981), maintenance is the "ultimate black hole":

> "It gradually became clear that programs are like cabbages, if you put them on the shelf and forget about them they go bad. The work of the maintenance programmer is formidable:
> - Trying to find original bugs
> - Trying to find the last maintenance programmer's bugs - Trying to find the bugs he put in last week himself

- Achieving compatibility with the latest operating system release
- Achieving compatibility with the latest compiler release
- Changing to use the new peripherals
- Changing machines
- Changing operating systems
- Changing standards
- Interfacing with new developments.
- Looking for bugs that do not exist (they are in the program of the accuser!)
- Statutory changes
- Making improvements for the user if there is any time left over"

(Feeney 1981, 266).

All of these activities demanded not only large amounts of machine time, they also demanded large numbers of programmers. The demand for programmers does not increase in simple proportion to the size of a program. As programs become more complex, the programming time required increases disproportionately: the larger the program, the slower the process of programming:

> "In 1964, the rule of thumb for estimating the manpower requirement of the programming of large systems was 200 or 300 machine instructions per man-month. However, in a graph developed that year by Nanus and Farr, summarizing experience gained in developing eleven complex systems, the function relating man-months to number of instructions curved upward rather sharply. It showed - on the average, though with quite a bit of variation - about 200 instructions per man-month on systems of 100,000 or 200,000 instructions, but fewer than 100 instructions per man-month on the largest of the eleven systems, which had 640,000 instructions" (Licklider 1969, 51).

The whole situation was made much more difficult by the fact that the programmers required for this explosion of programming activity simply did not exist. The shortage of programmers was a major preoccupation:

> "The supply of skilled computer personnel is far short of the demand. There are only about 120,000 programmers in the United States - and right now there's an estimated need for 175,000 or more. And the gap is widening. The number, power and widespread application of computers has far outstripped the supply of programmers" (*Computers and Automation* Feb. 1967, 11).

The shortage of programmers had important implications for the development of software. Most immediately, it meant that in order to meet their requirements companies often took on inadequately trained programmers, with obvious consequences for the quality of the programs produced, consequences that could multiply like a contagious disease:

> "The vast increase in programming requirements brought an influx of incompetent and poorly trained programmers. A lot of good people joined too! But it was the poor ones who soaked up the machine resource. It was bad enough to have a slow, inefficient and bug-ridden daily production job, but these characters really came into their own when they wrote slow, inefficient and bug-ridden compilers which produced deformed object code, which confused the hapless programmer endlessly. How could he resolve whether the problem lay with his lack of skill, or the operating system, or the compiler, or even with the intermittent hardware fault" (Feeney 1981, 265).

As the problems mounted, so did the costs. It was realised for the first time just how costly software was. Previously, the focus had been on the costs of hardware: now it became clear that software was even more expensive. Boehm (1973) provides some very striking illustrations:

> "For the Air Force, the estimated dollars for FY 1972 are ... an annual expenditure of between $1 billion and $1.5 billion, about three times the annual expenditure on computer hardware and about 4 to 5% of the total Air Force budget. Similar figures hold elsewhere. The recent World Wide Military Command and Control System (WWMCCS) computer procurement was estimated to involve expenditures of $50 to $100 million for hardware and $722 million for software. A recent estimate for NASA was an annual expenditure of $100 million for hardware, and $200 million for software - about 6% of the annual NASA budget. For some individual projects, here are some overall software costs:

IBM OS/360	$200,000,000
SAGE	$250,000,000
Manned Space Program 1960-1970	$1,000,000,000

Overall software costs in the U.S. are probably over $10 billion per year, over 1% of the gross national product (Boehm 1973, 48).

Programming was far more complex and required far more work than had been anticipated. As a consequence, the problem of costs was a double one: not only were costs high in absolute terms, but they also constantly exceeded estimates. It seemed to be almost a universal law that software took much longer than expected and cost more than was estimated. In the case of the Air Traffic Control System commissioned by the Federal Aviation Administration, to take just one example, Raytheon contracted (in January 1967) to deliver 16 computer display channels by the end of February 1968 for a fixed price of $44.8 million; more than four years later, in August 1972, not one operational model had yet been delivered and the "fixed price" of $44.8 million had been increased to $124 million (Hirsch 1972, 51).

Delays increased the costs for the user in two ways. Firstly, the increased cost of the software often fell on the user: obviously so if the software was being produced by the user company itself but often in other cases too - as in the case of the Air Traffic Control System. With experience, users were able to protect themselves in some cases at least: Wolverton, writing in 1972, reports that:

> "Our customers have shown a growing unwillingness to accept cost and schedule overruns unless the penalties were increasingly borne by the software developer" (Wolverton 1974, 615).

It was not only a question of direct costs. There were indirect, secondary costs too, which almost always fell on the users:

> "Big as the direct costs of software are, the indirect costs are even bigger, because software generally is on the critical path in overall system development. That is, any slippages in the software schedule translate directly into slippages in the overall delivery schedule of the system. Let's see what this meant in a recent software development for a large defense system. It was planned to have an operational lifetime of seven

> years and a total cost of about $1.4 billion - or about $200 million a year worth of capability. However, a six-month software delay in making the system available to the user, who thus lost about $100 million worth of needed capability - about 50 times the direct cost of $2 million for the additional software effort" (Boehm 1973, 49).

Cost was not the only problem arising from the complexity of third generation software. The other major problem was reliability. The baroque, seemingly uncontrollable nature of the software, in which complex application programs were often constructed on top of complex and error-prone operating sytems, led in the end to programs that were frequently unreliable. The unreliability of computer programs led to a proliferation of stories about people receiving electricity bills for a million dollars, but it also raised very serious problems about the consequences of depending upon computers.

The consequences of unreliability were particularly acute in the context of the "real time" computing applications then being developed, where the results of the computation were required immediately as part of a continuing process:

> "The issue of reliability takes on a dimension quite different from that encountered previously. If something goes wrong with the data processing environment...which cuts off real-time service, he has extended his troubles *right this minute* to the front line of the enterprise. In the case of an airline, it's the reservation sytem that does not work; for a bank, it's probably the savings accounts which are now inaccessible; a large wholesale distributor finds he no longer can determine his inventory condition...This entire sensitive matter of reliability takes on a degree of importance not easily equated directly to the costs of the system alone" (Boering 1967, 37).

The potential consequences of software unreliability are far-reaching. Speaking of a conference in 1969, Dijkstra recalls:

> "The...Conference in 1969 was shortly after Armstrong and Aldwin had walked on the surface of the moon and I knew that each new Appollo shot required 400,000 new lines of code - 400,000 is a lot and I was very

impressed. In Rome I met Joel Aron of the IBM Federal Systems Division who had been responsible as manager for quite a lot of that sort of thing and I said to Joel, 'How did you do it?' 'Do what?' said Joel. I said, 'Get all that software correct.' 'Correct!' he said, and then he told me that suppose they were computing trajectories and one of the programs computing a trajectory of the lunar module, if this was the moon, the module wouldn't go that way, it would go that way because in the program the moon had been repulsive, had been coded repulsive instead of attractive and that error had been found by accident five days before count zero. I got pale, and when I had regained my composure, I said 'Jesus Christ, Joel, those guys have been lucky!' 'Yes,' said Aron" (Dijkstra interview).

Dijkstra's story of the moon shot provides a particularly striking example, but the possible consequences of software errors are endless, from mistakes in electricity bills to plane crashes, from double-booking of airline seats to nuclear war. The implications of errors in the military software field, in which many of the most complex systems are applied, are obviously particularly frightening, especially as "the secrecy which shrouded their purposes served also to hide the extent to which such projects were often characterised by 'underestimates and overexpectations'" (Randell 1979, 5). Licklider cites the example of the Ballistic Missile Early Warning System:

> "Early in its operational life, the Ballistic Missile Early Warning System made its now-famous detection of 'incoming ballistic missiles' that turned out to be the moon. Fortunately, cool wisdom in Colorado Springs - and lack of confidence in the new system - prevailed over the reflex of counterstrike, and what could have been the greatest tragedy in history became a lesson" (Licklider 1969, 50).

The question of software reliability was an important issue raised in the discussion in the late 1960s of the US decision to develop an anti-ballistic missile system. As Licklider points out, the possibility of error arises not only from poor programming but from the sheer complexity of the programs involved:

> "Let us take time to make it clear that the presence of such errors in a program is not evidence of poor workmanship on the part of the programmers... the essential facts are that that all complex programs

> contain programming errors, that no complex program is ever wholly debugged and that no complex program can ever be run through all its possible states or conditions in order to permit its designers to check that what they think ought to happen actually does happen" (Licklider 1969, 51).

Software is costly and unreliable. These were the two most important features of the problems experienced by the users in the late 1960s. However, it is clear that there may be a conflict in trying to find a solution to these two problems: in certain circumstances reducing costs may have the effect of making the software less reliable. Testing software, for example, is time-consuming and costly. Boehm (1973, 52) points out that between 45% and 50% of the effort on large software projects was typically devoted to checkout and testing. At the end of any project, there is inevitably a decision to be taken on whether to spend more time and money on testing or to accept a certain possibility of software error. Thus, in his summary of a Department of Defense study of "the DoD software problem", Carlson states that:

> "Currently, software is quality controlled by testing its performance in as many situations as time and budget will allow" (Carlson 1976, 380).

The interest in reducing costs, therefore, is not synonymous with the interest in improving reliability. Furthermore, there may be differences in interest between producers and users in this respect, or between different groups of users. Alt (1969) points to this potential conflict in his discussion of the problems of software and their possible solution:

> "To say all this is a wish, not a prediction. It ought to happen, but will it happen? One straw in the wind is that there is plenty of economic interest in, and financial backing for, anything that reduces the cost of programming. But this condition does not exist for making programs safe. Thus it may be realistic to hope for a strong effort on cost reduction, leading to more specialized programming languages. But for making programs safe, for new methods of problem specification leading to higher

> quality in programming, only a continuation of the present slow rate of progress seems to be ahead. Programming errors hurt the unorganized users and, above all, the public; and they have no good way of protecting their interests. The case is somewhat analogous to automobile design, where appearance and performance pay off, but safety does not. Perhaps we need a Ralph Nader in computer programming" (1969, 16).

The difference between automobile design and software, however, is that dangerous software can do far more damage than a dangerous car.

IBM's announcement of the 360 series on 7th April 1964 had not only thrown the company itself into confusion, but it had released on the world a mass of problems arising from the unmastered complexity of the software required to run the new machines. Perhaps the most striking feature is not just the scale of the software but the bureaucratic transformation of the programming process:

> "The magnitude and complexity of the new software created immense and unsuspected development problems. It soon became apparent that little of the experience gathered in the development of earlier operating systems was applicable to the new endeavor. Close technical coordination had to be maintained over hundreds of implementation personnel; these, in turn, produced thousands of program modules which had to be integrated into a single system. These were entirely new conditions in an industry where individualistic practices had always flourished. All at once an enormous administrative and procedural apparatus had to be set in place. Naturally, development time and cost requirements soared, while pressure to meet delivery schedules caused the release of incomplete and untested products. As a result, and despite all the announcement fanfare, many users have found Third Generation software far more unreliable than the old" (Bouvard 1970, 121).

Bureaucratisation not only transformed the experience of the individual programmers and the process of programming: it affected also the definition of user requirements.

The disappointments of the computer users did not simply derive from the complexity of the software; it also had a lot to do with the nature of the demands

which they made on the software. The gap between expectations and experience expresses not just a technical problem, but a social relation, a relation between producers and users. The expectations were largely expectations promoted by the producers, the experience was the users'.

The relationship between producers and users is a complex one. It was seen in the discussion of the earlier generations that the development of the computer industry was characterised by the growing separation of producer and user. Whereas the relation between the process of production and the requirements of the user was a fairly direct one in the earliest computers, the two elements became more and more separated as the computer came to be a mass produced commodity. In the early days the user exercised a direct and continuous influence on the design of the computer, but as the industry developed, the user was progressively excluded from the process of production and design. The relation between what the producer produced and what the user wanted was no longer so direct: it was established through the market. A computer fetish was constructed, consciously and unconsciously: the producers tried to create an attractive image that would sell their computers, while the users turned to computers for the solution of problems that could not always be solved by computer.

By the late 1960s the relation between producers and users was often seen as antagonistic (cf. e.g. Laver 1970; Stevens 1970), and many users felt that their requirements had little impact on the development of either hardware or software. It is true that the manufacturers, if they are going to sell their computers, must pay some attention to the requirements of the users; but this does not mean that the interests of the manufacturers and the users necessarily coincide. Laver

(1970) suggests two examples to illustrate this point:

> "First, a supplier tends to be greatly concerned to reduce the initial cost of computer installations for this looms large in his negotiations with customers, but to do so may disadvantage the user by increasing his costs and diminishing the range of processing over the whole life of the machine. Secondly, the regular obsoleting of equipment is unsurprising as a supplier's response to a highly competitive situation, although it can also be (and often is) rationalized by presentation as an enlightened policy that brings the benefits of technical advance to users with the least possible delay. However, to produce new models every two or three years is too frequent, and merely keeps those users who have a morbid fear of being unfashionable in a costly state of flux" (Laver 1970, 104).

On the one hand, the interests of the producers are influenced by, but certainly not identical with, the needs of the users. On the other hand, the needs of the users are also influenced by the marketing of the producers.

> "This situation holds dangers. First there is the danger that users may be led into thinking of their needs in terms of improved machine performance. That they need faster, fiercer, smaller, cheaper computers is axiomatic but trivial; and when computer users focus on fascinating technicalities, whether of hardware or software, they become like those HIFI enthusiasts who, dazzled by decibels, tantalized by tracking errors and intrigued by intermodulation, hate music. The second danger arises because users' needs are plastic; they may actually come to want what computer designers want to design, or what computer marketing men want to sell... We all know now that users may need computers...for subconscious as well as conscious reasons; that stated needs may be hidden persuasions rationalised" (Laver 1970, 106).

It thus becomes very difficult to distinguish the "overselling" of Third Generation computers by the manufacturers from the "overbuying" of computers by the users. The story of the users' disappointments is not just a tale of wicked wolves and innocent sheep. The gap between expectations and experience was created not just by the fanfares of the manufacturers but also by the unreal expectations of the users:

> "Observes Donald Heaney, an internal consultant on computer usage with

> General Electric Co., 'Yes, there was overselling on the part of the manufacturers. But in companies everywhere the reasons for buying computers were not thought out. From the top, the attitude was that you can't let the competition get ahead of you; if they buy computers we've got to buy computers'" (Alexander 1969, 126).

The gap between expectations and experience is therefore not just a gap between producers and users. It is also reflected within the users themselves as a gap between uninformed hopes and the real world, or as a gap between desires and needs: "many business users may claim to be dissatisfied because they have got what they asked for instead of what they needed" (Laver 1970,106).

The users' frustrations had much to do with attempts to apply computers to new, more sophisticated applications.

Until the third generation, computers were used mainly for routine clerical and accounting purposes:

> "Computers entered industry as massive, fast, brute force arithmetic devices. They were bought because managers believed they might do known work more rapidly, accurately, and cheaply than organized sets of people. This was the EDP phase of computerization. Its greatest effect on organizations was chiefly just what was intended: a substitution of computer programs for human substructures that were then doing the same routine tasks" (Klahr and Leavitt 1967, 107).

This was reflected, for example, by the fact that within the private sector banks were the principal users of computers - in Britain it was calculated that the financial sector owned about 11% of the total computer stock (Stoneman 1976, 164). Within the field of routine accounting, however, applications were becoming more sophisticated. The introduction of magnetic-ink character recognition (MICR) encoding on all cheques in the late 1950s and early 1960s provided an important stimulus for the computerisation of record keeping for

cheque accounts (Ernst 1982, 112). Computer-based equipment for tellers was also introduced early in the 1960s to speed the processing of bank transactions. The first credit card systems were developed in the late 1960s (Ernst 1982, 115).

In the private sector more generally, computers were used mainly for the routine clerical tasks:

> "By the mid-1960s most large businesses had turned to computers to facilitate such routine 'back office' tasks as storing payroll data and issuing checks, controlling inventory and monitoring the payment of bills. With advances in solid-state circuit components and then with microelectronics the computer became much smaller and cheaper. Remote terminals, consisting either of a teletypewriter or a keyboard and a video display, began to appear, generally tapping the central processing and storage facilities of a mainframe computer. There was steady improvement in the cost effectiveness of data-processing equipment. All of this was reflected in a remarkable expansion of the computer industry" (Giuliano, 1982, 126).

Even in manufacturing companies, the first applications of computers were not directly connected with the process of production itself, but with the automation of routine administrative and financial tasks (Gunn, 1982, 88). By the late 1960s, however, this was changing. The introduction of computers into the design of products and their manufacture began to develop rapidly (Kaplinsky, 1984, p.33). Computer aided design was initially developed in the 1950s for the US early-warning nuclear defence system, but it was increasingly applied in the 1960s, primarily for military projects, and then spreading to the automobile and aviation industries by the late 1960s (Kaplinsky, 1984, 47). Similarly, numerically controlled machine tools were a technology which had been promoted by the military (the US Air Force) in the 1950s, but which really only took off commercially in the mid-1960s with the introduction of the cheaper and more

reliable computers of the third generation (Kaplinsky, 1984, 65).

In the more general area of management, attempts were being made to push computer applications beyond the performance of routine tasks. Starting in the late 1950s more complex programs were developed to automate some activities that had previously been performed with some element of judgment or had not been performed at all, such as sales forecasting, inventory control and production scheduling. Developments in this direction were supported by the development of more sophisticated mathematical techniques such as PERT (production evaluation and review technique), CPM (critical path method), simulation and dynamic programming (Gawne-Cain 1967, 158; Boehm 1962, 128). The various developments still affected only discrete, fairly routinised activities within a company structure:

> "All these avenues of advance were at first followed independently. Computers were still serving industry merely as overgrown book-keeping and calculating machines" (Gawne-Cain 1967, 158).

At the same time, however, the development of the new mathematical techniques for evaluating possible courses of action led people to speak of a new "decision theory" which could lead to a "marriage of judgment and mathematics" (Boehm 1962, 129) and provide a new rational basis for executive decision-making.

Roughly coinciding with the introduction of the first third generation machines, there is then the beginning of a third, "information systems" wave of computerisation (Klahr and Leavitt 1967, 108):

"During almost half a decade..., there has evolved from this heterogeneous growth the concept that the primary and all-important function of computers in business is management control. The separate, largely unrelated, single computer applications, already developed and currently being developed should be replaced by an 'Integrated Management Information and Control' system. This reflects a very important fundamental characteristic. The system is conceived and designed as a whole, to control an entire business organization rather than evolved as a loose combination of a number of more or less independent computer applications applied at a more tactical level of management" (Gawne-Cain 1967, 158).

For business users, one of the principal challenges of this period is to develop total Management Information Systems. Although computers were still being used predominantly in routine clerical tasks, by the mid-1960s this was seen as being relatively straightforward. From the point of view of management, it did not provide any major problems, since it involved the automation of a self-contained activity that could relatively easily be reduced to a system of rules.

> "Back in the days when corporate computer efforts centred on the conversion of accounting and administrative systems, management seldom had to concern itself with the issue of feasibility. With a relatively orderly manual system, the feasibility question centred on the technical problems of programming the computer" (McKinsey 1969, 29).

The desire to use computers for more sophisticated applications was of course prompted not only by the fact that such a development posed a challenge (although this may have motivated some programmers and data processing departments), but, much more concretely, by the pursuit of profit. By the late 1960s it was felt that the savings to be made by the further automation of clerical work no longer justified major expenditure on computers.

> "As a superclerk, the computer has more than paid its way. However, the areas (such as administrative and accounting systems) in which most companies have concentrated their computer activities are also those where the cream (and some of the milk) has already been skimmed. As a result, mounting computer expenditures are often no longer matched by rising economic returns" (Hertz 1969, 168).

The development of comprehensive management information systems was intended to achieve big improvements in managerial decision-making. The aim of such systems was to process data from the whole of the company in a comprehensive form. Information is distinguished from mere data:

> "A management information system is an assemblage of data (facts, opinions, etc) so processed (summarized, categorized, projected, etc) that it constitutes intelligence (information) for purposes of managerial decision-making and the attainment of organizational goals" (Tomeski and Lazarus1975, 104).

The ambitions of management information systems are well illustrated by a table in which Tomeski and Lazarus contrast MIS with traditional data processing:

"Data processing frequently has the following characteristics	*MIS frequently has the following characteristics*
Routine	Non-routine and novel
Procedurized	Difficult to procedurize
Accounting orientation	Extends beyond accounting
Internal data	External and internal data
Mechanistic	Some human judgment
Basic computations	Complex computations
Historical or current data	Predictive
Precision recordings	Estimates
Examples of Data Processing	*Examples of MIS*
Payroll	Collective bargaining strategy
Man-job matching	Manpower forecasting
Labor cost accounting	Human resource accounting"

(Tomeski and Lazarus 1975, 107).

The definitions of Management Information Systems vary, but the two features most often emphasised are the transition from the routine to the non-routine and the integration of information from different spheres in a form that will be useful for the process of decision-making.

The idea of using computers to construct Management Information Systems is

closely associated with the development of management science:

> "The machines in effect incubated the new discipline "management science", based upon an explicit faith in the quantitative approach to problems that had traditionally been looked upon as qualitative - and therefore susceptible only to intuition and experience" (Alexander 1969, 127).

More precisely, management science is said to involve:

> "1. Constructing mathematical, economic and statistical descriptions or models of decision and control problems to treat situations of complexity and uncertainty.
> 2. Analyzing the relationships that determine the probable future consequences of decision choices, and devising appropriate measures of effectiveness in order to evaluate the relative merit of alternative action" (Tomeski and Lazarus 1975, 114).

Since the quantification of so many variables leads to very complex computations which must be done in real time, the principles of management science only become practicable if the power of a computer is applied.

The application of management science and the development of Management Information Systems was, of course, not limited to business. As in other areas of computer development, the Department of Defense played an important role (the third generation is, after all, the time of the Vietnam War):

> "The United States Defense Department, the largest single user of computers, has been prominent in the forefront of computer-based information systems. Robert McNamara, when he was Secretary of Defense, introduced a plethora of advanced management techniques including computerized planning programming budgeting systems (PPBS) which appraised missions on a cost-benefit basis, rather than on the old line-item basis" (Tomeski and Lazarus 1975, 120).

The quantification of that which had previouly been considered qualitative, the integration of that which had previously been considered separate and the

routinisation of that which had previously been seen as non-routine obviously posed problems of a new order for programming. It is hardly surprising that users encountered enormous difficulties. Despite early optimism about what could be achieved (e.g. Boehm 1962), there was growing disillusionment by the end of the 1960s. There was a big gap between the claimed potential of MIS and the actual achievements. The happy pictures of users painted in the press were not always accurate:

> "It is widely reported that these have passed through the the tribulations of clerical mechanization and have achieved the blissful state of IMIS (integrated management information systems). A simple count, however, would probably show that only a few have gained the heights of IMIS, and that the main body is still struggling and cursing in the clerical swamps below. Those who report computer applications appear in fact to be unduly influenced by the primitive grammar of computer languages, for you will recall that these do not distinguish between the past and future tenses, nor between an intention and its execution" (Laver 1970, 107).

By the early 1970s, Morgan & Soden report that:

> "Abandoning multimillion dollar MIS development efforts is commonplace today" (Morgan and Soden 1973, 157).

Even more graphically, Tomeski &Lazarus report the laments of a company vice-president who had sponsored a study of his company's MIS needs:

> "Every year the computer department management claims it needs an expanded budget for more equipment, more programs and more staff. Yet, we have yet to obtain any management information. What information we do get is often of questionable value. The other day I received some forecasted data regarding the next year's sources of revenue. Some of the figures were absurd. I won't spend a nickel more on the operation until the mess is cleaned up" (Tomeski and Lazarus 1975, 117).

The first problem in the development of management information systems was that the software provided by the manufacturers did not provide as helpful as had

been hoped:

> "The data management and communications support facilities offered with operating systems were only of limited assistance to the user. Time sharing software was found ill-suited for use in a data processing environment where the emphasis centers more on sharing of data files than computational power between multiple users... File sharing, too, raised more complex problems which the generalized management software could not adequately resolve. It could only assist the user in implementing his own file design" (Bouvard 1970, 121).

Another problem was that effective management information systems depended on on-line processing, and the implementation of on-line systems proved very difficult:

> "On-line processing has been expensive for the most part, long in lead times and failure-prone. If one were to define the characterisitics of an on-line implementation, they might read as follows: on-line systems are traditionally installed late...significantly over budget...below performance expectations...low cpu utilization...difficult to expand" (Flynn 1974, 71).

However, the problems faced in implementing Management Information Systems go much deeper and relate to the nature of software itself. The contrast between the automation of clerical work and the development of management information systems is a contrast between "a relatively orderly manual system" (McKinsey 1969, 29) on the one hand, and a vague desire where nobody seems to have had a precise idea of what was required, on the other. Software is the reduction of processes to precise rules, but this can only be achieved if the requirements themselves are specified in a clear and orderly form.

> "Computers are a long, long way from aping the behaviour of the human mind. Before they can help with even the most routine chores, the rude chaos of the real world and the unpredictable behavior of humans at work must be reduced to an array of simple and highly specific tasks and instructions that the machine can handle" (Alexander 1969, 168).

This is the very core of software: the reduction of the "rude chaos of the real world and the unpredictable behaviour of humans at work" to an array of simple, specific rules. In the case of Management Information Systems, it was not clear just what were the processes that were to be reduced to rules and whether they could be reduced at all:

> "Management Information Systems today are not a well-defined application. Aside from saying that a manager shouldn't have to think, we know only in vague terms what a management information system is to provide. Now I don't mean to say that there aren't well-defined applications which do specific, useful things for management. There are, and they can be classified as management information systems. But too often today, the generic term management information system refers to 'terminal in the office, push the button, and get the answer to the competitive problem or next year's profit'" (Reynolds 1967, 19).

Moreover, it was not simply that the concept of the Management Information System was often very vague, but the process of management is inherently difficult to reduce to a set of rules. Managers, at least at a higher level, normally deal with decisions which do not fit into a pre-established pattern of rules, and the way that they deal with them will vary from one manager to another:

> "Because a manager deals with exceptional cases, it is difficult to predict and define clearly what he is going to do, what data he is going to need, and how he is going to use it...The analyst does not know what the exceptions are going to be until they occur. Moreover, until an exception occurs, the kind of data that will be needed to handle it is also unknown. The analyst does not know what kinds of alternatives will be developed, what can be explored and examined, nor what kind of historical data will be relevant until he knows which manager will handle the problem and how he "operates" (Gosden and Raichelson 1970, 76).

A fundamental problem in the concept of a Management Information System is the idea that the different factors to be taken into account in a managerial decision can be reduced to a common denominator, as quantities. Yet the effect of computerising the most routine business activities is to concentrate managerial

attention on the least predictable aspects of business, with the result that management becomes progressively less of a quantifiable activity:

> "This leaves top management grappling with a world of future possibilities, of people, politics, and policy. As it happens, these are the most important realms of all at a time when efficiency of operations weighs less in survival value than the ability to handle change. But this unstructured world tends to resist quantification of any kind and is certainly not reducible to a mathematical formula of the aesthetic exactness that scientists and computer people crave" (Alexander 1969, 128).

The whole idea of constructing a Management Information System is full of contradictions and the distinction made between data and information is not as simple as it seems. This does not mean that the concept of the MIS is not important. It simply means that these contradictions are carried over into the software itself.

The interface between the often vague and contradictory demands of management and the process of programming precise instructions for the computer is the work of the systems analyst or system designer. It was at this time (from the mid-1960s) that systems analysis emerged as a distinct branch of the programming profession. The task of the systems analyst (possibly in conjunction with an operations researcher or management scientist) is firstly to define the problem to which a solution is to be sought and secondly to decide on the overall structure of complex programming projects. The work of the systems analyst thus has two sides to it: in the phase of problem definition the analyst's face is turned towards the user (in this case the management of the company); the phase of deciding on the overall structure of the program involves an interface with the programmers who translate the analysts' specifications into instructions

to be fed into the computer.

In the case of Management Information Systems, however, to speak of the systematisation of the process is no help when it was often not clear what it was that was supposed to be systematised. Simon (1960, 8) distinguishes between programmed or routine decisions which can be easily automated, and non-programmed or ill-structured decisions. One of the aims of systems analysis is to extend the area of programmed (and automatable) decision making. However, it is clear that this is not just a technical task, but one which requires knowledge of the management process and also involves certain conflicts of interest.

The introduction of Management Information Systems almost inevitably involves a conflict of interest with certain managers. The aim of such systems is to alter the control structures within firms, so as to integrate areas that were previously separate. Information that previously remained within departments or sections is now to be made available more widely. One of the aims of the system is to subordinate the goals of the different parts of the organisation more firmly to the aims of the organisation as a whole. Inevitably, this means, if not a loss of power, then certainly a loss of autonomy for the managers of the different units. It is not surprising that one study of the problems of implementing Management Information Systems revealed "a great deal of massive resistance on the part of managers, particularly at the unit and subunit level" (Huse 1967, 291). Another study speaks of "aggression, avoidance and projection as the three types of dysfunctional reaction of unit managers to MIS" (Dickson and Simmons 1970, 63, cited in Tomeski and Lazarus 1975, 121). The reactions of management often led to a watering down of the idea of integrating the information between different

parts of a company as unit managers demanded, for instance, "that the programs, at the interface between different sections or subsections, come under manual control of the individual manager" (Huse 1967, 291). Inevitably, a data processing department's ability to implement a Management Information System in such a case involved a power struggle with individual unit managers.

However, it would be too simple to see data processing departments simply as a tool for strengthening central managerial control. The user companies were not monolithic units.

The commercialisation of computers, the growing distance that separated producers from users, had given rise to a distinct profession of computer and programming specialists:

> "The early computers were designed, developed, programmed, and operated at first hand by the mathematicians who wanted to apply their results: the influence of the users was immediate and continual; their needs were paramount. Then the professionals took over, computing grew into an industry, and it stratified. Designers worked for manufacturers, developing and producing what could most profitably be made and sold. Programming and operating became specialisms and, in the interests of efficiency, users - business users especially - were banished from the machine room. Users' experience of computing became vicarious..." (Laver 1970, 104).

The specialists were not just the computer manufacturers, as opposed to the users: rather, they were a distinct group within each type of company, both within the manufacturers and within the users, often moving easily from one type of company to the other:

> "Working managers feel..., not without reason, that computer experts are mercenary soldiers who are seeking high reward and fields chosen to demonstrate and develop their skill; and who remain uncommitted to any

cause but their own" (Laver 1970, 105-6).

The programming specialists had their own interests and objectives, which were not necessarily those of the company.

> "For instance, a technician's 'dream' may be a sophisticated computerised accounting system; but in practice such a system may well make no major contribution to profit" (Hertz 1969, 169).

In many cases programmers did not understand management's requirements. There were frequent complaints about the ignorance of computer staffs when it came to assessing what information was important to management. The education of programmers trained them to program computers but not to understand what information was required by managers:

> "The result of this training is to produce annually, by the tens of thousand each year, individuals who can indeed program the third-generation equipment produced, but who have little or no conception as to how organisational information is handled as part of the decision-making process for that organisation" (Trocchi 1968, 29).

Or again:

> "Computer department staffs - though they may be superbly equipped, technically speaking, to respond to management's expectations - are seldom strategically placed (or managerially trained) - to fully assess the economics of operations or to judge operational feasibility" (Hertz 1969, 169).

The limitations of the programming staff were seen largely as a question of ignorance but also as a matter of culture:

> "Most experts agree that another barrier to the most desirable use of the computer is the immense culture and communication gap that divides managers from computer people. The computer people tend to be young, mobile, and quantitatively oriented, and look to their peers both for company and for approval... Managers, on the other hand, are typically older and tend to regard computer people either as mere technicians or as threats to their position and status - in either case they resist their

presence in the halls of power" (Alexander 1969, 168).

There was often hostility or suspicion towards the computer specialists:

> "The ordinary business user has the impression that computer people do not much want to communicate with outsiders, that they enjoy a collective narcissism: nor is communication aided by the fact that computer men tend to be young and hawkish, and occasionally to theorise with breath-taking arrogance where working managers have learned to proceed with pragmatism and with hope" (Laver 1970, 105).

It was felt that programmers did not always pursue the interests of the company:

> "More often than not the systems designer approaches the user with a predisposition to utilize the latest equipment or software technology - for his resume - rather than the real benefit of the user" (Morgan and Soden 1973, 159)

In this, of course, the interests of the computer staff may be very close to the interests of the computer manufacturers. The lines between producers and users become blurred, and the tensions between manufacturers and users become reproduced within the user companies themselves.

From the point of view of senior management, therefore, the development of management information systems did not simply involve using computers to increase their control over the company. They also had to fight to assert control over the computer staff. This was made all the more difficult by the explosion of programming associated with the third generation. The shortage of programmers meant that it was necessary for management to offer not only good salaries but also good conditions of work. This was also a period in which more and more computer departments gained organisational autonomy - previously, many had been subdivisions of accounting or financial departments.

The assertion of managerial control meant that the question of the feasibility of computer applications could no longer be treated as a technical matter which could be safely left to the computer staff. It required managerial intervention, as McKinsey & Co, the management consultants, pointed out in their 1969 report:

> "Today the situation is very different. Applications are not only more complex, but also more far-reaching in their impact on different operating departments. Feasibility is no longer an issue that operating managers can ignore, for it is affected by complex economic and operational questions that the staff specialists are unequipped to answer. Yet many managers - far too many - are still leaving the whole question of feasibility to the computer professionals" (McKinsey 1969, 29).

It was necessary that the managers themselves should become involved in the process of automation and impose the interests of economic feasibility:

> "Many otherwise effective top managements, however, are in trouble with their computer efforts because they have abdicated control to staff specialists - good technicians who have neither the operational experience to know the jobs that need doing nor the authority to get them done right. Only managers can manage the computer in the best interests of the business. The companies that take this lesson to heart today will be the computer profit leaders of tomorrow". (McKinsey 1969, 33)

The problem of making computers do what they are supposed to do - contribute to the users' profitability - was thus translated into a question of control. The managers must take priority over the computer specialists, the criteria of economic feasibility must take priority over problems of technical feasibilty. For the manager, as Hertz points out, the interests of the company must take precedence over questions of technical elegance:

> "He must make the technology work for the company and not use the company's resources merely to support an elegant systems effort" (Hertz 1969,171).

The examination of the problems of the users thus brings the discussion back

to the central theme of earlier chapters. The tension between scientific-technical considerations and economic interests which can be seen in the conflict between FORTRAN and ALGOL, and again in the deliberations of the SPREAD Committee, and again in the history of the operating systems, reappears here as a question of control within the user companies, as a conflict between programmers and managers. Inseparably entangled with this tension between the technical and the economic is the question of the nature of software itself, the reduction of the "rude chaos of the world" to a set of rules.

Chapter 5

The Commodification of Software

Bundling

The difficulties experienced by both producers and users with Third Generation computers led to a new awareness of the significance of software. The new awareness of software found expression in its growing commodification at the end of the 1960s.

As was seen in the discussion of operating systems, it had become the practice for computer manufacturers to provide the software together with the hardware. In the earliest days of computers, the manufacturers would simply sell the computer itself, which would be programmed by the user: neither operating systems nor applications packages existed. As computers developed, however, and particularly with the advent of operating systems, it became established practice for the computer manufacturer to provide both operating systems and applications software "free" with their machines. This practice was referred to as "bundling":

> "Bundling is 'the offering of a number of elements that are considered to be interrelated and necessary from a customer's point of view, in the computer field, under a single pricing plan, without detailing the pricing of the component elements themselves'. The elements that were offered without a separate price were nonhardware items such as education, software, systems design and maintenance" (Fisher et al 1983, 172).

Thus, users bought a bundled package in which hardware and software were not priced separately. Software was regarded as purely secondary, as part of the service that users received when they purchased a computer.

"The positive reason behind bundling was you were selling or leasing the customer a complicated piece of equipment customers typically didn't understand and what the customer bought was not a piece of hardware that would sit there but some assurance that it would do what you said it would do and so, what you sold him was the assurance. One can really not imagine an IBM salesman in, let's say, the 1950s going out and saying 'By the way, after you have bought this thing, then I'll sell you the stuff that'll actually make it go" (Fisher interview).

Although it appeared that the manufacturers were providing a service free of charge to the users, in fact the practice of bundling had fundamental advantages for the manufacturers. Firstly, it played an important part in expanding the market for computers. One of the main obstacles to the development of the computer industry in the early days was the newness and unfamiliarity of computers. No matter how good the hardware was, the market would be limited until customers had learned to use the new machines. The provision of software and other support services was an important means of overcoming this barrier. The computer manufacturer dealt with the user not only through the salesperson, but also through the systems engineer. The systems engineer would work with the user to work out a hardware-software system configuration tailored to the user's needs and to write the specific applications software.

Even after the initial unfamiliarity had been overcome, bundling continued to be an important part of the marketing of computers. One of the reasons for the success of IBM was the extent to which it provided such bundled services as part of its marketing effort:

"At IBM the provision of bundled support began before the installation or even the acquisition of a computer by the customer. Such support was viewed both inside and outside IBM as an essential part of the marketing effort. The IBM systems engineer...worked with customers to define requirements, in system design, developing approaches to problems; they

also engaged in customer education and training, and in programming" (Fisher et al 1983, 172).

The practice of bundling was an important means by which manufacturers protected themselves against the risks involved in the leasing system. The fact that each manufacturer had its own way of writing software, forced users to be loyal to a particular manufacturer: switching software involved not only the costly process of rewriting programs, it also meant the upheaval of reorganising processes that had been shaped around a specific software configuration. Thus, the differences in software made it difficult for users to switch from one manufacturer to another.

Quite apart from the benefits that the practice of bundling brought for manufacturers, it was difficult in the early days to imagine how software could be marketed separately. People did not think of software as a thing that could be sold. Indeed, the generic term "software" did not exist before the 1960s. Software was seen as something that you shared with people:

> "The negative reason behind bundling was that it was very hard to imagine how to sell software as property. It's still quite hard, not to imagine, but to enforce property rights in software, the way people just copy - people with personal computers do that all the time. The concept of that was foreign... In the early days, software was something you wrote and shared with people" (Fisher interview).

This assumption was expressed clearly in a letter by Galler of the University of Michigan to the editor of the *Communications of the ACM* in 1960:

> "It has come to my attention that a 704 statistical program has been produced at Arizona State University which apparently does a pretty good job on factor analysis and a few other things. Unfortunately, the letter I saw indicated a charge of $3 for the manual, $4 for one box of binary cards and $20 for four and a half boxes of SAP cards (plus postage).
>
> I believe that this has very unfortunate implications for the computing

profession. When one has to cover printing and card costs, that is one matter. In this case, however, it is clear that what is being charged for is the development of the program, and while I am particularly unhappy that it comes from a university, I believe it is damaging to the whole profession. There isn't a 704 installation that hasn't directly benefited from the free exchange of programs made possible by the distribution facilities of SHARE. If we start to sell our programs, this will set very undesirable precedents" (quoted by Frank 1976, 92).

During the 1960s, the conception of software began to change. There were various developments that contributed to the emergence of software as an independent commodity.

Firstly, the difficulties of producing increasingly complex software, and particularly the traumatic experience of the 360 software, led to a new awareness: producers realised just how difficult software was to produce, and just how costly. In the case of IBM, not only OS/360 itself, but the whole package of bundled support for the 360 cost much more than the company had expected:

> "The training of the enlarged marketing staff, the support required by users to effect their conversion to the new and sophisticated operating system software associated with System/360, and the problems that IBM encountered with some of the 360 software caused the firm to devote an enormous portion of its resources to supporting the installation of System/360" (Fisher et al 1983, 176).

Although these costs were passed on to the users in the long term, it was IBM which had to bear the immediate costs involved. Within IBM, there was growing concern about the returns on their ever-increasing software expenditure: this led in 1966 to an attempt to quantify these returns (Fisher et al 1983, 176).

As the area of computer applications extended, there was also a growing community of sophisticated users who felt that they did not need the same level of support from the manufacturer and often preferred to write their own software.

These users resented having to pay for the whole package: what they wanted was "a lean, hard machine". This was seen already in the case of the 360/44 model, where IBM responded to users' demands by providing a machine without the usual software. As Thomas J. Watson later testified:

> "We had some very sophisticated customers by this time, Lockheed, Boeing and others, who felt that they were better at performing some of these services than we were. They felt it onerous to pay for them when they, themselves, could do it in their opinion better" (quoted by Fisher et al 1983, 176).

Another thing that helped to change the conception of software was the development during these years of independent software houses. The first software houses were founded in the late 1950s and early 1960s as consultants for the military and space agencies. By 1965 there were about 40 to 50 independent software vendors in the US and it was during the period of the third generation computers that software houses expanded. Software houses were then not just involved with government contracts. As computer manufacturers could not cope with the production of the software for the third generation computers, they began hiring consultants and university professors. In fact a lot of the system software for the third generation computers was done by software houses (about 30 to 50%). With the expansion and complexity in the use of computers, users increasingly began to use the services of software houses for their specialised applications. The software houses emerged as a bridge over the growing gap between producers and users. It was relatively easy to start a software house - not much capital was needed - and a lot of companies were formed by people leaving the larger firms. By 1968 there were nearly 500 software houses in the USA and their number was increasing.

Software houses influenced the conception of software: "the growth of the independent software industry showed that in fact it was possible to offer software separately as a product" (Fisher interview), that it was possible to price software and services independently of hardware. The fact that computers were sold or rented bundled created a barrier to the expansion of the software houses.

The concept of software as a product was also associated with the growing importance of software packages. Software packages were originally developed by computer users. The sharing of experience in users' groups led to "the concept of developing programs of general applicability and thereby saving the time and money that would otherwise be wasted in 'reinventing the wheel'" (Head and Linick 1968, 22). Packaged software had advantages for users not only in terms of costs but also in terms of standardisation:

> "Multi-divisional companies began to appreciate the advantages of application software which could be used by all divisions. This led to standardisation of operating and clerical procedures combined with increased generality in the computer programs" (Head and Linick 1968, 23).

As the advantages were realised, competitive pressures led manufacturers to provide application packages with their machines. The problems associated with the advent of third generation computers gave a boost to the development of software packages:

> "Manufacturers have been hard pressed to provide essential system software. Users have had to expend huge programming resources in conversion efforts. And, of course, good programmers are in very short supply" (Head and Linick 1968, 23).

In addition, compatibility between computers and the use of common programming

languages helped to open up a much wider market for application packages. Computer manufacturers were unable to meet the increasing demand satisfactorily and by the end of the 1960s many of the application packages were being supplied by software houses. By this time, software houses were selling not only applications software but also operating systems, such as the Mark IV System, an operating system developed by Informatics Inc. to be used with the IBM 360 computers (Postley 1968). Despite the fact that the products of software houses were sold on the market, while the manufacturers' software was offered free of charge, the superior quality or the greater specialisation of the software houses' products often made it worth while for the users to buy them.

Property Rights

The development of software packages brought to the fore another issue that had a bearing on the changing awareness of software: the protection of property rights. This problem was highlighted by a UK scandal concerning the "plagiarism" of sections of the British Overseas Air Corporation's $100 million international airline reservation system. It was alleged that some employees of BOAC had expropriated information for consultancy work and passed it to a software house which used it to win a public contract. The scandal caused "considerable consternation among manufacturers and software houses" (*Datamation* June 1968, 91).

The whole issue of property rights in software has been a topic of intense debate since that time. It was not clear whether software could be regarded as property and just what this would mean. To make this point clear, Dansiger makes an analogy with the protection of musical melodies. He cites a legal action in which

the composer of a tune called *Moonglow* claimed that the theme music of a Hollywood movie called *Picnic* was based on his melody and that he was therefore entitled to a large part of the royalties. The decision of the judge on whether the theme of the film was basically a copy of *Moonglow* or not was "necessarily subjective", because it is often very difficult to determine the degree of originality in music:

> "The music business has suffered for many years from this problem. No objective method has ever developed for settling these disputes" (Dansiger 1968, 32).

In just the same way, it is often very difficult to determine the originality of programs: "many claims have been made that one program is merely a slight reworking of another program previously in existence" (Dansiger 1968, 32).

Even if this problem could be overcome, it was still not clear just what would be covered by the protection of software: would it be the detailed coding of a program or the logic flowchart, or the results, or perhaps the algorithm underlying the program? (Davidson 1968, 12). Moreover:

> "To complicate matters, we have not so much an undefined product as a product whose definition keeps changing all the time. Software can be considered from several levels. It can be simply the program, as it appears in a deck of cards or on a reel of tape. It can be this plus the research effort which has gone into an over-all study of the problem" (Bigelow 1968, 32).

The two main existing forms of "intellectual property right" were copyright and patent. Under the US Copyright Law, the author or other copyright owner has exclusive control for 28 years of the right to reproduce the form of expression. Under the Patent Law inventors have exclusive right to control the use of their

inventions and the methods embodied therein during a period of 17 years. Copyrights are registered and any disputes come only after the registration; in the case of patents, the patent is applied for and the Patent Office examines the application to see if it meets the criteria of being (a) not already invented and (b) not obvious (Bigelow 1968, 32).

The US Copyright Office decided in May 1964 that computer programs could be registered if certain requirements were met. However, the acceptance of programs for registration did not necessarily give legal protection to the programs because the Copyright Office still took no position on whether a program could be considered to be "a writing of an author". The debate therefore continued on whether programs should be copyrightable. Opposition to the idea came mainly from the Interuniversity Communications Council (EDUCOM). EDUCOM claimed that if copyrights were permitted, it would be unlawful to use the program in a computer without the permission of the copyright owner. If copyright had existed from the beginning, they argued, so that program preparation had "been constantly carried out under the threat of infringement actions charging plagiarism of existing copyrighted programs, it is doubtful whether the growth of programs and programming techniques of recent years would have been possible" (quoted by Bigelow 1968, 33).

The response of the Copyright Office was that these fears were ill-founded:

> "The case of *Baker v Selden* decided by the Supreme Court in 1879 said, 'the copyright of a work on mathematical science cannot give the author an exclusive right to the methods of operation which he propounds, or to the diagrams which he employs to explain them, so as to prevent an engineer from using them whenever occasion requires'" (Bigelow 1968, 33).

By analogy with this case, the Copyright Office argued that the use of a copyrighted program would not be an infringement of the copyright whereas its reproduction would be. In effect the protection offered by copyright is limited to manuals, advertisements, and other documentation.

- Most of the discussion of the protection of software property in these years focused on the question of patents rather than copyright. The initial approach of the Patent Office was that computer programs were not patentable because they were not "methods or apparatus, but rather mathematical processes or formulae" (Bigelow 1968, 33) - which traditionally had been held not to be patentable. In August 1966 the Patent Office modified its position slightly by issuing guidelines which stated that a patent could be granted to a program if it could meet the requirements of either a "process" or an "apparatus".

These guidelines were discussed in a public hearing in Washington in October 1966. At this meeting the discusion focused around the question of whether computer programs deal with "mathematical" or "functional" entities:

> "The guidelines, which deserve a prize for murky syntax, seem to say that a program as usually written is an algorithm - i.e., a mathematical process - and hence unpatentable. But they go on to say that if the individual steps are described in functional terms- as changes 'in the state of certain electrical or mechanical devices within the computer'- the program would be eligible for patent consideration" (Hirsch 1966, 79).

As a number of the participants pointed out, such a distinction seemed unsatisfactory because any program written in algorithm terms could be patented simply by rewriting the algorithm in functional terms.

At the hearing the guidelines were universally criticised, but there were different views about the desirability of patents. Software users at the meeting were against the guidelines but in favour of patents. The computer manufacturers, however, were against any form of software patenting. Their argument was basically that software could not be seen as autonomous from hardware. This view was put forward by a representative of Honeywell:

> "He argued that a computer program can't be new, in the sense meant by the patent statute, because the corresponding logic design is 'contemplated fully in the design of the computer...In evaluating a computer program...it should be kept in mind that the hardware...at any particular instant..will be under the control of a program instruction which will...establish pre-determined...circuit paths. At that particular instant, the paths...established are (those) fully contemplated by the designer of the equipment'" (Hirsch 1966, 81).

The introduction of patents, manufacturers argued, could spoil their relationship with users. Manufacturers often developed their programs in cooperation with users and the introduction of patents could lead to arguments over ownership rights.

There were very few software houses represented at the hearing, but the head of a software firm interviewed after the meeting said that he was not interested in patents because "the programs we develop change too quickly. By the time a patent on a particular program was granted, we'd be using an updated version". Some software houses were in favour of patent protection, he added, but mainly because "it gives recognition to the programmers who do the work, and improves the company's image, not because of the property rights involved". Other software houses who produced relatively long-lived programs on contract, were interested in property rights, but, like the manufacturers they "have to worry about

customer relations" (quoted by Hirsch 1966, 81).

Over the period of the next few years there was a gradual shift in the position of the Patent Office. In April 1968 a patent was granted to Applied Data Research for a software sorting system. The president of the company claimed that this indicated that "software systems and programs are entitled to patent protection in much the same way as computer hardware...The issuance of a software systems patent is another milestone in the coming maturity of the software industry" (Richard Jones, quoted in *Datamation* July 1968, 91). However, an official of the Patent Office, when asked about the patent, stated "we do not think a program is patentable" (Erwin L Reynolds, quoted by Bigelow 1968, 34). It was not until the end of 1969, after a number of court cases in which the Court of Patent Appeals had overruled the Patent Office (*Computers and Automation* Jan 1969, 72), that the Patent Office finally revised its guidelines and announced that in future patent applications for software would be considered (*Computers and Automation* November 1969, 11).

In reality, the practical implications of patenting were limited. The development of software has been too rapid, and the machinery of patent protection too slow, for patents to have had a major significance in protecting software (Brock 1975, 64). In practice the most common way of trying to protect software has been through secrecy supported by contractual provisions either between suppliers and users or between employers and programmers. However, in an industry that has been developing so rapidly and in which people change frequently from one company to another, there are limits to the effectiveness of legal protection:

> "As a practical matter, the best protection for the software developer is to deal with an honest man, give him a square deal, and trust him" (Bigelow 1968, 38).

In the case of the BOAC scandal, the immediate response was to see the remedy not in terms of enforcing property rights, but in terms of "reinforcing codes of professionalism in an industry that is expanding so rapidly" (*Datamation* June 1968, 91).

Nevertheless, the discussion of software patents and property rights is very interesting because it reflected and legalised the changing conception of software. The acceptance that software could be patented was a victory for the argument that software was not simply the counterpart of the computer's design but something autonomous in its own right; and also for the view that software can be seen as closer to engineering than to mathematics.

Unbundling

The changing conception of software played an important part in preparing the ground for a change in the practice of bundling. The realisation of the problems and costs of producing software; the experience of the 360 and other installations; the growing sophistication of users who did not always want the whole bundle; the legal recognition of property rights in software: all pointed the way to the unbundling of software from hardware.

By the late 1960s there were increasing pressures on IBM to unbundle. These pressures came from some users, who wanted greater freedom from the manufacturers, and from software houses, who saw the practice of bundling as a barrier to their growth (Gilchrist and Wessel 1972, 8). These feelings were

given open expression in a meeting in May 1968:

> "Five revolutionaries advocated the overthrow of the computer industry before almost 600 witnesses at the SJCC [Spring Joint Computer Conference] last month. Members of the panel convened to discuss separate pricing of hardware and software, they suggested such blasphemous thoughts as forcing IBM (and the other manufacturers) to price software separately...and spinning off IBM's software production activity into a separate company" (*Datamation* June 1968, 72).

The "five revolutionaries" included the representatives of three important software houses, the director of the Center for Computer Sciences and Technology at the National Bureau of Standards and the president of SHARE, IBM's biggest users' association. They argued that the current pricing practices were in violation of antitrust laws. If IBM were to unbundle, it would lead to an increase in competition which would result in better quality software. One of the speakers argued that manufacturers have "little incentive to offer anything more than what will make the machinery function adequately". Separate pricing, he argued, "would offer the competition and the incentive to produce 'outstanding' software support" (*Datamation* June 1968, 72). The president of SHARE emphasised that "ultimately,...it will be the computer users who will have to insist on the separate pricing of hardware and software" (*Datamation* June 1968, 72).

In the same month, some software houses united to establish the Association of Independent Software Companies, with the aim of fostering their interests. The establishment of the Association was said to be "a sign of the continuing maturity of the computer software field" (*Computers and Automation*, July 1968, 14). One of their priority concerns was "the question of separating procurement and pricing of hardware and software" (*Computers and Automation*, July 1968, 14). Their first president was one of the "five revolutionaries", Richard Jones of

Applied Data Inc. In July of the same year, Jones published a letter calling for the separation of hardware and software pricing:

> "The issue of separating the pricing of hardware and software is the issue of free competition. In virtually every phase of our lives, we favor competition and the right of the consumer to make a buying decision. Our country was founded on this basis, and our legal system, presumably is constructed to protect this right" (Jones 1968, 12).

He argued that, apart from being in illegal restraint of trade, the current pricing practices meant that the user was paying for lack of competition in at least five areas:

> "1. The *pricing of equipment* includes the cost of highly specialized packages which are applicable to a very small segment of user population.
> 2. The *efficiency of software* is an issue over and over. The introduction of competition by companies who sell software on its merits would place emphasis where it belongs.
> 3. The *amount of equipment required* is of primary concern to manufacturers. Software that does not sell equipment or that reduces memory and peripheral requirements beyond those otherwise required are not in the manufacturer's interest.
> 4. *Excessive user personnel and unnecessary costs* result from the present approach which includes over- elaborate, difficult-to-use systems software.
> 5. The *maintenance of software* currently leaves much to be desired. Placing the economic emphasis where it belongs would improve the situation" (Jones 1968, 12).

The idea that unbundling could make a contribution to solving the problems of software was a major issue at a NATO Conference held in Garmisch in October 1968. At that conference, unbundling was such a hot issue that a special session on software pricing was arranged in response to demand. The session went on until well after midnight, but it was not fully reported in the published conference proceedings in order to preserve participants' anonymity:

> "The whole topic was regarded as so sensitive that this was the one session for which the conference report merely records the various arguments

put forward without attributing them to individual speakers" (Randell 1979, 2).

This "sensitivity" had to do with worries about the dominance of IBM:

> "Within the software engineering conference, amongst a number of people, there was an undercurrent of worry about the dominance of IBM, and that came out most obviously in the unbundling session. It was because of that commercial sensitivity, if I recall it correctly, that we made the deliberate decision not to attribute comments to particular people" (Randell interview).

The large majority of the participants in the discussion at Garmisch were in favour of the separate pricing of software (GCR 130). It was accepted by most people that software was "a sort of commodity":

> "The common thought was that software was a sort of commodity. It was a valuable commodity, it was a commodity whose reliability and whose timeliness mattered...it was a commodity rather than the almost accidental outcome of somebody writing instructions because he wanted to use the computer" (Randell interview).

This view, however, was not accepted by all the participants. One of the arguments against separate pricing was that "software belongs to the world of ideas, like music and mathematics, and should be treated accordingly" (GCR 132).

Although most participants saw software as a "sort of commodity", it was not clear what the consequences of unbundling would be: "people really didn't know what unbundling would do" (Randell interview). The discussion at Garmisch was centred on the relation between the market, power and the quality of software. At one level, the call for the separate pricing of software was clearly an attack on the power of IBM:

> "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 organisation with sufficient capital resources producing comprehensive software for any industry, educational activity, research organisation or government agency are considerable and far outweigh, for intance, 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" (GCR 132).

Other participants, however, warned that unbundling would not necessarily reduce the power of IBM:

"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" (GCR 132).

The increase in competition would not necessarily be to IBM's disadvantage:

"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" (GCR 132).

Related to the question of power is the question of quality. The practice of bundling and the power of IBM meant that many improvements in software were not widely diffused. Unbundling would mean a much better utilisation of talent:

"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 utilized. It is these sources and not IBM which have produced the majority of good systems and languages, such as BASIC, JOSS, SNOBOL, LISP, MTS" (GCR 131).

The quality of software, rather than any reduction in the price of hardware, was seen as the central issue. It was assumed that an improvement in software quality would result from the opening of the market to competition and from the clearer definition of the value of software:

"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" (GCR 130).

John Buxton, one of the participants in the conference, recalls that he made a similar argument at Garmisch in favour of unbundling:

> "As I recall, I was lobbying for it because...I think my own argument was that software at the time was not regarded as sufficiently valuable, and I felt that the crisis was so great than one ought to attach to software the right economic value, otherwise you weren't going to put the right amount of money into solving it. So that was my rationalisation for lobbying for unbundling, and I'm sure IBM were influenced a lot by that particular meeting. I was of course only half right and maybe even half wrong" (Buxton interview).

The reason that he was "only half right and maybe even half wrong" was:

> "What I'd not really worked out was that the cost of reproduction of software is of course almost zero. Therefore, if you sell a million copies and it costs you a million pounds, you will sell each copy at one pound, so what I hadn't worked out was that the consequences of unbundling would be that IBM software cost a thousandth of everybody else's software, because they sell a thousand times the number of machines. The IBM people at that meeting were sharper than I was and they spotted it imediately, and I recall some of them saying, 'You know, there could be some unexpected consequences if we unbundle'" (Buxton interview).

This argument was spelt out in much greater detail in an article published the same month by Conway, which Brian Randell (Professor at Newcastle University and also one of the participants in the Garmisch Conference) in 1979 referred to as "by far the most reasoned contribution to the debate" (Randell 1979, 2). The central point is that the relation between design costs and reproduction costs makes software a very special commodity.

The total cost of any mass-produced commodity consists of two elements: the design cost, which will depend on the nature of the article, its complexity, need

for reliability, etc; and the reproduction cost, which will depend to a large extent on the materials incorporated in the product. The cost of an individual article will consist of its individual reproduction costs plus a share of the total design cost. In the case of most mass-produced articles (a car, for example), it is the cost of reproduction that dominates: the share of the design cost is not significant. In the case of software, however, the relation between the design cost and the reproduction cost is the reverse:

> "Software, viewed as a mass-produced article, is strange because the design cost D can be very high and the reproduction cost R very low. For software which has a small sales and support cost the reproduction cost is essentially the cost of copying a tape or punching a deck. Thus the per-unit cost of unsupported software approaches zero as the volume becomes very large" (Conway 1968, 29).

It follows that the per-unit cost of software is peculiarly sensitive to volume:

> "Everything else being equal, the software producer with the highest volume will have the lowest per-unit costs. This effect is much more pronounced than in conventional mass-production equipment manufacturing where the reproduction cost dominates the design cost and the effect of volume on per-unit cost is only a second-order effect" (Conway 1968, 30).

As a result, it is the biggest producers who will probably gain most from unbundling:

> "When the edict comes to separate the pricing of software and hardware the lower-volume manufacturers may have to charge more for their software. This could have two consequences. First, it could force the smaller manufacturers into a hardware price war against IBM, a war they can not in the long run win. Secondly, it could lead consumers, other things being equal, to prefer machines which will run IBM software. Independent software houses will concentrate on developing proprietary software for 360's because of the larger potential market: this concentration, which favors the 360 owner is already quite evident" (Conway 1968, 31).

The long term effect of unbundling could, therefore, be the standardisation of hardware and software around IBM's products.

It was not only the small manufacturers but also the small users who could suffer as a result of unbundling, Conway argued. The system of bundled pricing meant that the larger users paid for a portion of the total software cost with each machine that they purchased. With separate pricing, they would pay only once for the software. As a result the small users would have to bear a larger proportion of the total software cost.

Another consequence of unbundling could be that users would no longer receive the support of the manufacturer in installing a new system or in integrating different components into a computer system. In this case the user would probably turn to a software company for support:

> "The major significance of separate software pricing, then, is that it promises to drive a wedge into the marketing relationship which now exists between the user and the computer manufacturer. The wedge is labeled 'prime contractor' and the role is most probably occupied by the software houses" (Conway 1968, 31).

It is clear that, although the call for unbundling seems an attack on the power of IBM, the matter is not so simple. There were pressures on IBM, but it is not clear that the pressures were contrary to IBM's interests.

The various pressures that were building up by 1968 had a certain influence, but they lacked real force as long as the state did not intervene. The US government might have used its power as a major purchaser of computers to enforce a separation of the pricing of hardware and software. In fact:

> "The General Services Administration and a few other government agencies did suggest that hardware manufacturers offer software separately, but only in a limited way. Moreover, such governmental 'unbundling' requests appear to have been motivated solely by the desire to obtain an effective price reduction and without any appreciation of the effect it would have on the economic pattern of the computer industry and resulting indirect costs" (Gilchrist and Wessel 1972, 9).

The government did not intervene effectively as a purchaser. The important area of intervention was through antitrust regulation.

Concern about the monopoly position of IBM was, to a great extent, a result of the 360 experience. The 360 family had been a success, it had consolidated IBM's position but it had taken IBM into a new era, an era of litigation:

> "Never before or since had any company been so involved in such a variety of legal actions simultaneously. Around Armonk, jokesters obseved that Watson senior had stressed sales, and Tom Watson, technology. The next chairman would have to be lured from the Supreme Court, for the company had entered an age of litigation" (Sobel 1984, 253).

As soon as the 360 family was announced in 1964, IBM's competitors had alerted their law firms and asked them to prepare the ground work for appeals to the Justice Department and the courts. IBM responded by starting to assemble its own strong legal team. Among the recruits to its legal staff was the former head of the Antitrust Division of the Justice Department during the Kennedy years and a former Attorney General. Later, when litigation began, "competitors complained that the very same individuals who had prepared the case for the government were now in IBM's employ" (Sobel 1984, 224). There is an anecdote that once Norris, the president of Control Data, was looking out the window with a friend at a long line of black limousines. "'Someone important must have died', said the friend. 'No', snapped Norris. 'That's just the IBM lawyers going out to lunch'" (Sobel

1984, 224).

The Justice Department started working on a possible antitrust suit againt IBM in 1965, but there was a split within the Department "on how best to proceed, on which of IBM's practices should be challenged, and even on whether or not the entire matter should be dropped" (Sobel 1984, 261-262).

Pressure continued to build up on the Justice Department to intervene. Competitors claimed that IBM's conduct thoughout the sixties had been against antitrust law in various ways. Software houses calling for unbundling had been putting pressure on the Justice Department to take action (Jones, 1968, 12). Computer manufacturers had been doing the same. Control Data in particular complained about IBM's practices, claiming that IBM had prematurely announced the 360/90 model and had priced it below costs in order to stop customers from buying Control Data's 6600 model. Control Data had collected information for a possible antitrust suit, which it submitted to the Justice Department in an effort to get the government to take action. The company maintained contact with the Justice Department from January 1966 to December 1968, when it was informed that no legal action was contemplated by the Department. In view of the inaction of the Justice Department, Control Data filed a private antitrust suit on December 11, 1968 (Brock 1975, 170; *Computers and Automation* Jan 1969, 72). The following month, on January 17, 1969, shortly before the end of the Johnson administration, the Justice Department filed its own suit against IBM. This suit was to last for more than a decade during which IBM was "obliged to amass no less than 46,726 tons of documents and incur untold legal costs" (Sobel 1984, 276); the case against IBM was dropped at the beginning of the Reagan

administration.

The Justice Department charged IBM with monopolising the general purpose computer market. The complaint was against four specific practices that were said to contribute to the monopolisation.

One of these charges was directly related to the complaint of Control Data. It was claimed that the announcement of the whole 360 family was an anticompetitive practice. According to the Justice Department the 360 family had been announced prematurely: the company had announced the future development of new models for competitive markets "when it knew that it was unlikely to complete production within the announced time". Further, in the case of some of the models where competitors were very successful, they had been announced as "fighting machines" (Models 360/90, 360/67 and 360/44), with prices deliberately set below costs (*Computers and Automation* Feb 1969, 8).

Another charge was that IBM had been dominating the educational market by granting "exceptional discriminatory allowances in favor of universities and other educational institutions" (*Computers and Automation* Feb 1969, 8). The educational market was important, it was argued, not just because it was large but because of the future influence that it had in the choice of computer manufacturer: "universities also often provide either ideas or programming which lead to useful commercial enhancements to the machines. In addition, universities are often very visible installations, providing substantial public relations value for the computer manufacturer" (Brock 1975, 158).

The remaining two charges related directly to software. One of these was that IBM was using its accumulated software and related support to preclude other computer makers from competing effectively for customers.

The other attacked the practice of bundling: the fact that IBM had maintained a policy of quoting a single price for computer hardware, software and support had limited the development of an independent software and support industry (*Computers and Automation* Feb 1969, 8). This charge was a response to the arguments of the software houses, some users and manufacturers.

Just before the Justice Department filed its suit, IBM announced that it was going to announce a change in its policy. On December 6, 1968, IBM announced that no later than July 1, 1969, it expected to "make changes in the way it charges for and supports its data processing equipment. The company had been "re-examining its methods of doing business in the United States to determine what support services should be separately offered and priced to better meet the future requirements of all users of IBM equipment" (*Computers and Automation* Jan 1969, 73).

Some idea of the upheaval within IBM is given by the account of one IBM employee who had been working on a new IBM computer:

"The product looked ready for announcement in December 1968. Then Joyce's boss announced he was going on a task force which would meet near group headquarters in Harrison. The purpose of the task force was top secret, but the rumor was that IBM, under that threat of incipient antitrust charges, was about to revise its price structure and start charging the customer for systems engineering, applications software and training courses. A week later Joyce received a call from his superior. 'He told me to forget everything we'd worked on for the past two years.

> IBM was going to *unbundle* and we'd have eight weeks to do the whole thing over'...The two-story building in Harrison now housed some two hundred men. 'It was under heavy guard, and you needed a badge to get in', Joyce recalls" (Fishman 1982, 135-136).

There were many practical problems to be solved before unbundling could take place. IBM would have to reduce the price of its hardware, but by how much? How were application programs to be priced? What were all the support services worth to the customer bundled and unbundled? The task force had a lot of work to do sorting out the practicalities of unbundling (Fishman 1982, 137).

On June 23, 1969 IBM unbundled: "the date is engraved on the hearts of the leaders of the software industry" (*Financial Times* Sep 30 1983). The company announced that in future a charge would be made for systems engineering activities, customer education courses, and new "program products as distinct from system control programming". Thus, it was only the application software that was being unbundled: operating systems were considered to be too much an integral part of the whole system to be unbundled:

> "When they unbundled in '69, they didn't unbundle the operating system, of course, because they thought at the time that that was just not feasible, there was no way you could separate the design of the operating system from the design of the computer. As time has gone on, they have discovered more and more ways to do that, they've unbundled more and more of the software" (Fisher interview).

At the same time, IBM announced that it was reducing both the leasing and sale price of its hardware by 3 percent, "stating that this reflected its 'best approximation' of the expenses that would 'no longer be provided for in prices of currently announced equipment" (Fisher et al 1983, 175; *Computers and Automation* July 1969, 8).

IBM's decision to unbundle is commonly seen as a response to the Justice Department's antitrust suit, but there are some who deny that the Justice Department had any influence:

> "Everybody speculates that it was the filing of the government case, that it had something to do with that. It's a perfectly plausible explanation, but I know not a shred of evidence that that is true, not a shred, and I expect the government would have produced it if there had been a shred. I can't imagine that it wasn't at the back of their minds, but there is really nothing to support that" (Fisher interview).

As can be seen from the whole development of commodification, the reasons why IBM unbundled were more complex:

> "They had to. Well, they had to and they wanted to. I think there were some people who saw an excellent opportunity for developing essentially a new business. There was a lot of concern internal to IBM, I remember this very well, about the possible effects of unbundling. There were those who saw it as a very positive thing to do. I think the majority knew it was something that had to be done to placate the Justice Department, but I think there were some who said 'well, we have to do this, all right', but they really wanted to do it, you know, to generate additional profits, to make a profit out of doing it" (Rasmussen interview).

Unbundling was not just a response to the Justice Department or to IBM's own interests, but confirmation of the changes in the conception of software that took place during the period of the Third Generation computers. Unbundling confirmed the emergence of software as an independent commodity: "unbundling made it more a commodity" (Randell interview).

The immediate reactions to IBM's announcement were very mixed. In August 1969 *Datamation* published an article under the heading "Industry Reacts with Approval and Dismay as IBM goes Separate Ways". The author reports that IBM's announcement has:

"Upped the giant's long-term income expectations.
Wounded the user, who is bewildered and moaning - but oh, so softly and ineffectually - about the prospect of increased costs.
Given the leasing companies a break for now, but false sense of hope for the future.
Opened significantly more of the software market to competition than had been dreamed.
Satisfied firms in professional edp education.
Pleased - perhaps - the Justice Department.
Forced more choice on mainframe makers in the setting of their own policies.
And earned the IBM task forces that designed the policies a week's vacation for a 'superb' job" (Pantages 1969, 105).

For users, unbundling had varying effects, depending on their size and sophistication. The immediate reaction of many users was hostile: it was estimated that for most users computing costs would rise by about 20% because of the additional prices they would have to pay for programs, systems engineering and education. Moreover, it was pointed out that companies which had already bought IBM equipment had already paid for the services which were now an added charge: "this is tantamount to selling a man a house and later removing all the bathrooms" (*Computers and Automation* Aug 1969, 39). Bigger users welcomed unbundling, they had pushed for it: IBM's software and support was not needed or wanted. But for many small users and newcomers bundling had represented an insurance policy, against the many, many unknowns of their data processing operations:

> "So the user sits, miserable and confused about the 'lush' prices, the skimpy reductions, the promised compiler that hasn't been announced and may be priced, the number and type of Systems Engineers he will need vs. what he can afford, what programming maintenance bills he will have, how he can afford that software package that must be paid for 10 times for the 10 computers he has scattered about the country, and what loyal programmers he will send for education" (Pantages 1969, 108).

The reaction of computer manufacturers varied. Some like RCA, Univac, and Honeywell saw unbundling as an opportunity to gain customers. They remained bundled hoping to get new accounts, particularly among small and unsophisticated

users who might find it difficult to choose separately priced products:

> "We could go to the customers, potential customers of IBM, and say to them that 'we would offer you these services...and you know what you will be getting from us, and under the IBM unbundled pricing policy, only time will tell what your real price will be'; and I think it was effective, at least for a period of time" (Fisher et al 1983, 177-178).

Other manufacturers took the same course as IBM. Burroughs had in fact already unbunbled before IBM; Control Data unbundled after IBM but it went further: it unbundled its operating systems as well as its application software (*Computers and Automation* Oct 1969, 11). IBM's unbundling gave computer manufacturers the freedom to choose their own pricing policy. Both strategies had advantages for them. If they stayed bundled, they could pick up the less sophisticated users who felt abandoned by IBM's new policy. If they unbundled, they relieved the strain on their own overstretched resources: they were no longer compelled to look after customers "from the cradle to the grave" (Sobel 1984, 263).

Software houses were the ones that gained most from unbundling. It was an exaggeration to say, as one commentator remarked at the time, that the announcement had "in effect given birth to the software industry as an industry" (*Computers and Automation* Oct 1969, 39). The software industry was already flourishing; but certainly it "opened the floodgates and made the fortunes of the founders of the software houses like MSA, Cullinet Software, Applied Data Research, Computer Associates and a score of others who wrote software for IBM mainframes which was better, faster and cheaper than the software IBM wrote itself, or carried out functions that IBM simply could not offer" (*Financial Times* Sep 30 1983).

There was some concern that "unbundling would dry up the source of new software, that unbundling would, since IBM was going to charge for software, make IBM the only source really for software, that others would have difficulties because they didn't have IBM's stamp of approval...What happened was just the opposite of course, that a lot of opportunities were created that IBM didn't always respond to, partly because they didn't have the resources and partly because it made mistakes too. So a combination of making mistakes and not having enough resources to discover and work on all the opportunities created a huge new world of opportunity" (Rasmussen interview).

Behind the ups and downs in the fortunes of users, manufacturers and software houses stood the power of IBM. Whatever else the effects of unbundling, IBM emerged with its power undiminished. Eight years after unbundling, Freeman commented:

> "The upshot of eight years' unbundling is now clear. It has benefited the computer industry as a whole, but IBM has lost little account control or market share" (Freeman 1978, 138).

IBM got the best of both worlds with unbundling. It showed virtue in giving way to anti-monopoly pressures, while doing nothing to reduce its power or its profits.

When IBM split software and hardware prices, it set the standard for the whole industry. Software had been emerging as a commodity, but IBM's unbundling gave this concept a new force: unbundling made users think of software as a product, "a psychological reaction which is difficult to translate into dollars" (Fishman 1982, 138).

With unbundling, IBM gave its blessing to the commodification of software. Unbundling was the culmination of a long-term process, which had gained momentum with the advent of the Third Generation. Once IBM made its announcement, the divorce between hardware and software was official: software was thrown out into the world as an independent commodity.

The existence of software as a commodity implied the recognition of software as property, as a form of intellectual property, with all the contradictions that this notion implies. By its nature intellectual property is difficult to protect; laws can exist but they are difficult to enforce. Everybody knows how easy it is to copy music or books illegally. This is even more true of software: the enormous disparity between production and reproduction costs makes it a very special form of intellectual property, and data communications make it very difficult to control its circulation. The fact that the law recognises software as property does not mean that it is respected in the same way as other forms of property. The concept of software as property brought other problems with it too: problems of taxation, import-export duties, import restrictions, insurance, liability for damage, and so on.

The commodification of software also means that the relation between the production of software and the market becomes more direct. The law of value imposes itself more directly on the production process. To survive in the competitive struggle, it is necessary to produce software as efficiently as possible. Pressures to produce software as efficiently as possible ran parallel to the process of commodification; both grew out of the experience of the third generation and the new awareness of software which that experience produced.

Before unbundling it was possible to hide inefficiencies in software production behind the hardware. With unbundling, software production was subjected increasingly to the disciplines of the market.

The subjection of intellectual work to the disciplines of the market raises particular problems. How is the productivity of intellectual labour to be measured? Especially in programming, how are standards of productivity to be set when there are enormous differences in capacity between one programmer and another, and when the object of the program is not to produce a certain number of lines of code but to solve a particular problem? This point is made forcefully by Dijkstra in his satirical letter from the chairman of "Mathematics Inc":

> "We have returned to our old method of productivity measuring: since February 1974 we measured mathematician productivity by the number of new results obtained per month; we are now back on the more realistic and, after all, also more objective technique of counting the number of lines of proof produced per week" (Dijkstra 1982, 185).

It is difficult to reduce intellectual labour to mere abstract labour. The contradictions are expressed in the tensions that run through the whole development of software: tensions between academics and industry, tensions between the technical and the economic, tensions between programming as mathematics and software as engineering.

Chapter 6

The Software Crisis

The Garmisch Conference

The tensions shaping software development came to the surface in Garmisch Partenkirchen where the NATO Conference on Software Engineering was held in October 1968. This conference was a turning point in the development of software.

The proposal to hold a conference on Software Engineering was made in late 1967 by the Study Group on Computer Science which had been established in the autumn of that year by the NATO Science Commitee. The actual idea for the conference came from Professor Fritz Bauer of Munich. It was Bauer too who appears to have chosen the title "Software Engineering" for the conference.

The choice of the title was very significant. The term "software engineering" had been used before the Garmisch conference. As far back as 1965, Eckert had used the phrase in his talk at the Fall Joint Computer Conference of that year. He argued that programming would be manageable only when it was possible to refer to it as "software engineering". The use of the term would "constitute recognition that, as a discipline, programming had come as far as the older, maturer engineering professions". Before the term could be validly applied, two events would have to occur: "One of these events is anticipated by the growing number of universities with 'computer science' programs. The second is the 'discovery' that computer programming projects can be managed" (Gordon 1968, 200).

The first university course on software engineering was taught in the spring of 1968 by Douglas Ross of the Servo Mechanism Laboratory at MIT, as a means of elaborating the principles of the pioneering work that he had been doing on numerical control (APT) and computer aided design (AED) languages. He recalls:

> "I've never seen anybody lay claim to the first use of the term 'software engineering'. I vaguely remember seeing it in a recruiting advertisement, I think from Boeing or some west coast aerospace company, but I also myself ended up teaching what I believe was the first software engineering course in 1968" (Ross interview).

When asked if the adoption of the term was in response to an awareness that a certain kind of methodology was needed, Ross replied:

> "Well, it wouldn't be that sophisticated at that point, you see. It would be that you realised that what you were doing was building things the same way that other people built bridges, right? And so engineering would be a natural way to express what it was that you were doing. As I say, there were lots of other groups besides ours that were doing this actual engineering work before calling it engineering" (Ross interview).

Although the term did not originate with the conference, it was the conference that launched it into common use. The title "software engineering" was deliberately chosen "as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering" (GCR 13). Bauer explained why he had chosen the term "software engineering" in a later account of the conference:

> "The time has arrived to switch from home-made software to manufactured software, from tinkering to engineering - twelve years after a similar transition has taken place in the computer hardware field...It is high time to take this step. The use of computers has reached the stage that software production...has become a bottleneck for further expansion. And saturation has not even been reached. This results in undesirable effects, like production and use of insufficiently sophisticated

> software with a resulting economic loss, or the strangulation of research in areas that need the computer intensively" (Bauer 1969, 189).

To prepare the conference, a meeting was held in Brussels in March 1968 of members of the Study Group together with the people they had invited to be the group leaders for the three main sections of the conference. It had been decided to organise the conference under three headings: design of software, production of software and service of software. The idea was to invite a limited number of people from all areas concerned with software: computer manufacturers, universities, software houses, computer users, etc. The participants were chosen from different backgrounds because it was intended that the conference should be a working conference which would throw light on the current problems of software and "discuss possible techniques, methods and developments which might lead to their solution". It was hoped that the results of the conference would "serve as a signpost to manufacturers of computers as well as their users" (GCR 14). The participants were carefully chosen:

> "The invitation list was carefully contrived. You know, we invited fifty or sixty people and it was done by the organising committee specifically trying to pick the leading figures in their country, so we had people from about twenty countries and they were all the top names" (Buxton interview).

Despite the fact that the participants were from a wide variety of backgrounds and countries and that many were meeting each other for the first time, "almost from the start a tremendous rapport grew up" (Randell 1979, 6), a rapport which was certainly helped by the beautiful location in the Bavarian Alps and the timing, "which enabled attendees to experience the Munich Oktoberfest en route to Garmisch" (Randell 1979, 6). On the first evening after dinner Bob Barton (who had been one of the leading members of the Burroughs B-5000 team) introduced

the discussion with a fifteen minute talk

> "to the effect that we were all guilty, essentially, of concealing the fact that big pieces of software were increasingly disaster areas and we were all sitting around actually worrying internally about it and doing precisely nothing. So Barton led the discussion off with an extremely lengthy confession himself of just how bad he believed the situation was...He took a very forceful line, and then everybody else just chimed in and said: 'Yes! Actually this is a unique occasion: there have never been at any one time so many distinguished people in software in one place. And perhaps we really should address the problem of what the hell is going on and what are we going to do about it, and what, if anything, is the next step.' So that set the sort of style for the whole thing, you see. And the result then was an immensely enthusiastic week" (Buxton interview).

This excitement can be felt in all accounts of the conference:

> "The atmosphere throughout the conference was really quite amazing. I've known nothing like it before or since" (Randell interview).

Dijkstra recalls:

> "The meeting in Garmisch Partenkirchen was very exciting. For me it was the end of the Middle Ages. It was very sunny. The meeting was a success in Garmisch Partenkirchen largely because most of the people present were sufficiently high in their local hierarchies that they could afford to be honest, and were" (Dijkstra interview).

What was exciting for most of the participants was the common recognition that there was a software crisis, that there were serious problems in the way that software was being designed and produced. There were those who disliked the term "crisis":

> "I do not like the use of the word 'crisis'. It's a very emotive 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" (Kolence, GCR 121).

However, the general feeling was captured in Ross's response:

> It makes no difference if my legs, arms, brain and digestive tract are in fine working ondition if I am at the moment suffering from a heart attack. I am still very much in a crisis (GCR 121).

This was quite new: before Garmisch, the literature tended to emphasise the "seductive fascination" of software rather than its failures (Randell 1979, 1). It seems that people felt a great relief in being able to talk openly about their worries, in being able to move from a feeling of individual responsibility to a collective confession that software was in crisis:

> "The general admission of the existence of a 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 the shortcomings is the primary condition for improvement" (Dijkstra, GCR 121).

This feeling that something could be done was central to the feeling of exhilaration: "an enormous atmosphere of enthusiasm; we felt we might actually achieve something and solve some problems" (Buxton interview); "the main thing from the first conference was that we were openly talking about the software crisis and that something needed to be done" (Ross interview).

The participants brought with them to Garmisch the experiences of the third generation: the trauma of the 360, the difficulties experienced in implementing large projects, the failure to meet deadlines, the disappointments of the users, the examples of disasters or near-disasters, the anxieties about unbundling. It had become clear that software was the problem, the bottleneck that was holding up further development. The manufacturers could build powerful computers, but could not program them to operate efficiently. The users could buy machines of

enormous capacity, but were often not able to program them to achieve their goals effectively. There were complaints everywhere and in the middle was software: people had come to realise how important and how difficult programming was. There was a strong feeling that software was in crisis.

The sense of crisis was not only concerned with the technical difficulties of software production; there was a new awareness of the potentially horrific consequences of software failure. Many participants emphasised the danger to life that could result from a software failure:

> "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 " (David and Fraser, GCR 120).

In this context, people referred a lot to the problem of aircraft safety. The faulty programming of aircraft design or air traffic control could easily lead to disaster:

> "It is my understanding that an uncritical belief in the validity of computer produced results...was at least a contributory cause of a faulty aircraft design that led to several serious aircrashes" (Graham, GCR 121).

Even greater than the danger of aircrashes were the potential dangers of faulty software in military projects.

> "I still remember the ABM debate vividly, and my horror and incredulity that some computer people really believed that one could depend on massively complex hardware systems to detonate one or more H-bombs at exactly the right time and place over New York City to destroy just the incoming missiles, rather than the city and its inhabitants" (Randell 1979, 6).

Although the anti-ballistic missile debate tended to be more an American than a

European concern, "certainly that was the sort of thing that people were worrying about" at the conference: "very complex software systems were being depended on to a very great extent at a time when people were beginning to admit that there had been a lot of very unsuccessful large software system development" (Randell interview).

Other people, however, worried more about the consequences for the industry rather than for their own personal safety. Thus Opler (from IBM) argued:

> "As someone who flies in airplanes and banks in a bank I am concerned personally about the possibility of a calamity, but I am more concerned about the effects of software fiascos on the overall health of the industry" (GCR 121).

However, although there were differences in the way that people saw the consequences of software crisis, it was this topic which dominated the conference. The problem was most generally seen in terms of a "gap". This was a concern that ran through the whole conference, and was seen as being so important that it was made the theme of a special session and was also discussed in the final plenary.

This gap was seen as a gap between expectations and demands placed on software, on the one hand, and actual achievements of software, on the other:

> There is a widening gap between ambitions and achievements...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. This is arising at a time when the consequences of a software failure in all its aspects are becoming increasingly serious" (David and Fraser, GCR 120).

There were different emphases in the way that people looked at this gap

between ambitions and achievements. Some saw it as a problem of the immaturity of the software industry. This argument was made very strongly in a paper by McIlroy:

> "We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialisation somewhere below the more backward construction industries" (GCR 139).

Gillette made a comparison between software production and the aircraft industry:

> "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" (GCR 17).

McClure also associated present difficulties with the immaturity of the industry:

> "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" (GCR 123).

Graham made a striking analogy with early aircraft production:

> "We build systems like the Wright brothers built airplanes - build the whole thing, push it off the cliff, let it crash, and start all over again" (GCR 17).

These comments on the "immaturity" of the industry could be seen as implying a basically optimistic picture: time and the natural evolution of technology would lead inevitably to "maturity", and would help solve the problems.

In contrast with this optimism, other comments emphasised rather the relation between the problems of software production and the environment within which that production takes place. The difficulties of software production were not just the result of immaturity, but of the pressures imposed on programmers by tensions between producers and users, data processing departments and marketing people and by the whole environment of market competition.

The notion of a software gap implied that there were two sides to the problem. It was constituted not just by the failings of software production but equally by the unrealistic nature of the demands made on software. This unrealism resulted partly from users' demands:

> "There are extremely strong economic pressures on manufacturers, both from the 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 the air traffic in Europe is such that there is a pressing need for an automated system of control" (Buxton, GCR 124).

In such cases, users were often trying to buy a technology that did not yet exist:

> "Some of the problems are caused by users who like to buy 'futures' in software systems, and then ignore the problem inherent in this" (Hastings, GCR 124).

However, it was argued that the users' illusions were actively promoted by the computer industry itself, especially by its marketing people. There was not only overbuying, but overselling too. Thus Randell said:

> "I am worried that our abilities as software designers and producers have been oversold" (GCR 121)

McClure, too, referring to the problems associated with the introduction of a new

operating system for the IBM 7090, concluded:

> "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?" (GCR 123).

Software was being produced in an environment of unreality, a fetishised, even fraudulent environment. One participant put it very forcefully:

> "You may be right in blaming users for asking for blue-sky equipment, but if the manufacturing community offers this with a serious face, then I can only say that the whole business is based on one big fraud" (quoted by Bauer 1969, 192).

The effects of this environment were, however, very real. Constant pressures from both computer manufacturers and users pushed technology forward at an unrealistic and dangerous rate. There never seemed to be time to produce software properly or to perfect production techniques. Thus, Buxton argued that any new software project involved not just production but a whole new research effort so that "you are in fact continually embarking on research, yet your salesmen disguise this to the customer as being just a production job" (GCR 122). Kinslow, fresh from his experiences in IBM, quoted the obvious example:

> "In my view both OS-360 and TSS-360 were straight-through, start-to finish, no-test-development, revolutions... At the time TSS-360 was committed for delivery within 18 months it was drawn from two things:
>
> 1. Some hardware proposed, but not yet operational, at MIT.
> 2. Some hardware, not quite operational, at the IBM Research Center" (GCR 122).

It is not surprising, then, that some participants (especially those from IBM) longed for a situation in which software production was not constantly pushed beyond its limits:

> "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" (Kinslow, GCR 124).

Opler's solution to the software crisis implied a direct connection between the crisis and commercial pressures:

> "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, 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" (GCR 124).

His solution, in other words, was to produce software in a non-competitive environment, far removed from the pressures of the market.

Despite the different emphases on the various aspects of the software crisis, there was a general consensus that software production lacked an adequate theoretical and practical foundation and that closing the software gap would "require metamorphosis in the practice of software production and its handmaiden, software design" (David and Fraser, GCR 124).

When the conference ended, "the majority...left...with a feeling of relief, some even in a state of great excitement: it had been admitted *at last* that we did not know how to program well enough. I myself and quite a few others had been waiting eagerly for this moment because now at last something could be done about it" (Dijkstra 1969, 35).

Ross, however, introduced a note of caution. The fact that the software crisis

was recognised did not mean that the solution would be easy:

> "My main worry is in fact that somebody in a position of power will recognise this - 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" (GCR 124-125).

The Rome Conference

In October 1969, a year after the Garmisch Conference, a second NATO Conference on Software Engineering was held in Rome. This conference was a sequel to the Garmisch one, but it was intended to have a more directly practical focus than Garmisch. Discussion was to be centred on the study of more detailed technical problems, and it was hoped that one of the outcomes would be the setting up of some sort of international institute for software.

The idea, as in Garmisch, was to bring people together from different backgrounds. The participants, many of whom had participated in the Garmisch conference, were chosen on a similar basis, although Douglas Ross, who was involved in discussions of who should be invited, recalls that the selection at Rome "was more manipulated, trying to make sure that certain areas were covered and get people to come and talk" (Ross interview).

From the beginning there seemed to be something wrong with the atmosphere at Rome:

> "The place where the conference was held was sterile - I mean it was too modern, too - it didn't bring people together with feeling and that combined with the fact that people were saying their own version in less

> compelling terms of what had been covered the year before...and that they were doing it on demand, or somebody had asked them to do that" (Ross interview).

In total contrast to the excitement and enthusiasm of Garmisch, the Rome Conference "just never clicked" (Ross interview); for Dijkstra, it "was a disaster" (Dijkstra interview), while for Randell, again co-editor (this time with Buxton) of the conference report, "the NATO Conference in Rome was bad-tempered, it was in a hotel which wasn't so nice, and in many ways I think a lot of people who had been at the first one wished the second one hadn't been held" (Randell interview). Buxton too felt that most of the participants were left with "an enormous sensation of disillusionment" (Buxton interview).

Clearly, however, the cause of failure was deeper than the poor choice of location. The harmony between people from widely different backgrounds, who in Garmisch had agreed on the existence of a software crisis and the need to develop software engineering, completely fell apart in Rome. The tensions between the different groups of participants became obvious:

> "Just as the realization of the full magnitude of the software crisis was the main outcome of the meeting at Garmisch, it seems to the editors that the realization of the significance and extent of the communication gap is the most important outcome of the Rome Conference" (RCR 7).

During the week the existence of such a communication gap became more and more evident, "and the realization that it was but a reflection of the situation in the real world caused the gap itself to become a major topic of discussion" (RCR 7).

At the end of the conference, many of the participants felt the need to discuss

this lack of communication, and an extra session was devoted to the topic of theory and practice. The lead in the discussion was taken by Christopher Strachey of Oxford University, who reported that he had heard people from industry complaining that they felt that they had been invited to the conference like "monkeys to be looked at by the theoreticians", while, on the other hand, the theoreticians felt isolated, that "they were not allowed to say anything" (Strachey, RCR 9).

The differences were not quite as clear as that: they could not be, when it is often difficult to distinguish the industrial people from the academics in a world in which computer scientists frequently move from IBM or the other large corporations to the universities and back, and in which so many from both backgrounds have an interest in a software house or a consultancy. For example, John Buxton was selected as an "industrialist" for the Garmisch conference, but was a professor at Warwick University by the time it took place; Brian Randell is listed as Mr Randell of IBM in Garmisch, but Professor Randell of Newcastle-upon-Tyne in Rome; Douglas Ross's address is given as MIT in Garmisch, and Softech Inc at Rome; Andy Kinslow, who had played an important role in IBM's TSS/360, is listed as Computer Systems Consultant in Garmisch; Bob Barton, who had been part of the Burroughs B-5000 group, is listed as Professor at the University of Utah and consultant in system design; Edsger Dijkstra, often seen as the most academic of the academics, was to become a Burroughs research fellow in 1973. It is impossible to draw very sharp lines, and yet there were certainly very important differences in approach, differences which were emphasised in the discussions in Rome.

For the academics, the software crisis was seen as the result of the unscientific methods and rotten techniques used by people in industry. Strachey pointed out, for example, that "recursive programming is not used in general in any large-scale software system, with a few exceptions such as the Burroughs people". Not only that, but there did not seem to be any willingness to learn on the part of the industrialists:

> "The thing that saddens me about the present situation is that there is not much sign that the large engineering set-ups have yet been able to change their basic techniques. They can change the peripheral stuff, the editing and the documentation and things like this, but so far they haven't changed the central core of what they are doing: that is to say the actual programming techniques. Right in the middle it is still...classical programming and classical mistakes"(Strachey, RCR 10).

Since they did not use scientific methods, it was not surprising, in this view, that there were so many failures in large software projects.

From the point of view of the industrialists, academics were arrogant amateurs without managerial capabilities who did not have to face up to the problem of the reality of large-scale software projects and commercial responsibilities. Since their work was simply carried on individually or on a small scale, they diid not worry about maintenance and documentation, so it was not surprising that systems collapsed once one person left the place. Their work was too abstract and they were unable to show that their programs could solve problems in the real world.

One of the main issues that emerged clearly from the discussion of this communication gap was the question of demonstrability and reliability. What industrialists felt they needed were techniques that could be shown to work and not

mere abstractions. They argued that their projects, although full of bugs, worked: not all of them were failures. They were facing strong demands from users to produce results, and they were not in the ivory tower of the university. They were professionals working collectively, trying to solve concrete problems, to come up with concrete solutions. Although academics complained that their techniques were unscientific, it was clearly unrealistic to think that they should simply adopt the ideas of the academics. As Strachey put it:

> "How can we convince people who are dealing with hundreds of programmers and millions of instructions that something as radical as changing the basic core of the way in which they program is a good thing to do? Clearly you can't expect anybody to change a very large project completely in a direction like that merely because you say it is a good idea. This is obviously nonsense" (RCR 10).

One of the proposals put forward for developing practical cooperation between industry and the universities was the idea of a pilot project. Strachey presented this as the solution of the problem of demonstrability and the way to construct a bridge between theory and practice:

> "We need something in the way of a proving ground, halfway between the very large projects which must rely on things that the managers already know how to use, and the most advanced techniques which we're quite sure are all right on a small scale but which we still have to develop on a somewhat larger scale. We need as it were a pilot plant...It could be done, I think, by a cooperative effort between the manufacturers who have the financial resources and the interest in very large systems, and the research institutions...You must build a reasonably sized system where management problems will arise and can be demonstrated to be soluble and where new techniques can be used by managers" (Strachey, RCR 10).

In the discussion on the idea of a pilot plant, various issues were raised about the meaning of a pilot project: how large would it have to be to be convincing to the industrialists without being an actual project? What would be its relation to the market pressures which, as seen at Garmisch, were often blamed for the failures

of software? Galler argued that one important difference between a pilot project and an actual project would have to be "that there be no predetermined time limit specified by marketing people" (RCR 12): again the solution appeared to be to develop software in an atmosphere sheltered from market pressures. This was seen as a possible role for the international software institute that the conference organisers hoped to create.

Doubts about the value of pilot projects were raised by Randell, who pointed out that there was no magic solution to the gap between universities and industry:

> "There is a well known English saying, which is relevant to this discussion about pilot projects. The saying is as follows: 'There's none so blind as them that won't see'. A pilot project will never be convincing to somebody who doesn't want to be convinced...If you have people who are completely stuck in their own ways, whether these are ways of running large projects without regard for possible new techniques, or whether these are ways of concentrating all research into areas of ever smaller relevance or importance, almost no technique that I know of is going to get these two types of people to communicate. A pilot project will just be something stuck in between them. It will be a Panmunjon with no way to it" (RCR 13-14).

The idea of a pilot project could also be seen as an inadequate and reformist approach to a problem that required a radical solution. Thus, Dijkstra criticised the whole idea that it was necessary to build a bridge between theory and practice:

> "I would like to comment on the distinction that has been made between practical and theoretical people. I must stress that I feel this distinction to be obsolete, worn out, inadequate and fruitless...Its inadequacy, among other things, is shown by the fact that I alsolutely refuse to regard myself as either impractical or not theoretical" (Dijkstra, RCR 13).

For him there was no need to construct a bridge between theory and practice simply because that distinction did not exist: it was a false distinction. For Dijkstra, the only way to show that software was of good quality was by *proving* it

correct: mathematical programming was the only way to achieve reliable software.

Recalling the discussion that evening, Buxton contrasts Dijkstra's and Strachey's approaches:

> "If we think it's mathematics, then we embark on single individuals writing big programs and proving them right and so on, but if we think it's engineering, then it's not like that. If we're real engineers, then we build prototypes and pilot plants and explore things and investigate things and then build the production version later, so we had a long discussion about the whole version of pilot plants, essentially advancing the ideas of prototyping, which have come back into prominence only in the last year or two" (Buxton interview).

The idea of setting up an international institute which might be reponsible for organising such pilot projects was one of the main points of controversy at the Rome conference:

> "The Rome Conference was distorted by consideration of the strategic problems of changing large-scale bodies such as the DoD or IBM in the way they worked. And the managers have to be convinced. The issue in Rome was whether a new international software institute should be founded, so that it could build a pilot model and show how it could be done." (Dijkstra interview).

The discussion on the creation of an institute appears to have become quite acrimonious, so much so that it was not included in the conference report. The idea had already come up in Garmisch, but according to Ross,

> "that was in an acceptable context of 'that's why we're all getting together is to see what this is and decide'. But between the first conference and the second, the concept of this institute had become a real political focal point, and the international politics kept getting in the way, because there was a lot of money and power associated with the idea" (Ross interview).

The institute was never founded. It got lost in international politics, in

quarrels over whether it should be in Munich or Luxembourg or Geneva:

> "All I remember is it ended up being a lot of time wasted, and no argument ever turned up to make something happen - which is probably just as well (laughs)" (Ross interview).

The failure to found the international software institute was the most obvious consequence of the failure of the Rome conference. But the problems went deeper than that. This time the participants did not go away with a feeling of euphoria and "certainly, there was no thought then to have any further such conferences" (Randell interview). The conference had shown that, although industrialists and academics might agree on the existence of the software crisis, they did not necessarily mean the same thing by that, and they certainly did not have the same views as to how it should be solved. The conference made clear important differences in approach, both differences in the interpretation of software engineering and, more obviously, differences that could be seen either in terms of theory and practice or, as Dijkstra prefers, in terms of the "Buxton Index", that is, in terms of a gap between long-term and short-term perspectives:

> "Rome suffered from a difference in Buxton index. The Buxton index is the number of years in the future over which the planning of a person or organisation extends itself...It's illuminating because if you try close cooperation between people or institutions with very different Buxton index, a man with a long Buxton index complains that the other party is short-sighted, and the man with the short Buxton index accuses the other one of being a day-dreamer and neglect of duty. The Rome Conference was a failure because it wanted to concoct a cure without having a considered opinion about why software development was so difficult...They were too rushed, they were too impatient to my taste." (Dijkstra interview)

In this view, the essential distinction is not between theory and practice, but between the hectic rush of the commercial world and the more careful, considered approach of the academics.

At the heart of the communication gap of the Rome Conference lay very different understandings of the nature of software, of the whole activity of programming, and of its relation to the "real world".

Chapter 7

Closing the Lid?

Garmisch can be seen as a turning point in the history of software development. The Conference was a major link in a chain of events that has been seen as bringing about a revolution in the way that people thought about programming, a change of paradigm which has been compared to the impact of intuitionist logic on mathematical thought in the early part of this century (Knuth 1974, 142).

In Garmisch everyone had seen Pandora's gifts. They had agreed that software was in crisis and that it was necessary to bring order into chaos, to lay a solid foundation for the further development of software. The term chosen to express the new direction, "software engineering" implied "the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering" (GCR 13).

However, behind the concept "software engineering" there were radically different understandings. For Dijkstra the importance of the concept lay in its connection with mathematics: "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" (GCR 82).

Not everybody understood the term in the same way, however. Dijkstra recalls that when he mentioned the term "Mathematical Engineer", "I noticed that my American colleagues started to laugh, because the mathematician was unpractical and the engineer was practical, and never the twain shall meet...As soon as you cross the Atlantic, all important words slightly change their meaning...Also engineering changes its meaning. You see, here an engineer is someone with a solid academic training...However, in the United States of America, if there is something wrong with your central heating, you get a maintenance engineer. They use 'engineer' for what we would call technicians" (Dijkstra interview).

These differences in interpretation were not just a matter of definition: there were real tensions, as was shown by the Rome Conference. The differences blossomed into distinct approaches, which can be loosely referred to as "structured programming" and "software engineering". In many cases, the two terms are used almost interchangeably and the situation is made more confusing by the fact that both terms are interpreted in different ways. Nevertheless, it is possible to distinguish two responses to the problems of establishing a firmer foundation for programming. The first is a response which insists on the importance of programming methodology and treats programming as a science. The second response has a more immediately practical focus and treats programming as engineering and as a craft. The different views as to the nature of software which had shaped its development from the very earliest days continued to shape the "new" approaches to programming that emerged in the late 1960s and early

1970s, and still continue to do so up to the present day.

Programming Methodology: Structured Programming

The birth of the structured programming approach is associated with two other events of 1968: the break-up of IFIP Working Group 2.1 and the publication of the GOTO letter.

IFIP Working Group 2.1 had been formed after the publication of the ALGOL 60 Report to maintain and develop the ALGOL language. The group met about twice a year. In the early 1960s they began to work on a new language, originally called ALGOL X, to be the successor of ALGOL 60. In May 1965 Niklaus Wirth was given the task of drafting the new language, and his draft was submitted to the group in October of the same year. The majority of the group rejected his draft in favour of a preliminary version of a much more ambitious language. Wirth subsequently left the group and published his version as ALGOL W. In the meantime, the more ambitious project grew more and more ambitious and more and more complex, despite the protests of Hoare, Dijkstra and some other members of the group (Hoare 1984).

The new project reflected the growing power within the group of Aad van Wijngaarden of Amsterdam (who had been one of the members of the ALGOL 60 Committee):

> "Van Wijngaarden gradually got more and more power within the group. Van Wijngaarden at that time I think was rather more interested in his method of language definition than the actual language he was defining. He produced this idea of a two-level grammar, or van Wijngaarden

grammars as they are sometimes called now, and one had the impression you could criticise anything you liked of the language and he'd make efforts to change it, as long as he could fit in his style of language design, his formalism for specifiying the language. And gradually more and more of the meetings of the group became concentrated on detailed issues, on van Wijngaarden's proposals and the little group around him" (Randell interview).

The crisis came at an "incredibly dramatic" meeting in Munich in December 1968, just a couple of months after the Garmisch conference. Van Wijngaarden's proposal for a new language, to be called ALGOL 68 (though Dijkstra thinks a better name for it would have been "van Wijngol"), was put to the vote and a minority group of seven voted against it and produced a short Minority Report: "what the minority group by then had formulated was a general belief that what became known as ALGOL 68 was much too complicated and that the report describing it was nothing like as simple as was needed...and the minority, as the name implies, lost the vote, and then seven of us resigned from the committee" (Randell interview). The split was reported in *Datamation* as a "serious setback" for "that dedicated band of software men, the European ALGOL loyalists" (March 1969, 205).

The break-up of Working Group 2.1 was due to two interrelated issues: the question of language and the issue of simplicity. The Minority Group felt that the emphasis on language design was hindering a simpler, clearer approach to programming.

In the 1960s much stress had been placed on programming languages:

"The sixties were very programming language oriented. It has been the decade in which there has been extensive research on all sorts of parsing algorithms. I remember that in the mid-seventies, when I came to some of the backward places, and even not so backward ones, people asked me,

'what are you doing, what's your specialty? My answer was always 'programming', and in hearing that, people didn't hear that, and said 'oh, programming languages!' I said, 'no, programming'. Programming languages were quite clearly recognised as a practical area of scientific research, but that programming itself, the activity, could be one, that was not perceived. And still isn't. There are still lots and lots and lots of people who think that the major problems of programming are due to the shortcomings of the languages we use, and if we have a good language, then all our problems are gone" (Dijkstra interview).

During the 1960s there was a great proliferation of high level programming languages: by 1969 there were well over a hundred languages in existence (Sammet 1969). This mushrooming gave rise to concern about the need for standardisation and warnings of the dangers of this new Tower of Babel.

Some people felt that the focus on language development was leading computer science in the wrong direction:

> "Much of the theoretical work now being done in the field of programming languages is concerned with language syntax. In essence this means the research is concerned not with *what* the language says but with *how* it says it. This approach seems to put almost insuperable barriers in the way of forming new concepts - at least as far as language *meaning* is concerned" (Strachey 1966, 75).

It was this type of critique that led to the break-up of Working Group 2.1 in December 1968: "some of us were getting to believe that those issues of language and language design weren't really so much the central issues of software, and we felt that issues of producing sophisticated systems that one could trust depended critically on questions of structure and questions of very great simplicity" (Randell interview).

The minority group then formed a new working group, IFIP Working Group 2.3, which was to concentrate on what they felt was the central issue of software.

At Brian Randell's suggestion, the group took the title "Programming Methodology".

From the beginning "the working group was very different from all the other IFIP working groups. It had started up with the avowed intention of not trying to produce a magnum opus like the ALGOL 68 report. We decided to act purely as a means whereby people who were working on a general set of topics met up together and exchanged notes every now and then" (Randell interview). As a result of this informality, the group had problems with the IFIP bureaucracy who felt that they were "just a sort of private club" (Randell interview).

In fact, the work of the people in the group was to become extremely influential in changing the orientation of software. This was not because of any group project, but because it provided a forum for discussion for a number of people who were trying to push software development in the same direction. "When eventually David Gries, one of the members, undertook the editing of the book which was collected papers from members of the group, that...absolutely transformed the impression that IFIP had and, from being one of the bad boys of IFIP, we suddenly became some of the highly approved of people" (Randell interview). The working group continues to exist, although its composition has changed.

The members of Working Group 2.3 differed in their emphasis, but they shared a common concern with increasing the rigour and raising the scientific level of computer programming. Within the group, there emerged a distinctive

approach to programming, advocated by some (Dijkstra, Hoare, Gries etc) though not all of the participants. This approach, which continues to be influential within computer science, is sometimes referred to as the "aesthetic school of computer science" (Wegner 1979, 206) or the "mathematical school", or sometimes the "Dijkstra school": Edsger Dijkstra is one of the most influential, and certainly the most outspoken, advocate of radical change.

This approach stresses the importance of scientific rigour. Scientific rigour is necessary because programming is difficult: "we must acknowledge that programming is a difficult intellectual task" (Gries 1979, 257). The difficulty of programming has not been sufficiently realised. The problems of programming are unprecedented in human history: "computers are so unlike anything we ever had before that the challenge to program them well is absolutely without precedent" (Dijkstra interview).

The difficulties of programming arise partly from size. Increase in size can make a problem incomparably more difficult: "one can close one's eyes and imagine how it feels to be standing in an open place, a prairie or a sea shore, while far away a big, reinless horse is approaching at a gallop, one can see it approaching and passing. To do the same with a phalanx of a thousand of these big beasts is mentally impossible: your heart would miss a number of beats by pure panic if you could!" (Dijkstra 1972a, 2). Underestimation of the difficulties arising from size is "one of the major underlying causes of the current software failure" (Dijkstra 1972a, 2).

The problem of programming is how to deal with this complexity. As humans

our intellectual powers are limited, and "as wise programmers...we should be aware of our limitations" (Dijkstra 1968, 147). The reason for the existence of the computer is that the computer can do things which humans can not do: "the automatic computer owes its right to exist, its usefulness, precisely to its ability to perform large computations where we humans cannot. We want the computer to do what we can never do ourselves" (Dijkstra 1972a, 2). Programming is about organising complex computations "in such a way that our limited powers are sufficient to guarantee that the computation will establish the desired effect" (Dijkstra 1972a, 3).

Complexity demands simplicity. The programmer's task, therefore, is to structure this complexity in such a way that it becomes intellectually manageable. Programming is about the organisation of complex problems into simple units: "programming...boils down to no more and no less than very effective thinking so as to avoid unmastered complexity" (Dijkstra 1982, 163). Complexity has to be avoided for the simple reason that humans cannot cope with it: "as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and to give them full credit, rather than to ignore them, for the latter vain effort will be punished by failure" (Dijkstra 1972a, 3).

The emphasis on simplicity and rigour is the basis of a radical critique of the existing practice of programming at the time. The starting point of the critique is the contrast between craft and science. Dijkstra, Hoare and others argue that the fact that programming had developed as a craft rather than as a science was the source of many of the problems of software.

The distinction between a craft and a science relates to knowledge, education and methods of work. The craftsperson begins by working as an apprentice under the guidance and supervision of a master, "absorbing gradually, by osmosis so to speak, the skills of the craft, until he may be called a master himself" (Dijkstra 1982, 104). The knowledge of craftspeople is never formulated explicitly, but is passed on from one to another, typically being kept a well-guarded secret among the craftspeople themselves. The methods of work are based not on formulated principles but on acquired intuition: "the craftsman knows what he is going to build, and knows how to build it. He has no need of elaborate plans, precise blueprints, careful measurements, precise quantities, progress charts, delivery schedules and cost estimates. When he undertakes to make something, he succeeds, because he knows how to make it and his customer knows what to expect. He seems to have an ingrained sympathy with his materials, and an intuitive knack for handling his tools most effectively. If by chance something goes awry, he knows how to adapt his work or his design to compensate for the error. And in the end, his product works, and gives good service, and endures. Or, on the other hand, it doesn'tl" (Hoare 1978, 1).

In contrast with the craftsperson, whose knowledge remains implicit, "the future scientist learns his trade as a student from a teacher...who tries to formulate the knowledge and to describe the skills as explicitly as possible, thereby bringing both into the public domain" (Dijkstra 1982, 4). This has important implications for education and particularly for the development of curricula in computer science. Whereas the craft approach leads to a curriculum which is a "cocktail", bringing together "any odd collection of scraps of knowledge and an arbitrary bunch of abilities (Dijkstra 1982, 65), the development of a

scientific discipline requires a high degree of internal coherence and rigour: "the internal requirement is one of coherence... The external requirement is one of what I usually call a 'narrow interface': the more self-supporting such an intellectual subuniverse, the less detailed the knowledge that its practitioners need about other areas of human endeavour, the greater its viability. In the terminology of the computing scientist, I should perhaps call our scientific disciplines 'the natural intellectual modules of our culture'" (Dijkstra 1982, 65-66).

Programming, in this view, was still predominantly treated as a craft rather than a science. From the early years of programming, the architecture and limited capacity of the machines had led to a tradition of seeing programming as clever tricks to get the computer to do what was required: the good programmer was "puzzle-minded and very fond of clever tricks" (Dijkstra 1978, 11), trying to squeeze the best results from an often unique and idiosyncratic machine, often achieving excellent results, but not always. This picture of the clever craft approach to programming can be seen very clearly, for example, in accounts of early programming. In a conference in 1979, Backus recalled that "programming in the America of the 1950s had a vital frontier enthusiasm virtually untainted by either the scholarship or the stuffiness of academia". However, the other side of this enthusiasm was that "just as freewheeling westerners developed a chauvinistic pride in their frontiersmanship and a corresponding conservatism, so many programmers of the freewheeling 1950s began to regard themselves as members of a priesthood guarding skills and mysteries far too complex for ordinary mortals" (Backus 1980, 126-127).

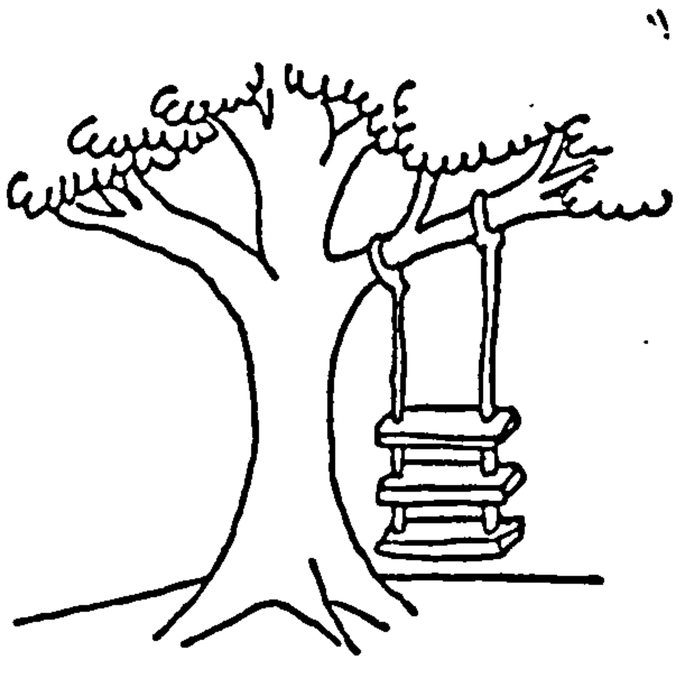
For the craftsperson, results are achieved through clever tricks, and professional excitement is derived from "not quite understanding what he is doing. In this streamline age, one of our most undernourished psychological needs is the craving for Black Magic and apparently the automatic computer can satisfy this need for the professional software engineer, who is secretly enthralled by the gigantic risks he takes in his daring irresponsibility" (Dijkstra 1972b, 223).

The result of this "daring irresponsibility" is, in this view, that one can never be sure of the correctness of a program. The way in which the correctness and reliability of software was ensured was, as Graham had pointed out at Garmisch (GCR 17), rather like the way that the Wright brothers had tested their aeroplanes - by pushing them off the cliff, letting them crash and starting all over again. The established method of ensuring reliability was first to write the program and then to test and debug it. Hoare describes the work of the programmer, systems analyst or project leader on the typical project:

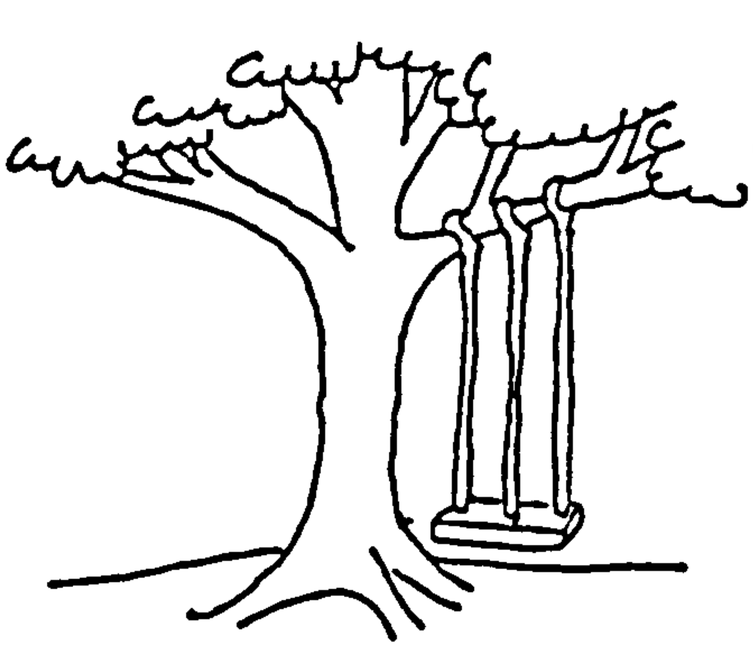
> "He may indeed start with a description of what his client thinks he wants; but the desription is so imprecise, inconsistent and even inconstant that it can serve only as a rough diagram rather than as a firm plan for implementation. Nevertheless a good programmer knows how to proceed. He seems to have an intuitive grasp of his programming language and ingrained feeling for what his operating system can be made to do. He starts writing and testing his code, and when it is all finished, it all miraculously fits together and works after its fashion. If anything goes awry, he hacks a bit at his already written code, modifies his plans a bit, and after some delay, delivers his product. If it is not quite what his client wanted, he can continue to hack until his client is satisfied; or more usually, until he gets tired of waiting. And if the product never gets to work at all, or is too inefficient or expensive to put into use, there is no point in trying to understand why - it is just one of those things that happen in programming" (Hoare 1978, 2).

For the craftsperson, the mechanism on which she or he is working remains not quite understood, a "black box" (Dijkstra 1972a, 5; 1972b, 221). Hence one

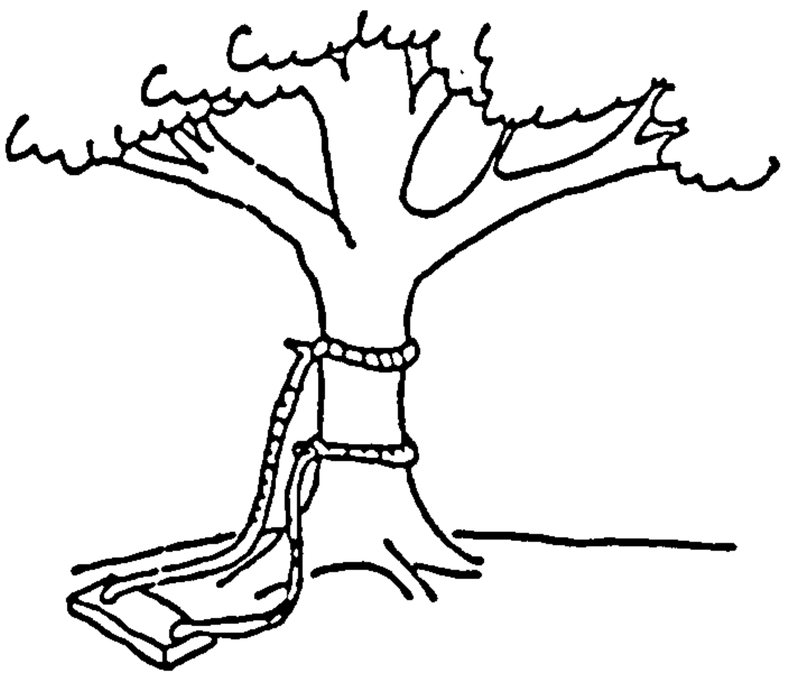
Scientific Management



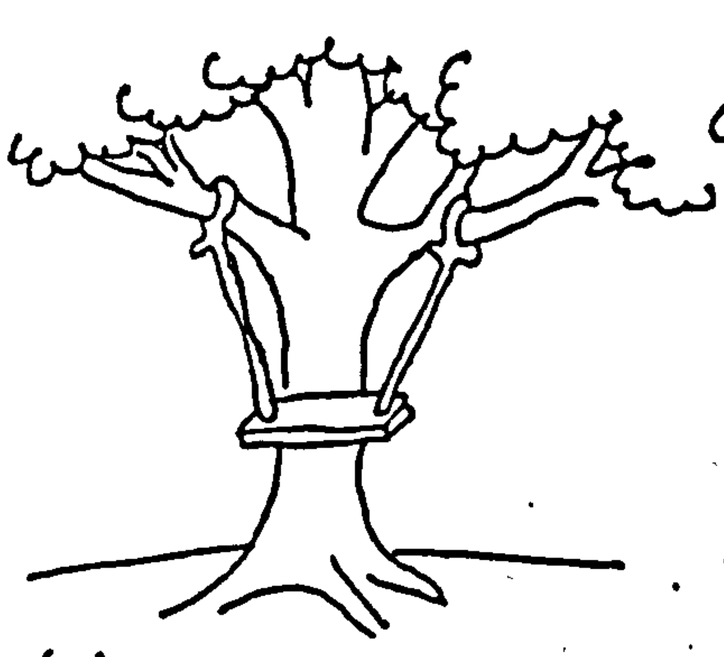
1. As management requested it



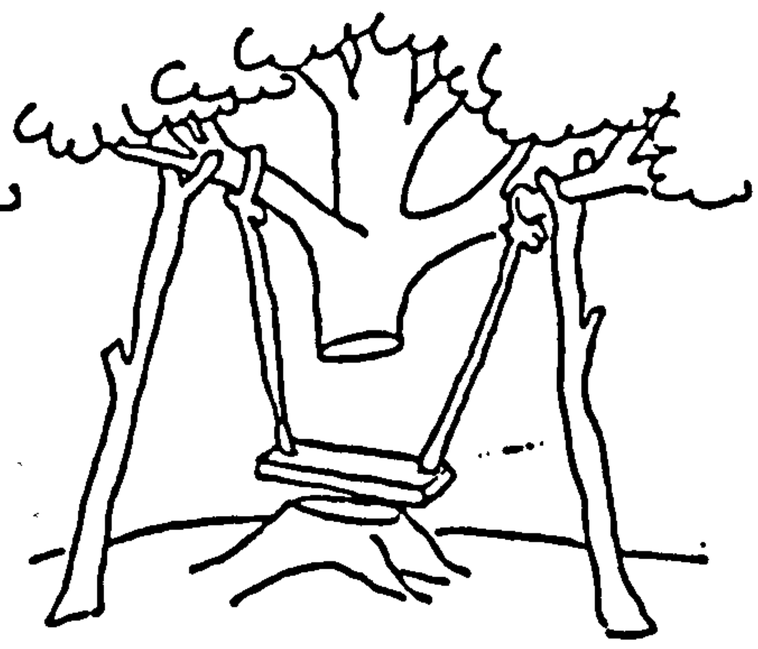
2. As the project leader defined it



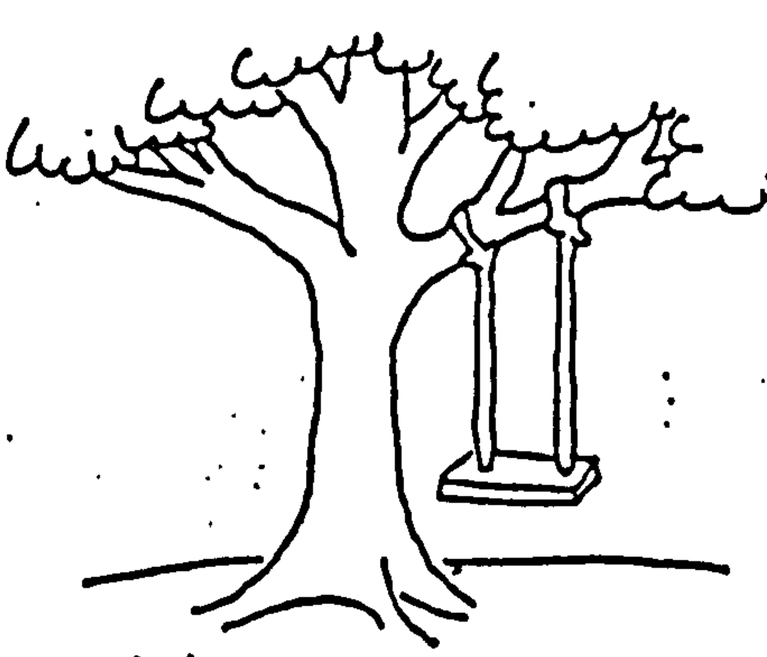
3. As systems designed it



4. As programming developed it



5. As operations installed it



6. What the user wanted

can never be sure what the output of the black box will be; one can test it and try to "debug" it, but one can never be sure of its correctness. Consequently, it follows that: "present-day computers are amazing pieces of equipment, but most amazing of all are the uncertain grounds on account of which we attach any validity to their output" (Dijkstra 1972a, 3).

Ensuring reliability by testing is, in this view, bound to be inadequate because, as Dijkstra points out, even for a simple program in the fastest machine, the exhaustive testing of all the computations that can be evoked by the program "is apt to take millions of years" (1971, 361). Exhaustive testing is thus impossible and "testing by random sampling is hopelessly inadequate as well, because even the most vigorous sampling will cover only a truly negligible fraction of the possible number of cases" (1971, 361). It follows that, in one of Dijkstra's favourite sayings, "program testing can be used very efficiently to show the presence of bugs, but never to show their absence" (1971, 361).

How can the reliability of programs be ensured? Dijkstra's "gospel", as he puts it, is that proving the correctness of a program cannot be separated from the program itself:

> "When concern for correctness comes as an afterthought, so that correctness proofs have to be given once the program is already completed, the programmer can indeed expect severe troubles. If, however, he adheres to the discipline of producing correctness proofs as he writes his program he will produce program and proof with less effort than just the programming alone would have taken" (1971, 366).

The only way to produce such a proof of correctness is through *mathematics*.

The application of mathematics to programming is the only way to deal with the

problems of reliability and complexity.

Mathematical assertions are seen as possessing three characteristics which should be intrinsic to programming: firstly, they are always general in the sense that they are applicable to many cases; secondly, they are very precise; and thirdly, "a tradition of more than twenty centuries has taught us to present these general and precise assertions with a convincing power that has no equal in any other intellectual discipline. This tradition is called mathematics" (Dijkstra 1969, 38).

Generality, precision and convincing power are seen as being essential to any program. Firstly, a program is general in so far as it computes a function that is defined for an enormous number of different values of its argument. Secondly, the specification of what a program can achieve must be precise if the program is going to be a safe tool to use. And thirdly, if the program is to be regarded as reliable, it is necessary to provide a convincing case for the assertion that such and such a program corresponds to such and such a function: "that program testing does *not* provide such a convincing case is well known...The only alternative that I see is the only alternative mankind has been able to come up with for dealing with such problems, and that is a nice convincing argument. And that is what we have always called mathematics" (Dijkstra 1969, 38).

As one can see if one thinks of Euclid's theorems in geometry, for example, the convincing power of the argument depends upon the way it is stuctured or built up step by step. The problem of correctness, therefore, can only be approached through the proper structuring of the program. This relation between

programming structure and mathematics was brought out strongly by another event in 1968, the publication of Dijkstra's GOTO letter and the ensuing controversy.

Dijkstra's letter, entitled "GOTO Statement Considered Harmful", was published in March 1968 in the *Communications of the ACM*. The letter was less than a page and a half in length, but it was to become famous as the symbol of the methodological attack on the established practices of computer programming and the focus of what became known as the GOTO controversy.

The title under which the letter was published was not its original title. Initially, it "was submitted as an article with another title - I don't remember the title any more - and I submitted it to Niklaus Wirth, who was then one of the editors of the ACM. And, in order to speed up the publication, he made a letter to the editor of it, because letters to the editor can be published immediately" (Dijkstra interview). The editor assigned the paper to Douglas T. Ross of MIT as a reviewer and he recommended that it should be published quickly since it was "a significant paper which should be brought to everybody's attention...it was an important thing for conceptualising, that people could talk about things that before had not been verbalised" (Ross interview).

Dijkstra's letter was concerned with controlling the progress of a process, or program in execution. As explained in the discussion of operating systems, the term "process" was introduced in the early 1960s as an abstraction of the processor's activity, to denote a program in execution at any instant in time, regardless of whether the processor is actually executing instructions from a

specific program at that instant. In terms of the analogy between an operating system and the coordination of a complex traffic system in a city, a process is a car on the road, whether it is currently in motion or not. To be able to keep track of the progress of one car on the road (process) is obviously an important precondition for being able to regulate a complex traffic system, or being able to coordinate the various processes within the computer.

In the 1960s the coordination of various programs in execution was a major problem in programming. The coordination of processes had been made much more complex by the advent of multiprogramming, real time computing and time sharing. It was a common occurrence for operating systems to crash, that is to stop suddenly for no apparent reason. Recovery from such crashes was both expensive and time consuming. The most common cause of system crash was the incorrect coordination of processes. One way in which this could happen was "deadlock", where two or more processes stop, each waiting for an action by one of the others. In this case the system comes to a complete halt, and recovery is often expensive, requiring the termination of one or more jobs in order to release the resources to allow the other jobs to proceed.

Dijkstra had made an important contribution to the study of the problem of coordinating concurrent processes. In 1965, he published a paper on cooperating sequential processes, in which he introduced the concept of semaphores, special signals designed to synchronise asynchronous processes. The operating system is seen as a "society of sequential processes, progressing with undefined speed ratios", and the semaphores are special variables which force a process to stop at a specified place and wait until it has received a specified signal (COSERS 1984,

674). In a Symposium on the Principles of Operating Systems, organised by the ACM at Gatlingburg, Tennessee in October 1967, Dijkstra presented an operating system based on these principles, the THE-Multiprogamming System (named after Dijkstra's university, the Technische Hoogeschool Eindhoven). It was in the course of the discussions there that the idea of writing the GOTO letter emerged:

> "I was explaining this point one afternoon. The sessions had finished and we were sitting outside with a number of computing scientists. I explained this and Brian Randell, who was sitting next to me and was listening to me, said 'Gee, Edsger, that's great. I never realised. You should publish that.' So then I wrote that explanation in the assumption that everybody would agree that there was something wrong with the GOTO statement. What I meant to publish was an explanation, and it created an unforeseen uproar" (Dijkstra interview).

The letter itself was not concerned directly with the coordination of different processes, but with the more basic question of how to control the progress of one process, a simpler problem than the orderly coordination of processes within a system:

The GOTO statement is an instruction to the computer to break the normal sequence of the program by "going to" or "jumping" to another location in the program. It was introduced by FORTRAN. It was seen in the first chapter that FORTRAN operates by first constructing a library of subroutines and then selecting or GOing TO each subroutine as and when it is required. The attack on GOTO was thus to some extent a continuation of the ALGOL-FORTRAN controversy in a different form. ALGOL also allows GOTO statements, but the GOTO is extraneous to the basic nested-block structure of ALGOL, while it is fundamental in a language such as FORTRAN that assigns data to fixed locations in a static storage, and then jumps to them as required.

Dijkstra began his letter by saying that he had been familiar for a number of years with "the observation that the quality of programmers is a decreasing function of the density of GOTO statements in the programs they produce". His aim in the GOTO letter was to establish a firm basis for the observation that the use of GOTO was harmful. The use of GOTO had already been criticised by a number of other people, such as Naur, Strachey, Hoare and Zemanek, during the 1960s (Arblaster, Sime and Green 1979, 105). Dijkstra himself had claimed that the use of GOTO created problems as early as 1965:

> "Two programming department managers from different countries and different backgrounds - the one mainly scientific, the other mainly commercial - have communicated to me, independently of each other and on their own initiative, their observation that the quality of their programmers was inversely proportional to the density of the GOTO statements in their programs...I have done various programming experiments...in modified versions of ALGOL 60 in which the GOTO statement was abolished...The latter versions were more dificult to make: we are so familiar with the jump order that it requires some effort to forget it! In all cases tried, however, the program without the GOTO statement turned out to be shorter and more lucid" (Dijkstra 1965, 213).

In his letter, Dijkstra claimed:

> "more recently I discovered why the use of the GOTO statement has such disastrous effects, and I became convinced that the GOTO statement should be abolished for all 'higher level' programming languages" (1968, 147).

Before developing his argument, Dijkstra introduced two general considerations. The first of these was that "although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is the process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications". Computer

programmers did not pay sufficient attention to controlling the process of computation: "once the program has been made, the 'making' of the corresponding process is delegated to the machine" (1968, 147)

Dijkstra's second general point was that it was essential to start from a recognition of our intellectual limitations: "our intellectual powers are rather geared to master static relations ... and our powers to visualise processes evolving in time are rather poorly developed". For that reason it was important "to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible" (1968, 147).

The problem, therefore, was to see how the programmer could control the progress of the process of computation. To do this, it was necessary to find a way of characterising the progress of the process: "you may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?" (1968, 147).

Dijkstra's argument was that, in the absence of GOTO statements, it is possible to construct a system of coordinates that will describe the progress of the computation: "with a straight line program, then just the instruction counter will do. If you have a repetition, well you can, behind the scenes, so to speak, introduce a counter that counts how many times it has gone through. If you have recursive procedures, you can stack the pending calls" (Dijkstra interview). In each case, the value of these indices are outside the programmer's control: "they are

generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not" (1968, 147)

The independent coordinates were important because their value would indicate the precise progress of the process: "if we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* whenever we see somebody entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!" (1968, 147)

The problem with the GOTO statement is that: "as soon you introduce GOTO there was no longer any means, any manageable means, of counting the number of instructions performed" (Dijkstra interview). Although it is possible simply to count the number of actions performed since the start of the program, "the difficulty is that such a coordinate, although unique is utterly unhelpful". The coordinate would indicate the number of actions performed but not which part of the program had been reached. By breaking the sequence of the program, the GOTO statement can lead to a logical spaghetti which is very difficult to implement, correct or change. "The GOTO statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program". Consequently, Dijkstra advocated the elimination of the GOTO statement from all programming languages.

After the publication of his letter, Dijkstra received "a torrent of abusive letters" (Knuth 1974, 265). As Knuth points out, the discussion was clearly threatening to some people. This was shown, for example, in the first published

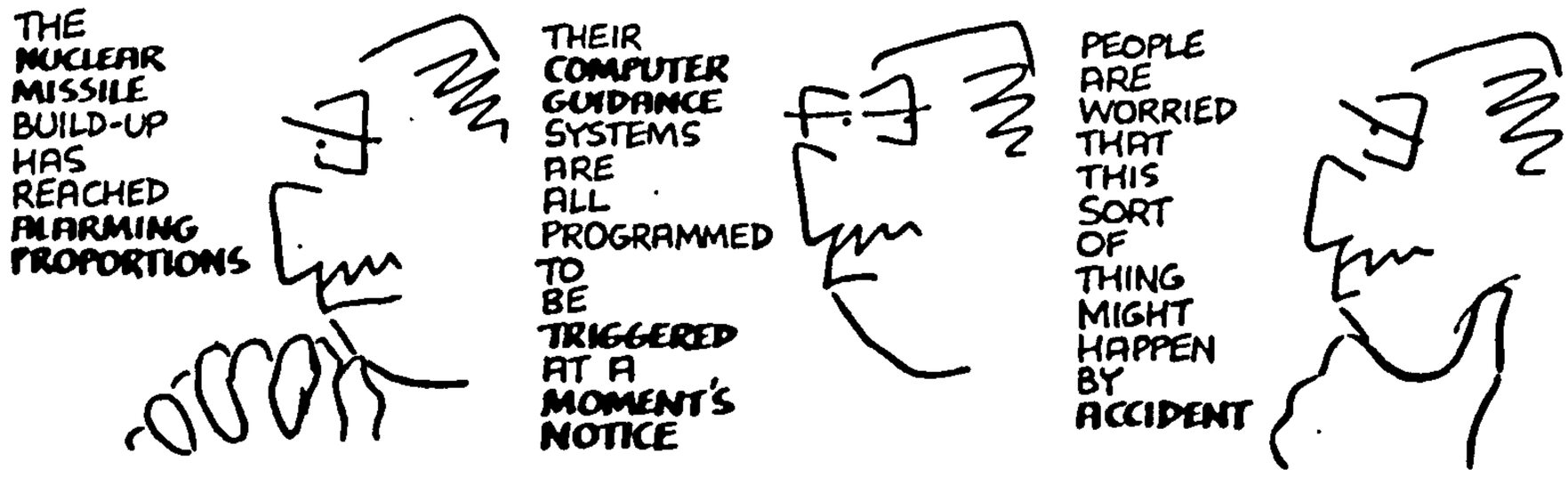
response to Dijkstra, a letter from John R. Rice in *Communications of the ACM* five months later. Rice's indignation is evident, even though, according to Dijkstra, "the published version had been slightly toned down" (Dijkstra interview):

> "I was taken aback by Dijkstra's letter on the GOTO statement, which is an obviously useful and desirable statement...I find the emotional tone of this attack as disquieting as the 'scientific' analysis. How many poor, innocent, novice programmers will feel guilty when their sinful use of GOTO is flailed in this letter?" (Rice 1968, 538).

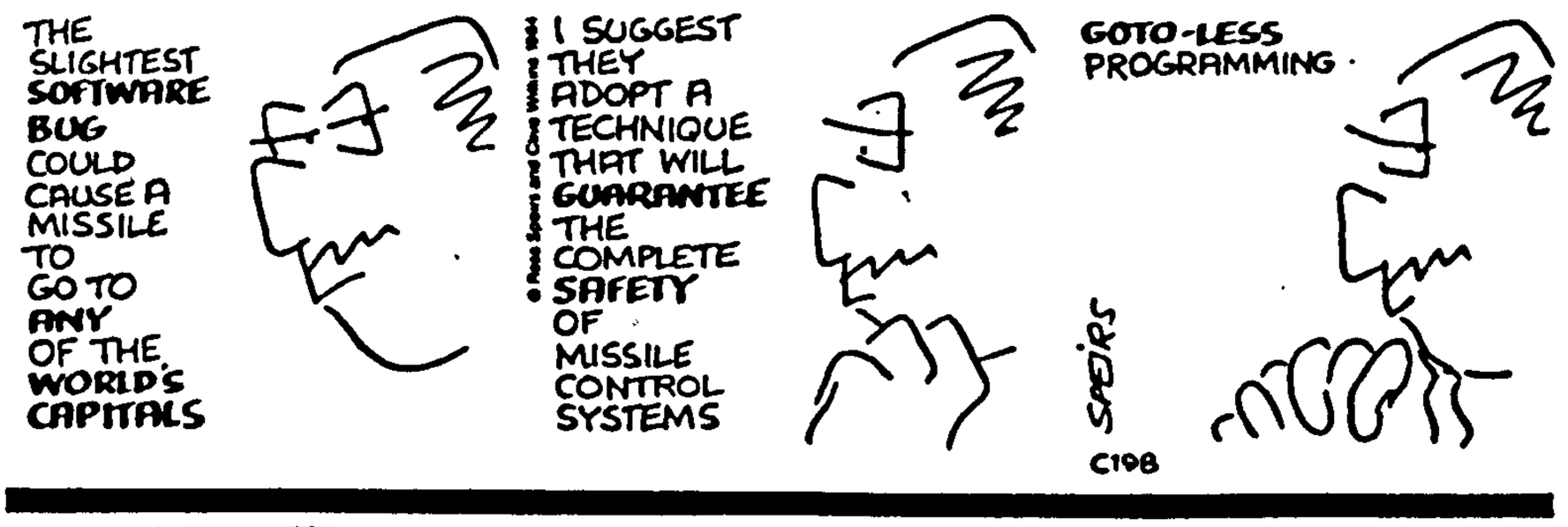
Rice's letter expressed what was probably a fairly widespread reaction to Dijkstra's suggestion that the GOTO statement should be eliminated. Dijkstra argues that the sense of indignation was "probably because that was a time in which the vast majority of programmers were monolingual. At the time there was definitely something called the FORTRAN community, and any suggestion that there might be something wrong with FORTRAN was taken as an offence" (Dijkstra interview).

The GOTO letter stirred up the "GOTO controversy". At first examination of the literature, however, it is difficult to see where the controversy is. Apart from Rice's letter, it is difficult to find anyone who openly defended the use of GOTO: "a number of years later ... there was a panel discussion about the GOTO controversy and I remember that at that time for the panel organiser it was hard to find someone who would argue in favour of the GOTO" (Dijkstra Interview). The person eventually found, Martin Hopkins of IBM, began his defence by saying that "it is with some trepidation that I undertake to defend the GOTO statement, a construct, which, while ancient and much used, has been shown to be theoretically unnecessary and in recent years has come under so much attack" (Hopkins 1972,

AS IT HAPPENS...



BY ROSS SPEIRS AND CLIVE WILKINS



787).

The fact that the GOTO statement was so weakly defended does not, however, mean that there was no controversy. The controversy took the form that many arguments take: those who responded to Dijkstra's argument did not (apart from Rice and Hopkins) openly disagree with him; but they did not entirely agree with him either. The controversy was not a "yes/no" conflict, but a "yes, but..." argument. Not only that, but the "yes, but..." operated at different levels, depending on how the letter was understood.

The letter was understood in different ways. As Randell says, Dijkstra "managed to hit the nail on the thumb with that letter... It caused quite a lot of confusion" (Randell interview). Partly because of the title that it was given on publication, "in the eyes of many, not using GOTOs was the main issue" (Dijkstra interview); or, as Hopkins put it in his defence of GOTO: "some people are now hoping that the royal road will be found through style, and that banishment of the GOTO statement will solve all" (Hopkins 1972, 787).

Consequently, much of the discussion focused simply on the elimination of the GOTO statement. However, if the discussion is focused simply on the use of the GOTO statement, the argument for its total elimination is rather weak: all that can be said is that it is possible to avoid the use of GOTO, that the GOTO is harmful because it can lead to a "rat's nest of control flow" (Wulf 1972, 791), but that there are certain circumstances in which its use is desirable. Nevertheless, after the publication of Dijkstra's letter, it became fashionable to eliminate the "sin" of GOTO at all costs, even if only to replace it by other forms of "jump" statement.

For example, the authors of the BLISS language replaced GOTO by eight "escape" statements and subsequently added a ninth. From this and other examples, Knuth concludes that "it seems that there is a wide-spread agreement that GOTO statements are harmful, yet programmers and language designers still feel the need for some euphemism that 'goes to' without saying GOTO" (Knuth 1974, 266). At this level, there is some justification for Hopkins' remark:

> "I also suspect that the controversy reflects something rather deep in human nature, the notion that language is magic and the mere utterance of certain words is dangerous or defiling. Is it an accident that 'GOTO' has four letters?" (Hopkins 1972, 787)

However, this was far from Dijkstra's original intention. A few years after the publication of his letter, Dijkstra wrote to Knuth, complaining of exactly what Hopkins had criticised, saying that he had "the uncomfortable feeling that others are making a religion out of it, as if the conceptual problems of programming could be solved by a single trick, by a simple form of coding discipline" (quoted by Knuth 1974, 262).

For Dijkstra, the main issue was the quality of programming, the clear structuring of the program: "not using the GOTO was a very minor part of that message; it was about something completely different and the ray of publicity has been responsible for it that, in the eyes of many, not using GOTOs was the main issue, and this completely distorted their view" (Dijkstra interview).

Dijkstra's "message" was that programs should be structured clearly: the use of the GOTO made it difficult to do this by generating unnecessary complexity. For Dijkstra and those who shared his views, it was only within this context that the

issue of the GOTO became significant: "the strong reasons for eliminating the GOTO arise in the context of more positive proposals for a programming methodology which makes the GOTO unnecessary" (Wulf 1972, 791). The important point is not to eliminate GOTOs from programs, but to program in such a way that the GOTO has no place: "in fact, the method of step-wise refinement of the programming task automatically leads to GOTO-free programming; the absence of jumps is not the initial aim, but the final outcome of the exercise" (Wirth 1974, 257).

The positive methodology referred to was structured programming. What is meant by structured programming and its relation to the GOTO statement can best be illustrated by returning to the simple program of morning activities discussed in the first chapter. In that example, the program of morning activities (eat breakfast; put on clothes; leave in car;) was broken down into a series of detailed and more detailed statements, so that the whole thing became a nested set of blocks. The example was used there to explain the principles of ALGOL, but it can be used equally well to explain structured programming.

Structuring complexity into appropriate blocks is the essence of structured programming: "it is the expression of a conviction that the programmers' knowledge must not consist of a bag of tricks and trade secrets, but of a general intellectual ability to tackle problems systematically" (Wirth 1974, 249). The clever craft programmer of the old days would have eaten toast in the car because it saved time, but the systematic, scientific programmer locates each activity in its proper block: first breakfast, then clothes, then car. It appears less exciting: there is no hectic rush, no clever tricks; but the challenge, Dijkstra and the others would argue, lies in the proper ordering of a complex set of activities.

Each block is built around an abstraction. The statement "Eat breakfast" abstracts from the details of what is eaten: it is a general description of a pattern of behaviour that covers a large range of possible variables. The abstractions help to impose order on what would otherwise be a confusing mass of individual statements: eat toast, put on shoes, start car, eat cereal etc: "our most important mental tool for coping with complexity is *abstraction*. Therefore a complex problem should not be regarded in terms of computer instructions, bits, and 'logical words', but rather in terms and entities natural to the problem itself, abstracted in some suitable sense" (Wirth 1974, 249).

The process of starting from the most general level of description and then breaking down the program into more and more detailed blocks is known as "step-wise decomposition" or "step-wise refinement": the general operations in the program "are then considered as the constituents of the program which are further subjected to decomposition to the next 'lower' level of abstraction. This process of *refinement* continues until a level is reached that can be understood by the computer" (Wirth 1974, 249).

By a process of abstraction and step-wise refinement, the morning's activities are decomposed into a hierarchical series of blocks, in which the abstractions are specified more and more concretely. Each block is self-contained, with a single entry and a single exit: first the breakfast, then the clothes, then the car, not backwards and forwards between the blocks with multiple entries and exits. A well-structured program is a disciplined sequence of blocks, with single entries and single exits.

Sequencing discipline is established not only through abstraction, but also through other "mental aids" or "typical patterns of thought which help us to understand complex problems" (Dijkstra 1972, 6). Of these, the two most important are enumeration and mathematical induction. Enumeration is the simplest way of ordering any series of computations, but it is not necessarily the most concise. Thus, an enumerative description of the process of eating the cereal might take the form:

```
begin comment Eat cereal;
Put milk on cereal;
Put sugar on cereal;
if hungry and cereal in bowl then Eat bite of cereal;
If hungry and cereal in bowl then Eat bite of cereal;
If hungry and cereal in bowl then Eat bite of cereal;
If hungry and cereal in bowl then Eat bite of cereal;
If hungry and cereal in bowl then Eat bite of cereal;
...
end .
```

In its simplest form, it would be possible to simply enumerate all the possible bites of cereal, up to the maximum number contained in the cereal bowl. However, "enumeration is only an adequate tool when the number of cases to be considered is relatively small" (Gries 1979, 259). By inductive reasoning, it is possible to shorten the sequence by using a **while...do...** statement. The program for eating cereal then becomes:

```
begin comment Eat cereal;
Put milk on cereal;
Put sugar on cereal;
while hungry and cereal in bowl do Eat bite of cereal
end .
```

Thus, using the three "mental aids" of abstraction, enumeration and induction, it is possible to express a very complex series of activities in a systematic and elegant manner. Together, these techniques help one to write a clearly structured,

tightly sequenced program. In a well-structured program, there is no confusion, no logical spaghetti.

In this approach, the program is visible. The well-structured program is easily understandable and requires no external documentation. Similarly, the proof of the program lies in the process of programming. A tightly sequenced series of logical, mathematical statements is its own proof. Correctness does not need to be established by testing after the event: "the wise programmer...develops program and correctness argument hand in hand; as a matter of fact, the development of the correctness argument usually runs slightly ahead of the development of the program: he first decides how he is going to prove the correctness and then designs the program so as to fit the next step of the proof" (Dijkstra, 1969, 39).

The program is its own verification: the proof of the pudding is in the cooking, not in the eating. It is the essence of a mathematical theorem that it does not depend on post-hoc testing: it is not necessary to use a measuring tape to establish that in an isosceles triangle, the square on the hypotenuse is equal to the sum of the squares on the other two sides. Once the terms of the problem are clearly specified, the process of programming should follow logically. In this conception, programming depends on the clear specification of the terms of the problem.

The description of structured programming contains no mention of GOTO. This makes it clear that structured programming is not "about eliminating GOTOs": the GOTO statement is simply redundant in a well-ordered program. The clever but unsystematic programmer who eats toast in the car would need a lot of GOTOs

connecting the Eat breakfast module with the Leave in car module. The end result might be faster, but the program would be hard to understand, hard to correct and liable to error.

In the light of the earlier discussion of ALGOL, it is clear that neither the attack on GOTO nor the elaboration of the principles of structured programming was dramatically new. Dijkstra and the others were elaborating and making explicit many of the implications of the ALGOL approach to programming.

It should now be clear why Randell said that Dijkstra "hit the nail on the thumb" with the GOTO letter. The fact that the letter was published under the title "GOTO Considered Harmful" contributed much to the form taken by the controversy.

Dijkstra's letter caused a lot of confusion; as one commentator wrote afterwards: "The letter attracted considerable attention and puzzlement at the time. I well remember asking people, 'Do you understand what Dijkstra is talking about?' The representative answer was 'I'm sure it's important, but I don't really quite understand it'" (McCracken 1973, 51).

Behind the appearance of agreement, there was actually a dispute - a dispute between people who agreed, but who understood that agreement in different ways. Some took the criticism of GOTO simply to mean that the elimination of GOTO statements led to better programs, while others saw behind the criticism of GOTO a revolutionary change in the whole concept of programming, a new approach to programming which they referred to as Structured Programming. But even the

term "structured programming" does not differentiate the two sides of the unstated controversy clearly, because, as will be seen, the term came to mean different things to different people.

Behind the apparent agreement, there can be seen what appear to be misunderstandings about the meaning of the GOTO letter. Misunderstandings, however, are rarely simply misunderstandings: behind the different interpretations of the GOTO letter lie divergent experiences, divergences that continue to shape the world of programming. When asked whether he expected the reaction that the letter received, Dijkstra replied:

> "No, I didn't, but that was because I was naive. In Europe I moved among computing scientists, among scientists in general, and the membership of the ACM probably consists largely of practitioners. Geldmongers. That's a reason for a shock too. Between the lines I raised all sorts of topics many people had never thought about: how to reason about a program. What I was essentially pointing out was that if you wish to see algorithms not as pragmatic pieces of text that you feed into an IBM computer, but if you prefer to treat them as a mathematical object, that in that case the GOTO statement is a complexity generator. And the suggestion that programs could and perhaps should be treated as mathematical objects, well that was a controversial one of course. Practitioners definitely did not like the suggestion" (Dijkstra interview).

Behind the GOTO letter and the whole methodological critique of programming practice was a web of conflicting interests. The essence of the GOTO letter was an attack on the established, FORTRAN-dominated practice of programming. Perhaps it was this which gave the GOTO controversy its imbalance: it appears a very one-sided controversy partly because only one side was stated: "I think a lot of people ignored it; anybody writing assembly code simply ignored the whole question and just carried on writing code as before" (Buxton interview). Probably this sort of pattern is common to a lot of scientific development, at least

where it aims to make an impact on practice: "the researchers are talking at or to some practitioners, while few of the practitioners are talking back; there is no real dialogue" (Gries 1979b, 360).

Dijkstra is very outspoken in his criticism of the "shockingly low" professional standards of the "average programmer" (1971, 360): "the world today has about a million 'average programmers', and it is frightening to be forced to conclude that most of them are the victims of an earlier underestimation of the programmer's task and now find themselves lured into a profession beyond their intellectual capabilities. It is a horrible conclusion to draw, but I am afraid that it is unavoidable" (1969, 36).

The criticism extends beyond the programmers, however. The condemnation of the existing practices of programming implies a condemnation of the whole environment in which programming takes place. The "rigorous approach" advocated by Dijkstra and the others is not promoted by the chaotic and frantic atmosphere that seemed to dominate software production. Dijkstra's description of the typical software project is reminiscent of accounts of the OS/360 and other large software projects:

> "One of the most common forms starts as an existing project, but as this project proceeds, deadlines are violated and what started as a fascinating thriller slowly turns into a drama, to be played by an ever increasing number of actors, the majority of which know perhaps their own part but have certainly lost their grasp on the meaning of the performance as a whole. At last the curtain falls, only because it is too late, not because anything has really been completed; the final piece of software is still full of bugs and will remain so for the rest of its days" (1972a, 219-220).

In the commercial world, everything is too rushed. In Dijkstra's terms, the

commercial world suffers from a low Buxton Index, a very short-term perspective, whereas scientific progress demands the long-term view: "in the industry, particularly the American one, but by now they've been copied in Europe as well, the top priority is speed, to have their product from the moment of its conception as soon as possible on the market". The way to move forward scientifically, however, is "not to be in a hurry...to guess which problems will be hot issues five to fifteen years from now" and to work on those (Dijkstra interview).

Dijkstra is very critical of the business world, and particularly of IBM and its influence. At one conference, organised by IBM, "I was severely shocked by the cultural level of the business participants. Their jokes were stale and sordid and - for people in business this amazed me - they could not drink their alcohol with style... But also technically, they were absolutely uneducated" (1982,128). He certainly does not see IBM as promoting scientific programming: "Turski's comments were short: 'They don't want computer scientists, nor software engineers, they want brainwashed mental cripples.' It is too true" (1982, 128).

The influence of IBM spreads far and wide. It shapes programming through the design of the IBM machines and the use of the IBM-sponsored languages. Thus, the architecture of the third generation computers, particularly the 360, (as contrasted with the Burroughs B5000) did much to shape the subsequent development of software. "When these machines were announced and their functional specifications became known, many among us must have become quite miserable; at least I was. It was only reasonable to expect that such machines would flood the computing community, and it was therefore all the more important

that their design should be as sound as possible. But the design embodied such serious flaws that I felt that with a single stroke the progress of computing science had been retarded by at least ten years; it was then that I had the blackest week in the whole of my professional life" (Dijkstra 1978, 12). As for the languages, "the intellectually degrading influence of what is becoming known as the pure FORTRAN environment is... grossly underestimated" (1971, 360).

The opposition to the influence of IBM is something which underlies a lot of computer science, especially in Europe. It was seen in the discussion of ALGOL that one of the considerations behind the original ALGOL project was the desire to resist the growing influence of IBM in Europe, and that ALGOL was seen by some as being specifically directed against FORTRAN and therefore IBM. Criticism of IBM is also an important theme in Dijkstra's work and is one element in a more general criticism of American and commercial influence, and this is combined with a defence of what Dijkstra sees as traditional European scientific values. In a note "On the Fact that the Atlantic Ocean has Two Sides", he explains the differences between European and American perspectives on computer science in two ways. The first of these is the Buxton Index, the length of time in the future for which a person or institution plans: "on the average, the European Buxton Index seems to be larger than the American one" (1982, 270). The second difference relates to the attitude of a society towards science. "The questions are: how does science justify itself, why does a society tolerate scientists? The way in which these questions are answered has a deep influence on the scientist's behaviour, not only on the way in which he presents his results, but also on his way of working and his choice of topics. Traditionally there are two ways in which science can be justified, the Platonic and the pragmatic one. In the Platonic way -

'l'art pour l'art' - science justifies itself by its beauty and its internal consistency, in the pragmatic way, science justifies itself by the usefulness of its products. My overall impression is that along this scale - which is not entirely independent of the Buxton Index - Europe, for better or for worse, is more Platonic, whereas the USA, and Canada to a lesser extent, are more pragmatic". Dijkstra's own "Pan-Academic preferences...are most definitely Platonic" (1982, 271).

There are two elements here. One is the cultural defence of traditional academic values against the influence of greater pragmatism. The second concerns the relation between scientific programming and the market. The problems of programming are not just cultural: they have to do with the fact that software and computers are produced for sale on the market. His criticism of IBM links their commercial success with their lack of scientific and mathematical quality: "technically, their machines were always very old-fashioned. It has been argued that it is misleading to consider IBM as a computer manufacturer. They didn't make computers, they made money. For the computing industry, it was, of course, a colossal *must* to hide the machine's mathematical nature as much as possible...If you have to sell the machines to a public that considers mathematics as the pinnacle of user-unfriendliness, then you just have to lie, so then you have to take out advertisements: our machines take 20,000 business decisions per second for you" (Dijkstra interview).

The market is not only opposed to mathematics, it favours complexity over simplicity: "a tool is useful to the extent that we can know its properties, and therefore its functional specification should be simple. However, many, many

commercial organisations, even if they were able to make a thing like that, wouldn't dare to market it, because they have a feeling that in order to promote their product, you have to include all sorts of user-friendliness... so the thing would be given all sorts of frills, bells and whistles, which may be nice to sell the product, but they're a pain in the neck to use" (Dijkstra interview).

In the market environment, computers and software appear just like any other commodity, and programming like any other production job. Many people in the computing industry "do their best to ignore" the specificity of computers and computer programming "and try to do the business as any other business... Apple has a new executive president, contracted a few years ago. He came from Pepsi Cola, and the reason they contracted him was that, having run Pepsi Cola, he knew what it meant to be not the first, but the second business in the area. Now you know how much understanding the executive president will have of technology. It doesn't matter whether a business sells computers or Pepsi Cola or chewing gum" (Dijkstra interview).

If software is a commodity like any other commodity, then this will have an effect on the way in which it is produced. Dijkstra is particularly critical of the way in which "programmer productivity" is measured: "for instance, one of the things I have heard structured programming being praised for is that it has been responsible for a dramatic increase in programmer productivity. But is it adequate to consider programming as a production job? And then you look at how software engineers measure programming productivity - number of lines of code produced per year or per month. Now, you shouldn't talk about number of lines of code produced, you should talk about number of lines of code consumed. They book

it on the wrong side of the ledger" (Dijkstra interview). It is absurd to measure programmer productivity in terms of lines of code produced when the aim should be to write programs in as few lines as possible: "In science we measure physical quantities, something that is meaningful because (the measurements of) these quantities are supposed to satisfy certain explicitly stated laws; the purpose of the measurements is to confirm or refute the supposed laws. Here, however, to 'measure' is used in the sense of 'attaching a number to', in very much the same way as psychologists construct an IQ. (It is a fallacy to assume that an IQ 'measures' something!" (Dijkstra 1982, 221).

From the clear connections that Dijkstra draws between the failings of software and the fact that software is produced for the market, it would seem legitimate to conclude that scientifically rigorous programming is not possible in a market environment. Dijkstra himself, however, does not draw such a politically radical conclusion. Where he does discuss the issue, his comments do not seem entirely consistent with the critical tone of most of his writing. A few years after accepting the fellowship from Burroughs, he comments in his discussion of the differences between the United States and Europe: "most of you must have been confronted with my Pan-Academic prejudices, which are most definitely Platonic, and by now you may wonder how in the world I could join not only an industrial organisation...but even an American one. But the answer is quite simple: in computing science the conflict need not exist - and that is what makes the subject so fascinating! To quote C.A.R. Hoare - from memory - : 'In no engineering discipline does the successful pursuit of academic ideals pay more material dividends than in software engineering.' I could not agree more" (1982, 271).

The solution for Dijkstra lies not in the abolition of the market but in laying greater stress on education, science and mathematics. It is necessary to convince the computer industry that "mathematical elegance is a matter of life and death... I have to convince the industry, by saying it over and over again" (Dijkstra interview).

Software Engineering

The other approach to laying solid foundations for software which emerged at the end of the 1960s and the beginning of the 1970s is *software engineering*. This approach too started out from a critique of pre-Garmisch programming and from the perception that software was in crisis. The experience of the third generation had shown that people did not know how to produce and manage good software.

Software was in a mess or, as Brooks put it, in a tar pit:

> "No scene from pre-history is quite so vivid as that of the mortal struggles of great beasts in the tar pits. In the mind's eye one sees dinosaurs, mammoths, and sabertoothed tigers struggling against the grip of the tar. The fiercer the struggle, the more entangling the tar, and no beast is so strong or so skilful but that he ultimately sinks. Large-system programming has over the past decade been such a tar pit, and many great and powerful beasts have thrashed violently in it. Most have emerged with running systems - few have met goals, schedules, and budgets. Large and small, massive or wiry, team after team has become entangled in the tar. No one thing seems to cause the difficulty - any particular paw can be pulled away. But the accumulation of simultaneous and interacting factors brings slower and slower motion. Everyone seems to have been surprised by the stickiness of the problem, and it is hard to discern the nature of it. But we must try to understand it if we are to solve it" (Brooks 1974, 4).

Brooks of course had considerable experience of the tar pit. After managing the OS/360, he left IBM to become a professor at the University of North

Carolina, and he later published his reflections on the problems of managing large software projects in a book called *The Mythical Man Month*. The way out of the tar pit, according to Brooks was to "try to understand it". The key was to learn from the mistakes that had been made: he optimistically prefaces the book with a Dutch proverb, "een schip op het strand is een baken in zee - a ship on the beach is a lighthouse to the sea".

Many of Brooks' conclusions reflected a new awareness of the nature of software. Two features were particularly important. One was that software is abstract, the product of mental labour:

> "The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds his castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures" (Brooks 1974, 7).

However, software is not simply abstract, it is also a product, a product that is part of a system, a "programming systems product": "the program construct unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms" (Brooks 1974, 7). A program, therefore, is not self-contained: it is part of a wider system and must produce useful results.

The third generation systems, in this view, had run into difficulties partly because of their failure to recognise the particular nature of software. One of the most important problems had been the scheduling of large projects: "More software projects have gone awry for lack of calendar time than for all other

causes combined" (Brooks 1974, 14). Because computer programs are constructed not from physical materials but from thought, it was expected that ideas would be easy to implement:

> "The programmer builds from pure thought-stuff: concepts and very flexible representations thereof. Because the medium is tractable, we expect few difficulties in implementation; hence our pervasive optimism. Because our ideas are faulty, we have bugs; hence our optimism is unjustified" (Brooks 1974, 15).

A second reason that Brooks gave for scheduling problems was the false assumption that people and months were interchangeable:

> "The second fallacious thought mode is expressed in the very unit of effort used in estimating and sheduling: the man-month. Cost does indeed vary as the product of the number of men and the number of months. Progress does not. Hence the man-month as a unit for measuring the size of a job is a dangerous and deceptive myth. It implies that men and months are interchangeable" (Brooks 1974, 16).

However,

> "When a task cannot be partitioned because of sequential constraints, the application of more effort has no effect on schedule. The bearing of a child takes nine months, no matter how many women are assigned" (Brooks 1974, 17).

On the contrary, increasing the number of programmers working on a project could easily lead, not to speeding up the project, but to increasing the time required. Firstly, the new programmers had to be trained for the project:

> "Each worker must be trained in the technology, the goals of the effort, the overall strategy, and the plan of work. This training cannot be partitioned, so this part of the added effort varies linearly with the number of workers" (Brooks 1974, 18).

Treating programming as though it were an ordinary mass production process

and adding more and more programmers to finish the job quickly - sometimes referred to as the Mongolian Hordes or Human Wave approach (Ogdin 1972, 21) - did not solve the problem at all:

> "When schedule slippage is recognised, the natural (and traditional) response is to add manpower. Like dousing a fire with gasoline, this makes matters worse, much worse. More fire requires more gasoline, and thus begins a regenerative cycle which ends in disaster" (Brooks 1974, 14).

A major difficulty arising from the employment of large numbers of programmers was the problem of communication. If the organisation of the project required communication between the programmers working on it, then, since communication took time, the time required for the project began to increase exponentially with the number of programmers:

> "If each part of the task must be separately coordinated with each other part, the effort increases as n(n-1)/2. Three workers require three times as much pairwise intercommunication as two. If, moreover, there need to be conferences among three, four, etc, workers to resolve things jointly, matters get worse yet. The added effort of communicating may fully counteract the division of the original task" (Brooks 1974, 90).

The question of communication between programmers was thus an important issue in the management of large software projects. Inevitably communication took time away from production. As communications between programmers increased, productivity fell:

> "Joel Aron, manager of Systems Technology at IBM in Gaithersburg, Maryland, has studied programmer productivity when working on nine large systems (briefly, large means more than 25 programmers and 30,000 deliverable instructions). He divides such systems according to interactions among the programmers (and system parts) and finds productivities as follows:
> Very few interactions: 10,000 instructions per man year
> Some interactions: 5,000
> Many interactions: 1,500" (Brooks 1974, 90).

Consequently, one of the lessons that Brooks drew from the OS/360 experience was that it was essential for management to reduce communication between programmers:

> "The purpose of organization is to reduce the amount of communication and coordination necessary; hence organization is a radical attack on the communication problems treated above. The means by which communication is obviated are division of labour and specialization of function. The tree-like structures of organizations reflect the diminishing need for detailed communication when division and specialization of labor are applied" (Brooks 1974, 79).

One of the problems facing software specialists was to find a way of organising an abstract labour process in such a way that there was a clear division of labour which would reduce the need for communication to the minimum. Reducing the need for communication was important both to increase productivity and to make the whole programming project more manageable and so increase the reliability of the program. The abstract nature of the production process had a number of implications: it not only meant that the reproduction costs of software were close to zero, but it also made the division of labour a particularly difficult problem.

Brooks was not the only software manager to reach this conclusion. An article by another manager, published in February 1968, spoke of "a structural theory of programs" as having emerged from industrial practice, a theory which he contrasted with the unstructured algorithmic approach still prevalent in the universities. The article concluded by calling on the universities to provide a more rigorous foundation for the practices being developed by industry:

> "The theory emerging from industry is tentative, incomplete and perhaps too compromised by its pragmatism. We vitally need the theoretical contributions which the university can make, not just in mathematical optimisation but in the structure of the programs, not only in compiler

design but in the psychology of programming" (Constantine 1968, 19).

In order to deal with the problem of division of labour, two other managers in IBM, Harlan Mills and Fred Baker, turned to the ideas of structured programming. It was Mills and Baker who first introduced the concept of structured programming into the practice of large-scale software production in 1969/70, while working on a project to index the New York Times. They saw the ideas of structured programming as being important in helping to establish a good, clean design for a programming project. Good design was the key to the internal division of labour within the project. Communication in a well-designed project should be limited to contact between programmers at the top of hierarchically organised groups.

The application of structured programming was said to be a "startling success" (Miller and Lindamood 1973, 55), "with reports of greatly increased programmer productivity and very greatly reduced coding error rates (one detected error per 10,000 lines of coding, or one per man-year)! Absolutely incredible, but these were the facts" (McCracken 1973, 51).

The New York Times index was followed by other projects applying the ideas of structured programming, so that by 1973, an article in *Datamation* commented *that "what was for a few* years an underground ivory tower - to mix metaphors a bit - has now come out in the open as a very important thing indeed. The practicality of the theory has been demonstrated in a fashion that simply cannot be ignored" (McCracken 1973, 52).

The application of structured programming led to a refinement of the

techniques of software management and particularly to the development of the Chief Programmer Team. As Baker put it at a conference in 1974:

> "FSD [Federal Systems Division] has been active in the development of structured programming techniques. This has led to organisations, procedures and tools for applying them to production programming projects, particularly with a new organisation called a Chief Programmer Team" (Baker 1975, 39).

The chief programmer team is described by Baker and Mills as "a new managerial approach to production programming. While the approach is made possible by recent technical advances in programming, it also incorporates a fundamental change in managerial framework which includes restructuring the work of programming into specialised jobs, defining relationships among specialists, developing new tools to permit these specialists to interface effectively with a developing, visible project; and providing for training and career development of personnel within these specialties". The new approach, they claim, "contrasts sharply with that of conventional programming groups which frequently suffer from lack of functional separation, discipline and teamwork" (Baker and Mills 1973, 58). The chief programmer team is seen as a reaction against the Mongolian Hordes approach of many of the large projects in the 1960s: the emphasis is on small teams with charismatic and highly specialised team leadership and a very clear assignment of roles within the team. The concept was seen as being similar to that of a surgical team, where skilled technical assistants provide the instruments to the chief surgeon who performs the specialist work required (Brooks 1974, 27).

However, the internal division of labour is only one aspect of the management problem. The other aspect derives from the fact that software is a *product*, which

must respond to users' requirements and is shaped by those requirements.

The emphasis on user requirements is at the core of another approach to software engineering. In this view structured programming is a useful technique but is of limited importance when one is dealing with real machines which must be responsive to real and changing requirements. Software is produced within the constraints of a market economy for use by users whose requirements it must satisfy. Software engineering is not of concern only to software engineers, but affects also users, managers, marketing people and all those who participate in the software environment. It is important, therefore, for all of these groups to understand and participate in the software engineering process. The focus is not on the program as a closed process but on the dynamic relation between programming and the environment which surrounds it.

To look at programming in terms of its relationship to users' requirements implies a different view of programming as an activity. Programming in this view cannot be seen as the rigorous application of a fixed body of principles; it involves a constantly changing relationship to a constantly changing environment. That shapes the activity of the programmer. There are no recipes, no hard rules that can be applied to an environment that is constantly evolving. That is why it is better to see the programmer as a software engineer who interacts with the surroundings and learns from them, through experience. In this sense, software engineering is a craft.

Software engineers, then, are craftspersons. Their ability will be shaped by the tasks they have to perform and by the environment within which those tasks

are performed. It is important for the development of the software engineers' ability that the environment should be a good one, "that it should have a 'strong' culture in good software engineering practice" (Macro and Buxton 1987, 3). Ability is also shaped by experience, by "the stage that software engineers have reached in consolidating their subject knowledge into a framework of practice sufficient for most eventualities, yet flexible enough to be extended and modified when necessary". It follows that in this sense the career of the software engineer "begins with an apprenticeship, and proceeds through junior and senior software engineer levels to that of 'master' in the craft" (Macro and Buxton 1987, 3-4). The education of the programmer cannot be purely theoretical.

In this perspective, rigidly hierarchical structures in the division of labour are a barrier to the making of good software engineers. The elitism in the traditional concept of the chief programmer team, as advocated by Mills, is counter-productive:

> "Now, I think the whole thrust of software engineering has very much gone the other way, that really high quality software is not written by COBOL-plonkers, and it's not written by an unduly hierarchical team with different skill levels, but it's actually written as a team activity with high skill levels on the part of everybody" (Buxton interview).

The contrast between the hierarchical approach and the craft engineering approach can be seen if you "go and have a look at the software houses that specialise in writing commercial programs, and go in particular and have a look at the big insurance companies, people who write very large commercial programs, and you will find they talk about systems analysts and programmers and these are different kinds of people. Go to Logica or SDL or SSL or whatever, and they don't. They talk about levels of programmer, depending on experience and

ability" (Buxton interview).

Craftspersons work with tools. Part of mastering the craft is the process of developing and mastering the tools and techniques of the craft. These tools (such as languages, editors, graphic packages, expert systems etc) help the software engineer to cope with the enormous complexity which arises from the changing relation between software and the multiplicity of users' requirements. For the craftsperson, building the right product in the right way is not a question of mathematical proof. The emphasis, therefore, is on devising good tools for testing correctness: tools such as prototyping, modelling, demonstrators etc.

The emphasis on user requirements implies not only a more dynamic but also a more long-term view of software. Programming is the programming of dynamic processes and the software must be flexible and dynamic to respond to the users' needs. Software production does not simply involve programming in the usual sense. This is just one stage in the entire *lifecycle* of the software, a central concept in this approach:

> "A major reason for the magnitude of the current software crisis is the failure of previous methodologies to analyse software costs over the *entire* life cycle of any given system. This life cycle extends from the design phase through initial implementation and testing, and includes modification and maintenance. Previous methodologies have failed to recognise the magnitude of the effort required to maintain and modify existing software. Software systems are 'living', 'growing' entities that require signficant care and attention in order to reach maturity" (Gillett and Pollack 1982, 4).

In this perspective, the actual process of programming is only part of the whole process of producing the programming product. A clear distinction is established between the program and the product. The product is external to the

process of programming. It is because of this externality that documentation, maintenance and testing acquire importance. Since the program is not the product, the testing of the software is a separate activity from the writing of the program; similarly, because software is an abstract product, documentation is required to make the program visible, and maintenance too is seen as an activity external to the programming process. These are all seen as different aspects of the engineering of a product.

The concept of "lifecycle" is both a technical and a managerial concept. From its origins and by virtue of its practical orientation, it is impossible to separate the technical from the managerial in the software engineering approach. For this reason, Macro and Buxton, in their recent book on the topic, argue that the definition of software engineering must include expicit mention of management considerations. The definition which they propose for software engineering is "the establishment and use of sound engineering principles and good management practice, and the evolution of applicable tools and methods and their use as appropriate, in order to obtain - within known and adequate resource provisions - software that is of high quality in an explicitly defined sense" (Macro and Buxton 1987, 3).

In the "real world" of programming practice, the production of software is inextricably linked with questions of management. In industry, software engineering "tends to mean primarily software management - programming teams and quality control and using the software life-cycle and specification and design and so on"; whereas "the academics look at software engineering and say - well, Edsger [Dijkstra] is the purest example, but Edsger says quite bluntly that the

only way that you'll properly engineer a piece of software is to realise that it is a mathematical construct and you've got to prove that it is right", for the industrialist there are practical problems of management: "the industrialist looks at a piece of software and says, 'I am going to need a million-line program to do this and it'll take a hundred people to write it for five years, and we'd better have all the necessary techniques and controls to actually do that'" (Buxton interview). In practice, therefore, an aim of software engineering is to provide the software manager with "the necessary techniques and controls" to produce good software.

Since software engineering is a practical real world activity, this means that it is necessary, according to its proponents, to come to terms with all the constraints that this implies. This means, firstly, a recognition that software has to be produced within the tight constraints of time and money imposed by the market. If the software is not efficiently contructed, it will not be competitive.

It also means recognising that programming can only be done with the programmers available. The rapid expansion of computer use during the 1960s led to an explosion in the demand for computer programmers. As a result, many people had been recruited into programming without adequate training: "the industry has grown so quickly that the demand for people has far exceeded the supply of well-trained people" (Guttag interview). Software engineering acknowledges that it is not possible to do away with the "average programmer": "on the large scale that the computing world has got to now, you can't be having the Dijkstras of this world do all the programming for you, and there are issues of people can only work with the resources that they've got, and the people that they have and the training that those people have got and their calibre" (Randell

interview).

The third way in which the "real world" orientation shapes software engineering is through the relation of programming to changing user requirements. The mathematical approach assumes the existence of a clearly defined problem which must be solved: a clearly defined set of user specifications establishes the framework within which the problem is to be tackled. In practice, however, the users' requirements will often be ill-defined, both in the sense that the users may not know exactly what they want and in the sense that, even if they know, it may be difficult to communicate those wants clearly. In the discussion of the experience of the users with software, it was seen that these problems of communication and definition reflect and are reflected in tensions between producers and users, and between line managers and data processing departments. The process of translating user requirements into specifications may succeed in providing a clear framework for the design of the program, but it may still be found that they do not accurately reflect what the user wanted, or that the user's requirements subsequently change. The tensions of the real world penetrate into the practice of programming: the orientation of software engineering implies, therefore, a relation not to a well-defined and stable specification but to ill-defined and unstable requirements.

The two approaches discussed in this chapter differ fundamentally in their conception of the nature of software and of the role of computer science. The central point is the relation of the computer scientist to the "real world" of software production.

On the one hand, the software engineers criticise the methodological school for having "too simple a view of the world" (Wegner 1979, 206). Their contribution is seen as being important but limited and ultimately unrealistic.

One aspect of this is the methodologists' preference for concentrating on small-scale problems. According to their critics, they insulate themselves from the difficulties that most programmers confront by the way in which they choose the problems they consider. Some of the main points of criticism are summarised in a discussion by Parnas of a paper by Gries; although Parnas shares the "belief that a more mathematical and disciplined treatment of programs is one of the keys to improved program quality", he argues that:

> "a. The programs by Dijkstra, Gries and others are the subject of months, years, or in some cases, centuries of study by disciplined, creative, and mathematically trained minds. Neither they, nor their working conditions, are typical.
> b. The problems have been selected because they are easily subject to mathematical treatment. Most programmers cannot select their problems.
> c. The programs are designed for publication, not for use. Few programmers can design their own, convenient notation and assume that it will be implemented.
> d. The methods work only where variables are uniquely and unambiguously identifiable. Major unsolved problems remain before one can deal routinely with programs that make sophisticated use of arrays or core addresses.
> e. Even with these advantages, incorrect programs have been published" (Parnas 1979, 353).

The insistence of the methodologists on program verification has similarly been criticised for not being applicable to the real difficulties of large programming projects. Even the concept of verification, it has been argued in a paper by De Millo, Lipton and Perlis, rests on a fundamental misunderstanding of the nature of mathematical proof, which denies the relation between proof and social processes:

"We believe that, in the end, it is a social process that determines whether mathematicians feel confident about a theorem - and we believe that, because no comparable social process can take place among program verifiers, program verification is bound to fail. We can't see how it's going to be able to affect anyone's confidence about programs" (De Millo, Lipton and Perlis 1979, 271).

Dijkstra's response to the "real-world" criticism is to deny the charge:

> "The appeal to the real world is always a hidden threat not to challenge the other one's tacit assumptions. The real-world (sometimes with a hyphen in between): that's the catchword of sometimes rabid anti-intellectualism. It's true that a lot of software is made without proper functional specification. My suspicion is that that's precisely why it's such a mess" (Dijkstra interview).

On the more specific argument that programming does not take place in an environment of clearly defined specifications, Dijkstra replies that:

> "The fallacy in this argument is to be found in the confusion between between exact and complete; although the program requirements, at a given stage, may be incomplete, a certain number of broad characteristics will be exactly known. The abstract program can see to it that these broad specifications are exactly met, while more detailed aspects of the problem specification are catered for in the lower levels...By successively adding more details in the lower levels [the programmer] eventually pins his program down to a solution for the given problem" (Dijkstra 1971, 367).

"Real world" approaches to programming in general are guilty of anti-intellectualism. Dijkstra originally welcomed the term software engineering, "because in my view of the engineer, it included the mathematical involvement" (Dijkstra interview). But then, under the influence of the American interpretation of engineering, "software engineering went down the drain; the current meaning given to software engineering, if you read a book on it, you'll discover that that group has accepted as charter 'how to program if you can't'. It's really dismal" (Dijkstra interview)

The structured programming of the real world, as practised by Mills and Baker, is no better. Dijkstra repudiates the "structured programming" which many people see as the product of his ideas, with almost the same horror as Frankenstein rejected his creature: "since IBM stole the term 'structured programming' I don't use it anymore myself" (1982, 341). There is a dramatic scene in which Frankenstein comes face to face with the creature: Dijkstra describes his arrival at a conference in Canada sponsored by IBM in the following terms:

> "From Monday through Wednesday IBM sponsored there a conference on Software Engineering Education, and in my innocence I had expected an audience of computer scientists. My driver, however, was a manager who opened the conversation with something like 'So you are the world expert on structured programming and chief programmer teams'. Then I knew I was out in the wilderness and politely refused to be associated with Harlan D. Mills" (1982, 126).

And later in the same conference:

> "Later I heard Harlan Mills give a summing up of some of the things I had said - together with some Harlanesk additions - for that business audience. It was terrible, a misuse of language to which to the best of my powers I could not give a meaning. So, every third phrase I interrupted Harlan 'please could you explain or restate what you tried to say', but it was hopeless. Tom Hull helped me and I was very grateful to him. Later, when it was all over, our eyes met, and Tom gasped 'Jezus!'. It was the first time that I had heard him use strong language. How to sell empty but impressive slogans under the cloak of academic respectability...." (1982, 128).

The real-world charge is, in this view, not only anti-intellectual; it also implies a short-term, blinkered view of what constitutes reality. This emerges, for example, in a comment by Dijkstra on a paper on software engineering education, and particularly in a remark on the authors' view of the proper role of universities:

> "They include producing the graduates industry and the government ask for. An alternative view is trying to educate the graduates the rest of the world will need in the future, independent of the question to what extent the rest of the world understands its future needs" (Dijkstra 1982, 220).

Dijkstra's long-term view of the world does not necessarily make his theories better than other theories, but it does give them a sharp critical force. In more positive terms, it is possible to argue that the limitations of the Dijkstra approach are also a strength. The methodological approach is a much more systematic and in many ways more cautious approach. It is an approach that rejects the "daring irresponsibility" of much current programming and insists that if something cannot be done properly, it should not be done at all. This implies a less hectic, and probably also less rapid progress in the development of software and therefore of computerisation, but perhaps a safer one.

When the possible consequences of software failure are so enormous, there is much to be said for this approach, but it is utopian. As has been seen throughout this thesis, it is not "scientific reason" that shapes the development of software. Software has been and is being shaped by the tensions of the world that surrounds it: a world of fierce competition between computer manufacturers, between marketing and technical people, between users and producers, between users themselves. Software development operates in a constantly changing environment, in a world of ill defined requirements. To try to convince IBM and the world not to rush and not to be too ambitious, that the only way forward is to program scientifically, in a world of good programmers and clearly defined specifications, is a hopeless task. The lid on Pandora's Box is not so easily closed.

end

References

Abelson P.H. & Hammond A.L. (1980), "The Electronics Revolution", in Forester T. (ed.), *The Microelectronics Revolution*, Blackwell, Oxford

Alexander T. (1969), "Computers Can't Solve Everything", *Fortune*, October

Alt F.L. (1969), "The Costs of Computing and Failure in Computing Programs", *Computers and Automation*, January, pp. 14-16

Arblaster A.T., Sime M.E. and Green T.R.G. (1979), "Jumping to Some Purpose", *The Computer Journal*, Vol. 22, No. 2, pp. 105-109

Armer P. (1970), "Computer Applications in Government", in Taviss I. (ed.), *The Computer Impact*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, pp. 123-129

Automatic Data Processing Newsletter Vol Ix No 25, 10 May, 1965, "EDP in State and Local Governments at Mid-Decade"

B-5000 Discussion (1987), "Discussion: The Burroughs B 5000 in Retrospect, *Annals of the History of Computing*, Vol. 9, No. 1, pp.37-92

Backus J. (1980), "Programming in America in the 1950s - Some Personal Impressions", in Metropolis N., Howlett J., Rota G.-C., *A History of Computing in the Twentieth Century*, Academic Press, New York, pp. 125-136

Backus J. (1981), "The History of FORTRAN I, II and III", in Wexelblat (1981), pp 25-74

Baker F.T. (1975), "Organizing for Structured Prgramming", in Hackl C.E. (ed), *Programming Methodology*, Springer Verlag, Berlin

Baker F. T. and Mills H.D. (1973), "Chief Programmer Teams", *Datamation*, December, pp. 58-61

Barron D.W. (1968), *Recursive Techniques in Programming*, Macdonald, London

Bauer F.L. (1969), "Software Engineering: A Conference Report", *Datamation*, October, pp. 189-192

Bauer F.L. (1980), "Between Zuse and Rutishauser - The Early Development of Digital Computing in Central Europe", in Metropolis N., Howlett J., Rota G.-C., *A History of Computing in the Twentieth Century*, Academic Press, New York, pp. 505-524

Belady L.A. and Lehman M.M. (1979), "Characteristics of Large Systems", in Wegner (1979), pp. 106-138

Bemer R.W. (1969), "A Politico-Social History of ALGOL", in Halpern M.I. and Shaw C.J. (eds), *Annual Review in Automatic Programming*, Vol. 5, Pergamon Press, Oxford, pp. 151-238

Bigelow R.P. (1968), "Legal Aspects of Proprietary Software", *Datamation*, October, pp. 32-39

Boehm B.W. (1973), "Software and Its Impact: A Quantitative Assessment", *Datamation*, May, pp. 48-59

Boehm G.A. (1962), "Helping the Executive to Make up his Mind", *Fortune*, April, p.128

Boering B.W. (1967) "Multi-Programming: Who Needs It?", *Computers and Automation*, pp. 36-37

Bouvard J. (1970), "The Translation of User Requirements into Fourth Generation Software", in Gruenberger F. (ed.), *Fourth Generation Computers*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, pp. 117-130

Brandon D.H. (1970), *Management Planning for Data Processing*, Brandon/ Systems Press, Princeton, New York

Braverman H. (1974), *Labor and Monopoly Capital: The Degradation of Work in the Twentieth Century*, Monthly Review Press, New york and London

Brock G.W. (1975), *The US Computer Industry*, Ballinger, Cambridge Massachusetts

Brooks F.J. (1974), *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, Massachusetts

Burck G. (1964), "The Boundless Age of the Computer", *Fortune*, March, April, May, June

Business Week 5th November 1966, p. 127 "Software Gap - A Growing Crisis for Computers"

Carlson W.E. (1976), Software Research in the Department of Defense", *Proceedings of the Second International Conference on Software Engineering*, pp. 379-393

Computers and Automation February 1967, p. 11, "Software Gap - A Growing Crisis for Computers"

Computers and Automation June 1967, p. 77, "World Computer Census"

Computers and Automation June 1968, p. 132, "World Computer Census"

Computers and Automation July 1968, p. 14, "Association of Independent Software Companies Formed"

Computers and Automation January 1969, pp. 72-23 "As we go to press"

Computers and Automation February 1969, p. 8 "As we go to press"

Computers and Automation July 1969, p. 8 "As we go to press"

Computers and Automation August 1969, pp. 39-40, "Reactions to IBM'S 'Unbundling'"

Computers and Automation October 1969, p. 11, "As we go to press"

Computers and Automation November 1969, p. 11, "As we go to press"

Constantine L.L. (1968), "The Programming Profession, Programming Theory, and Programming Education", *Computers and Automation*, February, pp. 14-19

Conway M.E. (1968), "On the Economics of the Software Market", *Datamation*, October, pp 28-31

COSERS (The Computer Science and Engineering Research Study) (B.W. Arden, ed), *What Can be Automated?* MIT Press, Cambridge Massachusetts and London, 1980

Dansiger S.J. (1968), "Proprietary Protection of Computer Programs", *Computers and Automation*, February, p. 32

Datamation March 1969, p. 205, "Software Group Splits over ALGOL 68"

Datamation June 1968, pp. 72-77, "Separate Hardware/ Software Pricing?"

Datamation June 1968, p. 91, "Program Plagiarism Alleged in UK Case"

Datamation July 1968, p. 91, "ADR Receives First Program Patent"

Davidson L. (1968), "Practical Considerations in Program Patentability", *Computers and Automation*, May, pp. 12-13

De Millo R.A., Lipton R.J. and Perlis A.J. (1979), "Social Processes and Proofs of Theorems and Programs", *Communications of the ACM*, pp. 271-280

Denning P.J. (1971), "Third Generation Computer Systems", *Computing Surveys*, Vol.3, No. 4, December, pp. 175-216

Dickson G.W. and Simmons J.K. (1970), "The Behavioral Side of MIS", *Business Horizons*, August, pp. 59-71

Dijkstra E.W. (1968), "GOTO Statement Considered Harmful", *Communications of the ACM*, Vol. 11, No. 3, March, pp. 147-148

Dijkstra E.W. (1969), "On the Interplay between Mathematics and Programming", in *Program Construction, Lecture Notes in Computer Science*, Springer Verlag, New York, pp. 35-46

Dijkstra E.W. (1971), "Concern for Correctness as a Guiding Principle for Programming", in *The Fourth Generation*, Infotech State of the Art Report, no. 1, Infotech International, Maidenhead, pp. 359-367

Dijkstra E.W. (1972a) "Notes on Structured Programming", in Dahl O-J, Dijkstra E.W., Hoare C.A.R., *Structured Programming*, Academic Press, London and New York

Dijkstra E.W. (1972b), "The Reliability of Programs", in *High Level Languages*, Infotech State of the Art Report, no. 7, Infotech International, Maidenhead, pp.218-232

Dijkstra E.W. (1978), "The Humble Programmer", in D. Gries (ed.), *Programming Methodology: A Collection of Articles by Members of IFIP WG2.3*, Springer Verlag, New York

Dijkstra E.W. (1982), *Selected Writings on Computing: A Personal Perspective*, Springer Verlag, New York

Doherty W.J. (1970) "Scheduling TSS/360 for Responsiveness", *Proceeding of the Fall Joint Computer Conference*

EDP Weekly, 6th September, 1965, pp. 14-15, "The House of Representatives Passed the Amended Version of the Brooks Bill"

Ernst M.L. (1982), "The Mechanization of Commerce", *Scientific American*, Vol. 247, No. 3, September, pp. 110-123

Feeney J.M. (1981), "Management Information Systems - The Failure of Technology", in *Business Information Systems*, Infotech State of the Art Report, Series 9, No. 7, Pergamon Infotech, Maidenhead

Financial Times, 30 September 1983, "Suddenly, the great software bonanza"

Fisher F., McKie J., Mancke R. 1983, *IBM and the U.S. Data Processing Industry*, Praeger, New York

Fishman K.D. (1982), *The Computer Establishment*, McGraw-Hill, New York

Flynn R.L. (1974), "A Brief History of Data Base Management", *Datamation*, August, pp. 71-77

Fortune January 1965, pp. 171-172, "The Battle of the Computer Marketeers"

Frank W.L. (1976), "The Second Half of the Computer Age", *Datamation*, May, pp. 91-100

Freeman D. N. (1978), "Software Unbundling and Program Products", in *IBM*, vol 2, Infotech State of the Art Report, Infotech International, Maidenhead, pp. 135-146.

GCR, Garmisch Conference Report (1969), *Software Engineering*, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, edited by Naur P. and Randell B., NATO, Brussels

Gawne-Cain H. (1967), "Integrated EDP Systems in Business", in Fletcher A. (ed.), *Computer Science for Management*, pp. 157-172

Gilchrist B. and Wessel M.R. (1972), *Government Regulation of the Computer Industry*, AFIPS Press, Montvale, New Jersey

Gill S. (1960), "The Philosophy of Programming", *Annual Review of Automatic Programming*, Pergamon Press, Oxford Vol. 1, pp. 178-188

Gillett W.D. and Pollack S.V. 1982, *An Introduction to Software Engineering*, Holt, Rinehart and Winston, New York

Giuliano V.E. (1982), "The Mechanization of Office Work", *Scientific American*, Vol. 247, No. 3, September, pp. 124-135

Goldstine H.H. (1972), *The Computer from Pascal to von Neumann*, Princeton University Press, Princeton, New Jersey

Gordon R.M. (1968) Review of *The Management of Computer Programming Projects* by Lecht C.P., *Datamation* April, pp. 200-204

Gosden J. and Raichelson E. (1970), "The New Role of Management Information Systems", in Gruenberger F. (ed.), *Fourth Generation Computers*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, pp. 75-88

Granholm J.W. (1963), "ALGOL on the 7090", *Datamation*, April, p. 28

Greenbaum J.M. (1979), *In the Name of Efficiency*, Temple University Press, Philadelphia

Gries D. (1979a), "Current Ideas in Programming Methodology", in Wegner 1979, pp. 254-275

Gries D. (1979b) "A Comment on Parnas' Counterpoint", in Wegner 1979, pp. 359-370

Gunn T.G. (1982), "The Mechanization of Design and Manufacturing", *Scientific American*, Vol. 247, No. 3, September, pp. 86-109

Head R.V. and Linick E.F. (1968), "Software Package Acquisition", *Datamation*, October, pp. 22-27

Hertz D. (1969), *New Power for Management*, McGraw-Hill, New York

Hilton A.M. (1963), *Logic, Computing Machines and Automation*, Spartan Books, Wahington D.C.

Hirsch P. (1966), "The Patent Office Examines Software", *Datamation*, November, pp 79-81.

Hirsch P. (1972), "What's Wrong with the Air Traffic Control System?", *Datamation*, August, pp.48-53

Hoare C.A.R. (1978), "Software Engineering: A Keynote Address", *Proceedings of the Third International Conference on Software Engineering*

Hoare C.A.R. (1984) "Der neue Turmbau zu Babel", *Kursbuch*, March, pp. 57-73

Hopkins M.E. (1972), "A Case for the GOTO", *Proceedings of the ACM Annual Conference*, Boston, Massachusetts

Hopper, G.M. (1981), "Keynote Address", in Wexelblat 1981, pp 7-24

Huse E.F. (1967), "The Impact of Computerized Programs on Managers and Organizations: A Case Study of an Integrated Manufacturing Company", in Myers C.A., *The Impact of Computers on Management*, MIT Press, Cambridge Massachusetts, pp 107-129

IBM (1967), "IBM Operating System/360: Concepts and Facilities", in Rosen 1967, pp 598-546

Jones F. (1963), *Datamation* March

Jones R.C. (1968), "Separate Pricing For Hardware and Software", *Computers*

and Automation, July, p 12.

Kaplinsky R. (1984), *Automation*, Longman, Harlow, Essex

Klahr D. and Leavitt H.J. (1967), "Tasks, Organization Structures, and Computer Programs", in Myers C.A., *The Impact of Computers on Management*, MIT Press, Cambridge Massachusetts, pp 107-129

Kleiman H.S. (1969) "The Economic Promise of Computer Time Sharing", *Computers and Automation*, October pp. 47-49

Knuth D.E. (1974) "Structured Programming with GOTO Statements", *Computing Surveys*, Vol. 6, No. 4, pp. 261-301

Knuth D.E. and Trabb Pardo L. (1980), "The Early Development of Programming Languages", in Metropolis N., Howlett J., Rota G.-C., *A History of Computing in the Twentieth Century*, Academic Press, New York, pp. 197-274

Kraft P. (1977), *Programmers and Managers: The Routinization of Computer Porgramming in the United States*, Springer Verlag, New York

Laver M. (1970), "Choosing for Using: Users' Influence on Computer System Design, in Gruenberger F. (ed.), *Fourth Generation Computers*, Prentice-Hall Inc., Englewood Cliffs, New Jersey, pp. 103-116

Licklider J.C.R. (1969), "Underestimates and Overexpectations", *Computers and Automation*, August, pp. 48-52

Lonergan W. and King P. (1987), "Design of the B 5000 System", *Annals of the History of Computing*, Vol. 9, No. 1, pp. 16-22

Macro A. and Buxton J. (1987), *The Craft of Software Engineering*, Addison Wells, Wokingham

McCracken D.D., (1973) "Revolution in Programming: An Overview", *Datamation*, December, pp. 50-52

McGovern P.J. (1967) ,"The Computer Field and the IBM 360", *Computers and Automation*, January, pp. 16-20

McKinsey & Co. (1969), "Unlocking the Computer's Profit Potential", *Computers and Automation*, April, pp. 24-33

Mealy G.H. (1966), "The Functional Structure of OS/360", *IBM Systems Journal*, Vol. 5, No. 1, pp. 3-11

Miller E.F. and Lindamood G.E. (1973), "Structured Programming: Top-Down Approach", *Datamation*, December, pp.55-57

Moreau, R. (1984), *The Computer Comes of Age*, MIT Press, Cambridge Massachusetts

Morgan H.L. and Soden J.V. (1973), "Understanding MIS Failures", *Data Base*, Winter, pp. 157-171

Naur P. (1981), "The European Side of the Last Phase of the Development of ALGOL 60", in Wexelblat 1981, pp. 92-138

Ogdin J.L. (1972), "The Mongolian Hordes versus Superprogrammer", *Infosystems*, no. 12, pp. 20-23

O'Rourke T.J. (1967), "The Many New Uses of Time-Sharing", *Computers and Automation* October, pp. 48-50

Pantages A. (1969), "Industry Reacts with Approval and Dismay as IBM goes Separate Ways", *Datamation* August, pp 105-111.

Parnas D.L. (1979), "Research Problems in Programming Methodology", in Wegner 1979, pp. 352-358

Perlis A.J. (1981), "The American Side of the Development of ALGOL", in Wexelblat 1981, pp75-91

Postley J. A. (1968), "The Mark IV System", *Datamation*, January, pp. 28-30.

Pugh E.W. (1984), *Memories That Shaped An Industry*, MIT Press, Cambridge Massachusetts

Radin G. (1981), "The Early History and Characteristics of PL/1, in Metropolis et al (1981), PP. 551-599

Randell B. (1979), "Software Engineering in 1968", in *Proceedings of the 4th International Conference on Software Engineering*, pp 1-10

Reynolds C.H. (1967), "Software Development and its Costs", *Computers and Automation*, pp 18-21

Reynolds J.C. (1981), *The Craft of Programming*, Prentice-Hall, Englewood Cliffs, New Jersey

Richmond W.H. (1965), "Integrated Circuits for Commercial Computers", *Datamation* , November, pp. 29-32

Rice J.R. (1968), "The GOTO Statement Reconsidered", *Communications of the ACM*, Vol. 11, no. 8, p. 538

RCR, Rome Conference Report (1970), *Software Engineering Techniques*, Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969, edited by Buxton J. and Randell B., NATO, Brussels

Rosen S. (1967), *Programming Systems and Languages*, McGraw-Hill, New York

Rosen S. 1968, "Hardware Design Reflecting Software Requirements", *Proceedings of the AFIPS Fall Joint Computer Conference*, AFIPS Press, Montvale, New Jersey, pp. 1443-1449

Rosen S. (1969), "Electronic Computers: A Historical Survey", *Computing Surveys*, Vol.1, No.1, March, pp. 7-36

Rosen S. (1972), "Programming Systems and Languages 1965-1975", *Communications of the ACM*, pp. 591-600

Rosin R.F. (1987), "Prologue: the Burroughs B 5000", *Annals of the History of Computing*, Vol. 9, No. 1, pp. 6-7

Rutishauser H. (1967), *Description of ALGOL 60*, Handbook for Automatic Computation, Vol. 1, Part a, Springer Verlag, Berlin and New York

Sammet J.E. (1969), *Programing Languages: History and Fundamentals*, Prentice-Hall Inc., Englewood Cliffs, New Jersey

Siekman P. (1966), "In Electronics, the Big Stakes Ride on Tiny Chips", *Fortune*, June

Sobel R. (1984), *IBM: Colossus in Transition*, Sidgwick and Jackson, London

Soma J. 1975, *The Computer Industry*, Lexington Books, Lexington, Massachusetts

SPREAD Discussion (1983), "Discussion of the SPREAD Report", *Annals of the History of Computing*, Vol. 5, No. 1, pp. 27-44

SPREAD Report (1961), "Final Report of SPREAD Task Group, Dec. 28, 1961", reprinted in *Annals of the History of Computing*, Vol. 5, no. 1, 1983, pp 4-26

Steel T.B. (1964), "Operating Systems: Boon or Boondoggle?", *Datamation*, May, pp. 26-28

Steel T.B. (1968), "Multiprogramming - Promise, Performance and Prospect", *Proceedings of the Fall Joint Computer Conference*, pp. 99-103

Stevens D. (1970) "The User/Manufacturer Interface", *Computers and Automation*, September, pp. 25-27

Stoneman P. (1976), *Technological Diffusion and the Computer Revolution*, Cambridge University Press, Cambridge

Strachey C. (1966), "Systems Analysis and Programming", in *Information*, a *Scientific American* book, W.H. Freeman and Co., San Francisco and London, pp. 56-75

Tomeski E.A. and Lazarus H. (1975), *People-Oriented Computer Systems*, Van Nostrand Reinhold, New York

Trocchi R.F. (1969) "Third Generation Hardware - First Generation Applications", *Computers and Automation*, September, pp. 28-29

Wegner P. (ed.) (1979), *Research Directions in Software Methodology*, Cambridge Massachusetts

Wexelblat R.L. (ed.) (1981), *History of Programming Languages*, Academic Press, New York

Wirth N. (1974), "On the Composition of Well-Structured Programs", *Computing Surveys*, vol. 6, No. 4, December, pp 247-259

Wise T.A. (1966a), "IBM's $5,000,000,000 Gamble", *Fortune*, September

Wise T.A (1966b), "The Rocky Road to the Marketplace" *Fortune*, October

Wolverton R.W. (1974) "The Cost of Developing Large-Scale Software", IEEE Transactions on Computers, June pp. 615-636

Wulf W.A. (1972), "A Case Against the GOTO", *Proceedings of the ACM Annual Conference*, Boston, Massachusetts

Zuse K (1970), *Der Computer, mein Lebenswerk*, Velag Moderne Industrie, Munich

Zuse K. (1972), Kommentar zum Plankalkül, *Ber. Ges. Math. Datenverarbeitung*, 63, Part 2; English translation in *Ber. Ges. Math. Datenverarbeitung*, 106 (1976), pp. 21-41

Zuse K. (1980), "Some Remarks on the History of Computing in Germany", in Metropolis N., Howlett J., Rota G.-C., *A History of Computing in the Twentieth Century*, Academic Press, New York, pp. 611-628

List of Interviews Cited

John Buxton	Professor of Information Technology, King's College London, co-editor of the Rome Conference Report
Edsger Dijkstra	Professor of Computer Sciences, University of Texas, formerly Professor of Mathematics at the Technical University of Eindhoven and Burroughs Fellow
Franklin Fisher	Professor of Economics, Massachusetts Institute of Technology; lead expert economist for the defence in the case of US vs. IBM
John Guttag	Professor of Computer Science, Massachusetts Institute of Technology
Brian Randell	Professor of Computing Science, Newcastle-upon-Tyne, co-editor of the Garmisch and Rome Conference Reports, one of the founder members of Working Group 2.3
Norman Rasmussen	Teleprocessing Inc., formerly head of IBM Cambridge Scientific Center, Mass.
Douglas Ross	SofTech Inc., formerly of the Servo Mechanisms Laboratory, Massachusetts Institute of Technology

List of Abbreviations

ABM	Anti-Ballistic Missile
ACM	Association for Computing Machinery
AEC	Atomic Energy Commission
ALGOL	ALGOrithmic Language
APT	Automatic Programming Tools
BNF	Backus Normal Form or Backus-Naur Form
BOAC	British Overseas Air Corporation
BOS	Basic Operating System
BPS	Basic Programming Support
CDC	Control Data Corporation
COBOL	COmmon Business-Oriented Language
CPM	Critical Path Method
CPU	Central Processing Unit
DoD	Department of Defense
DOS	Disk Operating System
EDP	Electronic Data Processing

ENIAC	Electronic Numerical Integrator And Computer
FORTRAN	FORmula TRANslator
GAMM	Gesellschaft für Angewandte Mathematik und Mechanik (Society for Applied Mathematics and Mechanics)
GCR	Garmisch Conference Report
GE	General Electric
IAL	International Algebraic Language
IFIP	International Federation for Information Processing
IBM	International Business Machines
IMIS	Integrated Management Information System
JCL	Job Control Language
JOSS	Johnniac Open Shop System
JOVIAL	Jules's Own International Algebraic Language
LISP	LISt Processor
MAC	Double acronym for Multiple-Access Computing and Machine Aided Cognition
MICR	Magnetic Ink Character Recognition
MIS	Management Information System
MIT	Massachusetts Institute of Technology

NASA	National Aeronautic and Space Administration
NATO	North Atlantic Treaty Organisation
NPL	New Product Line
OS	Operating System
PERT	Production Evaluation and Review Technique
PL	Programming Language
RCR	Rome Conference Report
SAGE	Semi-Automatic Ground Environment
SDS	Scientific Data Systems
SEAC	Standards Eastern Automatic Computer
SLT	Solid Logic Technology
SMS	Standard Modular System
SOS	SHARE Operating System
SPREAD	Systems Programming, Research, Engineering And Development
THE	Technische Hoogeschool Eindhoven
TSS	Time Sharing System
UNIVAC	UNIVersal Automatic Computer

Print bound close to the spine

Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of go to statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (if *B* then *A*), alternative clauses (if *B* then *A*1 else *A*2), choice clauses as introduced by C. A. R. Hoare (case[i] of (*A*1, *A*2, ··· , *An*)), or conditional expressions as introduced by J. McCarthy (*B*1 → *E*1, *B*2 → *E*2, ··· , *Bn* → *En*), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, while *B* repeat *A* or repeat *A* until *B*). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, *n* say, of people in an initially empty room, we can achieve this by increasing *n* by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n*, its value equals the number of people in the room minus one!

The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, *n* equals the number of persons in the room minus one!

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to

judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the go to statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the go to statement is far from new. I remember having read the explicit recommendation to restrict the use of the go to statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than go to statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the go to statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

REFERENCES:

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. BÖHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

EDSGER W. DIJKSTRA
Technological University
Eindhoven, The Netherlands

A Note on Other Sources

The bibliography does not represent all the sources drawn upon in the course of this research, but only those cited in the main text of the thesis.

The work was from the beginning one of exploration, an attempt to map out paths in an unknown territory. My supervisor Donald Mackenzie played a key role in showing me the possible ways forward.

One problem from the beginning was that all the literature seemed to be either extremely schematic in its treatment of the issues or else extremely technical and therefore very difficult to relate to a broader framework. Much of the work in the early days of the research involved a search for sources at the right level of comprehensibility. Having reached the end of the thesis, it is pleasing to look back and to see that much that at first seemed totally incomprehensible now seems a bit more straightforward. A number of guides and aids helped me to find my way into the jungle and, since these are not adequately reflected in the list of references, it is perhaps worth mentioning them here, as an indication to future explorers.

One aid was the discovery of a limited, but extremely useful literature that is both well informed and relatively easy to understand. Among books, the work of Fishman (1982) is perhaps the outstanding example; at a more specialised level, COSERS (1980) was a constant source of reference. Among journals, those addressed to practitioners rather than academics often provided a very useful introduction to difficult topics. *Datamation*, *Computers and Automation* and

Fortune were particularly helpful in this respect, at least for developments in the 1960s: the mixture of gossip, newsbriefs, advertisements and articles gave a good sense of what was happening at the time. Once some familiarity had been established with the topics, the more specialised journals such as *Annals of the History of Computing*, *Communications of the ACM* and the series of *Infotech* reports were constant sources of information and ideas.

I was also extrememly fortunate in having the benefit of numerous informal discussions with tolerant practitioners of computer science, to whom I owe many thanks. In this context, special mention must be made of my computer science supervisor, Stuart Anderson, who showed unlimited patience in explaining the significance of recursion, reentrancy, stack architecture, interrupts and semaphores, etc, etc.

Another important guide was provided by the more formal interviews which I conducted. The interviews were like markers in the jungle, giving me information, giving me ideas and giving me confidence that I was pushing forward in the right direction.

Given the nature of the topic, the interviews were conducted only after quite a lot of the research had already been done to identify key events, processes and controversies. Two considerations affected the choice of the interviewees. The first was of course their relation to the object of the research, the centrality of their involvement in these events, processes and controversies. The other was less positive, but is certainly a feature of all postgraduate research: the limited availability of funds for travelling around the world and interviewing all the most significant actors. Given unlimited funds, there are certainly other people I would

very much like to have interviewed, but since the main role of the interviews was to support the investigation and to bring more life into the style of the thesis, I do not regard this as a major drawback.

The interviews were 'semi-structered'. I prepared for each one by reading the published work of the person in question and by finding out as much as possible from published sources about their role. From this I produced a list of key questions where I felt the published record most needed supplement and where I had ideas that I whished confirmed or disconfirmed. But I did not stick rigorously to this list, allowing the conversation between us to develop, lead into new areas, and suggest new ideas.

The interviews were a rewarding experience. The people interviewed (Bruce Bertram, John Buxton, Edsger Dijkstra, Franklin Fisher, John Guttag, Philip Kraft, Brian Randell, Norman Rasmussen and Douglas Ross) were not only informative, but also very good-humoured. All the interviews were recorded and transcribed, and the recordings are full not only of information but of slurps of coffee, puffs of tobacco and bursts of laughter.

One of the most positive aspects of being led into the jungle was to discover that the trees were full of laughter: much of what at first seemed incomprehensible later turned out to be full of wit and humour. For this reason, if not for any other, the paths opened in this thesis are worth exploring.