

Evidence-based Software Production

James Kirby, Jr.
Naval Research Laboratory
Code 5542
Washington, DC 20375
1-202-767-3107

james.kirby@nrl.navy.mil

David M. Weiss
Dept. of Computer Science
Iowa State University
Ames, IA 50011
1-515-294-1580

weiss@iastate.edu

Robyn R. Lutz
Dept. of Computer Science
Iowa State University & JPL/Caltech
Ames, IA 50011
1-515-294-3654

rlutz@iastate.edu

ABSTRACT

“... [S]oftware remains NIT’s [Networking and Information Technology] greatest weakness. Although reliable and robust software is central to activities throughout society, much software is brittle, full of bugs and flaws. Software development remains a labor-intensive process in which delays and cost overruns are common, and responding to installed software’s errors, anomalies, vulnerabilities, and lack of interoperability is costly to organizations throughout the U.S. economy.” “... [T]he science of software development must be a focus of Federal NIT R&D. As software’s complexity continues to rise, today’s design, development, and management problems will become intractable unless fundamental breakthroughs are made ...”[2]

Current understanding of software development—largely based on anecdotes—is inadequate for this “science of software development.” Achieving the deeper understanding needed to transform software production requires collecting and using evidence on a large scale. This paper proposes some steps toward that outcome.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics—*process metrics, product metrics.*

General Terms

Management, Measurement, Documentation, Design, Reliability, Experimentation, Verification.

Keywords

Evidence-based, software production, data-driven improvement.

1. INTRODUCTION

The purpose of this paper is to make the case for evidence-based understanding and improvement of software production. Such evidence should be based on standardized, widely collected data, and not simply on unrepeatable and unrepeatable experiments and anecdotes. Evidence-based approaches have enabled rapid,

startling advances in manufacturing, agriculture, medical technology, and other fields. For example, Gawande describes how, in 1900, over 40% of a family’s income in the U.S. went to food. Farming was labor-intensive and engaged almost half the workforce. Productivity was low, and farmers viewed any change in their practices as too risky. The invisible hand of market competition was not improving the situation. A turning point came when the U.S. Department of Agriculture initiated a pilot project with a single farmer in 1903. Unlike other farmers in the area, he made a profit in what was the worst year for cotton in a quarter century. Experiments, pilot projects and demonstrations followed. Crop forecasting became possible; new hybrids and mechanization techniques moved from research into practice; and radio broadcasts on 163 stations supplied timely information so that farmers themselves could make rational planting decisions. Gawande describes the resulting transformation: “It shaped a feedback loop of experiment and learning and encouragement for farmers across the country. The results were beyond what anyone could have imagined. Productivity went way up. . . . Prices fell by half. By 1930, food absorbed just twenty-four per cent of family spending and twenty per cent of the workforce” [10].

Transformation of software production on this scale, at similar speed is likewise possible, but will require a similar investment in experiments and pilot projects, and in collection and analysis of detailed data. It will require better mechanisms to supply industry with information about what works and what doesn’t. The transformation of agriculture was not a result of relying on just a few breakthrough ideas, but rather a result of trying many different approaches and selecting those that evidence showed were useful. Achieving improved understanding of software production similarly needs evidence-based investigation of hypotheses. We can then provide timely feedback to software producers. Anecdotes and case studies, while useful, serve as starting points to stimulate the collection of evidence.

We use *software production*, instead of *software development*, to emphasize that we refer not only to software creation, whether starting fresh or reusing existing software, but also to software sustainment as the system evolves throughout its life. We understand software production to be (usually) the work of large, multi-disciplinary, geographically distributed, international teams of individuals and organizations creating and evolving a variety of artifacts, sometimes over decades. Artifacts may be formal with well-defined semantics such as models, specifications, and code; may be semi-formal with prescribed format and content such as requirements and design documents; or may be informal such as instant messages, wikis and email.

Copyright 2010 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
FoSER 2010, November 7–8, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-4503-0427-6/10/11...\$10.00.

We posit that current knowledge of software production is largely anecdotal, and that transforming software production requires developing a deeper understanding of it that must be based on evidence, both quantitative and qualitative. This paper proposes a strategy for gathering necessary evidence, and identifies some questions on which to focus to turn evidence into understanding that drives improvement. We are not the first to suggest that evidence is needed, or which methods to use to collect it, but we have not seen elsewhere a call to develop a wide-scale strategy for collecting and using evidence to put software production on a firmer basis [1], [2]. Nor have we elsewhere seen a call to collect such detailed data about software production.

Acquiring such knowledge will help producers make rational decisions that in turn can reduce software failures and make the systems on which we depend more reliable. SIGSOFT's Software Engineering Notes Risks column keeps us informed, entertained, and sometimes aghast at the variety of failures caused by software [4]. Kitchenham et al. discuss how decision makers could use "current best evidence from research" to improve software production [13]. What we don't have is evidence of systemic software production problems that lead to failures, of the distribution of such problems across software systems, and of the effects of various solutions that have been tried. The consequences of failing to collect and use evidence are, at best, a continuing sequence of flawed programs that deliver much less than was promised for much more time and effort than was expected and, at worst, programs that fail in operation with catastrophic effect.

With the ever-increasing demand for larger, more complex software-intensive systems and the current potential for improvement in our ability to build them, shaping a "feedback loop of experiment and learning and encouragement" [10] can propel our ability to produce software, much as it did agricultural productivity in a previous century.

2. ACQUIRING KNOWLEDGE

Developing an understanding of software production entails examining the decisions and assumptions that are reflected or recorded in artifacts. This includes examining the individuals, organizations, disciplines, and automated mechanisms that make the decisions and create and evolve the artifacts; examining the rationale, assumptions, and mental models and representations underlying the decisions; and examining the representations of decisions, rationale, and assumptions in various artifacts. We must observe how decisions, assumptions, and other forms of knowledge flow among the artifacts and artifact versions, as well as among individuals, disciplines, and organizations. We must study how decisions and assumptions are forgotten, misinterpreted, corrupted, and invalidated over time.

There are at least three types of knowledge that we need to predict the likelihood that we will be able to build and sustain successfully a particular type of system. First is *domain knowledge*, the knowledge of experts in a particular domain, e.g., pilots and aeronautical engineers in the domain of aircraft, physicians and nurses in the domain of health IT, physicists and chemists in the domain of sensor networks.

The second type is *software engineering knowledge*, which comprises knowledge of the processes, techniques, and tools used to produce the software. We must also know how well the producers of a system have mastered that knowledge. Estimating

the impact of these different factors is known to be a difficult research problem, but some results exist and can be used, such as those underlying [5], [6]. We may think of the domain and software engineering knowledge as primarily a summary of how good we are at building systems of certain types. Continuing baseline studies of specific domains will provide us with the evidence needed to make estimations and to identify areas where improvement is needed.

The third type is *knowledge of a particular software system*, which includes knowledge recorded in the artifacts discussed earlier, and which defines a particular software system being built or modified. This knowledge may be known to developers and sustainers, but personnel turnover can lead to its loss. A key question is whether or not the knowledge about the system(s) reflects the (haphazard) history of its development or has been rationalized, in the sense of [3], to support its future use.

Understanding the development, evolution and expression of the three types of knowledge is key to understanding how to improve software production. This is because software production is largely a process of making changes to artifacts. Whether building a new system, or sustaining an existing one, we proceed by making incremental changes. As knowledge about a system is lost, the effort required to make a change increases dramatically. System disorder and code interdependencies increase the probability of introducing errors when making a change. One might say that the entropy of the software production knowledge base, including the code, increases drastically.

There are currently no standardized, systematic techniques and mechanisms for assembling evidence that translates into knowing how much confidence we should have that we can build a system, or into predicting how well that system will meet its requirements if we do build it. Neither are there techniques and mechanisms for collecting evidence about what happens when we try to improve the basis for our confidence, that is, make changes in the way we build and sustain systems. If there were, we could start to think of software engineering as a traditional engineering discipline. We would also have a firm basis for knowing what problems to attack to make substantial progress in developing software engineering, and for estimating the value of particular investments in new software engineering R&D.

A few attempts have been made to establish such standardization within individual organizations [7], [8]. A key research (and technology) issue is how to extend such attempts to cover entire industries and market segments. Could we do this for all software developed for a government agency? For all software developed for use in the health care industry? For all software developed for use in a nation? We should be working on this issue now.

Collecting evidence is a critical contribution of this effort to improve software production. *Using* this evidence to reason about the state of software and what works and what doesn't, e.g., by hypothesis testing, pilot studies, and assembly of examples of successful innovations, is the second anticipated contribution of this proposal. *Educating and applying* the resulting new knowledge to improve next-generation workforce skills and production practices is where we can achieve big gains based on evidence. We can transition improved understanding to a wide set of stakeholders via updated course curricula, tutorials at conferences with high attendance by professional developers, and proposals to standards committees. Federally sponsored industry/academia experimental trials will be critical. We should

establish a strategy for making informed decisions by collecting, reasoning about, and disseminating evidence on which to base the decisions.

2.1 Recognizing Buildability and Correctness

Producing correct software confidently, consistently, and systematically requires explicit statement of the three types of knowledge. Changes in requirements, people, and technology during software production all complicate the job and require that the knowledge not only be made explicit, but be kept current and correct. Iterative development, including the spiral model [9], may be viewed as an attempt to do just that: create what you are sure of first, show it to the stakeholders, especially the customer/user, incorporate feedback, make it explicit (and rational), then proceed to the next iteration. Product line engineering also may be viewed as attempting to gather, make explicit, and thereby reuse the knowledge needed to produce a family of systems efficiently [12].

Again, we currently have little evidence that such techniques lead to improvements in our ability to produce software. There are few common evidence bases that industry can use to justify the initial investment costs in employing such techniques, or that universities can use to determine what to teach.

2.2 Recognizing Improvement

To gauge improvement, we must first characterize the starting point. Many factors make profiling the current state difficult. For example, the unavailability of data because of proprietary concerns is often cited as a major obstacle to empirical research. Consortia have had some past success in reducing the startup cost of data acquisition, and may provide a model of collaboration.

Determining whether a change to existing software production practice yields evidence of improvement requires a reasoned way to derive an identification of the data to be collected from the hypothesis to be investigated. To acquire the data, an *integrated instrumentation* of the product and the production process is needed. Such instrumentation also enables families of experiments to be run [11] more cost-effectively.

Improvement is heavily context dependent. What will lead to better software in one environment or domain may not have any positive impact in another. Consequently, we propose a broader use of sensitivity analysis to define which variations among domains (e.g., real-time, interactive, power-constrained) perturb the results of the experiments and may limit the applicable range of the candidate solutions. Results need to be parameterized in terms of their findings, much as research results on a new pharmaceutical drug are clearly constrained by the population on which it was tested.

Finally, what works for a project at one point in time may not work in the future, because of evolution of the system, changes in the workforce, or evolution in the market. Production of sustainable software systems is continual, while current best practices tend to focus on the pre-deployment state of the software. We thus propose investigation of what sorts of evidence are needed to support claims for improvement in operational systems. Databases of defect, near-miss and accident reports, for example, are a rich source of information about operational experience that have not been adequately incorporated into recommendations for future, similar systems.

3. ACHIEVING AN ENGINEERING BASIS

Many of the decisions and assumptions that comprise the three types of knowledge are recorded imprecisely and informally, in unnecessarily complex ways, and are subject to misinterpretation and misunderstanding. Many are not explicitly recorded. They may exist only in the minds of a small number of individuals for possibly a limited period of time, and are communicated verbally, in unmaintained notes, or in email. Events that invalidate the decisions and assumptions can go unnoticed too readily. Rationale is rarely recorded in a useful manner that can be maintained as software evolves. As an example, every time a programmer writes code, s/he makes decisions about what will be easy to change and what will be hard to change. Sometimes these decisions are made, reviewed, and documented before code is written, but often decisions are made when the code is written and are never documented or reviewed.

Software producers need both to preserve knowledge and to convey knowledge for future use and reuse. Mechanisms to preserve knowledge should be formal enough to allow machine readability so that the knowledge can be automatically maintained without incurring much added cost. Mechanisms to convey knowledge must be available both as "pull" technologies (to query and retrieve stored information) and as "push" technologies (pro-actively to communicate, educate and cause to be remembered needed information).

Making software production an engineering discipline means identifying and standardizing (1) the types of knowledge that we need to produce software, (2) the form in which the knowledge is expressed and preserved, (3) the manner in which the knowledge is communicated, and (4) the way in which future software engineers are educated about what the knowledge is and the process for recording, maintaining, and using it. For each of the preceding points we may formulate questions or form hypotheses regarding issues such as exactly what types of knowledge we need, how should it be expressed, etc. For each question to be answered or hypothesis to be tested, we seek to collect the appropriate data in the appropriate environment, as was done for the experiments in agriculture described in [10].

3.1 Some Initial Hypotheses

Collecting and analyzing detailed data enables us to increase our understanding of software production by testing the following initial hypotheses: (1) Decisions are recaptured many times, e.g., in needs and requirements documents, requirement specifications, design documents, source code, test specifications, often in slightly varying forms. (2) Artifacts other than source code are not maintained over time as decisions, assumptions, and rationale evolve, leading to their disuse, and to source code becoming the definitive artifact. (3) Efforts to rediscover lost decisions, assumptions, and rationale are expensive and ineffective, especially when they must be based on using existing source code for the recovery, most especially when the originators of that code are not available. (4) Most software production is mostly redevelopment, using existing decisions and assumptions, while changing just a few of them. (5) Most software production does not now include systematic planning for change, especially over long system lifetimes.

3.2 Conducting Tests

We believe these hypotheses are testable through the collection of evidence from software production from the start of production

until the time that the system is retired. We do not think that one can conduct controlled experiments for large-scale systems. Just as the medical and other communities collect evidence using baseline studies on large populations in which convincing trends appear, we can perform long-term baseline studies that show convincing trends in software production. For example, data showing the correlation of smoking with lung cancer was collected over decades and became increasingly convincing. Eventually, scientists identified the causal mechanisms at work.

Similarly, we can collect data about software artifacts and the organizations and individuals who create and evolve them that we can use to answer relevant questions and test relevant hypotheses. We can detect trends, identify correlations, and then find causal links that we can use to improve our software production capabilities. As noted before, this has been done on smaller scales than we are proposing [7], [8], but has not been tried on much larger scales. Standardizing, collecting, maintaining, making available, and analyzing the data automatically and unobtrusively are not easy tasks, nor are they tasks that we can do right now. However, other professions and industries have learned to do this as a matter of course, and our society benefits from it. It is time for software engineering to start the process.

3.3 Benefits

In addition to acquiring a deeper understanding of software production, we will learn how to capture data about it that we are not able to now. Shining the light of consistent, standardized, repeatable measurement on software production may itself lead to improvement. Asking for measures of recorded knowledge requires that there be recording and methods for retrieving what was recorded. Acquiring such knowledge will enable us to exploit advances in computer science, hardware, social media, and software engineering. It will help us create software development methods, processes, and tools that are well suited to the goals of organizations engaged in software production, including goals such as reducing total ownership cost, risk, and schedule. We will be better able to take advantage of human strengths and accommodate human weaknesses. Further, an evidence-based approach will provide greater confidence that we have done so.

Some may argue that “imposing” measurement will increase cost and time to develop and sustain systems. It may be true that in some cases, particularly where there is little recording of knowledge now, that some initial development costs will be incurred. However, studies of large organizations, such as [7], indicate small overhead in measurement costs, partly because much of the quantitative data collection and analysis is automated.

4. SUMMARY

Because “reliable and robust software is central to activities throughout society” [2], our known problems in software production impose costs throughout society. We need to put software engineering on an evidence basis, as other fields have done. Our goal is to improve software production based on a better understanding of it. We need to understand the current state of the practice, and the effects of trialed improvements, and to feed back this understanding to the software engineering community. While we need to do this for the software production industry as a whole, we must start on a smaller scale to understand the problems better.

Some of the key questions to answer to implement this strategy translate into the steps needed to start collecting and applying evidence: (1) What are the areas of knowledge that are critical to software engineering and how should we measure our effectiveness in defining and using them? (2) How do we standardize the collection of software measurement data across different organizations? (3) What will be the incentives for different organizations to collect the same types of data and provide them for analysis and archiving? (4) Who will be the keeper of the data? (5) How will data be made available to researchers and practitioners who want to use it in different ways? (6) Who will sponsor the research and development needed to answer the preceding questions? Similar questions have been answered, and the answers used to drive rapid, sustained progress, in fields as diverse as agriculture, genetics, automotive engineering, particle physics, health care, and semiconductor manufacturing. Software engineering should be no different.

5. ACKNOWLEDGMENTS

Thanks to Grady Campbell and Jon Bentley for helpful comments. We acknowledge NSF grants 0541163 and 0916275, and support from ONR and DDR&E/S&T/IS.

6. REFERENCES

- [1] Jackson, D., Thomas, M. and Millett, L.I., Eds. 2007. *Software for Dependable Systems: Sufficient Evidence?* Committee on Certifiably Dependable Software Systems, National Research Council.
- [2] President’s Council of Advisors on Science and Technology. 2007. *Leadership Under Challenge: Information Technology R&D in a Competitive World*,
- [3] Parnas, D. L. and Clements, P. C. 1986. A rational design process: how and why to fake it. *IEEE Trans. on Software Eng.* SE-12 (Feb. 1986), 251-257.
- [4] SIGSOFT Software Engineering Notes, Risks to the Public.
- [5] Boehm, B. 1981. *Software Engineering Economics*.
- [6] COCOMO, http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html
- [7] Hackbarth, R., Palframan, J., Mockus, A., Weiss, D. 2010. Assessing the state of software in a large enterprise, *Empirical Software Eng* 15, 3 (June. 2010), 219-249.
- [8] Grady, R., Caswell, D. 1987. *Software Metrics: Establishing a Company-Wide Program*. Prentice Hall.
- [9] Boehm B. 1986. A spiral model of software development and enhancement. *SIGSOFT SEN* 11, 4 (Aug. 1986), 14-24.
- [10] Gawande, A., 2009. How the Senate bill would contain the cost of health care. *The New Yorker* (Dec. 17, 2009).
- [11] Basili, V, Caldiera, G., McGarry, F., et al., 1992. The Software Engineering Laboratory: an operational software experience factory. *Proc of 14th ICSE*, 370-381.
- [12] Weiss, D. and Lai, C.R.T. 1999. *Software Product Line Engineering*. Addison-Wesley.
- [13] Kitchenham, B., Dyba, T, Jørgensen, M. 2004. Evidence-based Software Engineering. *Proc. of 26th ICSE*.