Research report 117

SOFTWARE ENGINEERING: THEORY, EXPERIMENT, PRACTICE OR PERFORMANCE*

Mathai Joseph

(RR117)

*from an Inaugural Lecture given at the University of Warwick, 15 February 1988.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

February 1988

Software Engineering: Theory, Experiment, Practice or Performance*

Mathai Joseph

For G.C. Bannerjee (29 Sept. 1905 - 18 Jan. 1988)

Forty years ago, the very first programmers were wondering how to make the best use of stored program computers. By twenty years ago, that handful of programmers had grown into a software industry so large that it faced a major crisis. Today, another twenty years on, we are beginning to see how that crisis helped to move the industry from an organised craft into a profession. We still have a software crisis, though it is no longer the old one and no longer even considered a crisis. It seems likely that we will always have a crisis, a situation where our propensity for producing programs outstrips our understanding of new programming concepts. What the last twenty years have accomplished in a quiet way is the subject of this talk and I would like to show you how, and how far, theory, experiment and practice have worked together, often unintentionally, to achieve performance in software engineering.

Some years ago there was a widespread belief that there was a simple, linear relation between the size of a task, the size of a program that would perform the task and the number of people needed to write the program. Any deviation from this linearity was due to inexperience, or to hardware problems, and there was certainly no contribution to be found from a more theoretical study of programs. Hardware problems were due to sloppy design or the use of poor manufacturing techniques and must therefore have a technological solution. Basically, writing a program was easy and getting it to run fast enough was only a question of having a fast enough machine, and that was a problem for computer designers and device technologists. Trying to make use of theory did not help, it just got in the way.

There were no sudden events to bring an end to this shining optimism. But there was a growing if reluctant admission that the evidence of the early part of the 1960's did not promise well for the future. Higher machine speeds and larger storage capacities had duly appeared as demanded and had immediately been swallowed up without trace by large software project teams; and often, the software that did emerge was months or years

*from an Inaugural Lecture given at the University of Warwick, 15 February 1988.

behind schedule and then singularly poor in performance. Some immensely large software projects produced immeasurably small results, at great cost, and many such projects were abandoned altogether or delayed indefinitely. It was not unusual to see teams of 100-200 programmers working on a project for several years and it was not unusual to find costs running even further ahead of estimates than ambitions were of achievements.

Smaller projects seemed to see some success: if they could be accused of being unadventurous, as indeed they often were, they could also claim to meet their aims. Such software was crafted by individuals or by small teams who had experience with some techniques and were careful not to attempt to use them beyond their limitations. They did not necessarily produce products that were easy to use - I can remember having to load three large reels of paper tape in order to compile and run a small Algol-60 program and then having to rewind the tapes which were lying in a tangled mess on the floor. But they tended to produce predictable results. Unfortunately, this was not the kind of evidence that impressed the large-scale software teams because they could not see how these techniques could be scaled up to solve the problems that they faced. So they had little alternative to doing a postmortem on their past efforts and trying again.

> *" ... the software scene is littered with the whitening bones of projects whose designers, with the ever present triumph of optimism over experience, thought that they could do better this time."*
> *(Editorial, Software Practice & Experience, Vol 1, 1971)*

Quite a lot of effort was, and still is, spent on developing statistical models for relating the number of errors remaining in a program with its size, the number of errors reported and those corrected, and so on. But this is basically a rather depressing view of things, one of desperation when all other methods fail, reminding us only that the number of errors is large enough to be a subject of statistical study. A buyer is unlikely to be pleased to know that the probability of finding errors in a piece of software has been reduced from 27% to 23%; the manager of the software team, lacking any other means of guessing how much effort will be needed for software maintenance, may find solace in these figures if there is reason to believe that the statistical model is valid. Unfortunately, building a valid model of this kind needs a large amount of qualitiative information specific to the kind of project and to the project team, so a good model is difficult to produce until enough of this information has been collected, by which time is is usually too late.

I first became interested in software engineering soon after the widespread publicity about a software crisis. At that time I thought, like many others, that a great deal of the crisis could have been avoided quite easily if people had written software using the right kind of language, for example a high level language. This was a forgiveable piece of enthusiasm when you consider how large programs were otherwise written.

But the term 'high level' is extremely misleading as the height of such a language depends very much on your point of view. For example, none of the high level languages then in use offered anything to solve the really difficult problems that systems programmers encountered. Very few of these languages were translated efficiently enough to compete with low level notations and the most suitable ones were not readily available. On the other hand, calling a language 'high level' gave it an impressive description as well as providing a slogan so the term has stayed in use. (It must have been some kind of retribution for my earlier *naivete* that the first course I taught when I came here was called High Level Programming).

It was easier to hold on to the belief that high level languages would ameliorate the tedious part of software engineering, leaving us to study the fascinating programming problems being discovered, than to justify such a belief with experimental evidence. One of the first large pieces of software that I designed in the early 1970's was a small real-time operating system written in assembly language (the typical low level from which the height of other languages is judged) and an incarnation of this, when I last heard about it a few years ago, was still in use. Years later I ran a comparatively large project where we designed a far more complex operating system using what we felt was the best possible high level language for writing such programs and I have to confess that none of it is in use today. However, there was a difference in objective in the two projects: the assembly-coded real-time operating system was needed for a project with tight delivery schedules, while the multiprocessor system was built as an experiment. I shall return to this point later ibecause it illustrates some of the distinctions I wish to make between experiment and practice.

In the crisis years, experimental evidence to justify the use of high level languages, or any other panacea, was difficult to come by. There was little sustenance from theory which had largely ignored the unstructured world of practical programming. 'Theory', in those days, meant the theory of computation and it was concerned with meta-theories about programs, rather than programs themselves. It was mathematically elegant but it was relatively unconnected with issues of real programs. Formal analysis of programming problems had just begun to arouse some interest and some solutions were beginning to appear.

At that time, I had started work in an institute devoted largely to basic research in the well established disciplines of physics and mathematics. I referred to myself as a computer scientist but it was not always easy to justify the term 'science' when talking about computing, except by pointing at the theory of computation, and this was some technical distance away from the level of my work. So it was reassuring to hear from a physicist friend that, whatever its failings, computer science had quite a lot in common with many other sciences. For example, for a long time, a discipline as distinguished as theoretical astrophysics had neither predicted anything that had not previously been observed nor explained all that *had* been observed: its first major success in prediction came with the discovery of pulsars in 1968. This reassurance was timely, because the immediate prognosis for the software engineer was rather bleak:

> *" ... one of the difficulties about computing science at the moment is that it can't demonstrate any of the things it has in mind ."*
> *(C. Strachey, 1969).*

Strachey's words were carefully chosen and laid as much emphasis on the lack of demonstrability as on the fact that many of the hopes of the theoreticians were still to be effectively articulated. And there were indeed clear divisions in computing, between hardware (which had to do with devices and circuits and computer architecture), software (which was everything to do with real programs) and theory (which seemed to have limited contributions to make to either of these areas).

But there was at least one area of computer science where there had been a fairly close and mutually stimulating relationship between theory, experiment and practice, though little distinction was made in those days between experiment and practice. From the work of Chomsky in the 1950's and Rabin and Scott's study of automata a few years later, enough was known by the 1960's about formal languages and parsing techniques for it to be put to use in designing compilers for programming languages and even to experiment with programs that would *produce* compilers - the so-called compiler-compilers. The need for practical improvements, to make implementation easier and to improve performance, led to better definitions of grammars and programming languages.

Now, however successful an experiment may be, it is usually not engineered in the same way as a final product. But this distinction was often overlooked and many compilers started life as experiments and were then delivered to customers with their experimental loose ends carefully hidden behind professional-looking documentation. So, though the construction of compilers was founded on sound and applicable theory, the compilers that reached eager customers sufferred from many of the defects that were found in other software.

In much the same way that the study of formal languages and automata profoundly influenced the design of programming languages and compilers, a close relationship developed during the 1970's between logic and programming, and this was built upon a growing theory of the mathematical semantics of programs. The origins of this connection go at least as far back as McCarthy, in 1962, probably also to Turing's work in 1949, and it found new relevance through the work of Naur, Floyd, Hoare and Dijkstra. There were important practical roots for this work and it gave rise to a number of new techniques for programming; not surprisingly, however, these found few practical users. This was despite the fact that the ideas *were* tested experimentally, in software large enough to manifest realistic problems and small enough to demonstrate the advantages of the new techniques.

Many of you will of course be familiar with the facts behind this potted history and some have actually played a role in causing them to happen. So the reason for presenting them here, and in this form, is to return to the title of my talk and to several questions that it provokes. What are the roles of theory, experiment and practice in software engineering when practical software has to demonstrate its usefulness by its performance ?

It is important for a theory to have a sound logical basis, and for an experiment to observe measurable properties of phenomena. It is not enough to say

> *"Every experiment is a success - it either proves or disproves a theory"*
> *(SDI spokesman, quoted in* The Times, *January 1988).*

because that would cast quite a lot of doubt both on the kind of hypothesis being tested and the kind of experiment being performed. The purpose of a scientific experiment has been examined quite closely, and for a long time: for example, these words were written in Charles Babbage's lifetime:

"Those who have condemned the use of hypotheses and preconceived ideas in the experimental method have made the mistake of confusing the contriving of an experiment with the verification of its result"
(C. Bernard, 1865).

So perhaps we should return to a more traditional view of an experiment before seeing what role it plays in software engineering:

" ... an experiment is ... a reproducible observation ... recorded under prescribed circumstances ... (which) bridges the gap between the empirical and the theoretical ... "
(J. Ziman, 1968).

The difficulties that are encountered in large-scale software often occur in situations that are not reproducible. For example, we have a digital telephone exchange at this University, and it has within it a computer and some software. Now, there are times when a number dialled on the telephone in my office produces no response at all - no engaged tone, no ringing tone, no dial tone, no noise at all other than my heavy breathing. Perhaps this problem only occurs when a number of other things happen simultaneously. But how can this hypothesis be tested ? Can we try making ten people pick their telephones up at the same instant, assuming that there is a satisfactory way of ensuring that they do this simultaneously ? Can we also arrange that they repeat this for every possible combination of incoming calls on the external lines to the exchange ?

An important role for experiments in software engineering is to re-create, in reproducible circumstances, problems that can arise in large programs in the same way that a good empiricist is able to contrive the right experiment - not just rubbing any two things together but choosing two pieces of wood with enough friction between them so that rubbing starts them smouldering. Much of the successful experimental work of the early 1970's was done in this style and led to the development of better analytical techniques and to better solutions. The experimentors abstracted problems from the reality of large programs where they could not readily be observed, re-created them in small-scale tests and devised ways of solving them. These experiments served as paradigms that could be analytically studied to provide solutions for use in practice. But the role of this kind of experiment was not well accepted and the experimentors found it difficult to convince practitioners that solutions demonstrated in small experimental systems could be applied in large systems.

The other classical purpose of an experiment is to verify a hypothesis drawn from some theory. This is commonly done in other sciences but the way it can be done in software engineering is very different. Much of the theory consists of proofs using some abstract mathematical structures and the relevance of the theory is determined by the extent of similarity between these structures and what is seen in the real world. So the second role for experiment in software engineering is to observe under experimental conditions how well theory can be applied to practical problems. But it is important not to build an experimental system so large and complex that it cannot be used for making accurate observations.

Experiments must make use of some experimental method, and one reason why the roles of an experiment in software engineering have not been clearly identified is that there is a genuine difficulty in talking about its experimental methods. Take a simple example, that of sorting or arranging a set of numbers in ascending order. A study of *sorting methods* may use measures for the time and space taken by each method under a particular set of circumstances. A study of a new programming notation may use a sorting method to illustrate its new features. And a new program development technique may be tested to see how effectively it can be used to produce a sorting program. The difficulty in using the traditional definitions of 'experiment' and 'experimental method' when talking about computer science is that the term 'method' tends to be overloaded. In software engineering, a method for constructing programs is as much a subject of study as a method used in a program to solve a particular problem; one is part of what is called programming methodology, while the other is part of the study of program techniques. A test for a programming method might be to examine what kinds of programs it is best suited for, while a test of a program technique might be to see if it simplifies or speeds up a program to solve a specific problem.

This may seem unnecessarily confusing to those of you who are not computer scientists, and the distinction I am making has not always been accepted even in computing. For example, an experiment was often included as one of the steps of the design of large software. What sort of experiment is this that fits into a tight development schedule ? If the time taken for an experiment can be precisely fixed, should we conclude that its results were not expected to change anything ? In fact, the term 'experiment' was used here to denote a test of whether enough factors have been taken into account, or perhaps a simulation to estimate performance timings but not a means of verifying a hypothesis. It may be no less valuable for that, but it is important to see the distinction between this and other kinds of experiment.

I had referred earlier to the success that small software projects had achieved and attributed this to the care exercised by people well versed in their craft. Some landmark projects were done in this way to focus attention on particular programming problems and others to examine how a *method* for constructing software can improve its reliability and its quality. This tests whether a set of well-defined steps, if rigorously followed, will result in producing a better software product. The steps may be performed by people or by computers, or by people with the support of computers, and they may be informally specified or they may require the application of mathematical principles. The use of such methods is not new but the methods we talk about today are rather different from earlier methods as they are based on mathematically defined rules. This has led to them being sanctified as 'formal' methods.

Formal methods have found a lot of support from many quarters, including, most important, both industry and major funding agencies.

> *" ... the highlight is the progress in bringing formal method from the academic world into use in industry and commerce ..."*
> *(B.W. Oakley, Alvey Annual Report, 1987)*

Interest in the use of formal methods is an outcome of the difficulties that the software industry has been facing. People working on development teams have surmised how these methods might have helped them avoid the problems they faced in their earlier work. Managers of software teams have been glad to adopt formal methods to help estimate the progress in a project and to reduce the costly maintenance of the finished product. Unfortunately, there is very little experimental evidence that the use of formal methods will achieve all that is expected, and there could be as much shortfall between expectation and performance as there was when the use of high level languages was advocated.

It is because there is so much variability in software projects that sound methods must be used for development. Formal methods require program development to be supported by rigorous analysis and this must lead to better software. But it would be as much a mistake to assume that all the necessary formal methods exist as to assume that all the problems have as yet even been discovered. When high level languages became more widely used, the cause of one class of problem was removed and other problems came into focus. There is no doubt that this will happen again when the use of formal methods is widespread.

There is a strong case for a middle ground, scaled somewhere between the high-pressure production work in industry and the toy environments often found in universities, where a new generation of experimental work in software engineering can be done. Once again, this is not new: computer science departments at many universities have been able to convince their masters that they need a realistically large computing environment in which to try out new ideas. Having won that battle, however, there is the severe danger of settling down to software development on an industrial scale and of measuring progress in terms of the amount of software produced. Very little of such software will usually survive after its creators have acquired their Ph.D.'s, because its hand crafted qualities will make it unresponsive to any other user. Little is gained from such a project, if its experimental evidence is lost along with its software.

If it is to be effective, this experimental framework must have common ground between the research environment and the practical application so that problems of large-scale software can be studied in realistic but smaller and controlled environments. One purpose of such experiments is thus to study both a problem and the situation in which appears. Such a problem can be solved either by the development of a new technique or even by changing the circumstances so that the problem is avoided altogether. The problems of practice call for such experimentation and it is a comfort to know that there are theoreticians lying in wait for such problems.

> *"It sems to me that we have desperate need for people who can look at a program and single out features of interest to a mathematician, and conversely someone who will take mathematical results and apply them."*
> *(R. Parikh, 1981)*

This framework must allow scientific experiments to be done, observations recorded and conclusions drawn in much the same way as for any other science. Not all the problems will appear at once, even under scrutiny, and re-examination of old observations may lead to new conclusions. With these failings too, software engineering would be in good company: I seem to remember that Anderson and Blackett discovered the positron by re-interpreting the evidence of old cloud chamber photographs. But here physics is on stronger ground, because Dirac had predicted the existence of positrons some years before that.

It is easy to forget that computer science has only had about 50 years in which to grow, and this has been a period when its equipment and its measures have changed dramatically. But there seems to be a levelling off in the kind of equipment an individual experimentalist might need, at least over a five year time scale, and new and powerful tools are available for use on such equipment. If the different roles of experiments in software engineering are kept firmly in sight, there is real hope that theory and experiment will interact with practice so that performance is the outcome of prediction.

References

C. Bernard, *Introduction a l'Etude de la Medecine Experimentale,* Paris, 1865.

Editorial, *Software Practice and Experience*, Vol. **1**, 1971.

B.W. Oakley, Alvey Annual Report, 1987.

R. Parikh, Comments from a discussion in *Logics of Programs*, D. Kozen (Ed.), LNCS 131, Springer-Verlag, 1981.

C. Strachey, Comments from a discussion in *Software Engineering Techniques,* J.N. Buxton, B. Randell (Eds), Nato Science Committee, 1970.

J. Ziman, *Public Knowledge: The Social Dimension of Science,* Cambridge University Press, 1968.

Acknowledgements
I am most grateful to Mike and Linda Paterson, and to Anita Joseph, for commenting so helpfully on an earlier draft of this material.