Empirical Answers to Fundamental Software Engineering Problems (Panel)

Bertrand Meyer (Chair)
ETH Zurich, Switzerland; ITMO Saint Petersburg, Russia;
Eiffel Software, USA
Bertrand.Meyer@inf.ethz.ch

Harald Gall
University of Zurich, Switzerland
gall@ifi.uzh.ch

Mark Harman
University College London, UK;
Microsoft Research Cambridge, UK
mark.harman@ucl.ac.uk

Giancarlo Succi
Free University of Bozen, Italy
giancarlo@giancarlosucci.com

ABSTRACT
Can the methods of empirical software engineering give us answers to the truly important open questions in the field?

Categories and Subject Descriptors
D.2 Software Engineering

General Terms
Experimentation

Keywords
Empirical Software Engineering

1. PANEL PRESENTATION (BERTRAND MEYER)
For all the books on software engineering, and the articles, and the conferences, a remarkable number of fundamental questions, so fundamental that just about every software project runs into them, remain open. At best we have folksy rules, some possibly true, others doubtful, and others — such as "adding people to a software project delays it further"[1] — wrong to the point of absurdity. Researchers in software engineering should, as their duty to the community of practicing software practitioners, try to help provide credible answers to such essential everyday questions.

The purpose of this panel discussion is to assess what answers are already known through empirical software engineering, and to define what should be done to get more.

"*Empirical software engineering*" applies the quantitative methods of the natural sciences to the study of software phenomena. One of its tasks is to subject new methods — whose authors sometimes make extravagant and unsupported claims — to objective scrutiny. But the benefits are more general: empirical software engineering helps us understand software construction better.

[1] From Brooks's *Mythical Man-Month*.

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
ACM 978-1-4503-2237-9/13/08
http://dx.doi.org/10.1145/2491411.2505430

There are two kinds of target for empirical software studies: products and processes. *Product* studies assess actual software artifacts, as found in code repositories, bug databases and documentation, to infer general insights. *Project* studies assess how software projects proceed and how their participants work; as a consequence, they can share some properties with studies in other fields that involve human behavior, such as sociology and psychology. (It is a common attitude among computer scientists to express doubts: "*Do you really want to bring us down to the standards of psychology and sociology?*" Such arrogance is not justified. These sciences have obtained many results that are both useful and sound.)

Empirical software engineering has been on a roll for the past decade, thanks to the availability of large repositories, mostly from open-source projects, which hold information about long-running software projects and can be subjected to data mining techniques to identify important properties and trends. Such studies have already yielded considerable and often surprising insights about such fundamental matters as the typology of program faults (bugs), the effectiveness of tests and the value of certain programming language features.

Most of the uncontested successes, however, have been from the product variant of empirical software engineering. This situation is understandable: when analyzing a software repository, an empirical study is dealing with a tangible and well-defined artifact; if any of the results seems doubtful, it is possible and sometimes even easy for others to reproduce the study, a key condition of empirical science. With processes, the object of study is more elusive. If I follow a software project working with Scrum and another using a more traditional lifecycle, and find that one does better than the other, how do I know what other factors may have influenced the outcome; and even if I bring external factors under control how do I compare my results with those of another researcher following other teams in other companies? Worse, in a more realistic scenario I do not always have the luxury of tracking actual industry projects since few companies are enlightened enough to let researchers into their developments; how do I know that I can generalize to industry the conclusions of experiments made with student groups?

Such obstacles do not imply that sound results are impossible; studies involving human behavior in psychology and sociology face many of the same difficulties and yet do occasionally yield

important and credible insights. But these obstacles explain why there are still few incontrovertible results on process aspects of software engineering. This situation is regrettable since it means that projects large and small embark on specific methods, tools and languages on the basis of hearsay, opinions and sometimes hype rather than solid knowledge.

No empirical study is going to give us all-encompassing results of the form "Agile methods yield better products" or "Object-oriented programming is better than functional programming". We are entitled to expect, however, that they help practitioners assess some of the issues that await every project. They should also provide a perspective on the conventional wisdom, justified or not, that pervades the culture of software engineering. Here are some examples of general statements and questions on which many people in the field have opinions, often reinforced by the literature, but crying for empirical backing:

- The effect of requirements faults: the famous curve by Boehm is buttressed by very old studies on special kinds of software (large mission-critical defense projects). What do we really lose by not finding an error early enough?
- The cone of uncertainty: is that idea just folklore?
- What are the successful techniques for shortening delivery time by adding manpower?
- The maximum compressibility factor: is there a nominal project delivery time, and how much can a project decrease it by throwing in money and people?
- Pair programming: when does it help, when does it hurt? If it has any benefits, are there in quality or in productivity (delivery time)?
- In iterative approaches, what is the ideal time for a sprint under various circumstances?
- How much requirements analysis should be done at the beginning of a project, and how much deferred to the rest of the cycle?
- What predictors of size correlate best with observed development effort?
- What predictors of quality correlate best with observed quality?
- What is the maximum team size, if any, beyond which a team should be split?
- Is it better to use built-in contracts or just to code assertions in tests?

When asking these and other similar questions relating to core aspects of practical software development, I sometimes hear "*Oh, but we know the answer conclusively, thanks to so-and-so's study*". This may be true in some cases, but in many others one finds, in looking closer, that the study is just one particular experiment, fraught with the same limitations as any other.

The principal aim of the present panel is to find out, through the contributions of the panelists — who are top contributors to empirical engineering, having helped to bring up the field to its current level of success and respect —, which questions have useful and credible empirical answers already available, whether or not widely known. The answers must indeed be:

- *Empirical*: obtained through objective quantitative studies of projects.
- *Useful*: providing answers to questions of interest to practitioners.
- *Credible*: while not necessarily absolute (a goal difficult to reach in any matter involving human behavior), they must be backed by enough solid evidence and confirmation to be taken as a serious input to software project decisions.

An auxiliary outcome of the panel should be to identify fundamental questions on which credible, useful empirical answers do not exist but seem possible, providing fuel for researchers in the field.

To mature, software engineering must shed the folkloric advice and anecdotal evidence that still pervade the field and replace them with convincing results, established with all the limitations but also all the respectability of quantitative, scientific empirical methods. The aim of this panel is to establish what we already know and what we should do to know more.

2. WHAT IS MISSING (GIANCARLO SUCCI)

Bertrand Meyer, the organizer of the panel, has proposed a challenge for helping systematizing the experience coming from projects with the intention of creating a sound set of propositions on the effectiveness of software production practices, languages, tools.

He spells out his goal very clearly: "To mature, software engineering must shed the folkloric advice and anecdotal evidence that still pervade the field and replace them with convincing results, established with all the limitations but also all the respectability of quantitative, scientific empirical methods."

I think we should step back and ask ourselves why this is not yet the case. In other disciplines is not so, like in medicine; however, there are also subjects with the same level of uncertainty on the effectiveness of practices as software engineering, like management, pedagogy, etc.

In the past, people have accounted this lack to the absence of suitable tools to collect data from industry without interfering with the work of who is developing software for business, who cannot devote time to activities who are not directly productive. However, now we have such tools, especially the Non Invasive Measurement Tools, like PROM or Hackystat.

So, something else is missing, and I would summarize here what, in my humble opinion, what it is.

First, we lack a method, agreed by all key stakeholders, to store the results of the studies in a way that is precise, unambiguous, reusable both for comparisons by researchers and for use by practitioners. Using an overly abused term, we lack a commonly agreed and widely used ontology to describe the studies – the hypothesis, the experimental design, the collected measures, etc. Indeed, building such ontology can be boring, tedious, time consuming, cumbersome and not provide any direct career reward, and this is also because it has not been done.

Second, we lack, as a community, a clear commitment toward such empirical evidence. Indeed, also thanks to the work done by many great scientists like Vic Basili, Dieter Rombach, Barbara Kitchenham, Norman Fenton, and many other, papers on empirical research are now fully accepted in major Software Engineering forums. In the past was more difficult – it took me 5 years to get my first work on measurement at ICSE (Maurer et al., 1999). However, we have papers in major forums presenting methodologies, tools, or practices that do not include any significant experi-

mental validation of what they present. Such papers would be completely rejected in medicine – and for a good reason. And I find this situation completely unacceptable.

Third, we do not appreciate enough that empirical studies require a very significant amount of effort. For a PhD or a junior faculty it is much easier to focus on some more theoretical work (which often, as said above, can be done without any scientific proof), rather than to get involved in a lengthy experimentation that might even not produce confirmation of the original hypothesis – and, in our discipline, we publish typically only papers on confirmed hypotheses, not on rejected hypotheses! In my experience, the whole cycle from designing an experiment to publishing a paper in a top forum lasts at least four years. For instance, we started the planning of the experimentation that lead to our 2012 ICSE Paper (Sillitti et al., 2012), in 2006, like this other involving a significant industrial experimentation (Clark et al., 2004). This is the duration of a long PhD or the time for a tenure evaluation...

Lastly, we lack a sense of empiricism in our empirical studies. Too often we forget that an empirical study carries results that, in the very best, are statistically significant, meaning that have margin of errors. We are not presenting truths proven by a theorem, but facts that sometimes can be wrong. Well, this is true also for medicine: it is enough to look for all the adverse effects that drugs can have, as described in their own leaflets – even an aspiring may cause the death of the patient!

This is why I really welcome this panel and I am really grateful to Bertrand for having initiated this work.

3. STRENGTHENING EXPERIMENTAL & EMPIRICAL SOFTWARE ENGINEERING KNOWLEDGE USING SBSE (MARK HARMAN, WITH YUE JIA[2] AND JENS KRINKE[3])

3.1 Overview

This position paper gives some examples from source code analysis and testing of experimental and empirical results, clarifying the difference between experimental and empirical studies and arguing that each is complementary to the other. We use these examples to illustrate the paucity of experimental and empirical knowledge. We conclude with an outline of an approach to empirical and experimental software engineering that uses Search Based Software Engineering (SBSE) to enhance rigour so that we can be more sure of the evidence that underpins experimental and empirical knowledge.

3.2 What Do We Know?

What do we know about software engineering? Let us be a little more specific. The most elementary question of all is surely

"what do we know about the fundamental engineering material with which we work; the programs themselves?"

One of the first things that we believed that we knew was that structured programming was superior to unstructured programming. This belief arose, initially, from Dijkstra's famous complaint about the use of the goto statement [2]. However, if one looks for empirical evidence to support this claim, one finds very little. It would seem that many of the things that we believed to be true about software are just that: beliefs. Surely many of these beliefs will turn out to be true. Nevertheless, we do not know the *extent* to which they are true, the *situations* in which the effects of maximal and minimal, nor any kind of *bounds* or assessment of the effect *size*. This is not a good situation for a quantitative engineering discipline. Are we really any better than a religious sect?

Side effects are widely believed to be harmful. Expressions should be side effect free. Surely this is obvious. This 'obvious' observation motivated an entire field of programming literature: Functional Programming [4]. However, the authors are only aware of a single paper that attempts to empirically assess the effect of side effects on programs [3]. Surely, such an important foundational belief (one that underpins a whole field of research) deserves a greater degree of scientific investigation.

Our very limited empirical studies of this problem were able to demonstrate that side effects are, indeed, harmful to program comprehension. Perhaps the most interesting finding was that experienced programmers' performance appears to be just as compromised as that of novices. This challenges the potential hubris of those more experiences programmers, who might believe themselves to be sufficiently experienced to avoid any cognitive penalty that might accrue from programming with side effects.

However, for such an important topic it is simply not enough to rely on a single study. Our study considered only a very limited type of side effect, it only considered the C programming language, and it only consisted of two studies: one on inexperienced programmers (students) and one on more experienced programmers (professional software engineers).

As well as the paucity of studies of important empirical problems there has also been a tendency to rely on a single study, particularly where this provides results we might want to believe. This is an understandable and entirely human behaviour, but the fact remains that it is detrimental to our science and engineering work.

For example, in software testing research, we are often concerned with the problem of evaluating the degree to which our testing techniques are effective and efficient. For this purpose it is convenient and expedient to investigate the proposed testing technique on simulated faults using mutation testing. This practice of using mutation testing is often justified with reference to the seminal work of Andrews, Briand and Labiche [1]. This is an important result for Empirical Software Engineering. It demonstrates that there is *evidence* that mutation faults are good simulators of real faults. However, there is too much temptation stretch this important finding; believing that it necessarily applies more widely in the authors would, themselves, claim.

For example, if one is using mutation testing on a language or with a set of mutation operators not considered in the paper by Andrews, Briand and Labiche, it would clearly be inappropriate to claim evidence that mutation faults adequately simulate the real faults. Since many authors do tend to make such claims, this highlights a pressing need for more studies that investigate the relationship between mutation faults and real faults in languages, paradigms and domains not previously considered in the literature.

3.3 Experimental vs. Empirical

It is important to recognise the difference between empirical and experimental software engineering research [7]. Experimental software engineering concerns findings that result from experiments conducted under what we might term 'laboratory condi-

[2] University College London, yue.jia@ucl.ac.uk

[3] University College London, j.krinke@cs.ucl.ac.uk

tions'. Empirical research is concerned with findings that are observed from the so-called real world. A study on 1000 machine-generated problem instances is an experimental study. A study on 10 real-world problem instances is an empirical study.

An unhealthy tendency has developed in which the community as a whole (and its referees in particular) tend to regard the empirical studies as inherently and fundamentally superior to experimental studies. In fact, both approaches are entirely complimentary. A robust software engineering claim should, ideally, be backed by claims supported by both empirical and experimental evidence.

Empirical studies are clearly valuable because they concern real world problems. Evidence is guaranteed to be applicable to at least one real-world instance. Unfortunately, of course, one seldom has sufficiently many real-world instances on which to base truly generalisable claims. Experimental studies can provide complimentary evidence to help to bridge this gap.

In an experimental study, problem instances may be *artificial*, but they are also *controllable*. This allows the researcher to study, experimentally, the effects of a particular algorithm, method or approach on varying degrees problem instance. Many problems are characterised by parameters that denote aspects of the problem 'difficulty'. It may not be possible to obtain real-world examples at sufficiently diverse granularity and in sufficient numbers to explore the impact of these parameters.

Fortunately, using a problem instance generator, we can generate an arbitrary number of synthetic solutions that exhibit the characteristics we seek to investigate. This is a fundamentally *experimental* approach and it allows us to consider, under controlled conditions, the effect of dependent variables on independent variables. This is the standard approach to experimental science and it is somewhat surprising that it seems to have become so thoroughly deprecated in the software engineering research community.

Returning to the example of mutation testing, a study of a proposed testing technique might consist of both an empirical and an experimental study. Mutation testing can be employed to investigate the behaviour of the proposed approach under controlled conditions. For example, we can investigate the testing technique's ability to find particular kinds of fault or its performance slowdown with respect to certain programming language features known to be problematic to be approach in hand. Ideally, these experimental results from mutation testing ought to be combined with empirical results in the form of an evaluation of the proposed technique on a suite of real software faults.

3.4 Using SBSE to Make Experiments more Robust

Many software engineering studies are concerned with claims about the performance of the particular kind of approach, algorithm or tool. Typically these studies concern experiments in which an algorithm or tool is applied to problem instances in order to make claims about improved effectiveness and efficiency compared to the state-of-the-art. Many of the algorithms and tools we develop are highly configurable. Sadly this poses a significant challenge when it comes to an experimental evaluation of one technique against another.

> *How can we be sure that we have not simply chosen that configuration which just so happens to make our chosen favourite technique appear to be superior to the current state of the art?*

In the remainder of this position paper we wish to address the question of how such experimental studies can be made more robust using search based techniques.

The first step is to formulate the problem as one in which the configuration space of each technique under investigation becomes a search space. The search space is the space of all possible parameter settings for the techniques. Because of combinatorial explosion, it is impossible for an experimenter to report results for every possible configuration choice. Even relatively well-understood tools have large configuration spaces. For example, GCC has more than 200 different parameter settings so its configuration space is larger than 2^200.

We cannot ignore the effect of configuration choices are experiments. Different configurations can have dramatic effects on experimental findings. Several approaches have been used to address this confounding configuration problem. We could simply report the configuration choices and thereby allow other researchers to try alternatives. We could sample from the possible configurations using design of experiments [9]. Alternatively, we could *search* the space of configurations seeking to optimise the parameter settings for the particular experimental questions we seek to answer.

Suppose we wish to compare Technique A against Technique B. Suppose that Technique A is the new proposed approach to solving our Software Engineering problem, while Technique B is the existing state-of-the-art. Clearly, we would like to demonstrate that we have evidence to claim that Technique A is superior. Such a claim can range from a 'weak existential' claim to a 'strong universal' claim in terms of configurations space.

Weak existential claims: If there exists a configuration setting for which Technique A outperforms Technique B, then we have a *weak existential claim* to superiority. This is the weakest possible form of evidence (in terms of configuration space) for the superiority of Technique A. Perhaps such a level of evidence might be suitable for a workshop. It demonstrates that there is, at least, some merit in considering the technique.

We can use SBSE to search for a configuration in which Technique A is superior to Technique B. For many software engineering techniques, this is not a demanding task. All that is required is to use the metrics that assess the performance of the techniques as a *fitness function* and to maximize this fitness over the configuration space. Since we must have a way of measuring the metric in order to make any claims about the performance of the techniques, it is only a small step to treat this metric as a fitness function [6].

In this way we would be searching for that configuration which is most sympathetic to our experimental goal. Our experimental goal, in turn, is most sympathetic to Technique A. This approach has a natural and compelling side effect that may be beneficial: The harder it is for the search to find such a sympathetic configuration, the less we should have confidence in the proposed new technique. The experimenters' confidence in Technique A will naturally grow commensurate with the ease with which the search finds sympathetic configurations. This confidence might reach the point at which the experimenter considers moving on to strong universal claims for their proposed new technique.

Strong universal claims: If there does not exist a configuration setting for which Technique A fails to outperform Technique B, then we can make a *strong universal claim* for the superiority of Technique A over Technique B. This is the strongest form of evidence possible (in terms of the configuration space) for Technique A.

Unfortunately, in order to provide evidence to support this claim we would have to present results for all possible configurations of the techniques in our experimental study. This is clearly infeasible in most cases. However, we can use SBSE to approxim-

ate the answer to this question. Suppose we searched the configurations that are sympathetic to Technique B. That is, we seek to make a *week existential claim* that the existing state-of-the-art (Technique B) is superior to our new propose approach (Technique A). The degree of difficulty experienced by the search in finding such configurations is one indicator of the approximate strength of the claim we can make the superiority technique A: The harder it is to find configurations that support the weak existential claim for Technique B, the stronger our belief becomes in the strong universal claim to the superiority of Technique A.

Our search algorithm will be optimising the configurations that favour the state-of-the-art. Therefore this is the kind of testing approach for evaluating our proposed new software engineering technique. Just as a tester gains confidence in the system under test the longer testing proceeds without failure, we will gain confidence in our technique the longer we search without finding a configuration that it fails to outperform. When the tester finds a fault in software system, it does not mean that the software system is abandoned. The fault is either fixed or a workaround must be found. Similarly, if an attempt to provide a strong universal claim for Technique A should fail, this does not necessarily mean that we must abandon Technique A. Rather, we may use SBSE to better understand those conditions under which the technique performs well. This understanding provides another spin-off benefit: Ultimately, we may arrive at a set of configuration conditions under which we have confidence that Technique A outperforms Technique B. In this way, SBSE will have helped us to investigate and construct experimental hypotheses.

Other claims derived from optimised choices over the configuration search space

SBSE has recently been used to find configurations of tools that are better suited to specific software engineering problems [5,8]. We have also recently demonstrated that it can be used to find configurations that maximise agreement among tools [10]. This allows us to be more sure that experimental differences observed are the result of inherent differences in the tools themselves, rather than the choices of parameters pertaining to these tools.

4. REFERENCES

[1] J. Andrews, L. Briand and Y. Labiche. Is mutation an appropriate tool for testing experiments? ICSE 2005.

[2] E. W. Dijkstra. Goto Statement Considered Harmful. CACM, 1968.

[3] J. J. Dolado, M. Harman, M. Carmen Otero and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension, TSE 2003.

[4] A. J. Field and P. G. Harrison. Functional Programming. Addison-Wesley, 1988.

[5] S. Lohar, S. Amornborvornwong, A. Zisman, and J. Cleland-Huang. Improving trace accuracy through data-driven configuration and composition of tracing features. In ESEC/FSE, 2013.

[6] M. Harman and J. Clark. Metrics Are Fitness Functions Too. Metrics 2004.

[7] M. Harman, E. Burke, J. A. Clark and X. Yao. Dynamic Adaptive Search Based Software Engineering. ESEM 2012.

[8] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia. How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms. ICSE 2013.

[9] S. Poulding, P. Emberson, I. Bate and J. Clark. An Efficient Experimental Methodology for Configuring Search-Based Design Algorithms. HASE 2007.

[10] T. Wang, M. Harman, Y. Jia, J. Krinke: Searching for Better Configurations: A Rigorous Approach to Clone Evaluation. ESEC/FSE 2013.

[11] Justin Clark, Chris Clarke, Stefano De Panfilis, Giampiero Granatella, Paolo Predonzani, Alberto Sillitti, Giancarlo Succi, Tullio Vernazza, (2004) Selecting components in large COTS repositories, Journal of Systems and Software, 73:2(323-331)

[12] Frank Maurer, Giancarlo Succi, Harald Holz, Boris Kötting, Sigrid Goldmann, Barbara Dellen (1999) Software process support over the Internet, Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, USA, May

[13] Alberto Sillitti, Giancarlo Succi, Jelena Vlasenko (2012) "Understanding the impact of pair programming on developers attention: a case study on a large industrial experimentation," Proceedings of the 34th International Conference on Software Engineering, Zurich, Switzerland, May