Specialising in Software Engineering

At the end of the millennium, software engineering will be 50 years old. Although its birth is often associated with the 1968 and 1969 NATO conferences, it is more properly marked by Wheeler's invention of the subroutine in 1950 ("The Preparation of Programs for an Electronic Digital Computer, with Special Reference to the EDSAC and the use of a Library of Subroutines"; M V Wilkes, D J Wheeler and S Gill; Addison-Wesley, 1951). The invention made possible an explicit consideration of program design: if procedures are nodes and calls are edges, the resulting graph can be considered a representation of the design. And it gave procedure call the pre-eminent place it still retains today in object-oriented languages.

It's significant that this seed invention of our field was a low-level and almost universally applicable device — in a 1983 workshop, Mary Shaw entitled her contribution "Procedure Calls are the Assembly Language of Software Interconnection". With this as the first step, it's not surprising that software engineering's first 50 years have been largely devoted to a search for low-level universal solutions to our problems: general-purpose languages and general-purpose methods for programming, design and specification. With few exceptions, every language and every method claims universal applicability. Advocates of structured, functional and object-oriented techniques exclude no class of application area or system; UML — Rational's attempt to become the Microsoft of object-orientation — carries no disclaimer that it is intended for use only on this or that kind of problem; Java's proponents see it as a universal programming language. There is some appeal in this. The computer itself is a general-purpose machine. Should its software not be developed by similarly general-purpose tools and techniques?

No, it should not. General-purpose tools and techniques are rarely the most effective for any specific purpose. That is why the established branches of engineering are all highly specialised. Bridges are not designed by generalist engineers who also work on automobiles and television sets and chemical plants. Automobile engineers don't design tunnels or electrical power networks. The field of the generalist engineer is — almost by definition — limited to highly speculative development of new products for which there are no reliable established precedents. In the first half of the nineteenth century a great engineer like Isambard Kingdom Brunel could work on designing artillery weapons and prefabricated hospitals, suspension, arch and girder bridges; on laying out the line for a new railway, and building railway tunnels, embankments and cuttings; and on designing huge steam ships to cross the Atlantic. Today that versatility would be impossible. Any one of Brunel's projects would demand mastery of its own substantial specialised branch of engineering — of a rich body of slowly accumulated knowledge that can scarcely be mastered in one lifetime.

Specialisation pays off in many ways. One is that as knowledge accumulates over a long period the most effective designs are identified and become largely standardised. Specialists, seeking to capture the market with the best possible product, examine their competitors' products with a keen eye, hoping to identify and copy new features and design elements that work well. A hundred and twenty years ago car designers were unsure whether the driver should sit at the front or the back, should sit in the middle or to one side, and should steer with a tiller or a wheel. Those questions, along with a thousand others, have long been settled, and today no car designer wastes time reconsidering them.

A second payoff is that sound theoretical principles can be embodied in standard designs and standard development practices. The specialised engineer works out few decisions by calculation from first principles; most decisions are made by recognising a standard situation and consulting standard design tables or rules of thumb or the results of standardised calculation procedures. It is true that established engineers are familiar with basic theory in their field in a way that too many software engineers are not. A software engineer who knows nothing of formal languages, data structures and process algebras is like an electrical engineer who does not know Ohm's law and Kirchoff's law; and there are too many such software engineers. Software engineers must be educated in the core curriculum that David Parnas wrote of ("Software Engineering: An Unconsummated Marriage"; David Lorge Parnas; Comm ACM 40,9, page 128; September 1997). Stronger education in theory is necessary. But it is not sufficient. To be applied regularly and easily, theory must be encapsulated in commonly understood and practised design procedures, and that can happen only in the context of a specialised field.

A third payoff is that specialised engineers have fairly reliable ideas of what they can't do and why they can't do it. No structural engineer would embark on the construction of a thousand-story skyscraper or a suspension bridge with a fifty-mile span. But the equally absurd proposal for SDI — the 'star wars project' — was given serious consideration by many software engineers before David Parnas and a few others convinced the world of its folly.

A fourth payoff is that specialised engineers have a stronger sense of personal professional responsibility — for selfish reasons if for no others. In a career of successive projects, each one in the same technical domain as its predecessors and conducted within the same professional social group, a significant personal failure can't be relegated to the limbo of the past. For the specialist, a really serious failure means the end of a career. To see what professional responsibility means, software engineers should read how William J LeMessurier, the structural engineer responsible for the Citicorp Tower in New York City, dealt with a serious flaw in his design ("The Fifty-Nine Story Crisis"; Joe Morgenstern; The New Yorker, May 29, 1995, pages 45-53).

Many eminent computer scientists rightly berate practising software engineers for paying too little attention to the theoretical foundations of our work. But they rarely notice how varied that work is. Building the software for a telephone switch has very little in common with building a compiler or a car braking system or email client. What they have in common lies at least as far below the level of working practice as what is common to designing an aeroplane and designing a tunnel. Yet the basic stance of software engineers — even the name itself of our discipline — has militated against specialisation for 50 years. We have aspired to take our place as just one more among the established branches of engineering, failing to recognise that we are really a confederation of loosely related branches.

Our achievements over our history of fifty years give us little reason for complacency about this approach. Peter Neuman's Risks Forum reminds us continually of a failure rate that would not be tolerated in any established branch of engineering. Of course established engineers have their failures, too. Often they are notorious and spectacular, like the Tacoma Narrows bridge that tore itself to pieces in 1940 in a moderately high wind, or the space-frame roof of the Hartford Coliseum that collapsed under a fall of snow in 1978, or the Titanic, lost in 1912. But established engineers are compelled to submit to public enquiry into each major failure, and — more importantly — they have social and professional mechanisms in place to digest the lesson and install it in a communal body of specialised knowledge where it can prevent repetition of the mistake. We shake our heads over the Therac-25 and the Ariane 5 disasters. But what software engineering lessons, exactly, did we learn? And in what repository of knowledge have we filed them? And what mechanisms guarantee that they will not be repeated? Learning from mistakes is the single most powerful force for improvement in engineering. But learning is effective only if engineering is specialised and highly structured.

I believe that the next 50 years will see an irresistible movement towards specialisation in software engineering. There are already a few established specialisations. Compilers are perhaps the most notable case. The first company to offer software development on a commercial basis set up shop writing compilers for hardware manufacturers in the 1960s. Compiler writing is now a discipline in its own right, with its own terminology, its own standard designs, and its own university courses. This is why the quality of today's compilers is so high.

Specialisations will also grow up around emerging standard designs. Object-oriented patterns and frameworks will have their most important impact here, and so too will the ideas of Domain-Specific Software Architectures. Some specialisations will form because only specialists can survive in a hostile environment. The frantic rush to produce compelling Web sites has led to a very complex environment of new HTML variants, scripting languages, database connections and much else. It is very hard for anyone but a dedicated specialist to put up a good site that works reliably.

Of course specialisation is a necessary, not a sufficient, condition for quality. There are several classes of software product where the field is dominated by a few, presumably specialised, producers, yet quality is still very poor. But in the new millennium the trend to specialisation will surely accelerate and intensify. In the process it will give us at least the probability of software of a better quality than we have learned to live with in the past 50 years. And I must say: it's long overdue.

[Michael Jackson; 1560 words including references; 22/07/99; SpecSE4.doc; IEEE Software 16,6 pages 119-121, November/December 1999]