SOFTWARE: BARRIER OR FRONTIER?

C. A. R. Hoare,
Oxford University Computing Laboratory, 11 Keble Road,
Oxford OX1 3QD.

Tel: +44 865 273841

November 23, 1999

In recent times, the capabilities of computers and of communication systems have improved by many orders of magnitude. Every year the frontiers are being pushed forward at an increasing rate by the amazing skill of researchers, inventors, designers and production engineers; and yet further advances are confidently predicted for the years to come.

But everyone knows that an exponential rate of progress cannot be maintained forever. Frontiers can be advanced only until they meet absolute barriers imposed by physics — the speed of light and the imprecision of quantum mechanics. It may be ten or it may be fifty years hence — that I do not wish to argue about. The purpose of my talk is to share with you a more immediate fear. I am afraid that the full potential for advance of the frontiers of communication technology is already inhibited by the problems of complexity of computer programs and software; that these problems are increasing; and that they threaten to build up an impenetrable barrier to the full exploitation of hardware improvements, preventing or seriously delaying their penetration into the market place. I will suggest that this barrier can be shifted only by wider appreciation and application of current understanding in software

engineering; and that this understanding needs development by continuing research, and propagation by continuing education in the necessary skills for successful application. I shall describe some of the goals and ideals of software engineering, and explain why I think that the telecommunications industries will be among the first to achieve these ideals and realise the benefits of doing so.

The main reason for this is the growing convergence of computing and communications, together with the widening realisation that it is software that provides the essential link between them. With a certain element of exaggeration, you could say that everything that the communications industry regards as its own preserve, all that vast network of hardware — handsets, displays, wires, fibres, switches, links, antennae, satellites — everything except the physical holes in the ground and the towers in the air — is controlled by software or soon will be; they are just peripheral equipment, as it were, to the computers which run the programs. Software is the magic ingredient which realises the growing potential of recent and predicted advances in the hardware. It is software that adds value, that assembles the components into saleable systems, products and services. Hardware components will be manufactured in increasing volumes and supplied at reducing cost to all the players in the communications marketplace opened up by deregulation. It is the software that will determine competitive advantage, and distinguish the winners from the losers.

So what is this discipline of Software Engineering? How does it compare and differ from other Engineering disciplines? And how far can it be regarded as mature?

The most strikingly visible difference between software and other engineering products is the almost total invisibility of software.

There is absolutely nothing photogenic about software, and absolutely no joy in building scale models of its operation. And even less can it be touched, felt, heard, smelled or tasted. It seems to be nothing more than the abstract disembodiment of pure complexity. As a corollary, software is a product where the cost and time required for manufacture and distribution are close to absolute zero. All the expense and delay is in design and development — and later in marketing and sales. Historically, this has made software a very difficult area in which to gain recognition for sound research in Universities, or in which to exercise sound planning, management, and control in Industry.

But of course the cruder parameters of software can readily be measured. Over the last ten years, such measurements often show that the length of computer programs embodied in a typical product have grown, perhaps by a factor of ten; that they have cost ten times as much to develop, that they are proportionately more likely to contain errors detected in service, and each error could potentially be ten times more damaging in its effect. Hardware has in the same time made equally rapid progress but fortunately in the opposite direction. It is now ten times smaller, faster, cheaper, and more reliable. I really shouldn't have embarked on this comparison, so dreadfully unfavourable and unfair to software engineers.

Let's change the subject quickly, and concentrate on the much more important similarities between software and other branches of engineering. Firstly we share the same goals; they were nicely defined for Civil Engineering by Thomas Tredgold in 1828 - "the art of directing the great sources of power in Nature for the use and convenience of man". Secondly, the success of any engineering project requires full attention to the implications of marketing,

commerce, accountancy, management, and even politics. But the single most important feature, the one that differentiates all of engineering and science from all these other important practical concerns, is the explicit, crucial and pervasive use of the techniques and notations of mathematics. Each branch of science seeks mathematical theories, or models of selected aspects of physical reality. The scientist uses mathematics to predict from the theory its observable consequences, which are then checked by careful experiment. The engineer on the other hand uses a validated scientific theory to check the performance parameters of a design before it is put into production. That for example, is why our buildings and bridges no longer blow down, — at least not very often.

In a mature engineering discipline the direction of the mathematical calculations is reversed. Start with a mathematical model of the customer's requirements. Decide the general strategy and structure of the solution; and then with the aid of calculation derive the content and detail of the design, including optimisation of relevant parameters. The design now needs no further test or check - it is correct by construction. This reversal from bottom-up predictive mathematics to top-down design calculations is the goal of all research into engineering method, in all branches of engineering, and especially software engineering.

But this simple story has ignored the incredible complexity of the symbolic and numerical calculations required by modern science and engineering. To limit this complexity, science presents not just a single model of reality, but rather a whole hierarchy of models, dealing with phenomena at differing scales, at differing degrees of granularity, and at differing levels of abstraction. For example, starting at the level of chemistry, physics offers the hierarchy

- molecular dynamics,
- atomic theory,
- elementary particles,
- chromodynamics.

Each level has its own autonomous concepts and its own model, which can be understood and used independently of the others. It is the general goal of the pure scientist to secure the links between the levels, defining the concepts at each level in terms of the previous one, and then proving its axioms as theorems in the previous theory.

A similar hierarchy of levels of abstraction is equally necessary in engineering. For example in design of computer hardware we separate

- instruction set
- register path
- microcode and control
- combinational design
- switch level design
- circuit electronics.

Here again, at each level there is a different conceptual framework, a different notation, and a different calculus of design. A complete design at each level of abstraction serves as a specification for the design at the next lower level. It is the particular goal of the practising engineer to ensure that a specification at each level is correctly and efficiently implemented by the selected design at the next lower level.

Communications engineers engaged in the design of protocols are familiar with the famous seven levels of the international standard

- Application
- Presentation
- Session
- Network
- Transport
- Data link
- Physical level.

The general principle of the hierarchy has been very successful. At the higher levels, there has been some delay in finding and agreeing on the appropriate abstract concepts for formulating the standard; but this kind of conceptual engineering is the necessary condition for breakthrough in any branch of science, or indeed any kind of intellectual endeavour. Just saying that is never going to make the discovery easy to make, however simple it may seem afterwards. At the lower levels of the protocol hierarchy, maintenance of the design structure throughout the implementation can cause problems of efficiency; but these are solved with the aid of correctness-preserving transformations, which combine the benefits of structured specification and design with highly optimised implementation. Software engineers can learn a lot from this transformational approach to specification and design.

Software engineers also have a hierarchy based on scale and granularity. They talk of

- systems
- modules

- classes
- objects or processes
- functions and subroutines
- straight line code
- individual instructions.

The mathematical theories which are useful for design calculations at each level have been to a large extent developed by software engineering research, and the transitions at the lower levels have been successfully formalised and even automated. The results of this research are being gradually assimilated into industrial practice. Progress is slowed by a general antipathy to mathematics among software engineers — but this feeling is yet another characteristic which is shared with other branches of engineering, and indeed with most of management, and with the general population.

Mathematics itself provides an outstanding example of the control of complexity by structure and abstraction. Its branches can be arranged in a hierarchy like those of physics — topology abstracts from analysis; analysis provides the basis for calculus; and calculus can be used by engineers who understand nothing of the more abstract foundations. Even within a single branch of mathematics, a lemma can be safely used without studying the complexity of its proof, a theorem abstracts from the complexity of its lemmas, and a theory from the collection of its separate theorems.

Similar structures are observable in good programming practice, where larger programs use smaller ones as subroutines, or subobjects, or subprocesses. But this analogy is good only if the formal statement of the function and purpose of each subroutine is as simple and complete as that of a mathematical theorem, and an

order of magnitude simpler than the code which implements it. Furthermore, the reliability of the code must also approach that of a mathematical theorem. A large system constructed from even slightly unreliable components can rapidly collapse, either before or after delivery. Reliability is the very essence of engineering, and it is achieved by explicit appeal to the concepts, methods, abstractions and structures of mathematics.

But, ironically, it is not to the traditional applied branches of continuous mathematics that the software engineer turns for guidance, but rather to the traditionally pure branches of discrete mathematics, — set theory, algebra, and even category theory. The reason is that software engineering deals very largely with discrete phenomena, transitions, events, values and structures. At the lowest level we have just the two discrete Boolean values, zero and one. In a large program, if just one of these digits is changed, even only for just one millisecond, the consequences for the whole program, indeed the whole system, are in practice quite unpredictable, and in principle potentially disastrous.

This means that the software engineer cannot rely on smoothness or continuity in the control of tolerances or error. Numerical approximation is simply not available as a technique to simplify calculations. Since there is no appropriate metric, worst case analysis and worst case testing are just not available. For the same reason there is no freedom to get it nearly right; even if there were, there is no way that this would simplify the task of design and implementation. Approximation, even to the extent of order-of-magnitude calculation, is the stock-in-trade of the engineer, the most important way of maintaining intellectual control at the early stages of design and throughout the later implementation. In the

discrete branches of engineering, for these purposes we have to rely almost wholly on structure and on abstraction.

The gap between continuous and discrete engineering is one that puts nearly all modern telecommunications and electronic hardware design on the same side as software engineering — certainly all of network design down to the individual switch, and all of VLSI design, down to the individual logic gate. To cover this range of disciplines perhaps I should use a more neutral term like Discrete Systems Engineering, or a more fashionable one like Information Engineering. Under whatever title, I believe that we will see a strong convergence in the practice of engineering of software, hardware and communications.

This convergence is simply and elegantly illustrated in mathematical theories known as process algebras, developed over the last twenty years by basic research in Universities. Such a theory combines the concepts of conventional sequential programming with the kind of concurrency which is embodied automatically in every combination of hardware components, and the kind of communication which occurs almost naturally whenever hardware components are connected by wires. The theory has already served as the basis of a draft international standard (LOTOS) for the definition of protocols, for the design of a programming language (occam), and a microprocessor (the transputer), and the design of several silicon compilation languages. But the mathematical theory is much more general; with slight variations, it can be applied on every scale and at every level of abstraction and every level of granularity, from the customer's view of the services required of the system as a whole, through the design of the major components of the network and the interfaces between them, down to their implementation on

a collection of processors and special-purpose application-specific hardware, interacting with each other at any distance. It is this appeal to abstraction that permits theories tested in the laboratory to be cautiously scaled up for industrial application. The uniformity of the mathematical foundation permits all stages of the design and implementation to be related by calculation and proof. Many of the stages of adaptation and optimisation can be codified and carried out (or at least checked) by automatic transformation systems. It is this that promises not only an increase in the quality and reliability of the product, but also a reduction in the time to market.

Experience with this kind of practically applicable theory leads me to predict that discrete systems engineering is making rapid progress towards the status of a mature engineering discipline. A fully mature discipline will have the following characteristics:

- It puts the customer first.
- It codifies corporate strategy.
- It puts management in control.
- It magnifies human intellect.
- It builds its own tools.
- It is the language for professionals.
- It is transmitted by education.

A mature engineering discipline puts the customer first. It starts with a scientific investigation of the actual characteristics and behaviour of the customer population, not just as individuals but as members of their societies; in the workplace, school or home. It

takes full advantage of the methods and results of the human sciences, - physiology, psychology, linguistics, sociology. It analyses stated requirements and stated assumptions until a clear picture emerges of some desirable future product or development that will satisfy the true requirements, which have often been left unstated. The mismatch between perceived and actual requirement is one which must be overcome by good marketing. Meanwhile, the engineer must attempt, as soon as and as far as possible, to construct a faithful mathematical model of both the numerical and discrete properties of the projected and desired interactions between the community of customers and the projected product or service. This is the first and most important interface to define; it is the basis of all subsequent engineering design, and any lapse of judgment here could lead to a product that is undeliverable or unusable. The pace of change is no longer driven solely by technology: in software especially, the technology must be driven by the customer.

A mature engineering discipline formulates strategic policy. No large enterprise can afford to design and deliver a single product at a time, no matter how advanced the technology or how timely its introduction to the market. The real challenge is to design an architecture for a family of products, covering not one but a range of markets, with not just one product in each market but a series of complementary, supplementary, enhanced and eventually replacement products, stretching into the foreseeable future. The strategy must be presentable in abstract terms at Board level, so that it can be correlated with financial management, marketing, resource planning, and ultimately with the image that the enterprise wishes to have of itself. An engineering discipline must provide

the appropriate abstractions and theories to define the structure and interfaces of the entire family, long before any of the detailed design begins; and this must be backed by enterprise-wide engineering standards which give assurance that the strategy can be implemented as planned. The days of innovation as adventure are over. In the framework of strategic policy, innovation is routine.

A mature engineering puts management in control. Each level and branch of management can understand, within a self-contained intellectual framework, all the objectives and activities of subordinate levels, so that it can take confident responsibility for the way in which these activities contribute to the goals of superior levels. The confidence is based upon abstract but precise formalisation both of the vertical and of the horizontal interfaces throughout the management hierarchy. The confidence is justified by mathematical calculations, which establish in advance of implementation that if each subordinate goal is met, the superior goal is guaranteed. Complexity is controlled by correlating levels of management with levels of abstraction, so that problems and delays can no longer be hidden under a morass of technical detail. In spite of the intangibility of software, signs of trouble are immediately visible, and if change is required in the design or in the interfaces, the manager can explore and report all the wider consequences of the change before authorising it. As a result, there are few last-minute surprises. When the components are delivered, they slot together without prolonged integration testing, and they can be delivered immediately with minimal risk of feature interactions discovered in service. "Design right first time" is no longer a slogan but has become a habit.

A mature engineering discipline releases the full potential of the

human intellect. Because specifications are expressed at the highest possible level of abstraction, they give the widest possible scope for exercise of the design skill, ingenuity and inventiveness of the human engineer. The mathematical theory defines the boundaries of the design space, and provides the method by which it may be thoroughly explored. As new ideas emerge, they can be crystallised with the aid of mathematical formulae, which can be objectively discussed, evaluated, justified or even admired. Finally when the design is frozen, it can be made sufficiently precise for reliable implementation by teams of less experienced or inventive engineers or technicians.

A mature engineering discipline constructs its own design tools. The validity of the methods which transform specification to design and design to implementation is assured by their basis on the well-established scientific theories which underlie the discipline. Only parts of the tool are fully automatic; at all crucial stages, guidance is needed from the skilled and experienced engineer, who has the understanding and inventive talents to direct the design towards a cost-effective solution. The only contribution of the tool will be to calculate a few parameters, and to organise the mass of associated detail in a manner which ensures correctness by construction. In future, the programming of individual lines of C-code will seem as archaic as laying out individual transistors and wires on a silicon chip. But even when a tool is really successful, the general impression should be that it only does the easy bits.

A mature engineering discipline provides a language for communication among professionals specialising in its various branches. The underlying mathematical theory not only explains the common foundations of all the branches, but explains why and for what pur-

poses it is necessary to differentiate between each of them. There is no longer any need for clamorous conflict between the various branches of software engineering, each claiming exclusive merit for a single computational paradigm: the functional programmers, the logic programmers, the object-oriented programmers, and the no-nonsense hewers of hardware or hackers of C. Any large system will have components constructed from a variety of technologies; and the interfaces between the technologies, which is where most of the problems of engineering arise, are controlled by the abstractions of the underlying general theory.

Finally, a mature engineering discipline is transmitted to future generations of engineers by further education. Its theoretical foundations, their abstraction and elegance, can be taught as a free-standing mathematical discipline at University or even at school. Its methods and principles can be illustrated on a small scale by student demonstrations, experiments and exercises. By repeated exposure at many different levels to the transition between abstraction of specification and details of implementation, the student comes to understand how the techniques generalise to an industrial scale of application. When this education has been complemented by a period of industrial experience, the educated engineer is intellectually equipped to rise through the management hierarchy to the very highest levels.

This brief and idealised account of engineering education contrasts strongly with the training on the job, which was the only training available when I entered the profession in 1960, and is still the norm today. We learnt programming in a wholly operational fashion, by trying to understand the behaviour of the computer which is executing the program. Execution traces were the only

means we had of understanding and removing errors. Errors were regarded as inevitable, because we had no technology to avoid them, even in principle. We hardly recognised the possibility that a complex program might have a simple specification, of far greater benefit to the customer than the implementor. Lengthy and total immersion in operational detail actually inhibited progress towards understanding of the necessary simplifying abstractions.

When it became necessary to learn a new programming language or use a new operating system, training was based on the voluminous manuals which accompany the software. Because there was no common culture or education in the understanding of abstraction, these manuals too have to be based on the lowest level of operational detail. Their volume, complexity, and structural deficiencies absorb all the intellectual energy of the student; and yet they were so incomplete or even inconsistent in detail that, when used in earnest, the only way of finding out what the software will actually do is by experimental trial and error. The tool which should be helping has become part of the problem.

The absence or even conscious avoidance of mathematical abstraction in programming education explains why many programmers have often been regarded more like craftsmen or technicians than engineers. They are wonderful people, with experience and skills greatly to be admired and valued. But they work best in isolation on self-contained tasks. They have no language to discuss, explain and justify their work to their colleagues and superiors. Documentation is their bane. They do not read the technical literature to keep abreast of their field. On promotion, they find it difficult to establish or maintain intellectual control of the work of their teams. That is why it is rare for the best programmers to rise

to the higher levels of management. Yet it is not conducive to the health of the enterprise when the worse ones are highly promoted.

The transition between a craft and a mature engineering discipline is always fraught with confusion, difficulty, animosity and charlatanism; and the intangibility of software has certainly prolonged the agony. But the more far-sighted enterprises can see the competitive advantage to be obtained by rapid transition to an engineering attitude towards software. Among them, the telecommunications industries are playing a leading role. It is essential to them to raise the educational level of their software engineers, by in-service courses for experienced programmers and their managers, by promoting the quality and relevance of the subject in higher education, by promoting research at the interfaces between technologies; and by attracting the very best of the graduate population into their teams and eventually into their management. Divergence of culture between traditional communications engineers and the new software engineers educated at leading Universities must not be allowed to hinder the flow.

I have painted for you an idealised picture of the potential benefits to be obtained from a rigorous approach to the discipline of software engineering. I have suggested a useful contribution can be made by education and research at Universities. But there are many familiar and even unavoidable difficulties and dangers that can delay achievement of the ideal. They include

- short term commercial goals
- false analogies
- premature standardisation
- late standardisation

- inadequate tools
- misguided research
- shallow education
- failure in planning.

To counteract all these widely prevalent difficulties and dangers, all I can offer is my picture of an ideal of software engineering. I hope to inspire you with a vision of what might be possible if the difficulties can be overcome. But that will depend on the determination of political operators, commercial managers, as well as working engineers and programmers, perhaps some among my audience today. It is not for me to predict whether my ideal will be approached or achieved in this century or in the next; whether software remains a barrier which inhibits the spectacular march of progress in all other branches of telecommunications technology, or whether the frontiers of software can be moved forward in step with other advances, to ensure rapid delivery of new products and services, to the benefit of customer and supplier alike.