Empirical Studies of Software Engineering: A Roadmap

Dewayne E. Perry
Electrical and Computer Engineering
University of Texas at Austin
Austin, TX 78712
+1 512 471 2050
perry@ece.utexas.edu

Adam A. Porter
Computer Science
University of Maryland
College Park, MD 20742
+1 301 405 2702
aporter@cs.umd.edu

Lawrence G. Votta
Motorola
1501 W. Shure Dr.
Arlington Heights, IL 60004
+1 847 632 2706
votta@cig.mot.com

ABSTRACT
In this article we summarize the strengths and weaknesses of empirical research in software engineering. We argue that in order to improve the current situation we must create better studies and draw more credible interpretations from them. We finally present a roadmap for this improvement, which includes a general structure for software empirical studies and concrete steps for achieving these goals: designing better studies, collecting data more effectively, and involving others in our empirical enterprises.

Keywords
Empirical Studies, Software Engineering

1 INTRODUCTION

An empirical study is really just a test that compares what we believe to what we observe. Nevertheless, such tests, when wisely constructed and executed and when used to support the scientific method, play a fundamental role in modern science. Specifically, they help us understand how and why things work, and allow us to use this understanding to materially alter our world.

Yet in software engineering research, empirical studies have not had the same success. This seems odd given their wide use in other sciences. This problem has been widely discussed and many articles have pointed out possible causes. We argue, however, that many of these articles are "implementation-oriented". That is, they suggest that the biggest barriers to using empirical studies lie in the details of conducting them.

For example, Norman Fenton et al. [1] point out that many empirical studies have poor statistical designs, don't scale up to large systems, and are conducted over too short a time. Victor Basili [2] suggests that the many differences between individual software projects make comparison difficult. Philip Johnson also remarks that practitioners may resist being measured. [3].

Surely, these and many other factors affect the use of empirical studies. Nevertheless, we believe that even if all these issues disappeared, empirical studies would still fail to have the impact they have had in other fields. This is because there is a gap between the studies we actually do and the goals we want those studies to achieve.

Our experience in attempting to use empirical studies to change how a development group builds software has convinced us that we must also take a "requirements-oriented" view. That is, that we must think harder about what experiments really are and how they can be most effectively used to improve software development.

We came to this conclusion while trying to improve the software inspection process used in a Lucent development setting. We found that our greatest difficulties were not in designing and conducting individual studies (which was by no means easy). Our greatest difficulties were in conceptualizing and organizing a body of work that could be relied on as the basis for changing an organization's long-practiced development processes.

Moreover, we believe that this problem – defining and executing studies that change how software development is done - is the greatest challenge facing empirical researchers. Therefore, in this essay we will examine the nature and purpose of empirical studies, discuss how they are currently used, and offer some suggestions for improving them in the future.

2 WHY EMPIRICAL STUDIES?

All large software projects follow some underlying development process that includes stages such as requirements definition, functional design, unit implementation, integration, and so on. The way in which these stages are conducted, the tools that are used to support them and the rationale for doing so, however, varies widely.

Some companies have rigid processes that all projects follow. Others allow individual managers to make

decisions based on their personal expertise. Others simply follow institutional traditions for lack of suitable alternatives. No matter which approach is taken, in almost all cases, there is little hard evidence to inform these decisions, and their costs and benefits are rarely understood. One reason for this is that software engineering research has failed to produce the deep models and analytical tools that are common in other sciences.

The situation indicates a serious problem with research and practice in software engineering. We don't know the fundamental mechanisms that drive the costs and benefits of software tools and methods. Without this information, we can't tell whether we are basing our actions on faulty assumptions, evaluating new methods properly, or inadvertently focusing on low-payoff improvements. In fact, unless we understand the specific factors that cause tools and methods to be more or less cost-effective, the development and use of a particular technology will essentially be a random act. Empirical studies are a key way to get this information and move towards well-founded decisions.

Empirical studies take many forms. They are realized not only as formal experiments, but also as case studies, surveys, and prototyping exercises as well. No matter what its form is, the essence of an empirical study is the attempt to learn something useful by comparing theory to reality and to improve our theories as a result. Therefore, empirical studies involve the following steps:

- formulating an hypothesis or question to test
- observing a situation,
- abstracting observations into data,
- analyzing the data, and
- drawing conclusions with respect to the tested hypothesis.

Of these, the last step – drawing conclusions - is the most important and too often the least well done. It's important because it's here that we get the information that will enable us to guide, to change and to push our field. It's here that we pinpoint inefficiencies, identify where large improvements can be made, and determine whether our still-forming ideas are on-track. It's the reason why we do empirical studies. The other steps, however indispensable, are only prologue.

Of course, doing all of these steps well is difficult. Done well, however, the payoffs will be large, including that:

- knowledge is encoded more rapidly,
- low-payoff or erroneous research ideas are discarded quickly,
- high-payoff areas are recognized and correctly valued, and
- important practical issues are considered.

3 THE STATE OF EMPIRICAL RESEARCH

We have said that empirical studies are used to compare what we believe to what we see. Ideally, these tests should allow us to positively affect the practice of software development. In this section we will explore to what degree we, as a research community, are living up to this ideal.

Current Strengths

Empirical software engineering has matured considerably over the last 10-20 years. Consider for example:

In some software engineering sub-fields empirical validation is considered, if not a standard part, then a powerful addition to research papers. This has been especially notable in the testing community.

The quality of the average empirical study is rising. Researchers are becoming better educated about empirical studies and how to conduct them. Consequently, we are seeing increasingly more comprehensive studies conducted on increasingly realistic programs and processes.

Funding agencies are recognizing the value of empirical studies. In the U.S. for example, National Science Foundation (NSF) programs such as the Experimental and Integrative Activities program supports research with a decidedly experimental flavor. The recently proposed Information Technology Research (ITR) program also stresses that proposals include a strong validation component. Other examples include National Academy of Sciences sponsored workshop on the topic of statistics and software engineering [4].

We've had many talks with currently active researchers who have become interested in and are beginning to do empirical studies.

And finally, there have been several empirical studies-related tutorials, panels and state-of-the art presentations at major software engineering conferences such as ICSE, FSE, ICSM and others.

Of course many factors contribute to this situation. Many researchers and practitioners have tackled the problem of increasing the use and effectiveness of empirical studies. For example:

There have been several influential and widely quoted articles attempting to raise our consciousness about the state of empirical studies in software engineering. Tichy et al. [5] and Wallace and Zelkowitz [6] both argue that empirical studies are underused in software engineering relative to other areas of engineering. Both ferociously condemn software engineering researchers for not validating their research ideas and both have been invaluable making this a high profile issue.

There is a growing awareness that software engineering researchers must be educated about conducting empirical studies. To this end, Kitchenham and Pfleeger wrote a series of articles for ACM SigSoft Software Engineering Notes. These articles covered a variety of topics including the logical foundations and design of empirical studies, their operation, and techniques for collecting, analyzing and interpreting data.

Several research groups were instrumental in increasing researcher access to industrial data. Today we find many papers with significant, detailed accounts of industrial experience based on industrial data. One of the forerunners of this approach was the Software Engineering Laboratory of NASA, the Computer Sciences Corporation, and the University of Maryland [7].

Finally, many fine researchers have waded in and done their own empirical studies.

Systemic Problems

Despite, or maybe because of, the strengths listed above there are some serious problems. These stem from misunderstandings about what empirical studies are and why we do them. Before we can improve our use of empirical studies we have to eliminate some problematic practices and beliefs.

Often when someone says that we need more empirical studies in software engineering, they really mean that research results should be empirically validated. They want researchers to demonstrate the value of their new ideas as early as possible. This is a good idea for many reasons. We believe, however, that it is important to remember that empirical studies can be used not only retrospectively to validate ideas after they've been created, but also proactively to direct our research.

For example, in compiler optimization research empirical studies have identified common code usage patterns. Knowing, for instance, that branching behavior is not usually random, helps identify and justify the potential value of research on branch prediction, aggressive pre-fetching, etc. In short, we should use empirical studies also to drive our research

In program committee meetings we often hear lengthy discussions over the exact statistical tests used in a study or whether it wouldn't have been better to have done one thing or another. These discussions reflect a vain search for the perfect study. Well, we've done many studies and we've never done one perfectly! Of course, we want to see proper statistics used. But as we will discuss shortly, what's important is not whether the study is textbook perfect, but whether the study and its conclusions taken as a whole are credible.

Too many empirical studies study the obvious. As this sometimes shows that the obvious isn't so obvious, we wouldn't discourage anyone from doing such work. Nevertheless, it makes us wonder, "if empirical studies mostly just confirm the intuitively obvious, then what's wrong with argument by intuition". Clearly, we believe that there are things that are true, but that are not intuitively obvious. Furthermore, we believe that some of these findings will be valuable to software research and practice. Therefore, we need to think much harder about the questions we are studying empirically.

There are too many papers whose only selling point is that they have lots of data. Data is not enough. Just presenting data or simply applying curve-fitting algorithms to them may be useful. But they don't usually help us understand why the data is as it is. Our data should be used to answer questions, not just to fill graphs.

A more fundamental aspect of this problem is that many empirical studies simply lack hypotheses. They pose no questions, they serve no well-defined end. Thus at the end of the study the researcher can only present observations about the data. All studies, even case studies, should be designed to answer some question.

As we said earlier, the most important part of doing an empirical study is drawing conclusions. Many papers fail to do anything with their results. We need to learn something from every study and relate these things to theory and practice.

Since many researchers are reluctant to draw conclusions from their data, it's easy to imagine that they aren't too happy to generalize them either. Instead of speaking thoughtfully about their work they cloak the results in "weasel words". So much so that, often, in the end, they say nothing. There's obviously a balance to be reached here because we don't want researchers to over-generalize, But on the other hand, if we can't discuss what a study's results might mean then it's hard to make progress.

4 FUTURE CHALLENGES TO EMPIRICAL STUDIES

The goal of all research, not just empirical studies, is to improve the state of research and practice. If we want to empirical studies to improve software engineering research and practice, then there are two things that we need to do better in the future. Said simply, we need to create better studies and we need to draw more credible conclusions from them.

Creating Better Empirical Studies

Creating better studies means doing studies that have some chance of directing our research. It implies that we must be clear about the goals of our studies, design them more effectively, and maximize the information we get out of them.

To do this we should consider at least the following issues.

Our studies should strive to establish principles that are causal, actionable and general.

For a factor A to cause outcome B it's necessary that A and B are correlated, that A precedes B in time and that there is a constructive, testable theory explaining how A affects B. Without causality you have no ability to control your situation.

A principle is actionable if the causal agent A can be effectively controlled. For example, knowing that larger systems normally have more bugs may not be an actionable principle if the developer can't make the system smaller.

The principles should be applicable in as wide a variety of circumstances as possible.

When we have a causal relationship we know why something happens. If the agent is actionable, then we have a knob that can be turned to control the outcome. If it is general it will be useful to a wide range of people in a wide set of contexts.

Our studies should try to address important questions. There are many questions to answer. Answering some of them will be cheaper than answering others; using those answers will have more significance in some cases than in others. This consideration implies that we need to spend a good deal of time understanding why we're doing our studies and what results might come from them.

Individual studies are rarely, if ever, unequivocal. Instead of trying to solve large issues with a single study we must attack it with several; each examining different, but complementary aspects. Here the critical issue is to use each new study to generate and refine our hypotheses.

Empirical studies are expensive and take time. If we must do multiple studies, then we have to find ways to get the information we need at a low cost. This may also mean that we have to take some shortcuts in our experimental designs or tackle smaller, more focused problems.

We will also need to enlist the help of others. Empirical studies gain credibility when they are redone and rechecked. We need to find ways to help others to reproduce our results.

Credible interpretations

The credibility of a study refers to the degree of confidence we have in its conclusions. If studies aren't credible, then the time spent doing them was wasted. To improve the credibility of our studies we must consider several issues.

If we are trying to establish the existence of causal relationships, we need to design experiments with high validity. Validity, as we will explain later, is a characteristic of an empirical study and is the basis of establishing credible conclusions. There are three types of validity that are particularly important: internal, external, and construct validity.

Our studies (no matter how they are done) should always have hypotheses. With every study we must define what we are comparing and why.

Often a study won't be powerful enough to show a causal relationship. Still, in many cases we can posit several alternative explanations for the data and then use other data to discredit them. This still doesn't show causality, but it can at least remove obvious alternative explanations from consideration.

We should avoid the temptation to measure everything to the finest possible precision. Sometimes it will be enough to identify an upper and lower bound; other times it will be enough to measure at a gross resolution. The definition of adequate precision will depend on the problem, but using coarse measurements may be one way to limit study costs, while still getting important information.

Our data and procedures need to be made public so that others can understand, analyze and possibly replicate our studies. Frankly, this can be really difficult, and we haven't always managed to keep up ourselves, but we believe it's worth the effort.

Designing an Empirical Study

In our careers we've designed and conducted a number of studies. None have been without flaws. Our conclusion is that no study is perfect and that the real challenge is to create, design and conduct high-impact, credible studies. This involves managing trade-offs in such a way that we maximize:

- accuracy of interpretation - the results we see are not really the result of some unknown influence,
- relevance - our results tell us something important about software engineering, and
- impact - our results affect the practice of or research into software engineering

subject to

- resource constraints - studies are expensive; we must work within resource limitations, and
- risk - studies, especially those done in industry, can disrupt or put at risk industrial partners; we must minimize these problems.

5 THE STRUCTURE OF AN EMPIRICAL STUDY

In this section we discuss the structure and components of empirical studies. We expect that good empirical studies will have each of these components and that papers written about the studies will discuss them as well. These components are:

- research context,
- hypotheses,

- experimental design,
- threats to validity,
- data analysis and presentation, and
- results and conclusions.

Research Context
All studies focus on a problem. Here the problem is defined and its terminology explained. This section links the study goals to what's currently understood about the problem. This section has two parts.

Problem Definition: We define the problem and explain it's important terminology.

Research Review. We provide the historical context surrounding the problem. We describe what we know about the problem, what has been done previously, what questions still remain to be answered and what questions will we be focusing on.

Hypotheses
Hypotheses are essential. They state the research questions we are asking. Sometimes there is confusion surrounding the term hypothesis. In fact there are really two kinds of hypotheses. The trick is to think of a study as a procedure for making a comparison. Therefore, we start at with high-level, abstract questions and refine them into low-level, concrete questions.

Abstract hypotheses are high-level, natural language statements that are usually stated in everyday terms. They say things like, "meetings are an indispensable part of the inspection process".

Concrete hypotheses are stated in terms of the study's design. They may say things like, "teams who do inspections with meetings find more defects than teams who do inspections without them."

We begin by stating our hypotheses first in everyday terms. Then we translate them to terms that exist in the study's design. To the degree that this mapping is done well, comparisons made at the level of concrete hypotheses can be mapped back to the comparisons made at the level of abstract hypotheses.

Study Design
A study's design is a detailed plan for creating the data that will be used to test its hypotheses. It has several components:

One component is a set of variables that link causes and effects. Typically, there are two kinds of variables: dependent and independent.

Independent variables are attributes that define the study setting. In some cases, especially when comparing two situations, these variables are actively manipulated.

Dependent variables are end-process outputs whose values are expected to vary predictably when the values of independent variables change.

The study design may also include a plan for systematically manipulating the independent variables while observing the dependent variables.

The final component is the operational context of the study. This is a description of the physical, intellectual and cultural surroundings in which the study takes place. It is included so that the study's users can better interpret the data.

Threats to Validity
Threats to validity are influences that may limit our ability to interpret or draw conclusions from the study's data. There are at least three kinds of validity that must be protected from such threats.

Construct validity means that the independent and dependent variables accurately model the abstract hypotheses.

Internal validity means that changes in the dependent variables can be safely attributed to changes in the independent variables.

External validity means that the study's results generalize to settings outside the study.

Data Analysis and Presentation
Two general approaches to presenting and analyzing data are called Quantitative and Qualitative analysis.

Quantitative analyses, as the name suggests, deal mainly with comparing numeric data. The comparisons are typically aimed at rejecting or not rejecting a null hypothesis. Two of the tools used in quantitative analysis are hypothesis testing and power analysis.

Hypothesis testing determines the confidence level at which the null hypothesis can be rejected. The confidence level is a measure of the probability that the null hypothesis will be erroneously rejected. Some people believe that this confidence level must be less than 1 in 20 or 0.05 for a result to be significant. It doesn't have to be. In situations where data is plentiful and measurements precise, higher confidence levels may be called for. Since data is often limited and measurement imprecise in studies of software engineering, lower confidence levels may be justified. In any event, we suggest that researchers report the confidence level (without predetermining the significance level) and let the reader decide its significance.

Power analysis determines the likelihood that the null hypothesis will not be rejected when it really should be. This analysis depends on the magnitude of the effect and the amount of data we have. This isn't quite a standard practice yet, but something that we should consider more.

Qualitative analysis, on the other hand, tends to use data that is less readily quantified: observations, interviews, diaries and such. These techniques tend to be used when we want to understand people's perspectives of a situation. Typically, researchers must be very careful about how their biases affect their data. One technique for doing qualitative analysis is call Grounded Theory [8]

In software engineering research qualitative analysis is less widely-used than quantitative analysis, but we can expect to see more of it in the future. As Glasser and Strauss [8] point out "In many instances, both forms of data are necessary—not quantitative used to test qualitative, but both used as supplements, as mutual verification and, most important for us, as different forms of data on the same subject."

Results & Conclusions

After analyzing the data we have to make sense of it. This steps leads us back to our original questions. Here we need to focus on the following things.

We have to understand and explain the limits of the study. What conclusions can we draw? Where are we limited in drawing conclusions? What might have influenced our results?.

Given our understanding of the validity limits and any other information we might have, what does the data really say? Are there ambiguities in our interpretation? Can we think of other explanations for the data we see? Are our results really believable?

Tie results back to the initial questions. Try to explain what questions we answered; don't simply present the data.

Discuss the practical significance of the results. If these results proved to be general what could a manager or developer do with them?

Ensure that you have given enough information to others to help them repeat the study if they want to.

6 CONCRETE STEPS

As we argued above, software engineering researchers must realign their thinking about the goals of empirical studies and improve how they conduct and evaluate them. In this section we discuss some concrete strategies for doing so.

Designing the Studies

Asking Insightful Questions

Ultimately, the most important thing researchers can do is to ask insightful questions. Just as with software development, clear requirements improve the likelihood of a high quality outcome. Note however that an important question isn't necessarily an insightful one, especially if it's very difficult to answer. For instance, it's certainly important to ask whether object-oriented programming is effective, but it's hard to see how a small number of studies can be expected to answer it. Instead, we have to narrower the questions, make them more precise, and ask the ones that lead to important answers.

Knight and Leveson's study on N-Version programming is a good example of such an insightful question [9]. N-Version programming refers to using software redundancy in the hopes of achieving very high reliability. Knight and Leveson noted that this hope depends heavily on the assumption that redundant modules fail independently. If they did not, then the reliability of the total system would not be as high as expected.

Thus, they studied whether independently-developed modules do indeed fail independently. The conclusion was, instead, that the module failures were not independent and that, therefore, N-Version programming did not deliver on its promise of high reliability.

This sparked a great deal of discussion, raising questions about the validity of the study itself, the exact effect of dependent failures on the reliability calculations, and whether failure dependence could be avoided. This is exactly what a good study can do.

Families of Studies

Not every question lends itself to a single empirical study as well as N-Version programming did. For many issues we will have to do many studies. In these cases we design and conduct not just a study, but a family of studies. Here we have to think about the range of questions we will ask and design individual studies to support our overall goals.

Schneiderman et al. [10] did a family of studies on the value of flowcharts as a programming aid. They began by determining how flowcharts might theoretically be useful. That is, they decided that flowcharts might support program composition, program comprehension, program debugging, and program modification. Next, they studied each of these four possibilities in isolation. In all cases they could not demonstrate that having a flowchart was better than not having one. Thus they concluded that flowcharts were not as useful many people believed them to be.

In some cases, we will not know the range of questions beforehand but find them as we conduct our experiments. It may well be that we raise more questions than we answer and so need a sequential family of studies to resolve these related issues as they arise.

The key observation here is that with some thought we can design and conduct a series of studies that together help us answer a larger question.

Building Partnerships

The kinds of experiments we're suggesting often will be difficult for a single individual. Deepening the questions and broadening the number of studies will make it more unlikely that any one person will have all the required information or resources. One way to handle this problem will be to create partnerships.

One kind of partnership involves placing students in industrial environments. This serves several purposes. The student can conduct and monitor the study, while at the same time learning about the practice of software engineering and developing professional contacts. Another important benefit is that the student can handle some study's paperwork, relieving the developers of that burden. This is a powerful benefit as the fear of extra work was one reason our industrial partners had for not wanting to participate in studies.

Another kind of partnership involves creating interdisciplinary research teams. Sometimes a problem is so large that different areas of expertise are needed. In these cases it can be useful to create partnerships with people outside of software engineering.

One example is the Code Decay Project [11] based at Lucent Technologies. It is a long-term, multidisciplinary project examining the fundamental causes, symptoms, and remedies for code decay. The primary data source is the Lucent 5ESS™ switching system. It is composed of more than 50 subsystems and contains over 18 million lines of code. Along with the source the data includes the system's change control history for the past 15 years covering 3.6 million code changes implementing 672,000 change requests. There is also data on its planned and actual development milestones, effort and testing data, organizational history, development policies, and coding standards. The goals of this project are to define response variables and document the existence of code decay, develop code decay indices, identify factors causing it, and create and evaluate prevention strategies.

Obviously one person can't carry out such a project. In fact, the project team contains researchers in Statistics, Experimentation, Organizational Theory, Programming Languages, Software Engineering, and Visualization.

Long-running, in vivo, *experiments*
Many people argue that empirical studies can't be done in live software developments (*in vivo* or *in situ*). Their reasoning is that since different groups can't be asked to build copies of the same system, there are no controls. This isn't false, but the example assumes that we always want to study entire development projects and that doing the project twice is the only way to have controls.

These assumptions aren't always correct. Some development tasks such as bug-fixing, testing, and inspections lend themselves to *in vivo* studies. This is so because they are executed frequently, are of short duration, and, relative to an entire project, are inexpensive. Also, we can establish controls by insuring that tasks are randomly assigned to different treatments.

Nevertheless, care must taken in any study to preserve the rights of the subjects. This problem is harder still in *in vivo* studies because the studies often last longer subjects and the subjects have many other work responsibilities.

One thing we have done in both *in vitro* and *in vivo* studies is to give each study participant a "bill of rights", reminding them of their right to withdraw from the study at anytime with no recriminations from the researchers or their management [12]. We ask each participant to acknowledge this right at the beginning of the study by signing a release form.

Another important problem is knowing when to stop the study. Studies using professional developers creating professional products can have very strong validity, but can put the participating project at risk. One solution is to discontinue any problematic treatment once there are enough observations to convince yourself that nothing "unlucky" has happen. This will require some statistical modeling and will definitely require closely monitoring the study.

Getting the Data
Many studies get their data by measuring subjects as they perform predetermined tasks. This is a costly way to get data. We should, therefore, explore other methods for collecting data.

Retrospective artifact analyses
One resource to which we haven't paid enough attention is the version control system (VCS). Many analyses of the long-term effects of different processes and tools depend on the ability to recreate snapshots of the software at different points in time. A version control system (VCS) tracks each change a developer makes to the system and, as a result, can recreate a consistent snapshot at any point in time. Examples of VCSs include RCS [13] and SCCS [14]. While this basic functionality of VCSs is essential for version control, there is much data in a VCS that is ignored when simply using it to extract snapshots of source code.

For instance, A VCS tags each change with a substantial amount of additional contextual information. Knowing what code was changed, when it was changed, who made the change, and so on, can yield valuable insights into what actually went on in the course of code development, sometimes better than developers' memories. Also, VCS data is amenable to automated analysis. Furthermore, most large software development organizations employ some form of VCS. Thus analysis methods built for VCSs will be widely applicable to many software projects.

Furthermore, this kind of data can be used in many other ways.

- It can be the basis for building program testbeds (well-documented, publicly-available artifacts that can be used by other researchers);
- it can be used to better study system evolution;

- it can be used to help understand work patterns (for creating benchmarks, for example); and
- it can be used to study fault and failure models for new programming languages.

Simulation and Mathematical Modeling

Another way to generate data may be by using simulations or mathematical models. These approaches can be very powerful, but have their own limitations. We'd like to see greater use of simulation and modeling together with directed studies.

One interesting example of this is a study of system integration strategies done by Solheim and Rowland [15]. For this study the researchers built a number of artificial systems (shells of the systems with only rudimentary code inside) whose failure characteristics they could alter. They then tested these systems under different integration strategies and measured their fault detection ability and system reliability. Other examples include using mathematical models to examine the cost-effectiveness of certain maintenance changes [16] and the use of experimental design theory to generate test cases [17].

Involving Others

Meta-Analysis

No single study gives unequivocal results. Therefore, it is imperative that the research community integrates and compares studies that address common hypotheses. This is the only way to gain confidence that empirical results are real and not just due to random variation. Below we outline three approaches.

Integrating multiple studies in a credible way isn't simple. Two studies can address the same issue, but be conceived and executed quite differently. Thus, direct comparison of the results is often impossible because the studies differ considerably in their designs, instrumentation, subject population, and analysis methods.

A classic approach to understanding what several studies say about some phenomenon is to conduct a literature review, qualitatively summarize existing results, and manually synthesize them. The drawback of this approach is that it lacks precise methods for combining different results.

A statistical approach for integrating multiple studies is called Meta-analysis [18]. This approach has two steps. First, the experimenters attempt to reconcile the primary experiments—i.e., define a common framework with which to compare different studies. This involves defining common terms, hypotheses, and metrics, and characterizing key differences. Next, the data from the primary experiments are transformed or recalculated according to agreed upon definitions. In the second step the transformed primary data is combined and reanalyzed. Unfortunately, it is not always clear when Meta-analysis is appropriate, what statistical models should be used, or when it is acceptable to combine data from disparate sources.

And, of course, there are ad hoc approaches that fall between the two. Sometimes you can reconcile two experiments without combining any of their data. This process will often highlight similarities and differences between the two experiments, allowing you to better understand what data are comparable and which are not [19].

Educational Laboratories

Several authors claim that the quality of many CS experiments is poor. Whether or not you agree with these assessments, it is clear that the quality of CS experiments needs to be improved. One factor contributing to this situation is that researchers are rarely trained to perform high quality experiments. An easy way to remedy this is to integrate experimental methods into the CS graduate curriculum.

One way to do this is to create short (say 4-week) teaching modules in which students perform experiments, collect and analyze data, and test hypotheses as part of their graduate software engineering courses.

These teaching modules would support three primary objectives.

- Show how experiments can be used to evaluate hypotheses concerning open research issues,
- Teach students to design and conduct experiments to evaluate their own research, and
- Teach basic statistical procedures for collecting and analyzing data from their own experiments.

These modules could be captured in the form of educational laboratories. Educational laboratory exercises are a standard part of physical science education. These "labs" require students to learn and apply the scientific method, and examine physical principles. While conducting a lab a student monitors a physical process, gathers and analyzes data about the process, and uses the data to test hypotheses—which often challenge his or her intuition.

To construct these labs researchers would package empirical studies into a laboratory "manual." The manual contains the training materials for lectures, reference articles, sample specifications, data collection forms, a description of the experimental procedures, and a post-experiment survey and take-home assignment.

After the lab has been performed the instructor collates the data, recoding it to ensure the anonymity of the participants. Next, the hypotheses behind the experiment are fully explained and the students are taught the statistical rationale for the experimental design, and learn statistical procedures for data analysis and hypothesis testing.

The final step could involve a take-home assignment in which the students are required to propose an experiment to evaluate some hypotheses in which they are interested.

7 SUMMARY

This article has provided an overview of the current state of empirical studies and delineated its strengths and weaknesses. We also discussed the important issues that must be addressed in creating a rigorous and credible empirical discipline for software engineering.

To improve this current state, we must create better designs and draw more credible interpretations from them. As a background for where we need to go in the future, we have outlined a general structure for software empirical studies. We concluded with concrete steps that can be used achieving these goals: designing better studies, getting the data and in involving others in our empirical enterprises.

While we are still relatively immature as an empirical discipline compared with other sciences and engineering disciplines, progress has been made and we are optimistic that we can and will achieve the needed rigor that will underpin the development of deep understandings of software engineering.

8 REFERENCES

1. N. Fenton, S.L. Pfleeger, and R. Glass, *Science and Substance: A Challenge to Software Engineers.* IEEE Software, 1994. **11**(4): p. 86-95.

2. V. Basili, *Editorial.* Empirical Software Engineering Journal, 1996. **1**(2).

3. P.M. Johnson, *Project LEAP: Lightweight, Empirical, Anti-measurement dysfunction, and Portable Software Developer Improvement*, in *Department of Information and Computer Sciences*. 1997, University of Hawaii, Honolulu.

4. D. Pregibon, *et al.*, *Statistical Software Engineering*, . 1996, National Academy of Sciences: Washington, D.C.

5. W.F. Tichy, P. Lukowicz, L. Prechelt, and E.A. Heinz, *Experimental Evaluation in Computer Science: A Quantitative Study*. Journal of Systems and Software, 1995. **28**(1): p. 9-18.

6. M.V. Zelkowitz and D. Wallace, *Experimental validation in software technology.* Information and Software Technology, 1997. **39**(11): p. 735-744.

7. V.R. Basili, *et al. The Software Engineering Laboratory--An Operational Software Experience Factory*. in *14th International Conference on Software Engineering*. 1992. Melbourne, Australia.

8. B. Glasser and A. Strauss, *The discovery of grounded theory: Strategies for qualitative research*. 1977, Chicago: Aldine Publishing.

9. J. Knight and N. Leveson, *An Experimental Evaluation of the Assumption of Independence in Multi-Version Programming.* IEEE Transactions on Software Engineering, 1986. **SE-12**(1): p. 96-109.

10. B. Schneiderman, R. Mayer, D. McKay, and P. Heller, *Experimental Investigations of the Utility of detailed Flowcharts in Programming.* Communications of the ACM, 1977. **20**(6): p. 373-381.

11. S.G. Eick, *et al.*, *Does Code Decay? Assessing the Evidence from Change Management Data.* IEEE Transactions on Software Engineering, (to appear).

12. C.M. Judd, E.R. Smith, and L.H. Kidder, *Research Methods in Social Relations*. 1991, Fort Worth, TX: Holt, Rinehart and Winston, Inc.

13. W.F. Tichy, *Design, Implementation, and Evaluation of a Revision Control System*, in *Proceedings of the Sixth International Conference on Software Engineering*. 1982: Tokyo, Japan. p. 58—67.

14. M.J. Rochkind, *The Source Code Control System.* {IEEE} Transactions on Software Engineering, 1975. **1**(4): p. 364—370.

15. J.A. Solheim and J.H. Rowland, *An Empirical Study of Testing and Integration Strategies Using Artificial Software Systems.* IEEE Transactions on Software Engineering, 1993. **19**(10): p. 941-949.

16. W. Harrison. *Change-Prone Modules, Limited Resources, and Maintenance*. in *wess*. 1996. Monterey, CA.

17. S.R. Dalal and C.L. Mallows, *Factor-covering designs for testing software.* Technometrics, 1998. **40**: p. 234-243.

18. G.V. Glass, B. McGaw, and M.L. Smith, *Meta-analysis in social research*. 1981, Beverly Hills, CA: Sage.

19. A.A. Porter and P.M. Johnson, *Assessing Software Review Meetings: Results of a Comparative Analysis of Two Experimental Studies.* IEEE Transactions on Software Engineering, 1997. **23**(3): p. 129-145.