The Limits of Empirical Studies of Software Engineering

Prof. David Lorge Parnas, P.Eng. (Ontario)
Software Quality Research Laboratory
College of Electronics and Informatics
University of Limerick
Republic of Ireland

Abstract

Some advocates of empirical studies of software engineering appear to be claiming that empirical studies alone can tell us how we should do software development. This paper argues that what can be learned from empirical studies, while important, is very limited. Mathematical studies and empirical studies must be seen as mutually supportive ways to increase our knowledge of software development methods. The role of empirical studies should be limited to confirming that what works in theory can actually be used (and useful) in practice.

1. Were Computer Science departments a good idea?

As a graduate student, I was present at the birth of one of the earliest, and by some measures one of the most successful, Computer Science departments. I was surprised to learn that there was opposition to starting the department. The opponents believed that students of the new department, would be cut off from the nurturing roots of the well-established disciplines of engineering, science, and mathematics.

More specifically:

- The mathematicians felt that computer software was mathematical in nature and that students of computer science would lack an understanding of mathematics as a set of rules that was complete in itself, i.e. without any physical interpretation.
- The Engineers felt that the construction of computer systems, whether hardware or software, was an engineering task and that students would not understand how Engineers apply science and mathematics to make sure that their products are fit for their intended use.
- The scientists felt that much of what people would have to learn about computers and their use was empirical in nature and students would not understand how difficult it was to make accurate, substantiated, statements about the world. Further, they argued that the graduates would not understand the difference between fundamental scientific facts and the arbitrary properties of technological products.

I was surprised by these arguments. As a student, I had been enrolled in all three types of programs. I agreed about the importance of each issue raised, but could not see why a new department could not design a program that would convey the necessary understanding of each field. With the perspective that comes with experience (and age) I now see the wisdom of those arguments. What those critics feared has come to pass.

- A great deal of work in “formal methods” makes extensive use of mathematical notation but lacks the elegance and clean structure of classical mathematics. The papers seem to alternate between intuitive reasoning, introducing new symbols, and writing formulae that express the author’s intuition or observations. Computer scientists often invent “half-baked” mathematical concepts when mature mathematical concepts would do the job better. Derivation of surprising new results from a small set of clearly stated axioms on the basis of a previously agreed set of rules of inference is very rare. It is common to find complicated proofs of facts that are easily derived from previously known results. Many formal methods seem suited for comparing one mathematical model to another but have little relation to actual programs in practical languages.

- Computer Science education does not adequately prepare people to be professional software developers. Graduates end up with their knowledge in two compartments, theory and technology. The theory is not used when developing software and the technological knowledge is soon out of date. The discipline of designing software according to a set of “engineering principles” is either not taught at all or restricted to brief mention in one or two courses. In the classical engineering fields, teaching disciplined design pervades the educational program. Graduates often lack the traditional engineering knowledge needed to analyse the effect of their software on its environment [6].
• Computer scientists completely underestimate the difficulty of applying the "scientific method". They confuse anecdotes with data, trials with experiments, and data with results. Moreover, they do not use mathematics and empirical studies to complement and support each other.

This paper will expand on the third of these points, discussing what can and cannot be done with empirical studies in the area of Software Engineering.

2. "The first empirical study of software engineering"

About a decade ago, one of those who was first to stress the need for empirical studies of software development, hearing some scepticism in my remarks, told me that I was the author of one of the first such papers. He was referring to [1], a paper that describes a successful application of the principle that I call information hiding. My problem is that if I were asked to review that paper today, I would be very critical of it. The title, "Some Conclusions from an Experiment in Software Engineering Techniques", is misleading. It was not an experiment in any scientific sense; it was merely a trial. The main conclusion was actually a meta mathematical theorem with a simple proof.

2.1. The "experiment"

My paper described a demonstration that one could build sets of interchangeable modules. At that time, the idea of interchangeable software components was a dream. A very simple programming problem was decomposed into five modules. Each module had a formal specification. Four implementations of each module (each using a different method) were built. The 4 implementations were intended to be interchangeable; if they were, it would be possible to construct a family of 4^5 different systems. We selected 25 combinations randomly and got them all to run correctly on simple data. Even today, "almost interchangeable" components are common but one often finds small (annoying) differences between them. I have no doubt that our success was a real achievement.

The "experiment" itself however, was of very limited value. It showed that a group of 20 very bright students, with the support of a dedicated graduate teaching assistant under the supervision of a young professor (whose career turned out to hinge on their success) were able to build sets of interchangeable modules. By itself, the trial did not show anything. It would not be viewed as an experiment by any true scientist because there were too many variables that affected the outcome; they had not been identified let alone controlled. It was simply a trial. Nothing, beyond the fact that one trial could succeed, had been proven.

2.2. The information hiding theorem and its proof

I designed the system described in [1] in the hope that the experience of building it would teach the students a better way to design software. I wanted to show them there were benefits to thinking about a system in terms of information to be hidden rather than by starting with a flowchart or algorithm. I based the design on the following trivial "theorem". The story of how the importance of the theorem became clear to me can be found in [2].

Theorem:

Given two collections of programs (modules), A and B, and specifications for each module, if module A can be proven to meet its specification without using any knowledge about module B other than B’s specification, then, if module B is changed in such a way that it is still in compliance with its specification, module A will still meet its specification.

Proof:

If the only information about B used in the proof of module A’s correctness remains valid, the proof of A’s correctness can be reused without change.

Terminology:

• Information that is needed to prove a module correct, is known to that module.

• Information that is known to only one module is a secret of that module.

Corollary:

If a software system is designed so that each internal design decision or external interface is a module’s secret, the changes to the software that are required when one of those design decisions or interfaces changes can be limited to one module.

2.3. What the theorem does not say

It is important to note some of the many things that the theorem stated above does not say.

• The theorem does not say that you have to prove programs correct. It merely talks about what would be true if a proof were possible.

• The theorem does not say that you can always design systems so that each decision that is going to change will be the secret of one module.

• The theorem does not say that the specifications will actually contain all the information used in a proof. Proofs, whether written or oral, formal or intuitive, are often invalid because they are based on assumptions that are not stated. For example, program correctness proofs often assume that no other program/process will change the values of certain variables, but do not state this assumption explicitly.
• The theorem does not state that the designers will have enough information about the future to identify all decisions that might be changed and then make each one the secret of its own module.
• The theorem is trivial if there is only one module. The theorem does not say that it is possible to design any piece of software so that it conforms to the requirements of the theorem and has more than one module.
• The theorem does not say that a software product designed in accordance with this theorem (i.e., with every changeable fact the secret of one module) will make effective use of computing resources.

In other words, the theorem, even if fully formalized, does not say that the it is generally useful and usable.

2.4. What the “experiment” did not show

The “experiment” did not prove very much.
• It did not show that we had applied the theorem because we did not prove that the programs were correct.
• It did not show that it is generally possible to hide critical design decisions. In fact, several years later, I realized that the published design did not hide a very important and subtle design decision and, as a result, had eliminated an important class of implementations from the family of programs [3] or product line [4] defined by the specifications.
• It did not show that the performance penalty that resulted from applying the principle was acceptable. The particular design, as implemented, resulted in a major performance penalty.

The experience did demonstrate that there was something to be gained by trying to apply the theorem in one application with one group of people and one (small) data set.

Subsequent experience has taught me that the principle is much harder for people to apply than I expected. For example, even after being warned that flowcharts lead to structures that do not hide critical information, students and professionals will base a structure on a flowchart. In reviewing designs by both students and professionals (in a consulting capacity) I almost inevitably discover that the supposed secret of one module is actually known by other modules and have to remind the developers that shared secrets are not secrets. In other cases, I have to show them how to design an interface to hide a secret. As discussed in [5], producing an information hiding design requires systematic study and analysis.

The “theorem” shows that information hiding has theoretical advantages, but no amount of theoretical work can show that people can exploit the advantages in real systems.

I often find examples of designs that are successful and good because they do hide information. Unix and its imitators are full of examples of information hiding. I also see many horror stories that result from ignoring the principle. These examples cannot be regarded as experimental evidence of the utility of the principle. They are just anecdotes.

In the “real” sciences, when a principle appears to be confirmed by an experiment or trial by one group of scientists, other scientists design other experiments or trials that are designed to either refute, or independently confirm, the principle. In Computer Science, probably because of the strong influence of mathematics, this is rarely done. In our field there is no glory in confirming other people’s results; you have to invent your own. As far as I know, there has never been a serious attempt to confirm or refute my work. One skeptic wrote and asked for samples of the assembled systems. I sent him listings but never heard from him again.

3. Can we do experiments in Software Engineering at all?

As an Electrical Engineer, I was taught (and later applied) Kirchoff’s circuit laws to design electronic circuits. Every Electrical Engineer that I have ever met knows these laws and can apply them. They do this in spite of the fact that the usefulness of the laws has never been proven by empirical research of the type advocated in Software Engineering. As far as I know, nobody has ever asked one group to design using Kirchoff’s laws and compared their circuits to the product of a group that is not allowed to use them.

The fact is that an empirical verification of Kirchoff’s laws is not needed. The theory is known to be sound and we have anecdotal evidence that it can be applied.

Moreover, an experimental validation worthy of the name would be very expensive. In any valid experiment it is essential to identify all variables that might affect the outcome and either control the values or have a very large sample in which the effects of the uncontrolled variables can be shown to be insignificant. When studying how human beings perform complex design tasks, there are so many variables, the variables are so hard to control and each trial so expensive that it is rarely practical to do such an experiment.

It is here that I see evidence that the scientists who opposed the founding of Computer Science departments were right. I see many papers drawing conclusions based on observing a very small number of inexperienced students solving relatively simple tasks in classroom settings. There are always many variables that affect the outcome; the investigator ignores all but a small subset, identifies some as “the control variables” and claims to have done a controlled experiment. Suggesting that these “experiments” can give
us information that is relevant for experienced professionals solving complex problems in an industrial setting, requires a giant leap of faith. No researcher with training in classical experimental science would do such a thing.

4. What about surveying practitioners?

Another form of empirical study replaces experiments with surveys. People are asked how they solve a problem or whether they use some method. I was taught that one of the cardinal rules of psychological research is, “If you ask a subject how he/she solves a problem, don’t believe what they say.” People often do not know how they solve problems, and asking them how they do it often changes the way that they work. The Computer Scientists who conduct these surveys often seem to ignore that rule and assume that people actually do what they say they do.

I also see people drawing unjustified conclusions from survey data. For example, one survey of programmers revealed that a large majority said that they do not use logic when programming. Accepting their answer as factual, the researcher concluded that students interested in a career in software did not need to be taught logic. Reflecting on the large number of logical errors in software, one might equally well conclude that those errors are caused by programmers not using logic and they should be taught more.

In reading survey results, one must always remember that the vast majority of those who are now developing software did not receive a professional education in software engineering. I am sure that surveys would reveal that most believe that they learned more on the job (from others with similar backgrounds) than in their education. We must be careful not to portray folklore and bad habits as something worth teaching to future professionals.

5. Why do we study Software Engineering?

A growing number of empirical researchers are studying how today’s software developers carry out their tasks. In fact, for much of the research one sees, training in Psychology or Sociology would be more useful than an education in Computer Science.

While knowledge of how things are currently done often provides insights, I think some researchers have lost sight of why our field is so well supported. We are not here to find out how things are done now. That is a means to an end. We are here to find ways to do things better. Statistics about how today’s professionals spend their time will not tell us what tomorrow’s developers should do. Observations about how untrained people solve problems, will not, by itself, tell us what we should teach to the ones that we are training.

6. Conclusions

Both groups, those who develop theories about Software Engineering and those who do empirical studies, need to re-think what they do. No amount of theoretical analysis will reveal that a theory-based method makes an assumption that is not true; only empirical studies will reveal that a model is an invalid abstraction of reality. On the other hand, our inability to carry out truly scientific experiments and surveys means that empirical studies by themselves will yield anecdotes of limited value. Empirical studies should be used to confirm that what works in theory can actually be used (and useful) in practice.

Neither theory alone nor empirical studies alone show us how to improve our software development capabilities. In fields such as Physics, experimental researchers make measurements to test theories, and theoretical researchers adjust their theories to make them consistent with the facts. Software Engineering researchers will have to learn to do the same.

7. References