Will There Ever Be Software Engineering?
by Michael Jackson

In his opening essay in this issue, Ed Yourdon forecasts both a happy and an unhappy future. In his bright future we have challenging projects, exciting technologies, innovative applications, giant salaries and lucrative stock options. In his gloomy future we have US Federal and state government departments unable to solve their Year 2000 problem, a business revolt against expensive and troublesome software that delivers no apparent economic benefits, and a consequent drying up of money to buy new releases of COTS software or to finance new software developments. The good news and the bad news are essentially commercial. But if the bad future comes to pass, there might be demands for certification and licensing of software professionals and for a formal approach to software development. In a word, we might be expected to become serious software engineers.

We won't, of course. Yourdon is confident that we have learned a lot about software processes, methods and techniques; he says that we have a vast body of knowledge in requirements, risk management, metrics, testing and quality assurance, incorporated into the SEI Capability Maturity Model. So that's all right, then. We don't really need software engineering in the narrow academic sense. He quotes Jerry Weinberg with approval: "There are only three problems in developing software: people, people and people." Then he tells us that he wasted six hours struggling with fatal system freezes on his Macintosh Powerbook running OS8 and Microsoft Word 6.0.1; his Intel-based PC running Microsoft Office97 had been even more problematic. The people, people and people involved here seem to be Microsoft, Apple and Ed Yourdon. But his tale is not intended as a criticism of Microsoft or Apple. I don't understand why, unless perhaps he thinks the fault is his own.

But do these errors really matter anyway? Yourdon says he is a busy professional whose time is relatively expensive and valuable. He can't afford to have that kind of experience very often. Later, on the other hand, he cites a more optimistic view: "That's the price of progress. You're bound to have some rough edges with new, innovative applications that we're building on this year's new, advanced hardware". And we might agree that word processing is not exactly a safety-critical application. Just look over the printed output and try again if it's not right. If your system crashes, reboot. If that doesn't work, just go back to an earlier release of the operating system. The errors don't really matter that much, just so long as they don't happen too often.

But the errors surely do matter. First, the world of computer-based systems is becoming ever more interconnected and interdependent. One of the more startling entries in Peter Neumann's monthly Risks column in ACM Software Engineering Notes reads: "Word Perfect upgrade crashes Utah phone system". That sounds very like a result of the exciting technology and innovative applications that Yourdon's bright future holds. What will happen when developers really start revelling in his "opportunities to create platform-independent, user-friendly, functionally sophisticated Web applications in Java"? Second, this ready interconnection of systems reflects the concept of a single software industry with a single set of shared basic assumptions. The idea that software must always be full of errors, that shipping buggy code in time for Comdex is better than shipping no code at all, that a rich feature set is ample compensation for an even richer set of errors and design defects — these ideas come to infect most, if not all, of the world of software development. Forty thousand million dollars says that these are the ideas that bring success.

They are the very antithesis of the idea of software engineering. Thirty years ago the organisers of the famous NATO conferences chose the title "Software Engineering" as a conscious provocation. They expressed an aspiration to which we all pay lip service today. In 1968 it was novel. Four years later Edsger Dijkstra expressed a closely related vision in his 1972 Turing Award lecture:

> "The vision is that, well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining our programming ability at the expense of only a few percent in man-years of what they cost us now, and that besides that, these systems will be virtually free of bugs."

But the aspiration and vision are still just that. Software systems are still infested with bugs, and software development is still as far from becoming an established branch of engineering as it was in 1968. Recently David Parnas explained in CACM why he thinks that Software Engineering is an "unconsummated marriage". He focuses his complaint on the failure to distinguish computer science from software development. Engineering is using science and mathematics to build products for other people to use. Software engineers should not themselves be computer scientists, but must command a core body of knowledge derived from computer science, wedding this knowledge to the concepts and disciplines taught to other engineers. The vital need is to identify the core body of knowledge. Then software engineering can be recognised as a new branch of professional

engineering, with the attendant regulation and licensing to guarantee the minimum standards of professional competence that our customers are surely entitled to expect.

The argument sounds very convincing. Like automobile and chemical and aeronautical engineers, software developers build machines. A program is a description of a machine: a general-purpose computer accepts the description and, by executing it, becomes the machine described. As software engineers we are not concerned to build the computer hardware; but we are full partners with the hardware engineers in responsibility for the resulting machine and for its fitness for its stated purpose. Some famous software failures, such as the control software for the Therac-25 radiotherapy equipment that killed several patients, can be traced directly to the software developer's ignorance of standard results of computer science that would certainly be in Parnas' recommended core body of knowledge.

Failures in safety-critical systems for medical equipment or nuclear power stations or automobiles or aeroplanes attract plenty of attention. Major failures also occur in tax systems, information systems, telephone switching, payroll programs, banking systems, electoral systems — and of course in systems of the kind that wasted Yourdon's expensive time, in the office world of word-processing and spreadsheets and email systems. What, then, are we waiting for? The computer scientists have a rich body of knowledge; general concepts and principles of engineering have been developed over nearly three centuries. Why has the marriage not yet been consummated? Why is Yourdon's time-wasting experience still typical? Why has Software Engineering not yet taken its rightful place among the established branches of engineering?

One obstacle, undoubtedly, is the entrenched interest of dominant software companies. Their domination, and its continuance, depend on an ever-accelerating provision of new products and new — and preferably incompatible — versions of old products. The purchase of 'upgrades' — often a euphemism for what in a more innocent age were called 'bug-fixing releases' — is a central component of their income. Car manufacturers can rely on physical degradation of their cars to encourage owners to buy new ones every three or five or seven years. Software manufacturers are more ambitious. They aim at a replacement cycle of one or two years, and must rely partly on the errors and defects of today's version to persuade many of its owners — the 'early adopters' — to buy tomorrow's. They rely also on the stranglehold of the de facto standard. Tomorrow's version will use an internal format that is incompatible with today's: to read the files that come to you in your email you must certainly buy tomorrow's version — rather as if last year's telephones were unable to receive calls from this year's. In this way the early adopters drag other users along with them. In such a hectic commercial environment, sound engineering has little payoff. That's convenient, of course: the rush to market leaves too little time and energy for sound engineering.

Another obstacle is the entrenched interest of practising software developers. Most — like me — have no computer science degree. Like the old barber surgeons they are naturally unenthusiastic for a change that might take away their livelihoods. Nor, for the most part, are they convinced that Parnas' core body of knowledge is essential to their work. A major structured methods company once advertised one of its methods seminars like this:

> 'Building workable software systems is not a "science". So-called computer scientists try to convince us that our systems are really directed graphs or n-tuples of normalized forms or finite state automata — their pronouncements are more relevant to Zen than to the no-nonsense business of building useful, maintainable programs and systems. They have no answers to real-life problems like users who change their minds or requirements that are in a constant state of flux.'

The advertisement struck a chord in the breasts of many working developers. It pulled in over 5000 registrations for the seminar (which, incidentally, was billed as a seminar in Software Engineering). These software developers, like all practitioners of a craft, trade or profession, value their practical know-how above the more abstract explicit science of the theoreticians. The name of Software Engineering is attractive: the reality is not.

Another obstacle is presented by the ever-increasing complexity of the computing environment, often greatly aggravated by the need to incorporate legacy systems from several generations. Many companies today are struggling with a multitude of programming and scripting languages, with database, communications and operating systems, with bought-in application packages, terminal emulations, and web interfaces. The crucial knowledge needed to keep this alphabet soup of software in working order is knowledge of the messy and poorly documented details of the particular ingredients, not knowledge of the core computer science curriculum. The Year 2000 problem illustrates the point perfectly: for many companies the crucial skills today are knowledge of COBOL and CICS, and even IMS. Concurrency, data structures, invariants and automaton theory come a distant second.

Even if there were a general determination to improve the technical practice of software development it is not clear where the improvement should be directed. The practice of software development forms a continuum from the most trivial amateur work to the most demanding professional project, from the writing of the simplest Word or Excel macro to the design and construction of an avionics in-flight system. Where in this continuum does the putative discipline of software engineering become necessary? The choice is not simple. Nor is it static: today's trivial Excel macro becomes tomorrow's library of coordinated macros, which in turn will later become a component in a large interconnected system on which the business eventually comes to depend. But surely we can not sensibly demand that every writer of an Excel macro be a qualified software engineer? This continuum is scarcely present in established engineering disciplines. A makeshift hut can not be elaborated into a skyscraper; there is no do-it-yourself kit for building suspension bridges; there are no home-brew nuclear power stations.

The difficulty of the continuum of software development is merely a special case of a more fundamental obstacle to the establishment of a discipline of Software Engineering. Software products and projects are so various that no single intelligible discipline could possibly embrace them all. What is common to an operating system, the software for a telecommunications switch, a word-processing package, a trading floor system, a car braking system, a compiler and an air traffic control system? Very little indeed. They differ radically in every important respect: the problems they solve; their operational environments; their economic and technical goals; their underlying technologies; their critical risk factors; their product lifecycles; the technical challenges they pose; the design techniques they demand. The common core body of knowledge lies deep below their surfaces. In effect, they have no more in common than a power generating station has with an aeroplane.

In his characterisation of software engineering Parnas has omitted an essential ingredient. There is indeed a core body of knowledge derived from computer science; and there are general concepts and principles of engineering; but sandwiched between them is a much thicker layer of knowledge specialised to particular product groups. This is not knowledge of the detail of a particular application system. It is knowledge analogous to the knowledge that a bridge engineer has of different bridge structures, or a naval engineer has of different ship architectures. Bridge engineers don't design ships, because they don't have the appropriate specialised knowledge. Too much is known about bridges and ships for one person to be able to design both. In the early nineteenth century it was possible for a great engineer to design both bridges and ships. Now it is no longer possible; increasing knowledge has separated them into two distinct specialised branches of engineering.

This has already happened in certain fields of software development. Compiler writing and operating system development are two obvious specialisations. Enough is known about each of them for it to have largely split off from the main body of software development: they have their own body of practical knowledge, their own core selection from the body of knowledge derived from computer science, their own university courses, their own literature and their own conferences. The same is true of database systems and switching systems. And the emerging work on frameworks and Domain-Specific Software Architectures will encourage the growth of further specialisations.

So even if there is not yet software engineering, there is at least compiler engineering and operating system and database system engineering — or there soon will be. But there will never be software engineering. As these specialisations flourish they leave software engineering behind. It becomes only the residue of problems that we do not know how to solve, of products that we do not know how to design and build — not the most promising definition for a new engineering discipline. Every successful area of software engineering immediately leaves home to set up its own independent specialisation, depriving the forlorn parent of the credit for its success. A professor of software engineering must, by definition, be a professor of unsolved problems. This is why we have persisted for so long in the illusion that there can be universal constructive methods of software development — methods equally useful for designing a word processor and an avionics system. Promoters of software development methods don't offer compiler designs as worked examples: wisely, they stick to problems for which there is no well-understood solution.

Gloomy as it is for the prospect of a general discipline of software engineering, this conclusion is hardly surprising. The counterpart of software engineering in the established engineering branches would not be chemical or electrical or aeronautical engineering. It would be a mythical discipline that we might call "Physical Engineering". There is no such discipline.

The way forward must therefore lie through increasing specialisation. Software developers must specialise by requirement and product, just as established engineers do. They must refine their products over several generations, applying the relevant parts of Parnas' core body of knowledge in their specialised area, and demanding additions to that body of knowledge as particular problems come into better focus. They must recognise the importance of other disciplines for their projects, but not allow those other disciplines to dilute or

dissipate their concentration on their own particular branch of engineering. A software development project may involve law or ethnography or sociology. When it does, the software developers must join a multi-disciplinary group with lawyers, ethnographers or sociologists. But they must not think that law and ethnography and sociology are parts of their engineering discipline. If you want to understand anything, you must not try to understand everything. Someone must mind the shop.

This specialisation is vitally necessary; but it will not succeed unless these specialised engineering disciplines are properly rewarded. And they will not be rewarded while we continue to accept needlessly complicated and buggy software as a fact of life to be patiently tolerated. Perhaps in Yourdon's sad tale of problems with his Microsoft and Apple system the software producers were not the only people to blame. In the end, we will get the software we deserve.

[ENDS: c 2710 words; for IEEE Software January 1998; 09/10/97]