Empirical Software Engineering, 6, 195–200, 2001.

Viewpoint Article: Conducting and Presenting Empirical Software Engineering

NORMAN FENTON norman@agena.co.uk
Agena Ltd, 11 Main Street, Caldecote, Cambridge, CB3 7NU, UK and Faculty of Informatics and Mathematical Sciences, Department of Computer Science, Queen Mary (University of London), London E1 4NS

Despite the heroic efforts of a small group of people, like those involved with this journal, a truly "empirical" basis for software engineering remains a distant dream. In the current academic year I have been teaching software engineering (a double unit module) at Queen Mary (University of London) where I have been a Professor part-time since March 2000. Although I had been teaching courses on software metrics and quality assurance regularly in recent years, this was the first time I had taught a "standard" software engineering module since 1992. As the leader on a module with over 100 students, publishers have been keen to send me all their latest offerings. As a result, in the last few months I have received more than a dozen new or revised dedicated software engineering text books, and around two dozen "software engineering with Java" or "object oriented software engineering" type books.

The good news is that, compared to the text books that were available when I last taught software engineering the new bunch are, almost without exception, a massive improvement. They provide students with techniques and methods that they can actually apply to their own programs and group projects. This compares favourably with the previous generation that simply documented a set of research ideas dreamed up in academia and never applied successfully in practice. This makes it easier and more satisfying to teach, and more rewarding for the students to learn (primarily because they can learn from doing which they could not in the past). Moreover, in this respect, the impression is that software engineering has come closer to being true "engineering". However, if we accept that an empirical basis is one of the other components that mark out a true engineering discipline, then the latest round of books confirm that any progress we may have made in this area has had an almost negligible impact.

The primary motivation for my own original interest in empirical software engineering was the desire to see a more rational basis for decision-making. For example, I was concerned that methods were being adopted on the basis of who, among the methods' proponents, shouted the loudest. In many cases methods were being pushed, not only without adequate tool support, but without any quantitative justification of their effectiveness. I am not talking about the need for

full-scale experiments here (something I have always been sceptical of in software engineering anyway) but I would have expected at least a single case study on a non-trivial system with fully documented results. In addition to decisions about which methods to use and how, I was concerned about the lack of any empirical basis for decisions affecting all aspects of the software life-cycle. For example: if we use method X in its fullest form what measurable benefits are likely and at what cost; which parts of method X could we leave out if our reliability requirements drop by 50%; which testing strategies are most cost-effective in which contexts and how much effort should be allocated; how many defects is it "good" or "bad" to find in different contexts; how "big" or "small" does a project have to be for it to have to include or exclude various methods and QA techniques. In brief what was needed was a set of empirically based guidelines for decision-making about all aspects of software engineering. Sadly, this kind of empirical software engineering is still completely missing from the new books. While they provide more focused practical methods to use, the belief in the methods is still based on blind faith, with no understanding of what true benefits lie ahead or what trade-offs can be made at which phases.

So what do we need to do to ensure that the software engineering textbooks of 2010 have the missing empirical jigsaw pieces; the material that enables us to determine (like in other engineering disciplines) not only how to use some particular technique, but what to use, when to use it, and why? One answer is, as I hinted at above, to put behind us the mindset of believing that only formal experiments can provide valid empirical results. Generally, such experiments are prohibitively expensive and technically infeasible to set-up properly. Even in those rare circumstances where it has been possible to run a formal experiment to test some highly specific hypothesis, the results are in any case not accepted by those who see them as a threat to their own ideas. For example, by using a large number of Masters level students Finney (1998) was able to test the hypothesis that trainee programmers subjected to an intensive training course in the formal notation Z could understand simple Z specifications. The experiment rejected the hypothesis showing that less than 30% of students could understand the simplest of Z specifications well enough to trust them to program a simple requirement stated in Z. Yet, even this experiment was criticised on the basis that Masters students were not at all representative of trainee programmers in industry. Although these students undertook a full semester of Z training, far more than would normally be expected of a trainee in industry, the critics who rejected the "validity" of the experiment still argue that "almost any programmer" can understand Z specifications after a 1 week training course.

So what is the answer? I propose that we can gradually build up an empirical body of knowledge, simply by providing relevant quantitative information about real projects that we are involved with. A model of this approach was provided in Fenton and Ohlsson (2000). Our stated intention there was "to provide a very small contribution to the body of empirical knowledge by describing a number of results from a quantitative study of faults and failures in two releases of a major

commercial system.'' We did not claim that the results presented were in any sense truly novel; on the contrary, we believe that similar analyses have been performed (with similar results) for major systems throughout the world. However, it appears that few organisations publish such results, even in the ''grey literature'' and so there is little if any similar published data. We made no claims about the generalisation of the results, but hoped that in time they could form part of a broader picture.

The key thing to note about the study was that we had very little control over the quantitative data that was available. We were not able to ''define a metrics program'' and we were not able to dictate in any way which data should be collected. The lack of such control is often regarded as a fundamental impediment to carrying out an experiment or case study. In fact, it should be regarded as the norm. I am constantly amazed that *any* commercial organisation would ever agree to any kind of instrumentation of their process that goes beyond what they do anyway as a matter of course. In fact, if they do agree to such actions, I would be sceptical that the relevant projects are in any way representative. What we did was to look at the data that *was* available and retrospectively consider the most general and useful software engineering hypotheses that we could test with the data. Hence, we focused on providing small pieces of evidence that one day (if a reasonable number of similar studies are published) may help us test some of the most basic of software engineering hypotheses. In particular we examined the extent to which the data provided evidence for or against the following hypotheses:

- Hypotheses relating to the Pareto principle of distribution of faults and failures
  - 1a) a small number of modules contain most of the faults discovered during pre-release testing;
  - 1b) if a small number of modules contain most of the faults discovered during pre-release testing then this is simply because those modules constitute most of the code size
  - 2a) a small number of modules contain the faults that cause most failures
  - 2b) if a small number of modules contain most of the operational faults then this is simply because those modules constitute most of the code size.
- Hypotheses relating to the use of early fault data to predict later fault and failure data (at the *module* level):
  - 3) A higher incidence of faults in function testing (FT) implies a higher incidence of faults in system testing
  - 4) A higher incidence of faults in pre-release testing implies higher incidence of failures in operation.

We tested each of these hypotheses from an absolute and normalised fault perspective.

- Hypotheses about metrics for fault prediction
  5) Simple size metrics, such as Lines of Code (LOC) are good predictors of fault and failure-prone modules
  6) Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules.

- Hypotheses relating to benchmarking figures for quality in terms of defect densities
  7) Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system
  8) Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases.

For the particular system studied we provided some evidence for and against some of the above hypotheses. We were careful to note that our study was based on just two releases of a major system and therefore, we made no attempt to generalise the results. However, (and this is the key point) it must surely be possible for many other researchers to provide similar fragments of empirical knowledge so that a richer picture can emerge.

To give a feel for the kind of results we reported, there was some support for Hypotheses 1a and 2a, while 1b and 2b could be rejected. Hypothesis 3 was weakly supported, while curiously hypothesis 4 was strongly rejected. Hypothesis 5 was partly supported, but hypotheses 6 was weakly rejected for the popular complexity metrics. However, certain complexity metrics which could be extracted from early design specifications were shown to be reasonable fault predictors. Hypothesis 7 was partly supported, while 8 could only be tested properly once other organisations publish analogous results.

We encourage the results to be summarised as we did, in Table 1 below.

In summary, I feel that the recent progress that has been made in empirical software engineering has failed to impact mainstream practice. Practitioners still rely on unquantitative methods of selection and analysis for all key decisions in a software project. I feel that the notion of formal experiments places a prohibitively onerous and impractical burden on researchers to provide ''valid'' empirical results. On the contrary, we can build up a valid empirical knowledge base simply by analysing retrospectively projects for which we happen to have data available. For example, if you have worked on (or know of) a project which used UML, then why not help the entire software engineering community by testing simple hypothesis such as ''the effort spent in producing all the diagrams was less than 10% of the total project effort'' or ''less than 20% of the maintenance effort was spent fixing bugs''. By presenting your results in the form I have shown the software engineering text books of 2010 may yet be able to provide the missing empirical jigsaw piece.

Table 1. Support for the hypotheses provided in the case study.

Number	Hypothesis	Case study evidence?
1a	A small number of modules contain most of the faults discovered during pre-release testing	Yes—evidence of 20–60 rule
b	If a small number of modules contain most of the faults discovered during pre-release testing then this is simply because those modules constitute most of the code size	No
2a	A small number of modules contain most of the operational faults	Yes—evidence of 20–80 rule
b	If a small number of modules contain most of the operational faults then this is simply because those modules constitute most of the code size	No—strong evidence of a converse hypothesis
3	Higher incidence of faults in FT implies higher incidence of faults in system testing	Weak support
4	Higher incidence of faults in all pre-release testing implies higher incidence of faults in post-release operation	No—strongly rejected
5a	Smaller modules are less likely to be failure prone than larger ones	No
b	Size metrics (such as LOC) are good predictors of number of pre-release faults in a module	Weak support
c	Size metrics (such as LOC) are good predictors of number of post-release faults in a module	No
d	Size metrics (such as LOC) are good predictors of a modules' (pre-release) fault-density	No
e	Size metrics (such as LOC) are good predictors of a modules' (post-release) fault-density	No
6	Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules	No (for cyclomatic complexity), but some weak support for metrics based on SigFF
7	Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system	Yes
8	Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases	Yes

Norman Fenton is Professor of Computing at Queen Mary (London University) and is also Managing Director of Agena, a company that specialises in risk management for critical systems. Between 1989 and March 2000 he was Professor of Computing Science at the Centre for Software Reliability, City University. Norman is a Chartered Engineer (member of the IEE) and a Chartered Mathematician (Fellow of the IMA). He has been project manager and principal researcher in many major collaborative projects in the areas of: software metrics; formal methods; empirical software engineering; software standards, and safety critical systems. His recent research projects, however, have focused on the use of Bayesian Belief nets (BBNs) and Multi-Criteria Decision Aid for risk assessment. Also, Agena has been building BBN-based decision support systems for a range of major clients.
Norman Fenton Agena Ltd, 11 Main Street, Caldecote, Cambridge, CB3 7NU, UK. Phone: +44 (0)20 7882 7860 or +44 (0)20 8530 5981 or 44 (0) 1223 263880 Fax: +44 (0) 1223 263899 Email: norman@agena.co.uk, www.agena.co.uk Mobile: 07932 030084
Professor of Computer Science, Head of RADAR (Risk Assessment and Decision Analysis Research) Computer Science Department, Faculty of Informatics and Mathematical Sciences Queen Mary (University of London) London E1 4NS. Email: norman@dcs.qmw.ac.uk www.dcs.qmw.ac.uk/research/radar/ www.dcs.qmw.ac.uk/~norman/.