(R)

Proc. 1st Int. S/W Process Workshop.
Egham, Surrey, 1984. ed. C. Potts, IEEE Press.

1984b

# ARE WE REALLY SOFTWARE ENGINEERS?

Ian Sommerville

Department of Computer Science, University of Strathclyde,
Glasgow, Scotland.

## 1. ABSTRACT

This paper argues against the notion that the problems encountered in large scale software development are necessarily due to deficiencies in the software process model. Rather, it suggests that these problems arise because software engineering is more akin to a craft than a modern engineering discipline. The paper suggests that the subject lacks secure foundations, that software engineering education is inadequate and that working practices which are common in other engineering disciplines are not adopted. A number of proposals suggesting how this situation might be rectified are made. These include suggestions for a research programme to define computer science, new educational initiatives in software engineering, more effective use of software tools and the establishment of a professional body to certify software engineers.

The 'classical' software life cycle model views the software development process as a sequence of phases namely analysis, specification, design, implementation, validation and maintenance. This has close parallels with the development model for other large and complex engineered structures. For example, like a software system, a warship takes a long time to build, is very expensive, has a long life and its design may be radically changed after the ship has gone into service. This process of change is called refitting (not maintenance!) and is required to adapt the ship to new weapons systems, communication systems, etc.

Although it is not uncommon for ships to be delivered late and to cost more than originally planned, it is my impression that this happens less commonly than with software systems. Why then are marine engineering and other engineering projects, in general, less likely to 'fail' than software engineering projects. By 'failure', I do not mean, of course, that the project is necessarily abandoned. Rather, the term 'failure' encompasses a range of situations from late delivery and above-estimate costs through incomplete system delivery to complete cancellation of the project.

One explanation for these failures might be that we have completely misunderstood the nature of the software development process. Thus we fail because the view of the life cycle which we have is an inadequate model of what really happens during large systems development. By attempting to fit systems development to an inadequate model, we may actually make that development much more difficult. On the other hand, this model seems to be valid across many engineering disciplines and diverse projects. It is my contention, therefore, that it is unwise to reject this model before examining other possible causes of the problems of software development.

The basic premise which underlies this paper is that current problems in developing large software systems have arisen because we do not adopt proper engineering practices in the development of software systems. There is nothing fundamentally wrong with the classical model of the software life cycle although it may benefit from some refinement and we must be careful about rigorously applying it to every project. Our problems are not simply due to deficiencies in the model but are a direct result of the fact that our approach to the construction of large software systems has been that of a craftsman (or perhaps a 'hacker') rather than a modern engineer. Although lip-service is paid to software engineering as a discipline, most organisations rely on empirical or ad-hoc systems development methods and cannot quantify any aspect of their systems except, perhaps, the total systems cost.

Given that the model is fundamentally sound, why then are software projects so much more likely to fail than other large engineering projects? I believe the answer to this is that other engineering disciplines have a sounder base than software engineering, have a means of ensuring that engineers are properly qualified for the tasks which they undertake, and adopt more 'professional' practices when building systems. Let us now look at these points in more detail.

## 2. FOUNDATIONS OF SOFTWARE ENGINEERING

Collins English dictionary defines the term 'engineering' as follows:

> The profession of applying scientific principles to the design, construction, and maintenance of engines, cars, machines, etc.

This definition explicitly states that engineering is dependent on some underlying science and the dependence of traditional engineering on the physical sciences of physics and chemistry is quite clear. Without these sciences as a foundation, modern engineering as we know it would not exist.

It is important to emphasise here that this is a definition of engineering as it is now practised. It is not a definition of craftsmanship which is not based on scientific principles and which has been practised for thousands of years. Whilst it is possible to create sophisticated and well-made systems, such as Chartres cathedral, without such principles, this process is so dependent on the skill and experience of individuals, that it is very expensive and often unrepeatable.

If we substitute 'software systems' for 'engines...' in the above definition of engineering, we should then have a definition of software engineering. However, if we then try and identify the 'scientific principles' on which software engineering is based, we are in trouble. There aren't any! Software engineering is still a craft rather than an engineering discipline.

Whilst some may say that software engineering is rooted in computer science, I believe that the name 'computer science' is completely misleading. There is about as much science in this subject as there is in astrology! If we return again to the dictionary to find a definition of science, we find the following:

> The systematic study of the nature and behaviour of the material and physical universe based on the observation, experiment and measurement and the formulation of laws to describe these facts in general terms.

Of course, the study of computers and software systems is not directly analogous to the 'material and physical universe' because of the man-made and abstract nature of these systems. However, what the above definition does give us is the implication that there is such a thing as a scientific method and that the aim of a science is the formulation of generally applicable laws.

In sciences such as physics and chemistry, a body of accepted knowledge has arisen through the process of hypothesis formation and testing using objective experiments. Hypotheses are formulated by observing the physical world, applying existing theories to these observations then deriving a hypothesis to explain inconsistencies between the observation and the theory. Alternatively, the hypothesis may be derived from theory. In any case, after a hypothesis has been formulated, it is normal practice to attempt to verify that hypothesis by experiment, measurement and observation and, if verified, the hypothesis becomes the new theory.

If we compare this scientific method with the activity which we call computer science, we see that computer science is certainly not practised as a science like physics or chemistry. The notion of objectively testing hypotheses and repeating experiments to verify their conclusions is one which is almost completely alien to the computer science community. Indeed, a research proposal to repeat some other reported work is very unlikely to bee funded.

Although I am suggesting here that computer science is, or should be, considered as an experimental science, this does not preclude aspects of the subject being looked on as a mathematical science. Just as applied mathematics has a vital role in other engineering disciplines, its role in computer science is equally important. As well as helping us understand algorithms, mathematics should also help us to predict and analyse experimental phenomena. Whilst the former role has been explored in studies of program verification, complexity analysis, etc. there has been relatively little use made of mathematics in experimental computer science.

One of the best examples of this lack of scientific method in computer science can be seen in the computer science community's attitude to what are sometimes termed 'Lehman's Laws'(1). I think it is fair to say that these are mostly unheard of in traditional computer science departments yet they are one of the few examples of generalised laws which may be universally applicable to software systems.

The exact nature of these laws is not relevant here - what is relevant is the fact that there have been few attempts to verify Lehman's hypotheses or to try and use these laws to derive principles of software system construction. Similarly, Halstead's software science (2), although it has been investigated for small systems with conflicting results (3, 4), is untried for large systems and we do not know if it is a technique which is generally useful in the analysis and measurement of software systems.

One consequence of this lack of elemental knowledge is that salesmanship is more important than science when a new technique is proposed. For example, techniques such as modular programming, structured programming and, the latest example, object-oriented programming (5) have all been hailed as significant advances. This may or may not be the case but what all those methods have in common is that their proposers did not attempt to justify their claims by means of comparative experiments.

This lack of demonstrable improvement or otherwise means that industry is naturally very conservative in accepting new techniques - after all, they may have succumbed to salesmanship in the past and found the product inadequate. Conversely, academics tend to embrace many new techniques with fervour in the hope that they aare an improvement over existing working practices. Neither attitude is correct - we must retain the best of the established methods whilst replacing inadequate methods with newer and better approaches.

Part of the blame for the lack of scientific foundation to software engineering must lie with the scientific establishment. Proper, comparative experiments involving large software systems are very expensive and computer science research has always been under-funded. However, part of the blame also lies with those of us actively involved in computing research. A 'not-invented-here' philosophy pervades university computer science and this, combined with our distaste for proper scientific experiments has meant that far too much time has been spent on playing with computers rather than in productive work.

In the conclusions to this paper, I make some suggestions as to how this situation might be changed. I believe that until it is changed and until some fundamental laws of software systems are established, software engineering cannot become a true engineering discipline.

## 3. SOFTWARE ENGINEERING EDUCATION

The lack of foundations which means that software engineering is a craft rather than a science-based engineering discipline is probably the most fundamental reason for our failures in software system construction. It is not the only reason however - some of those practices which have been developed and tested and which have proved useful are sometimes unknown and often unused. In short, we are not even making the best of the limited resources which we have.

At least part of the reason for this is that many, if not most, of the staff involved in developing large software systems are inadequately educated. It is common practice for companies to hire individuals trained in disciplines as diverse as music and marketing to be programmers and to give them relatively complex work to carry out. 'hilst these individuals may understand the echanics of programming and may also understand what is being programmed they are unlikely to understand the reasons why particular methodologies are used, why programs should be readable, etc. Would you like it if the designer of the bridge you drove over every day was a biologist with no formal qualifications in civil engineering? In fact, government regulations probably exclude this possibility yet there is a high probability that you fly in aircraft whose software systems were built by 'amateur' software engineers.

This situation is compounded by the fact that, throughout most of industry, there is little or no investment in postgraduate training except of the most specialised kind. Staff are not given the support or the opportunity to learn of new developments so naturally continue to use and to be proud of their undisciplined methods. As a result, software productivity throughout most of industry is much lower than it ought to be and there is constant reinvention of already known techniques.

Of course, universities and other educational institutions are also responsible for inadequate software engineering education. University partments of computer science have been slow to ecognise that there is a qualitative difference between large and small software systems and to include a software engineering component in their courses. They have tended to concentrate on courses in specialised topics (compiling techniques, simulation, etc.) rather than devote resources to the practical study of large software systems. In essence, we place too much emphasis on teaching topics which allow the graduates to be immediately productive and not enough on teaching core knowledge which is generally applicable and which will not rapidly go out-of-date.

I believe that there are four reasons for the failings in many current courses in computer science. These are as follows:

(1) Many computer science academics drifted into the subject from mathematics and have no appreciation of what's involved in developing a large software system. They prefer to concentrate on aspects of the subject which appear to be more tractable and which have some affinity to mathematics. The lack of solid computer science makes it difficult to argue against this approach.

(2) Until very recently, there were virtually no student textbooks which covered software engineering as a coherent subject. However, this situation has recently changed for the better and a number of general texts are now available.

(3) Many university staff seem to be more concerned with assessment rather than with education. Hence, individual working which is easier to assess is encouraged and group working (or copying!) is discouraged. This is in direct contrast to the situation in most software development projects so students never get experience of anything but relatively small programs.

(4) Industry and commerce have made and continue to make unreasonable demands on computer science departments to produce graduates who are fully productive immediately after graduation. This may be contrasted with industry's attitude to graduates in other engineering disciplines where it is assumed that the university course should provide the foundation for further vocational training rather than the vocational training itself.

I discuss in the conclusions to this paper how this situation might be changed. However, this must be preceded by a recognition in both industry and in universities that software engineering is an engineering discipline and that graduates need practical experience and postgraduate training to develop their skills. One point worth noting is that other engineering disciplines have the benefit of professional institutions, such as the Institute of Mechanical Engineers, who maintain professional standards. Membership of these institutions is only awarded to those who have both academic qualifications and sound post-qualifying vocational experience. Membership of such institutions is an indication that the individual is truly a professional engineer.

Essentially, membership of an engineering institution signifies that the member has both academic ability and qualifications and practice in the subject. Notice that it is not expected that a new engineering graduate should be immediately useful and that membership of an Institution is only granted after considerable post-graduate experience.

## 4. PROFESSIONAL PRACTICES

As engineering has developed, a number of standard practices have evolved which, I believe, make an essential contribution to systems development. These practices are not generally adopted by software engineers and this is partly responsible for some of the difficulties which we encounter in building large software systems. Consider some practices which are the norm in other engineering disciplines.

(1) Quantification of requirements

When a mechanical engineer needs to use a beam, say, in some structure he does not specify that a 'strong beam' is needed. Rather, he quantifies the strength of the beam that is required. Software specifications, on the other hand, are full of 'weasel words' like 'easy to maintain', 'readily portable' etc. We need to express such concepts in unambiguous terms not woolly words.

Although attempts are now being made to develop notations to express functional specifications, there seems to be little work in progress which is concerned with the quantification of the reliability, readability, portability, etc. of software systems.

(2) Prototyping
When building a large and complex system, it is unrealistic to expect a complete design of that system to be completed on paper. Rather it is expected that a prototype or model be built to iron out problems in specification and design. How many organisations who develop software systems use prototyping as a matter of course?

It has been suggested than an evolutionary approach to software development through more and more sophisticated prototypes, is a better way to attack certain software problems rather than the traditional software process model. This may indeed be true where the system requirements are very difficult to define. However, engineering prototypes are usually intended to be throw-away systems and it seems to me that some of the requirements of prototype construction, such as the requirement that a prototype should be developed quickly, are likely to conflict with robustness, performance and reliability requirements.

I thus believe that prototypes should be mostly used as in other engineering disciplines to help with requirements definition and design. Only in exceptional circumstances should they be retained and used, in their entirety, as a basis for further development.

(3) Standardisation
When a complex system such as a warship is built there are many completely new components involved but there is still great reliance on the use of standard components which have already been tried and tested. The use of standard components, even if they are not ideal, significantly reduces system costs yet component sharing in software engineering is a practice which tends to be informal in some organisations and non-existent in others.

Part of the reason for this is the lack of tools to facilitate sharing but I believe that the root of the problem is the 'not invented here' syndrome which means that everyone rewrites everything every time it is needed.

Other engineering disciplines also use another level of standardisation in the notations which are used to describe their systems. By the universal use of blueprints, a design produced by one engineer can be understood by more or less any other engineer in the same discipline.

However, there is no commonly understood notation for expressing a software design except, perhaps, the now-discredited flowchart. In fact, a recent count of software development methodologies and their associated notations revealed about 200 different methodologies in use! Thus, the problems associated with the communication of software designs are immense.

In this respect, the programming language Ada may be very important. Although that language has been criticised for its size and complexity, it seems likely that it will be generally adopted and understood by the software engineering community. At last we may have a common language which allows us to talk to each other!

(4) Quality control
Quality control is built into large engineering projects from the very beginning in that components used in the system have been checked, their materials have been checked, and so on. In software projects, quality control is often equated with testing and, in many cases, is something which is applied after a great deal of work has been done. Quality control does not just mean assessing if the software works but is something much broader which evaluates the way in which the software has been constructed as well as its correctness.

Of course, quality assurance is practised by all organisations involved in software development. What I believe, however, is that the qualities tested by QA practices are only a subset of those qualities which should be assured. For example, is the readability of the software tested, the functional independence of software components, even the testability of components? These tests are quite distinct from testing if the system 'works' yet have a very important bearing on overall life cycle costs.

Of course all these points are interrelated. Quality control is difficult without standards and standards are difficult to establish without metrics and without some foundation knowledge about the most appropriate techniques for systems development. Nevertheless, the fundamental problem is in our attitude to these practices - we make little attempt to apply professional practices, however inadequate, to the development of our software.

## 5. CONCLUSIONS

Curing the ills of software engineering is not a short term process. Fortunately, perhaps, governments have been forced into thinking about the problem by the Japanese Fifth Generation Project and, in the UK at least, there is some evidence that a change is at hand.

As I see it, the steps which we should now be taking are:

(1) The immediate commencement of a proper scientific research programme whose aim is to derive the foundations of software engineering or, if you like, to define computer science. This will objectively compare and evaluate existing techniques and methodologies and find out which of these are 'best' for particular tasks. The first priority in such a programme is to investigate how to quantify the subject so that we can dispense with terms like 'easy-to-use' and 'properly structured' and so that we can measure and compare different approaches to software development.

Some examples of experiments which immediately spring to mind are:

An experiment to find out what notations are suitable for prototyping particular classes of system. This would involve developing prototypes in various notations and developing a system without prototyping. Development times, system sizes, ease of modification, number of requirements changes, etc. could then be measured. The programme might be speeded up by introducing artificial requirements changes and extrapolating the results.

An experiment to verify Lehman's Laws by examining a number of large systems and their maintenance.

An experiment comparing Prolog and Lisp as notations for the development of expert systems. This would involve developing a number of expert systems in both Prolog and Lisp and comparing system sizes, development times, difficulties encountered, etc.

An experiment to compare different software development methodologies such as JSP, Structured Design, etc. to find out which are best suited to particular classes of system.

Such experiments would make up a very expensive programme of research as the aim should not be to produce working systems but to derive results. However I suggest that such a programme is less expensive than research in high-energy physics or astronomy, say, and has much greater potential economic benefits.

The results from such a programme, could be the basis of standards which would be demonstrably adequate and which are likely to be adhered to by the software development community in general.

(2) The establishment of an Institute of Software Engineers with membership only awarded to those with acceptable qualifications and practical experience. This should be integrated with other ngineering institutions and should monitor university courses, etc. for acceptability. It should be the norm for practicing software engineers to be members of this institution in the same way as practicing civil engineers are members of the Institute of Civil Engineers.

I do not think it is possible for the existing computing societies such as the British Computer Society and the Association for Computing Machinery to take over this role. They are not seen by either members or the industry in general as being in the same position as the engineering institutions and they are too well established for this situation to change. It may however, be possible for one of the existing engineering institutions to embrace software engineering although there are many problems involved in reconciling the different qualifications involved.

(3) The development of a new approach to the teaching of computer science which is oriented towards producing people who can design and evaluate software. Many if not most current computer science courses spend too much time in teaching commercial data processing and other applications simply so that they seem relevant to industry. This time could be spent on more general aspects of the subject. Both universities and industry must accept that, in the long-term, it is better to equip university students with the ability to learn new techniques than simply train them to be immediately productive.

Ultimately, I hope that university courses in software engineering will be established which would have a practical bias. I am not sure that the time is yet ripe for this because of the insecure foundations of the subject.

(4) The adoption of professional engineering practices discussed above. Whilst this may lead to a short-term cost increase, the ultimate benefits will be a reduction in overall software costs.

Whilst it is true that proper professional practices rely on secure foundations, it is not good enough to do nothing until foundations are established. Practically any standards are better than none, we probably have adequate notations for systems prototyping in languages like APL and Prolog and even a qualitative evaluation of software quality is bound to lead to overall improvements.

(5) The adoption of the view that software technology is capital intensive rather than labour intensive. Historically, the greatest productivity improvements have been realised when well-trained staff have been equipped with powerful tools. We must build and promote the use of powerful software tools and accept that software productivity is likely to be improved when staff have sufficient computer power so that software development is never constrained by hardware limitations.

Fundamentally, however, we need a change in attitude by software buyers, software management and by software engineers themselves which must encourage and reward professionalism rather than system hacking. Unless professional engineering practices are adopted, we will continue to produce systems which do not meet the user's needs, are unreliable and which cost far more than they ought to. No amount of tinkering with process models will compensate for the fact that we still act like amateurs at this business!

## 6. REFERENCES

(1) Lehman, M.M. & Belady, L.A. 1976. 'A model of large program development'. IBM Systems Journal, 15(3), 225-52.

(2) Halstead, M.H. 1977. 'Elements of Software Science'. Amsterdam : North-Holland.

(3) Hamer, P.G. & Frewin, G.D. 1982. M.H. Halstead's Software Science - A Critical Examination. Proc. 6th Int.Conf on Software Engineering, pp 197-206

(4) Shepherd, S.B., Curtis, B., Milliman, P., Borst, M. & Love,T. 1979. First year results from a research program in human factors in software engineering'. AFIPS. 44, pp 1021-1027

(5) Booch, G. 1983. Software Engineering with Ada. Reading, Mass. : Addison-Wesley

## DISCUSSION

### 1. Science and Engineering

**Jackson:** One of the fundamental differences between computer science and natural science is that we, the so-called computer scientists, are ourselves free to manufacture more software, whereas natural scientists are not free to manufacture more of the world - that is some greater power's responsibility. So the question that always arises in theories about software is whether Halstead is just talking about programs that he knew of, and they may be radically different from the ones that are going to be written the day after his book is published, or is he saying something general and universal? Because if it is only the first, then perhaps we shouldn't be paying any attention to it anyway? What is your position on that?

**Sommerville:** There is no question whatsoever that software is a unique type of commodity and any comparisons with other sciences are not 100 percent valid - I'd be the first to accept that.

**Kolence:** My position has always been the following: a science has the job of understanding how things work, and an engineering discipline has the job of building something using the understanding of how things work. There is a mixture now in the field of software engineering and computer science because we can't do our engineering jobs if we don't know how things work and we don't see the people in computer science turning their attention to the things we need to understand.

**Sommerville:** I agree with you absolutely.

**Turski:** The lack of foundations - it causes laughter, but ships were built long before Archimedes discovered that principle of flotation. Sound principles have nothing to do with the engineering profession. Aircraft fly even though there is no satisfactory mathematical theory of aerodynamics. Roman roads are better than some roads are today!

**Sokol:** Maybe it is not possible for software these days to be any more methodical or manageable than highway systems, which have a great social element in them that makes them much less tractable than if they were mere artifacts. I wonder if your points are possibly misdirected, in the sense that you are trying to treat software engineering as though it dealt with a natural artifact, whereas it may not be.

**Sommerville:** It is more like an economic system, you mean? That may be true, yes, I accept that.

### 2. Software Engineering Education

**Pyle:** Isn't assessment checking that people have the ability -have acquired the necessary professional standard? You don't want to have a person who cannot be assured by the university that he has achieved a professional standard. That is assessment.

**Wasserman:** Do you do quality assurance on the software industry? If I turn in a project working for a company, don't they want to see if it has bugs in it and if it works and decide if they want to promote me or hire me or give me a raise or whatever? It is just a different form of assessment, but the goals are the same.

**Sommerville:** I'm not arguing that assessment is a bad thing. I'm just saying that we have courses that are constrained by assessment. We are not teaching the right things because they are hard to assess.

**Wasserman:** I thought that statement applied to industry and not to the universities.

**Lavi:** I don't think that we have to handle very large objects in a university in order to teach people proper engineering - you can use small objects.

**Sommerville:** I disagree with that entirely, because it is always possible to handle small projects in an undisiplined way, whereas I believe for a large project, you must have discipline.