The Gentle Art of Computer Programming

An Inaugural Lecture
delivered before the University of Newcastle upon Tyne
on Monday 2nd Narch 1970

by B. Randell
BSc ARCS FBCS

Professor of Computing Science

UNIVERSITY OF NEWCASTLE UPON TYNE 1970

Printed for the University of Newcastle upon Tyne
by Titus Wilson and Son, Ltd., Kendal

The Gentle Art of Computer Programming

Introduction

IN JANUARY 1966, Professor Page gave an Inaugural Lecture to mark his appointment to the newly-created Professorship of Computing and Data Processing. I take it as a mark of recognition by the University of the high rate of development of the subject that now, just four years later, a second chair has been created in the Computing Laboratory, and of course I consider it a great honour to have the task of inaugurating this chair.

In his Inaugural Lecture Professor Page gave an admirable survey of the history of the development of computers, and of the wide spectrum of problems to which they had been applied. As indicated by the title I have chosen for this lecture, I intend to concentrate on surveying just one aspect of computing, that of computer programming.

Ada Augusta, Countess of Lovelace

Computer programming, like computers themselves, can fairly be stated to have originated with the invention by Charles Babbage, well over a hundred years ago, of the 'Analytical Engine'. The design for this machine incorporated the basic principles on which all modern computers are based, but the machine, which was entirely mechanical, was far ahead of the capabilities of the technology of its day and was never completed.

The main sources of information about this early computer, and in particular about how it was programmed, are the writings of Ada Augusta, Countess of Lovelace, who was the daughter of Lord Byron. She was an extremely gifted student of mathematics, and took a great interest in Babbage's work. For example, a contemporary wrote of a visit to Babbage's workshop: "While the rest of the party gazed at this beautiful instrument with the same sort of expression and feeling that some savages are said to have shown on first seeing a looking glass or hearing a gun, Miss Byron, young as she was, understood its working and saw the great beauty of the invention".

In fact I regard her as having been the world's first computer programmer. Certainly, the lengthy notes that she appended to her translation of Menabrea's description of the Analytical Engine virtually amount to a programming manual, and explain the operation of the analytical engine far more clearly than Babbage himself ever did.

The Analytical Engine used the angular position of each of a column of discs to represent the sequence of decimal digits making up a number. Its memory, or 'storehouse', was to have had a capacity of a thousand numbers, each of fifty digits. Its arithmetic unit, or 'mill', was capable of performing the four arithmetic operations of addition, subtraction, multiplication and division on such numbers. The machine could receive its data, and produce its output, on punched cards, of the type that were then in use for controlling Jacquard looms, used for the weaving of damasks and other elaborately patterned fabrics. It could also produce a copper plate impression of its output for printing.

Lady Lovelace's notes explain how the machine was controlled using sequences of punched cards. These cards indicated which variables, i.e. columns of discs, were to provide operands for the mill, which arithmetic operation was to be performed, and which column or columns was to receive the result. Repeated sequences of operations could be carried out using the same sequence of cards, and the future course of a computation could be made to depend on the outcome of a given operation. Several examples of what we would now call programs were given by Lady Lovelace including the complete text of a program to calculate the Bernouilli Numbers.

Even this short piece of program text should enable anybody who has not previously been exposed to the delights of programming a computer to get a first impression of the nature of the programmer's task. It shows that even a very simple calculation has to be broken down into a series of minute steps, and that a detailed set of conventions regarding the assignment of variables to storage locations has to be drawn up. Clearly, the amount of work involved in programming a complex calculation could be very considerable indeed.

However, Lady Lovelace's notes amount to much more than just a programming manual. In one section she comments on the fact that some accounts of the Analytical Engine concentrated on its mechanical construction, others on topics concerned with its programming:

> It is obvious that in the invention of a calculating engine, these two branches of the subject are equally essential fields of investigation and that on their mutual adjustment, one to the other, must depend all success. They must be made to meet each other, so that the weak points in the powers of either department may be

compensated by the strong points in those of the other. They are indissolubly connected, though so different in their intrinsic nature, that perhaps the same mind might not be likely to prove equally profound or successful in both.

Since her time we have invented the jargon ‘hardware’ and ‘software’ for these two aspects of computing, and a wide gulf has opened up between the hardware designers and the programmers, or ‘software designers’. In my view many modern computers would have been much better designed if the “mutual adjustment, one to the other” of hardware and software considerations had been more effectively carried out.

Even more perspicacious were Lady Lovelace’s comments on the limitations of the Analytical Engine:

> It is desirable to guard against the possibility of exaggerated ideas that might arise as to the powers of the Analytical Engine. In considering any new subject, there is frequently a tendency, first, to overrate what we find to be already interesting or remarkable; and, secondly, by a sort of natural reaction to undervalue the true state of the case when we do discover that our notions have surpassed those that were really tenable. The Analytical Engine has no pretensions whatever to originate anything. It can do whatever we know how to order it to perform. It can follow analysis; but it has no power of anticipating any analytical relations or truths.

These remarks are as true and as apt now, with respect to present-day large-scale electronic computers, as they were of the Analytical Engine when they were written in 1843–I shall return to this point later.

Grace M. Hopper

In view of the title of this lecture, it will perhaps not surprise you that the second programmer whose work I want to discuss was also a member of the fair sex. After Babbage’s work, the next big step towards the development of the modern computer was made by Professor Howard Aiken, of Harvard University Computation Laboratory, who during the years 1937 to 1944, in collaboration with the IBM Corporation, designed and built the Automatic Sequence-Controlled Calculator. It was on this machine that Dr. Grace Murray Hopper, Lady Lovelace’s major rival for the title of the world’s first computer programmer, started her extremely successful, and I am pleased to say, still continuing career.

The Automatic Sequence-Controlled Calculator, or Harvard Mark 1 as it is usually known, was a monstrous machine, over fifty feet long, built mainly out of electro-magnetic relays and other punched-card equipment components. It was very similar in concept to the Analytical Engine, with the sequencing of arithmetic operations being controlled by a punched paper tape. The techniques of ‘preparing and planning sequence control tapes’, i.e. for programming the Harvard Mark I, were described in 1946 in a paper that Grace Hopper co-authored with Professor Aiken. Each row on the control tape contained places for up to twenty-four punched holes. Each row represented a single machine instruction, in general of the form ‘Take a number from register A, put it into register B, and perform operation C’. The machine contained 72 variable registers and 60 constant registers, each of 23 decimal digits-it was thus considerably smaller than Babbage’s grand plans for his Analytical Engine. The major example of programming given by Hopper and Aiken was a program to tabulate the polynomial:

$$F(x) = x^4 + 3x^3 - 3x^2/4 - 22x + 3$$

Programs such as this, written in ‘machine-language’, i.e. directly in terms of the basic operations of a computing machine, are extremely detailed and intricate. Such examples of early machine language programs, I feel, go far towards explaining why, in 1967, at a special conference to mark the twentieth anniversary of the Association for Computing Machinery, Grace Hopper stated: “My primary purpose in life has not been in making programs, not in designing languages, not in running computers, but rather in making it easier to use computers”.

In fact, her work on this topic is greatly renowned. In the early 1950’s she was responsible for the development of some of the first compilers, i.e. special programs that would translate a description of a computation from some convenient ‘high-level’ notation, to machine language. She concentrated on the problems of commercial data processing, as opposed to scientific calculations, and was one of the originators of COBOL (Common Business Orientated Language), which is now one of the most widely used programming languages. This, incidentally, is a fact that tends to get overlooked in many University computing laboratories, where people are content to limit their debates to either ALGOL versus FORTRAN, or in the case of the American (or Americanised) computing laboratories, to FORTRAN versus PL/I.

An Early High-Level Language

Let me at this point digress slightly, and discuss a widely-used high-level language which antedated all of the languages that I have mentioned so far, and yet includes many of their features. In particular it includes facilities for iterating, i.e. specifying that a group of instructions are to be executed repeatedly, and for introducing and using procedures, or subroutines, i.e. shorthand notations for frequently used sequences of instructions. I am referring to the language in which knitting patterns are expressed. Any knitting pattern can be conveniently used to illustrate a little more clearly what I mean by the phrase 'a high-level language'. It will be seen that the pattern is couched in terms which relate to the object being knitted, and uses convenient mnemonics for the various different types of stitch. It is not written in the excruciating detail which would be needed to specify every movement of the knitting needles and of each of the knitter's fingers.

By now you have probably received the impression that I class the gentle art of programming with the more conventional maidenly pursuits such as flower-arranging and embroidery, and perhaps more appropriately, the solving of jig-saw puzzles. In fact the way people are sometimes taught to program, and the form that some advertisements for programmers take, would do little to dispel this image. Perhaps my views on the nature of computer programming will become clearer as I continue this brief account of its development.

The Fortran Compiler Project

No survey, however brief, of the development of programming could omit mention of the original FORTRAN compiler project, by a group led by John Backus. The importance of this work, carried out during 1954 to 1957, was that it demonstrated for the first time that it was possible to design a compiler which could generate a machine language program that was virtually as efficient as that which a good programmer could produce. This fact, plus the commercial success of the computer for which the compiler was programmed, did much to establish FORTRAN as the major language for scientific applications–a position which it still holds, despite the subsequent development of languages which have considerable technical advantages over FORTRAN. There was another 'pioneering' aspect of the first FORTRAN compiler–it took about twenty man-years to produce. At the time this was viewed with either awe or scorn, the latter mainly by people who did not understand the difficulty of the task that John Backus and his group had undertaken. However, as we shall see, it was but a foretaste of what was to come.

The Changing Scale of Programming

The examples of programs (and knitting patterns) that I have shown will I hope have conveyed, to those of you who have not experienced it at first hand, at least a first impression of what a programmer's task involves. However, I assume that many of you are in fact familiar with the University Computing Laboratory, and have tasted the delights of writing your own programs. But do you realise how far removed this impression, or even this first-hand experience, is from the realities of programming many of the current advanced applications of computers? The task of programming an airline passenger reservations system, for example, bears about as much resemblance to that involved in writing the typical programs that are produced at the University Computing Laboratory as the Concorde supersonic airliner bears to a bicycle! The difference is basically one of scale, but as Professor Page said, of developments in computer speeds, in his Inaugural Lecture, "great changes in magnitude may be equivalent to a change in type".

Operating System/360

As a first illustration, let me take Operating System/360, one of the two large-scale operating systems that we use on the IBM System 360, Model 67, at the Computing Laboratory. (The main purpose of an operating system is to allocate and schedule the use of various computer resources such as storage and input/output equipment; the 'computing system' which the typical user has to concern himself with consists of the basic computer hardware, plus such an operating system.)

Now it is always easier to grasp the size and complexity of an object that one can see. One look at the inside of a fairly large-scale computer, such as our Model 67, is enough to convince anyone of its complexity. The task of the team of engineers who create such a computer is obviously quite formidable. One of the techniques that such engineers use is simulation. They represent their computer design, which details the basic electronic components and their interconnections, by a computer program. This program, when executed, simulates the behaviour of the computer specified by their design. This, of course, provides a powerful aid to checking the correctness of the design, and to investigating the relative merit of various design alternatives. The size of this simulator program is, I claim, a fairly clear indication of the complexity of their design, and hence of the magnitude of their task. Operating System/360 is a suite of programs, written in the same language as such simulators. We can, therefore, obtain at least an impression of the complexity of Operating System/360, relative to the computer hardware on which it runs, by comparing the number of instructions in Operating System/360, with the number in the simulator. I understand that the ratio is in fact of the order of one thousand to one!

I mentioned earlier that the original FORTRAN compiler had taken the then awesome total of twenty man-years of effort to produce. It was admitted some time ago that OS/360 had involved 5,000 man-years of effort. By now I would assume that the actual total had reached a considerably higher figure. Incidentally, OS/360 forms just part of the set of programs which constitute the basic software that is provided with the System/360 series of computers. In fact, the amount of software which computer manufacturers choose, or perhaps find necessary, to supply with a computer is increasing dramatically, with no signs of any levelling-off.

The man-millenia that have gone into the creation of OS/360 have come from many hundreds of programmers, working in programming development laboratories in many different parts of the world. In 1965 a photograph was published showing the computing centre at one of these laboratories, which illustrated the incredible scale of this project. The computing centre portrayed covered half-an-acre, and included twenty-two computers. The fruits of the labours of all these programmers, at all of these separate development laboratories, have to be integrated together to form OS/360. This system is now in daily use, on the vast majority of the medium and large scale computers that IBM has produced. Thus, even though the technical aspects of the system have been criticised, often strongly, by many people including myself, the organisation of the programming effort is clearly a triumph of technical management.

The style of programming effort that is used to produce OS/360 has been very aptly described as the 'human wave' or 'Chinese Army' approach. It is abundantly clear that the results of this approach are, technically speaking, far from satisfactory. The word 'system' should imply organisation and coherence, but here one has almost unbelievable size and complexity instead. The software projects which are generally agreed to have been of high technical quality, such as the T.H.E. multiprogramming system, and the JOSS, BASIC and APL time-sharing systems, have all been produced by quite small groups of people, and have been of comparatively limited scope. Unfortunately, no one has managed to demonstrate the viability of any alternative to the human wave approach, which involves overwhelming the programming task by the sheer force of numbers, for systems of the scope of OS/360.

A major source of the complexity of OS/360 is that it in fact is not just a single system. It is used on many different sizes and configurations of computers, in many different types of installations. Therefore it is really a huge complex of software components, which can be put together in an almost limitless number of different combinations to produce a system which attempts to meet the needs of a particular computer installation and its users. Furthermore, OS/360 does not attempt to provide one single fixed set of facilities. Rather, the system has been continually extended by the addition of new facilities ever since the first version, consisting of some 400,000 lines of program text, was released in 1966. Now, just four years later, it consists of approximately two million lines of program text.

The Nature of the Programmer's Task

Let us examine the effects that all of these factors have on the 'gentle art' of programming. Clearly each programmer will be working on only a minute fraction of the whole system. He will be in an organisation which is governed by plans and schedules and progress reports and deadlines. In theory each task he is given will have a detailed specification, against which he can assess the correctness of the program text that he creates. In practice, checking that his program meets its specification will be relatively easy. However, because the specification is almost certain to be inadequate if not inaccurate, there will be no guarantee that, even when his program meets its specification, the system will work correctly when his program has been incorporated in it. And remember, he is but one of the many programmers concurrently making amendments to the system!

The co-ordination of this vast effort is currently done by conventional managerial methods. However, both IBM and General Electric have disclosed plans for using computers to help supervisory personnel to monitor and co-ordinate the activities of large teams of programmers on a big software project. Each programmer will work at a terminal connected to a huge central computer. This computer will contain all the information pertaining to the project–all the program text, all of the specifications and documentation, test cases and their results, statistics regarding the progress being made by individual programmers towards meeting their schedules, etc. Well, at least there is a certain poetic justice in the fact that this Orwellian development is intended for use by, or rather on, computer programmers.

Large Software Systems

Just because I have dwelt at some length on one particular software project, and the man-millenia of programming effort that it has absorbed, you should not think that it is the only such project. Since the first really large scale system was produced in the mid-1950's, of the order of 100 further large scale systems have been developed. The first system, SAGE, was an aircraft surveillance system, incorporating a network of computers, displays, radar tracking stations, etc. The programming support system alone, i.e. the software that was produced in order to aid the development of the actual operational software, amounted to approximately one million instructions.

Like SAGE, most of the early systems were developed by one or other of the various military agencies of the United States Government. However, nowadays an increasing number of large scale systems are being produced by non-military organisations. Examples of such systems are the sophisticated general purpose operating systems, management information systems, and airline reservation systems. Furthermore, these systems are no longer a purely American phenomenon. In Britain for example, banks and airlines are developing very large software systems.

Airline reservation systems and on-line banking systems are examples of what are called 'real time systems'–that is, systems whose activity has to be closely co-ordinated with events occurring in the external world, perhaps even on a second by second basis. For example, the American Airlines SABRE system has to cope with several thousand messages a minute and yet respond virtually immediately to any message, such as a ticket agent's request for information about a particular flight. The emergence of these sorts of system marks the growing dependence of many organisations on their computers. No longer can they tolerate unexpected and lengthy periods of computer mal-functioning. Rather, the continued correct functioning of their computer hardware and software, perhaps on a minute by minute basis in the case of real time systems, has become of vital importance. (If you doubt this, then consider for a moment the possible consequences of undetected programming errors in the large software systems being developed in connection with automated air traffic control, and with the anti-ballistic missile system.)

The Problems of Large Software Projects

At this point I would like to return to a comment I made earlier, which perhaps surprised some of you. I remarked that I viewed the development of OS/360 as having been a triumph of the art of technical management. My opinion is based on my respect for the immensity of the task of co-ordinating and controlling the development of such a huge software system. Hence the fact that the system was eventually produced, and has since achieved widespread use, is as far as I am concerned, sufficient reason to regard the management of the project as having been successful. However, by any of the normal measures that management uses to assess a project, such as the extent to which cost, completion date, performance and reliability estimates were attained, OS/360 was far from successful. It was by no means unusual in this respect, since the sad fact is that the vast majority of large software projects, and particularly the large real time systems, have exceeded their budgets, missed their completion dates, and failed to meet their intended performance and reliability levels, often by spectacularly wide margins. For example, the one million instruction SAGE program support system was not completed until after all need for it had passed!

One of the major reasons for this unsatisfactory state of affairs is, in my opinion, directly concerned with the very nature of the programmer's task. It is, at least for problems with some degree of novelty, essentially creative in nature. It is, therefore, hardly surprising that estimates should have little correspondence to reality in most large software projects, which typically contain so much that is novel that they are essentially research projects, though this is not often admitted beforehand. As the problems involved in a particular type of programming project become more understood, the accuracy of estimating, and the speed with which the project can be completed, increase dramatically.

Program Validation

The other major cause of difficulties is the problem of validating a program. People attending their first programming course soon learn that only the most trivial programs that they write are likely to be free of errors, or 'bugs' as they are somewhat euphemistically called. They learn to 'debug' a program by trying it with sets of sample input. When the program does not exhibit the required behaviour the sometimes tedious, sometimes exhilarating, process of finding the bug and correcting the faulty instructions begins.

Clearly, the value of this procedure depends critically on the choice of sets of sample input. For any but the most exceedingly trivial of programs it will be completely unfeasible to try more than a minute fraction of the various possible sets of input data. Randomly chosen sets of input data may allow some bugs to be found, but will not inspire any confidence that no remaining bugs exist–testing shows the presence not the absence of bugs! The idea of 'worst-case' testing, which is used by engineers to test, for example, the strength of their constructions, depends on their ability to predict worst-cases, which in turn depends on their understanding of the physical properties of their materials and methods of construction. In programming, on the other hand, predictions as to 'worst cases', i.e. sets of data which are likely to demonstrate the existence of a programming error, must be based on one's understanding of the program that is being debugged rather than on any well accepted physical properties of materials. Obviously this understanding cannot be complete–if it were there would be no reason for there to be any bugs. In fact it would be more accurate to say that 'worst case' testing depends on the programmer's assumptions about the program he is testing; as his testing proceeds he will almost certainly find that these assumptions are themselves error-ridden.

Clearly this analogy has its limitations–a more appropriate analogy, perhaps, is to liken the programmer's task of validating a program to that of an engineering draughtsman in validating a set of blueprints. Apart from the fact that a typical large program is, I believe, far more complex than a typical set of blueprints, the major difference is in the different standards of accuracy required. Blueprints are used by human beings who will be capable of coping with minor misprints and ambiguities, and who will know when to refer back to the draughtsman for clarification; programs will be used to control the activities of a mindless computer, which can no more correct the most minor of misprints than it can correct a major conceptual blunder by the programmer.

The debugging of programs is indeed very much of a black art even though it has been facilitated by the developments of various debugging tools. Must this always be so? I fear that for the immediate future this is almost certain to remain the case. In the longer term there is perhaps the hope that current work on techniques for proving the correctness of programs will have an impact. Implicit in this work is a recognition of the fact that programs have much in common with mathematical theorems, and that what is needed are techniques for creating rigorous correctness proofs. A notable example of the use of such a technique is the work of Dijkstra and his colleagues on the T.H.E. multiprogramming system. This system was validated almost entirely by deductive reasoning, rather than conventional debugging.

The word ‘proof’ is perhaps a little misleading. Just as in mathematics the value of a proof depends on its clarity and simplicity. Few people would feel satisfied that a theorem was correct if the only proof consisted of 100 closely-typed pages of intricate reasoning. They would rightly suspect the possibility that the ‘proof’ itself was in error. Not surprisingly, therefore, the major results of Dijkstra’s work have been the insights gained into the problems of structuring a complex program so as to facilitate the construction of a trustworthy correctness ‘proof’. In fact this topic of the structuring of complex programs in general, not just for purposes of correctness proofs, is of great interest to me personally, and is I feel capable of much further development.

Other important work on the validation of computer programs has been done by Floyd and Lucas. Floyd’s approach has been based on the idea of a ‘verifying compiler’. A program which is to be compiled would be accompanied by detailed propositions, in a formal mathematical notation, about the relationships that the programmer believes will hold amongst the variables when the program is executed. The compiler will check the consistency of these propositions with each other and with the statements which make up the program proper. The work by Lucas and his colleagues stems from their very successful research on the formal definition of programming languages, in particular the language PL/ I. Like Dijkstra’s work on program correctness it has already been of some practical utility, as it has been used to help validate a PL/I compiler.

A student of programming, struggling to debug one of his programs, often consoles himself with the thought that a program once debugged will stay that way; that it is thus quite different from, say, an electronic device whose components will age and develop faults. This is of course true, but it is also misleading, at least in the context of large software systems. A program can only be debugged or validated with respect to some given specification of what its actions are to be. If the specification is incomplete or changes, then any validation or debugging can at best be partial. This is invariably the case with a large software system, and is frequently the case for most of its component pieces. A large system will undergo frequent and extensive modification (whether for purposes of correcting errors, or adding new facilities) so that pieces of program that were previously considered to have been debugged will, in effect, develop faults. This fact, plus the sheer magnitude of the complexity of large systems, explains why there is general agreement with the statement that no large system has ever been completely debugged. By way of illustration, let me use once again that fertile source of statistics, OS/360. Twice a year a new version, containing corrections and extensions to the previous version of the system, is released. It has been reported that on average, by the time a version has been superseded, one thousand distinct errors have been found in it.

Project Apollo Ground Support System

Now, by way of contrast, I would like to discuss briefly one very large and complex software project, which can fairly be said to have been spectacularly successful. I am referring to a real time system, known as the Project Apollo Ground Support System. This system is one of several large software systems involved in Project Apollo. During space missions its functions are to receive telemetry data from the space vehicle and from tracking stations all around the world, to perform orbit and trajectory calculations, and to provide the data for the display screens at each of the consoles in the mission control room at which the NASA controllers work. In between actual missions it is used for extended simulations of space missions for training and testing purposes.

To give you some idea of the scale of this project I should explain that, largely due to the demands of testing and simulation, no less than six System/360 Model 75 computers are used, even though only one is needed for an actual mission, plus a second on standby. There are approximately three hundred programmers working on the development of this system, who have to reprogram a considerable fraction of the software afresh for each mission.

It is clear that the ground support system plays a critical part in the overall Apollo project, since it is a vital link between the astronauts and the mission controllers. Its correct functioning is as crucial to the safety of the astronauts as its readiness on schedule is to the timing of each mission. In fact both these aspects of this complex real time software project have been as successful as the spectacular success of the Apollo II mission might lead you to assume.

In retrospect it seems clear that the outstanding success of this project is due, not to some breakthrough in the art of programming, but to several somewhat mundane, but nevertheless important, factors concerned with the personnel on the project, its management, and the environment in which the project is conducted.

The ground support system, and the team which developed it, originated with the Vanguard project in 1959. This was followed by Project Mercury, which involved putting a single astronaut into orbit around the earth. There followed Project Gemini, and eventually Project Apollo. Each of these projects involved a sequence of missions, each one a little more complex than its predecessor. During this time the ground support system evolved, to keep pace with the increasing complexity of each mission. Thus the version of the ground support system that was used for Apollo II and the team of programmers which produced it were the product of a gradual evolution, spread over ten years. In contrast, the vast majority of large systems, such as those I discussed earlier, have been more the product of a revolution than of an evolution.

Other factors beside this evolutionary development have also played their part. The reliability of the ground support system was achieved by a careful mixture of programmed and manual techniques–there was no attempt to rely exclusively on sophisticated programming techniques. For example, programmers sit at the side of the NASA controllers at the monitor consoles during a mission. They can decide that something is not working correctly and that the system should be switched manually to the standby computer, or even that there should be a switch to an earlier version of the software system. The success of the project in meeting deadlines, is not solely due to the quality of the technical management, but is also due to the fact that the basic essential core of the system changes comparatively little from mission to mission. The major part of the reprogramming that is involved with each new mission concerns various telemetry experiments. As a last resort, if it is necessary to meet a deadline imposed by a scheduled launch-date, one or more of these experiments can be discarded.

Conclusion

With this brief analysis of one very successful large software system I must conclude. My aim in this lecture has been to try to convey to you how much programming has changed since the early days when it was perhaps indeed a 'gentle art'. Since then there have been many major developments in programming but our basic understanding of the nature of programming has progressed comparatively slowly. Thus it is in my view vital that we re-double our efforts to improve this understanding, and to produce better techniques for constructing and in particular for validating complex prgrams. Indeed it is considerations such as these which have prompted our plans for research, here at the Computing Laboratory of the University of Newcastle upon Tyne, on topics concerned with the problems of constructing highly reliable computing systems. However, in the present situation, we would do well to heed the warning given by Professor Gill at the NATO Conference on Software Engineering:

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

Finally, to close the circle, let me return to Lady Lovelace and some remarks of hers that I quoted earlier. Her purpose in making these remarks was to allay possible fears that the Analytical Engine might be capable of being developed to the point where it would become man's master rather than his slave. However, her remarks can equally well be directed at those who embark on over-ambitious software systems, in ignorance of our comparatively limited mastery of the not so gentle art of programming:

> It is desirable to guard against the possibility of exaggerated ideas that might arise as to the powers of the Analytical Engine. . . It can do whatever we *know how to order it* to perform.

The Lecture was illustrated throughout.

Bibliography

H. H. Aiken and G. M. Hopper.	The Automatic Sequence Controlled Calculator. *Electrical Engineering* (1946) pp. 384-391, 449-454, 522-528.
H. P. Babbage.	Babbage's Analytical Engine. *Monthly Notices of the Royal Astronomical Society* 70 (1910) pp. 517-526, 645.
H. P. Babbage (Ed.)	*Babbage's Calculating Engines:* E. and F. N. Spon, London (1889).
J. W. Backus, et al.	The FORTRAN Automatic Coding System. *Proc. 1957 Western Joint Computer Conference,* Los Angeles, California. Institute of Radio Engineers, New York (1957) pp. 188-198.
J. N. Buxton and B. Randell (Eds.)	*Software Engineering Techniques*. NATO Science Committee (in preparation).
E. W. Dijkstra.	The Structure of the T. H. E. Multiprogramming System. *Comm. ACM*. II, 5 (1968) pp. 341-346.
R. W. Floyd.	Assigning Meanings to Programs. *Proc. Symposium in Applied Mathematics*, Vol. 19. American Mathematical Society (1967) pp. 19-32.
G. M. Hopper.	Automatic Programming–Definitions. *Symposium on Automatic Programming for Digital Computers*. Office of Naval Research Washington (1954) pp. 1-5.
P. E. Ludgate.	Automatic Calculating Engines. In *Napier Tercentenary Celebration: Handbook of the Exhibition*. (E. M. Horsburgh, Ed.). Royal Society of Edinburgh (1914) pp. 124-127.
P. and E. Morrison (Eds.)	*Charles Babbage and his Calculating Engines*. Dover Publications, New York (1961).
P. Naur and B. Randell (Eds.)	*Software Engineering*. NATO Science Committee (1969).
E. S. Page.	*Computing: How, where and what*. University of Newcastle upon Tyne (1967).