SOFTWARE ENGINEERING MANAGEMENT

BY

K. GEARY, B.Sc.,M.B.C.S.,G.I.M.A.
(*Sea Systems Controllerate*)

This article is based on a paper presented by the author at the Eighth Ship Control Systems Symposium in the Hague in October 1987.

ABSTRACT

Software introduces its own style of management problems. If software development is not made visible and tangible, it becomes difficult to control. Poorly controlled software development can take up a major proportion of equipment development costs, cause delays to delivery, and degrade equipment performance. This paper describes how these problems can be avoided and discusses the management practices involved. The impact of software on equipment configuration, ease of operation and the reliability and safety of equipment is also considered.

Introduction

Software is the logic that is embedded within the hardware of equipment control mechanisms. Software therefore can have a significant effect on equipment performance in terms of availability, reliability and maintainability. Additionally, it can have an indirect but significant impact on equipment configuration management, equipment to equipment communications, ease of operation, and safety of personnel.

Ever-increasing requirements for improved facilities on marine equipment have promoted computer software as the prime means of implementing control logic. Software offers significant advantages over hard-wired logic by giving designers the ability to implement orders of magnitude greater complexity. Software enables the hardware control element of equipment to be constructed from general purpose digital components that are relatively inexpensive, very reliable, and compact. Logic design changes have little effect on the hardware production line as any changes can be implemented in software.

Software, therefore, brings major advantages in equipment control, but it also represents a major change in technology. The impact of software is far greater than just providing for enhanced functionality of equipment. It introduces its own problems of project management, equipment development costs, and quality assurance. Software development is purely a design exercise. In the equipment production phase, reproducing software becomes merely a matter of automated replication from a master copy.

Software Appreciation

There are now very many cheap computers available in consumer shops. The general public can go out and buy a home computer which can be used by all the family, enabling the creation of software to be accomplished by children and their parents alike. Digital technology is relatively new and some equipment project managers may have gained an appreciation of software through this route.

Amateur programming is useful in that it gives familiarity with computing and raises awareness of some of the problems that can occur. However, programming in an uncontrolled environment also allows poor practices to develop. Problems can arise when computer programming practices that are adequate for programming small quantities of recreational software in the home are transferred to the management of software development in the engineering environment. There are major differences between recreational software and marine equipment software (TABLE I) which will significantly affect requirements for management and quality control.

It is necessary, therefore, for project managers to possess an awareness of the principles of management of software development. Managers need to be aware of the stages in the software life cycle and the milestones that identify progress. They need to know what estimates and plans must be made in order to avoid future problems and they need to know what items must be delivered to complete each stage of work. Those managers that treat software development as a single self-contained entity are likely to find that the software will cause the project to run over time and over budget. Software, like other engineering disciplines, needs careful control. The controlled development of software through staged management practices, as shown in FIG. 1, is known as 'Software Engineering'.

TABLE I—*Comparison of recreational and operational software*

Recreational Software	*Operational Software*
Implemented on general purpose computer purchased to suit leisure	Implemented on special purpose or bespoke computer designed to suit the application
Recreational/interest/hobby	Functionally necessary
No software development costs (except leisure time)	Expensive software development costs
No software maintenance costs (except leisure time)	Expensive software maintenance costs
Maintained by the author	Life of software greater than the availability of the author
Limited lifespan (up to 2 or 3 years)	20 to 30 year lifespan
Specified, designed and used by author and immediate family	Specified, designed and used by different people
Up to a few hundred lines of code	Several thousands or millions of lines of code
Reliability unimportant	Reliability very important
Not time critical	Often time critical

Tangibility

When compared to hardware development, software development appears intangible. Traditional supervisory inspection of hardware can give an experienced project manager some measure of progress of work. With software, this is not so easy. Counting the number of lines of code written does not give a good measure of progress. Furthermore, if a manager were to use such a technique it would encourage the bad practice of writing code before the design has been well thought out. This would lead to many lengthy sessions being spent by the programmers at terminals, as they try to find and correct a plethora of errors, referred to as 'testing out the errors'. Such practice is indicative of poor software design, weak management and ineffective quality control practices. The result will almost certainly be soaring development costs, late delivery, and system reliability problems.

Software must be made in relation to its development cycle. The only way to achieve tangibility is through formal issue of documentation covering each stage of work. This practice also has the benefit of ensuring that the documentation reflects the design and is not a chore that must be completed after programmers have experienced the accomplishment of producing working code.

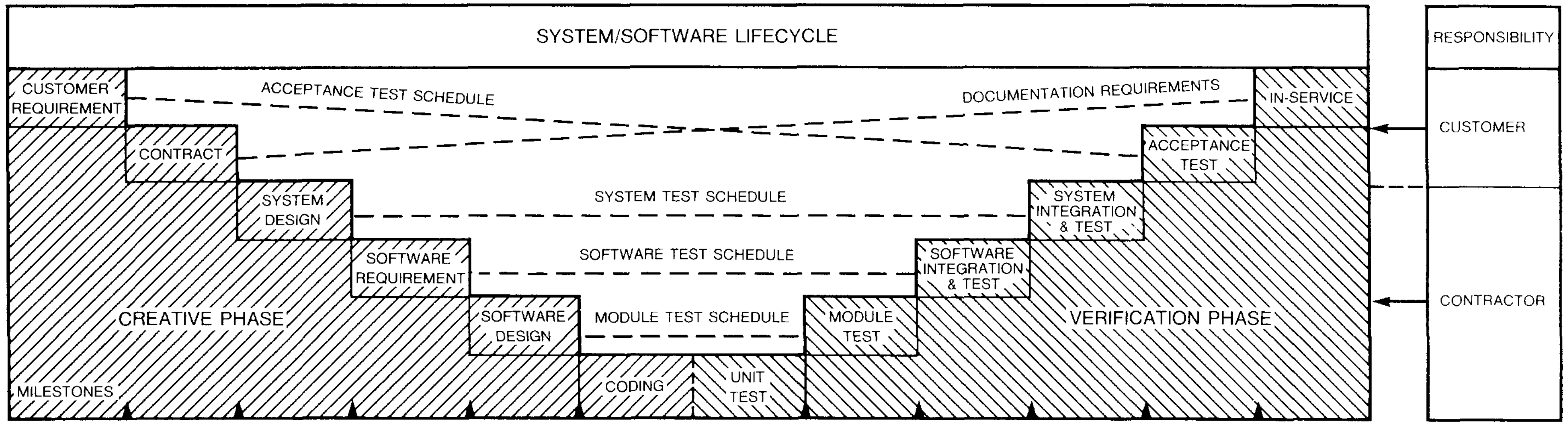

Fig. 1—The controlled development of software

Once software development has been made tangible, it can be controlled. Deliverable documentation must be placed under configuration management, with formal issue states. So often, software and its documentation is not brought under configuration management until after equipment delivery. Such practices avoid the process of formal change control, but they also avoid managerial control. The software then becomes a constantly changing entity; it is never complete and costs and timescale escalate. By placing completed documentation under configuration management, project managers can maintain visibility of design changes.

By using the life cycle stages, shown in FIG.1, the project manager can make software development visible. By demanding documentation to demonstrate the completion of a unit of work, software can be made tangible. By placing completed items of work under configuration control during development, software can be made controllable.

Estimation

In the planning phase, one of the most difficult problems to overcome is accurate estimation of the work involved to develop a new piece of software. Past experience of similar projects is a useful aid to this. There are various models that may be used to assist in the task, and most of them are now implemented on one of the new generation of personal business computers. None of the models is guaranteed to give absolute accuracy, but they are all more consistent and useful than intuitive methods.

Cost estimation models require input of estimated parameters, such as number of lines of code, or some other sizing parameter. Managers using these models must be aware that if the original sizing estimates are wrong then the costing and work schedule proposed by the model will also be wrong. It is therefore necessary for managers to ensure that the sizing estimates are realistic. One of the benefits of such models is that they do encourage realistic estimates because they require the planner to give more thought to sizing than a quick guess.

Work load estimation is also useful to the customer procurement officer during tender assessment. If, in the invitation to tender, the supplier is required to complete an appropriate questionnaire, such as Def Con 143, the procurement officer can process the information through one of the models to check whether estimates are realistic. With current trends towards fixed contracts, cost checking would appear irrelevant. However, the customer will also be interested in whether timescales are realistic. Contractual clauses may offer compensating protection, but they do little for timely delivery of equipment if a supplier is put out of business.

A useful spin-off from cost estimation is that all the sizing information will be available to estimate processor loading and memory requirements. Such data will enable designers to choose a processor with sufficient power to suit the application. This will help to avoid the bad practices and escalating costs associated with fitting the software into hardware that has insufficient power and capacity to suit the application.

Quality Assurance

Quality assurance is the monitoring of quality control practices. Both quality assurance and quality control are on-going activities throughout the project life cycle. Application of quality assurance and control to hardware engineering disciplines is generally well understood. However, available quality assurance requirements are often written in general engineering terminology and therefore require considerable interpretation to relate to software.

A problem occasionally encountered in the area of software quality control is a lack of awareness amongst supplier and customer project managers. Often, because programs are listed on paper, the only formal quality control applied to software can be configuration management of the paper code listings, which are considered to be equivalent to hardware design drawings. Quality control of the software design itself is sometimes left to informal and uncontrolled practices which may or may not be applied by the programmer.

Where there is insufficient quality assurance and control, design reviews become poorly conducted or may be omitted altogether. A common mistake in design reviews is to discuss the documentation format and conventions used to express the design, and omit to review the actual design. Formal design review procedures should be employed on the specification as well as the design, the specification being a greater source of errors than the whole of the remainder of the software development cycle. Furthermore, as all subsequent quality control practices are carried out to ensure compliance to the specification, errors in the specification will not be realized until after the equipment is accepted into service. This accounts for the very high support costs of some software-based equipment. Correction of errors is much more expensive if discovered at a late phase in the life cycle.

Planned and controlled progressive testing is part of quality control. Experience has shown that it is a mistake to place full reliance on equipment acceptance tests, allowing the software to receive no formal unit or module tests and little formal integration testing. Once software modules have been integrated, the number of possible control paths increases to a point where it is not feasible for equipment acceptance tests to exercise the code adequately. Programs must be tested and successful completion formally recorded at unit test, module test and integration stages. Records of progressive development testing should be required as part of the acceptance procedure. At the formal completion of each stage, the accepted unit of code, program module or system software must come under configuration management to ensure that any subsequent design changes are controlled and adequately tested.

Project managers should beware of using research or prototype software in production equipment. As with some prototype hardware, such software is produced quickly with little or no quality control, the objective being to produce a study report or feasibility demonstration. It is not possible to enhance software quality retrospectively and it is therefore necessary to discard poor quality software and rewrite new software with proper quality controls.

Integrity/Reliability

The issue of reliability should be considered from a 'systems' viewpoint. It is of secondary interest to the user whether it is a hardware or software problem that is resulting in the system failing. Software, like hardware, can contribute to the unreliability of a system and therefore must not be discounted, but the methods used in determining or obtaining high integrity software differ from those used for hardware. Hardware reliability considerations are dominated by determination of the 'wear-out' characteristics of components in the system, enabling designers to predict quantitatively the likely order of the resulting system's reliability. Other factors influencing the reliability of the hardware design are therefore largely discounted from the ensuing calculations.

Software on the other hand has no tangible existence other than through the medium of hardware. It follows therefore that it has no 'wear-out' characteristic. Any functional failure is due to inherent error in the design or implementation of the software. A principle element in most reliability calculations is 'time'. However when considering software the passage of 'time' may have little impact. Any fault in the software will have been there since its design and will only become apparent when that feature is called into use. So, one may have a situation where the system is exhibiting total reliability but the software component contains errors that have yet to manifest themselves.

The customer will state a reliability objective for a system. In apportioning the reliability objectives to the constituent portions of the proposed system the design must consider the software and in doing so it would be unacceptable to assume that the software never fails. The software element will make a finite contribution to the overall reliability of the system and therefore software integrity objectives should be stated in the supplier's equipment design outlining the software functions.

The greatest influence on the quality and reliability of software will be the management and quality control procedures and the skill of the people involved in the specification and design of that software and its test schedules. Confidence in the software should be based on the achievement of the software team members in designing similar systems under similar quality control procedures and the perceived reliability and performance of those systems.

Equipment Configuration

Equipment design changes may be carried out much more quickly and easily by changing software than by changing hardware. Hence there are often many more changes to the software than to the hardware. Traditional configuration management for equipment is adequate for infrequent and slowly changing hardware designs; however, software may be changed rapidly and frequently and installed in a hardware configuration item without any externally visible alterations.

It is necessary to review hardware-oriented configuration management procedures in the light of engineering advances into the use of software. The configuration management system must control the many software design changes that affect pre-programmed integrated circuits known as 'firmware'.

Information Transfer

Software-based equipment control has lead to a wide availability of control data. It is now possible to integrate different equipments on a platform into a co-ordinated management system. This raises many issues concerning communications, but the one relating directly to software is the integrity of information transferred.

Project managers need to be aware of the old software adage, 'garbage in, garbage out'. When equipments were isolated, any errors in the control logic would affect only that piece of equipment. Now that data can be readily transferred between different equipments, it is possible for a software malfunction to have an impact on other areas of the platform. Hence, systems which accept information from other sources should be designed to validate incoming data before making use of that information.

A further design consideration is the way in which data transferred from one equipment to another will be put to use. Misconceptions concerning the interface data specification could lead to data being used in a way that was unforeseen by the data source designer. It is therefore necessary to ensure that all interfaces are clearly specified and understood.

Man Machine Interface

The user's view of the equipment is through the man machine interface (MMI), or human computer interface (HCI) as it is sometimes known. Increasingly this interface is being designed and implemented using software. Equipment designers are becoming aware of the need for an easy-to-use MMI but often they impose their own ideas on the design.

The MMI is probably one of the most difficult areas for the customer to specify, but from the operator's point of view it is one of the most important. If an equipment is designed with a poor MMI, changes to make it easier to use may involve a major software rewrite with significant cost escalation and time delays. It is usually difficult to get the MMI design correct first time, so it would be beneficial to be able to create a prototype version that can be refined. There are now several software tools available that enable designers to construct a prototype of the MMI. Operators can then be asked to evaluate the design and advise on refinements before the operational MMI software is finally written.

Safety Critical Software

Software is now being used in applications which require very high integrity, such as control of equipment that can be a hazard to the safety of personnel. It is not possible to prove software 100% correct. Safety assessors are, therefore, becoming concerned over the risk to human life posed by possible software errors.

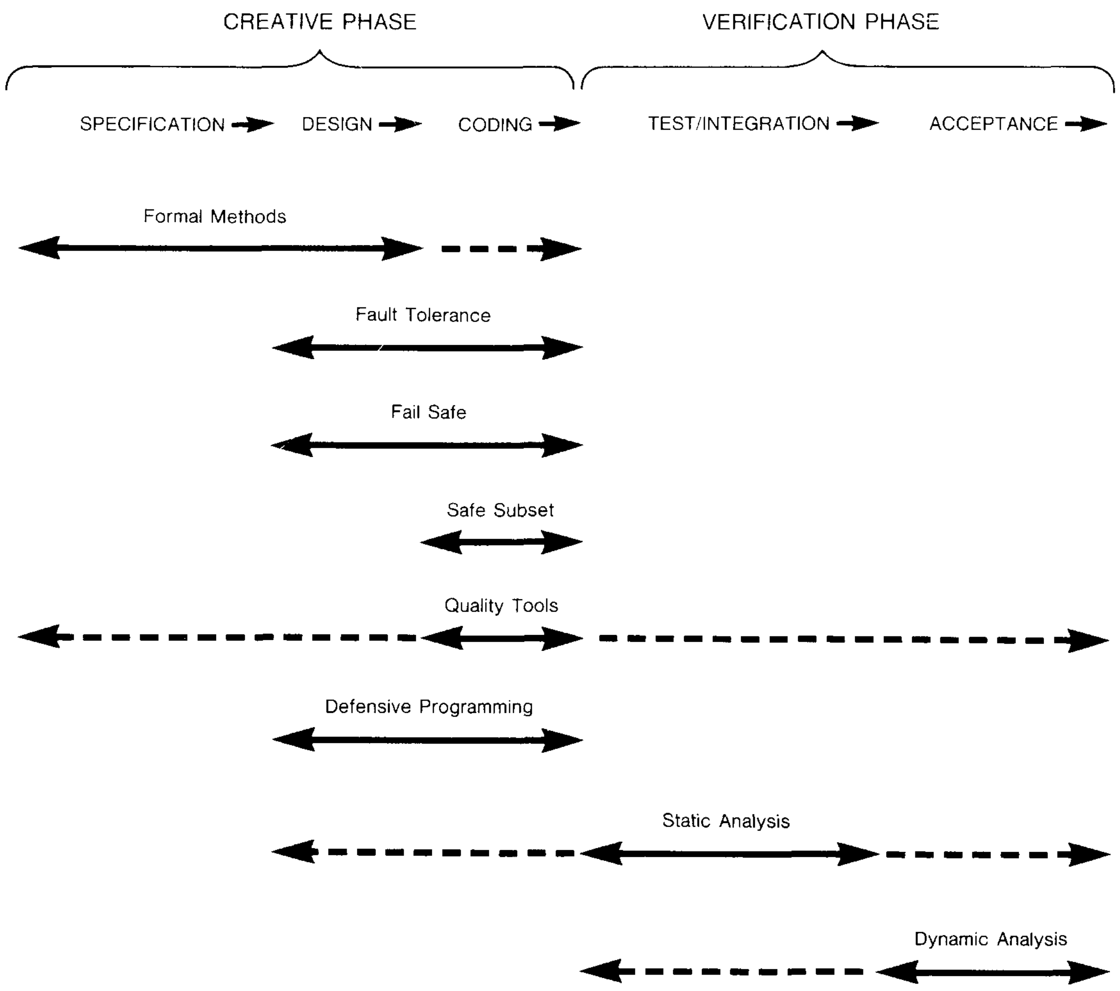

FIG. 2—HIGH INTEGRITY SOFTWARE MEASURES IN THE SOFTWARE DEVELOPMENT CYCLE

Technology to achieve the very high integrity required for safety critical software is relatively new. There are a number of measures which should be applied in addition to the conventional software engineering practices discussed above. These measures, specified by the High Integrity Annex to *Naval Engineering Standard 620*, are shown in FIG.2 in relation to their application in the software development cycle. Further guidance may be found in a MOD paper, the Chief Naval Weapon System Engineer (CNSWE) *Guide to Software for Safe Control of Weapon Systems* (AD/PWS Memo 1/87).

Until recently, there has been a preference for fault-tolerant techniques such as 'diverse' or 'redundant' software. This involves two or more independent software teams designing and coding separate programs to carry out the same functional task, usually running in parallel. Whilst this technique may be useful in hardware technology to avoid the effects of random hardware failures, its usefulness with software is not so clear. Any errors in the originating specification will be reflected in the design of all versions of the software. There are also several other factors which may compromise avoidance of common mode design errors, such as a common technical monitoring point or a common designers' educational background.

Intuitively, it may appear that more testing should be applied and that such testing should be independent of the software designers. Extensive independent testing is a useful technique, identified in FIG.2 as 'Dynamic Analysis', and should be included in the range of measures applied. However its limitations should be recognized. The input domain, even for a small program, will almost certainly be too large to test exhaustively. For example, with only four bytes of information (i.e. a 32-bit word) the input domain is in the order of 4000 million possibilities.

Current preference is for the use of formal mathematically-based methods and tools. Formal mathematical notation can now be used to express specifications and design precisely. Notations receiving most interest in the UK are 'Z' and VDM. In addition, software 'Static Analysis' tools are becoming available. These tools enable detailed analysis of the design and code, and thus can be used to identify logic errors which may cause the program to malfunction.

Use of formal mathematical techniques has largely been confined to experimental application. However, they are now starting to be used on real safety critical equipment projects. There is some debate over their cost of application, but experience is beginning to suggest that the use of mathematical methods results in more cost-effective software than that developed using non-mathematical engineering design practices.

Conclusions

Software is now an important area of technology that must be properly addressed by customer and supplier project managers alike. The management practices of software engineering are well established and managers need to ensure that these practices are applied effectively to equipment software. *Naval Engineering Standard 620* provides a concise set of requirements which should be referred to in contracts for equipment containing software. It is not possible for MOD project managers to understand detailed aspects of the many technologies used to construct modern operational equipment but the engineering control mechanisms should be known. Awareness and early recognition of possible problems is important and managers need to be able to call on knowledgeable advice when required.