[illegible]

Problems of Software Development

C.A.R. Hoare

Summary.

The main part of this paper is a reprint of the summary of a "teach-in" devoted to problems of software development. Of continuing interest is the diagnosis of the causes of failure of large software projects, and a summary of the seventeen criteria of the quality of software. The historical background is surveyed in an introduction.

1. Quality of Software
2. Failure

1962

↓

0 Introduction.

In the early nineteen-sixties a small but profitable Computer Manufacturer decided to embark on a comprehensive operating system project for a new machine, of the type now known as "third generation software". It included an automatic single-level store (SPAN), automatic buffering of peripheral transfers (TSS), a filing system, job control, editing, a new assembly language and new compilers for FORTRAN and ALGOL 60, more general than anything gone before. This was known as the Mark II Software Project. Documents describing "concepts and facilities" were sent to customers, and deliveries were due to start about 18 months after initiation. A team of about twenty 15? programmers started work. 1964.

Meanwhile the attention of the Company turned to a new machine, cheaper, smaller and more fashionable than the other. Software designs for the new machine were to be scaled-down versions of the software of the previous machine, and work started with a team of about five programmers. It was soon decided that sales of the old machine were falling off, and that software development for it would cease on completion of the existing software commitment, thereby releasing programmers for the new machine.

But somehow the completion of this software did not seem to get any closer. Promised delivery dates passed, and the promises were duly renewed. Increasing numbers of inexperienced programmers were recruited. Eventually Early in 1965. some experienced programmers (and even managers) were recalled to the project, and the specifications were drastically curtailed, and progress charts were made giving targets week by week for each member of every team. These showed that the delivery of the first item of Mark II software (the ALGOL 60 compiler) could take place in about four months. By dint of extremely arduous work on night shift this target was met - the first actual software delivery the Company made during a period of two years.

teach programmers to predict.

Unfortunately, it was undeliverable. Its speed of compilation was only two characters per second, which perhaps would be acceptable by present-day standards, but compared unfavourably with a speed of near a thousand characters per second of the previous ALGOL compiler for the same machine.

The main reason for this was insufficient planning in the use of the "single-level" store concept, which was crippling even though the backing store was what was subsequently known as "extended core", only fifteen times slower than the main store. Over the next few weeks, an intensive investigation of this trouble was conducted and resulted in a factor-of-four improvement in performance. Further improvements were possible but would give diminishing returns. There was a hardware limit on the size of the main store of the machine, which prevented the usual solution of free store handouts to all customers.

With some angulsh, the whole Mark II software project and thirty man-years of programming effort was abandoned. A meeting of customers was called to explain the situation. It was galling to find that many of them were not surprised.

despair
OS 360 } even worse
MULTICS }

It was at this point, on October 22, 1965, that a general "teach-in" was held of all senior programmers and section leaders to attempt to understand the causes of the disaster and agree on a remedy. The following sections are an unedited transcript of notes which I made of the discussion - only the names of the machines have been removed. A final section on the management structure of the software group has also been omitted.

what to do?
on offered machine
scrap it go back to original
ALGOL & extend it.
make small teams
with no interfaces.
put team leaders
in direct contact
with customers.
undertake
only ~~do~~ things
which I could
understand — both
to use and
to implement.

It is pleasant to report that the Company went on to design and produce (more or less on schedule) some quite presentable software, both for its old machine and for its new one, for example, the February 1966 date for delivery of ALGOL on the new machine was met. However great care was taken to avoid promising software with over-ambitious specifications or impossible delivery dates. Unfortunately its competitors were not so cautious (or perhaps scrupulous) in avoiding such promises, and many customers were lost.

Eventually the Company was taken over by a larger one, which was shortly taken over by an even larger one still. Its identity is thus so completely submerged that I cannot even give the usual acknowledgement. Nevertheless, my thanks go to all who worked for me at the time and afterwards, since they have taught me (the hard way) all I know about the design and management of large software projects.

Of course, since the events described above, many larger software projects have culminated in even more spectacular and better publicised failure. The reason for publishing the following notes is in the hope that it will help others to avoid similar failures in the future. It is not only by reports of success that the state of the art can be furthered.

The reasons why the report of an event which occurred some seven years ago has now been edited is partly for historical interest, but mainly because I think most of the problems it describes are still occurring today. The only comment which I would now add is that the seventeen objectives of software are far more irreconcilable than I imagined at the time, and that any software design can only be based on a clear appreciation of the relative weight to be given to each of them. It would be as well that these weights were laid down by Management in accordance with the commercial interests of the Company. I myself would prefer to give weights in accordance with the practical interests of the software user, which are often very different. I believe that the right order of priority for the user to be roughly:

1. Clear Definition of Purpose.
2. Simplicity of Use.
3. Operating Ease.
4. Clear, Accurate and Precise User Documents.
5. Ruggedness.
6. Reliability.
7. Early Availability.
8. Brevity.
9. Efficiency.
10. Suitability to Configuration.
11. Wide range of application.

But of course the other criteria must not be wholly ignored.

define a standard of acceptability on each criterion and ~~see~~ test whether they are all achievable befor before embarking on the software.

1. The Purpose of Software

To increase sales of the Company's computers by providing a suitable degree of software support; and to give maximum satisfaction to customers with resources available.

Software support comprises those programs, compilers, sub-routines, etc., which simplify and expedite the task of applying a computer in a relatively wide area of problem solving.

2. Recent Major Customer Grievances

In recent times the software situation at the Company has given much cause for dissatisfaction among customers. Their major legitimate grievances are:

2.1. Cancellation of Projects.

Several projects have been cancelled at a very late stage even after their descriptions have appeared in the Technical Manual. The specifications of other projects have been significantiy reduced.

On our new machine, TSS has been virtually shelved, and the specification of SPAN will probably be simplified before implementation on backing store.

2.2. Failure to meet Deadlines.

For our Mk.II software on our current machine, deadlines set in March and again in June 1965 were passed without notice; and on our new machine, the November deadline for ALGOL has had to be postponed to February 1966.

2.3. Excessive size of Software.

In the specification and planning of software insufficient thought was given to predicting the size of the subroutines and of the programs which used them; and in the event the size has proved not to be justified by the usefulness of the facilities provided.

On our new machine, early predictions of the size of the ALGOL compiler and systems programs were over-optimistic, but lack of effort has protected us from the worse excesses of the earlier machine.

2.4. Excessively slow programs.

In the design of software insufficient thought was given to predicting the speed of the programs; and in the event the ALGOL and probably also the FORTRAN compilers have been found to be quite unacceptably slow.

On our new machine, the absence of two-level storage has protected us from the worst errors of the previous machine.

2.5. Failure to take account of Customer Feedback.

Earlier attention paid to the quite minor requests of our customers might have paid as great dividends of goodwill as even the success of our most ambitious plans.

2.6. Poor information flow to Education Dept. Sales and Customers.

As a result of hasty improvisation undertaken at the last minute to get the programs delivered, the documentation of the programs actually delivered is sadly lacking; and the education department has had no chance to keep up to date and train customers programmers in the use of the programs actually available.

3. <u>Investigation of Causes</u>

The causes of the failure summarised in the previous section may be classified roughly as:

1. Failings within the software group due mainly to inexperience in planning and prediction.
2. Failings within the software group due mainly to organisational and communication problems.
3. Grievances, which relate to circumstances wholly or partly out of the control of the software group.

3.1. Failure in prediction.

The failure to predict successfully the consequences of the Company's plans and policies might be said to be the main cause of the trouble, since if we had foreseen the trouble, we would certainly never have embarked upon the plans.

3.1.1. Overambition.

The goals which we have attempted have obviously proved to be far beyond our grasp. Even if it were theoretically possible to implement so great a volume of interlinked programs, it was obviously not a practical possibility in the prevailing circumstances. On our new machine, the attempt to achieve compatibility with commercial software developed outside the group, has basically failed because it was more difficult to achieve than was expected.

3.1.2. Estimation and consideration of program size, and speed.

Early and correct estimates of the size and speed of compilers and other programs would have led to the use of more successful techniques of implementation - or abandonment of the project.

The fact that it was extremely difficult to make these estimates in advance might have been a warning not to adopt such complex techniques, or at least to keep a closer watch on these factors during the course of implementation.

3.1.3. Estimation of effort required.

The amount of effort required to get a program to a state of delivery has been consistently underestimated. For simple applications programs, a typical productivity figure for a programmer engaged on flowcharting coding, and testing is 100 single-address instructions a week. For complex interlinked programs, compilers, and for software in general, a figure of 40 instructions/wk is quoted and borne out by our experience. For systems programs floating point routines, input/output routines, and other programs which occupy core store space during the running of customers' programs, the utmost care in coding is required; and 20 instructions/week would be a good figure, if it can be afforded.

All these figures are based on:

1. feasible original specifications
2. minimal changes to specification
3. the best planning, coding, and testing techniques

There is no known way of estimating how long it will take to deliver a program which has been partially developed without satisfying these criteria.

3.1.4. Failure to Plan Coordination and Interaction of Programs.

Where the coordination and dependency of programs has been accepted as desirable or necessary, the greatest possible care must be exercised in planning and clearly defining the simplest possible interface between the programs. On both sides of the interface the programmer should try to understand the major requirements of the other side, and a continuous watch should be kept on the situation by both sides. Even when the greatest care has been exercised, some trouble is to be expected and allowed for in the final linkup.

3.1.5. No Early Warning of Failure.

Even the best based predictions will sometimes prove incorrect, so that it is wise to draw up a list of the assumptions on which each prediction has been based, and to check regularly whether the assumptions are still justified or not. Since many predictions are based on a breakdown of a task into its component parts, a continuous check of actual against estimated achievement on each of the parts is the best indication of likely success or failure.

omit.

3.2. Communication and Organisational Problems.

The problems of organisation and communication in software design and implementation are extremely severe, owing to the great mass of specialised and interacting details which must be controlled and coordinated within some clear general framework.

The severity of the problem means that especial care must be taken by all members of the department that information is distributed and gathered in the right places at the right time.

3.2.1. Proper Procedures for Program Changes.

It is in the matter of changes to program specification that failure to consult and communicate result in the greatest difficulties. Each programmer must maintain a continuous awareness of the activities and needs of other programmers who are relying on him, as well as the progress of those on whom he is relying. Draft changes to specification must be circulated to interested parties, and authorised by the group leader before they are implemented.

3.2.2. Coherency of Series of Documents.

When a series of documents is issued relating to the same project or subject, the documents should be numbered, and each successive document should indicate in what way it is related to its predecessors; for example, whether it supplements it, adapts it, or supercedes it.

Failure to do this can lead to grave confusion.

3.2.3. Liaison with Education and Sales.

In addition to the problems of communication within the programming group, there are equally severe problems of passing up-to-date information to the Education Dept. for teaching to our customers, and to Sales Department so that Sales Policy may be based on a realistic appreciation of the situation.

3.2.4. Management Liaison.

A clear picture of all the aims, activities, and difficulties of the department must be presented to management, so that longer term and even shorter term decisions may be based on a correct appreciation of the situation rather than upon guesswork.

3.2.5. Customer Liaison.

Ultimately, the most important form of liaison is that with the customer, who must be given a fair and accurate indication of the nature and purpose of the programs which he is to expect, and when he is to expect them. Comments and feedback from customers must be given the greatest weight, but must not lead to any rash promises.

3.2.6. Clear and Stable Definition of Responsibilities.

In the recent past, the clear division of responsibilities among the members of a programming team has often been lacking; and programmers have been uncertain about the extent of their responsibilities for the programs they are developing. Furthermore, programmers have been moved from one project to another rather more freely than is consistent with the establishment of proper long-term responsibilities.

Fundamentally, the task of each software programmer is to deliver programs which other programmers (internal or external) are willing and glad to use.

3.3. <u>GRIEVANCES</u>.

3.3.1. Lack of Machine Time.

The lack of suitable machine time during software development is likely to be a permanent feature, since software usually has to be developed simultaneously with the hardware which it uses. The original design and planning of the software should take this fact into account.

3.3.2. Struggle for Machine Time.

Small amounts of machine time would be more acceptable if its availability is considered to be beyond question. In fact, it has been a constant struggle to maintain any allocation inviolate from encroachment; it is this struggle which militates against efficient planning of the use of the time available.

3.3.3. Unpredictability of Machine Time.

When machine time is made available (especially from the production floor) this is usually only at short notice. This makes planning impossible, and encourages precipitate and unprofitable use of even that time which becomes available. This is particularly true when a programmer attempts to make up for a long period without machine time by a long session at the console.

3.3.4. Lack of Suitable Peripherals.

The non-availability of suitable peripheral equipment is often even more difficult to overcome than that of central processor time, which is sometimes offered in exchange. Again, this fact should be taken into account in the planning of software for peripherals.

3.3.5. Unreliability of hardware.

Even when machine time is available, the unreliability of processor and peripheral equipment often causes more frustration and waste of time than its complete absence. This is also due to the need to test programs on early prototype or production models, and the unavoidable inexperience of commissioning and maintenance personnel.

In general, a regular guaranteed allocation of reliable machine time, even quite small, is far better than the constant struggle to make up for loss of machine time by other means.

3.3.6. Dispersion of Staff.

Problems of communication, which are already sufficiently troublesome in software development, are further aggravated by the dispersion of staff throughout the building.

3.3.7. Lack or Absence of Tape Preparation Equipment.

On our new machine the absence of proper paper tape preparation equipment has and will continue to prevent interchange of programs and data between ourselves and our customers. It necessitates maintenance of two versions of every program.

3.3.8. Lack of Firm Hardware Delivery Dates.

Delivery dates for hardware on which software development depends are extremely difficult to obtain, and when obtained are often worthless.

3.3.9. Lack of Technical Writing Effort.

For some time it was hoped that deficiencies in original program specifications could be made up by the skill of a technical writing department. When this hope proved to be unjustified, there was a natural ground for complaint.

In fact the original hope was misguided; the design of a program and the design of its specification must be undertaken in parallel by the same person, and they must interact on each other. A lack of clarity in the specification is one of the surest signs of a deficiency in the program it describes, and the two faults must be removed simultaneously before the project is embarked upon. This means that the program designer, usually the section leader, must be responsible for the description as well.

3.3.10. Lack of Software Knowledge Outside Programming Group.

The lack of knowledge and interest in software which is prevalent outside the programming group is not entirely a grievance, since it can be remedied by more care taken by the programming group in presenting the necessary information in a simple and palatable form.

3.3.11. Interference from Above.

On our new machine/certain decisions impinging on software design (e.g. character codes, paper tape preparation equipment, etc.) were taken by management possibly without a full realisation of the more intricate implications of the matter. Again this circumstance can be partially remedied by better information flow from the programmers.

3.3.12. Over-optimism to Customers.

There has been some lack of caution on the part of salesmen and programmers in making promises or near promises to customers about the merits and availability of software. In fact, the programmer should always promise just a little less than he knows to be achievable, and quite a bit less than he hopes will ultimately be attained. The salesman should assist the programmer in avoiding excessive commitments, rather than encourage more ambitious promises.

4. The Criteria for Quality of Software.

In the recent struggle to deliver any software at all, the first casualty has been consideration of the quality of the software delivered. The quality of software is measured by a number of totally incompatible criteria, which must be carefully balanced in the design and implementation of every program, and in the incorporation of improvements in programs already delivered.

4.1. Clear Definition of Purpose.

The aim of every item of software must be most clearly defined; and recommendations on the circumstances of its successful use must be fully explained. Vagueness in this matter is intolerable; and of course, every feature of the program delivered must be oriented towards the declared purpose.

4.2. Simplicity of Use.

The software must be very simple to understand and use in the majority of cases, and the extra complexity of its use in less normal circumstances must be kept to a minimum.

4.3. Ruggedness.

As well as being very simple to use, a software program must be very difficult to misuse; it must be kind to programming errors, giving clear indication of their occurrence, and never becoming unpredictable in its effects.

4.4. Early Availability.

Software must be made available together with the first delivery of the hardware to which it is relevant. If software is not available at this time, the customer is forced to develop his own ad hoc techniques, and may never be weaned to use the more elegant methods of software which has arrived too late.

4.5. Reliability.

The original program delivered must be as free from errors as possible, and if errors are discovered they must be capable of correction with the greatest rapidity. This is dependent on simplicity of programming techniques and first-class program documentation.

4.6. Extensibility and Improvability in Light of experience.

Since lack of hardware and lack of experience makes it impossible to deliver perfect software in a Mark I version, the software programs should be capable of further development and improvement. This again demands simplicity of approach and good documentation.

4.7. Adaptability and Easy Extension to Different Configurations.

The purpose of software is to satisfy as many customers as possible, including those with a wide range of configurations. Furthermore, a customer who expands his hardware after purchase will wish to avoid changing his programs or his programming techniques and concepts.

4.8. Suitability to Each Individual Configuration of the Range.

In addition to adaptability over a range of configurations, the software actually available for each member of the range should be well suited to the capabilities and needs of that particular configuration.

4.9. Brevity.

Software programs should be as short as possible, particularly those which have to co-exist in the store with the programs which use them. Furthermore, the use of the software should not involve extra length in the program which makes use of it.

4.10. Efficiency (Speed).

The speed of software programs should be sufficient cto justify their use in most circumstances.

4.11. Operating Ease.

The most critical factor in the efficiency of an installation is often the smoothness of the operating system. Software should be designed to make the job of the operator as simple as possible.

4.12. Adaptibility to Wide Range of Applications.

Since the purpose of software is to find wide applicability, it is necessary to consider the widest possible areas of application in its design and implementation.

4.13. Coherence and Consistency with other programs.

As far as possible, software programs should be compatible with each other, and capable of being used either separately or in conjunction with each other. Furthermore, any overlap between the programs available should only be accepted if the need justifies it.

4.14. Minimum Cost to Develop.

The cost of software development in manpower and machine time is a vital factor in the planning of a suite of software programs.

4.15. Conformity to National and International Standards.

When national or international standards for character codes, tape formats or languages have been set up, or seem likely to be set up, these should be observed to the maximum extent.

4.16. Early and Valid Sales Documentation.

Sales documentation describing the most important features of the software under development should be available early in the life of the project; and they should remain valid when the product is finally delivered.

4.17. Clear Accurate and Precise User's Documents.

In addition to sales documents, the user's documents should contain clear instructions on a program's proper method of use, and an accurate description of its properties.

The document should be available early in the life of the project, and should be kept scrupulously up to date.