By Victor R. Basili *and*
Marvin V. Zelkowitz

EMPIRICAL STUDIES
TO BUILD A SCIENCE OF COMPUTER SCIENCE

We learn to develop software by building, testing, and evolving models.

Computer science has been slow to adopt an empirical paradigm, even as practically all other sciences have done so. Since the time of Aristotle 2,500 years ago, the "natural sciences" (such as physics and biology) have observed nature in order to determine reality. In computer science this rarely happens [12]. Experimentation generally means the ability to build a tool or system—more an existence proof than experiment. While experiments are often organized around the evaluation of algorithms (such as performance and workflow), little has been done that involves humans (such as the development of high-performance codes and

test sets based on specified test criteria). But any future advances in the computing sciences require that empiricism takes its place alongside theory formation and tool development.

Here, we explore how to apply an empirical approach toward understanding important problems in software development. Understanding a discipline demands observation, model building, and experimentation. Empirical study is about building models that express our knowledge of the aspects of the domain of greatest interest to us (such as those that cause us the most problems). Learning involves the encapsulation of knowledge, checking that our knowledge is correct, and evolving that knowledge over time. This experimental paradigm is used in many fields, including physics, medicine, and manufacturing. Like other sciences, many disciplines within computer science (such as software engineering, artificial intelligence, and database design) likewise require an empirical paradigm.

Because software development is a human-based activity, experimentation must deal with the study of human activities. Experimentation, in this context, involves evaluating quantitative and qualitative data to understand and improve what the development staff does (such as defining requirements, creating solutions to problems, and programming).

Experimentation requires real-world laboratories. Developers must understand how to build systems better by studying their own environments. Researchers need laboratories in which to observe and manipulate development variables, provide models to predict the cost and quality of the systems, and identify what processes and techniques are most effective in building the system under the given conditions to satisfy a specific set of goals. Research and development have a synergistic relationship that requires a working relationship between industry and academe.

If we take the example of the development of software systems, the developer needs evidence of what does and does not work, as well as when it works. A software organization must be able to answer questions like: What is the right combination of technical and managerial solutions for my problem and environment? What is the right set of processes for my business? How should they be modified? How do we learn from our successes and failures? And how do we demonstrate sustained, measurable improvement? All must be supported by empirical evidence.

For example, a specific development question might be: When is a peer review more effective than functional testing? A number of studies [4, 7] suggests that under specified conditions, peer review is more effective than functional testing for faults of omission and incorrect specification and that functional testing is more effective for faults related to numerical approximations and control flow. In some situations, the cost of the review meeting may outweigh the benefits of the meeting [11].

Empirical evidence sometimes supports and sometimes does not support intuition. When it does, one might feel that empirical evidence is unnecessary. This is fallacious reasoning since the way we build knowledge is through studies, first recognizing relationships (that is, A is more effective than B under the following conditions), then evolving the relationship quantitatively (such as A provides a 20% improvement over B). Demonstrating that gravity exists satisfies our intuition, but being able to measure it adds detail to our understanding. When the evidence does not support our intuition, we must change our mental models and identify hypotheses and conditions for why it doesn't.

Software Engineering Laboratory

We write this from the point of view of our personal experience motivating and encouraging experimental research in computer science. We start with an early example of empirical studies—the NASA Goddard Software Engineering Laboratory (SEL) from 1976 to 2002—where the goal was to observe project development as a means to better understand, control, track, and improve software development for ground-support systems.

Basic scientific and engineering concepts were adapted to the software engineering domain. The relevant elements were captured in the quality improvement paradigm (QIP), which functioned as an evolutionary learning approach for using packaged knowledge to better understand how to build systems [3]. The process of building, refining, and testing models was encapsulated in a model called the Experience Factory (EF) [2]. Data was collected and interpreted via the Goal Question Metric (GQM) Approach [5]. Product development was monitored through observation and data collection so adjustments could be made in real time and synthesized into models. The results were packaged and deployed in future projects.

The SEL conducted two major classes of study: controlled experiments and case studies. The controlled experiments were applied to new techniques to identify key variables, study programming in the small, check out methods for data collection, and reduce the risk of applying the new technique on live projects. Case studies were used for live projects to check the scalability of the technique and understand how to adapt, tailor, and integrate the techniques.

The use of QIP, GQM, and EF enabled NASA to improve its software over the lifetime of the SEL. Additional studies were carried out in classroom settings or in training studies at NASA. Many processes were tried, sometimes over several projects, which were carefully monitored to reduce risk and determine where optimization was possible.

Some quantitative results (see Figure 1) show a dramatic increase in the reuse of source code and decrease in defects from 1985 to 1995 across many SEL projects. At the same time, costs decreased 55%, then of the remaining 45% a further 42% decrease of that. An independent study in 1993 estimated that from 1976 to 1992, the functionality of these systems increased fivefold [6]. The SEL was instrumental in not only demonstrating that large-scale empirical studies could be run but in developing results that showed the value of cleanrooms, inspections, Ada, and the reuse of source code; it also produced many documents for managing software development and for collecting and using data useful to the software development community.

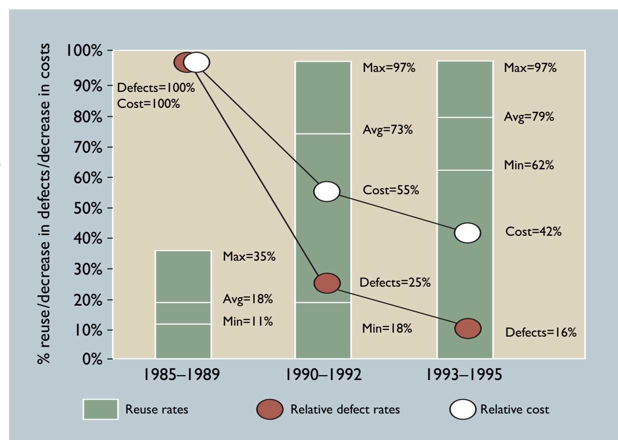

Figure 1. Increases in reuse rates and decreases in defect rates and costs in the SEL.

The key point of this example is that empiricism is needed to build knowledge about a domain and that experimentation focused on people is possible and necessary. Only knowledge gained through experimentation can lead to new opportunities (such as improved software development). The process is slow, but each study advances our knowledge.

High-End Computing

While the need for experimentation and data collection in software engineering is an accepted part of software engineering culture [1, 10] and experimentation is accepted as an increasingly important requirement for publication [12], the need for experimentation is not the case for computer science in general. Experimental concepts can and must be applied in a variety of computer science environments. Since 2003, we have been studying high-end computing (HEC) where multiple processors are used to achieve computational rates in teraflops (10^{12} floating point operations per second), and the goal of the Defense Advanced Research Projects Agency High Productivity Computing System program (HPCS, www.highproductivity.org/) is to achieve petaflops (10^{15} flops) speed by 2012. Given the need to produce results, the DARPA-funded theme of HPCS was "Time to Solution = Development Time + Execution Time." Prior emphasis in the domain was on execution time, that is, to run benchmark programs that exercise hardware efficiently, even though most programs fail to achieve the performance of these highly tuned benchmarks.

In addition to the hardware question of how fast these machines are able to execute, the HPCS program wants to know if the machines can be programmed effectively. How does an HEC environment affect the development of an HEC program? What is the cost and benefit of applying a particular technology (such as MPI, OpenMP, UPC, Co-Array Fortran, XMTC, and StarP) in the development of HEC programs? What are the relationships among the technologies, work flows, development cost, defects, and performance? Can we build predictive models of these relationships? And what trade-offs are possible and desirable?

Understanding, predicting, and improving development time requires empirical methods to properly evaluate programmer, as well as machine, performance. We need theories, hypotheses, and guidelines that allow us to characterize, evaluate, predict, and improve how an HEC environment—hardware, software, developer—affects development of these high-end computing codes. Running a variety of studies involving experts and novices collecting empirical data generates experiential knowledge in the form of predictive models and heuristics useful for identifying knowledge.

As with the SEL, the approach is to develop knowledge based on empirical studies. But while the SEL studied a software-development domain involving professional programmers, within the HPCS program computer scientists interact mainly with computational scientists and physicists. They are more interested in answering science questions with the help of a computer and less interested in how well the computer programs work. Very different drivers (such as context variables) compared to the conventional computing domain operate in the HEC domain where empirical studies are used to try to understand these drivers.

Our HPCS research model is more complex than

what we used in the SEL (see Figure 2). Pilot studies involving single-programmer assignments to identify the important variables expose data-collection problems, characterize workflows, and debug experimental designs. We ran studies 2004–2006 to calibrate an effort model against actual effort using small, limited-scale assignments (such as array compaction, parallel sorting, LU decomposition) developed in graduate courses at a number of universities.[1] Observational studies simulated the effects of the treatment variables in a realistic environment, validating the data-collection tools and processes. These studies led to replicated controlled experiments of single programmers to increase confidence in the results and provide hypotheses about novice developers (such as controlled experiments comparing the effort required to develop code in MPI vs. OpenMP).

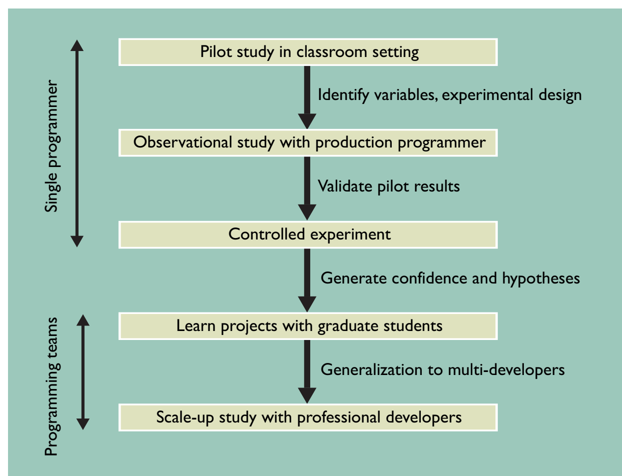

Figure 2. HPCS experimentation research model.

Team projects with graduate students in the participating universities studied scale-up and multi-developer workflows. Full-scale applications dealt with nuclear simulation, climate modeling, and protein folding developed at the five Advanced Simulation & Computing Centers[2] or run at the San Diego Supercomputer Center. Compact applications were developed for bioinformatics, graph theory, and sensor networks involving combinations of kernels developed by experts testing key benchmarks. They are used to understand multiprogrammer development workflows.

Mixed throughout these studies are interviews we conducted with developers and users in a variety of environments. We collected "folklore," or unsupported notions, stories, or sayings widely circulated, from practitioners in government, industry, and academic research labs. They are used to generate hypotheses that guide further experiments and case studies. Formalizing folklore involves four activities:

- Identify the relevant variables and terminology and simple relationships among variables, looking for consensus or disagreement;
- Identify the context variables that affect their validity, using surveys and other mechanisms;
- Develop hypotheses that can be specified and measured; and
- Verify the hypotheses via experimentation.

The end result of this research was to package our acquired knowledge by:

- Identifying the relevant variables, context variables, programmer workflows, and mechanisms for identifying variables and relationships;
- Identifying measures for the variables that can be collected accurately or proxies that can be substituted for these variables; and
- Identifying the relationships among the variables and the contexts in which the relationships are true.

Although validity may be threatened by using students in classroom experiments, there is support for the following hypotheses among this (interviewed) population [9]:

- OpenMP offers greater speedup for novices in a shorter amount of time when the problem is more computationally based than communication based;
- OpenMP saves 35%–75% of effort over MPI on most problems, UPC/CAF saves approximately 40% of effort over MPI, and XMT-C saves approximately 50% of effort over MPI;
- The programming model has a greater influence on performance than on experience with the problem to be solved; and
- When performance is the goal, experts and students spend the same amount of time, but experts produce significantly better performance.

On the other end of the size spectrum, characterizing processes based on full-scale applications being developed at the ASC Centers and run at the San Diego Supercomputing Center, we identified three classes of user:

Marquee. Running at very large scale, often using

[1]University of California, San Diego, University of California, Santa Barbara, University of Hawaii, Iowa State University, University of Maryland, Mississippi State, MIT, and University of Southern California.
[2]California Institute of Technology, University of Chicago, University of Illinois, Urbana-Champaign, Stanford University, and University of Utah.

the full system and supported by a consultant to help improve performance;

Normal. Typically using 128 to 512 processors and less likely to tune their codes; and

Novices. Just learning parallel programming.

Determining inputs to systems can take weeks and are themselves research projects. Addressing the challenge of debugging (such as in modules) may work in isolation but fail when connected together. Programs may work on 32 processors but break down on 64 processors; it is especially difficult to debug failures on hundreds of processors. Performance is treated as a constraint, not a goal to be maximized; that is, performance is important until it is "good enough" for the developers' machine allocation. Portability is a must, as new computers are being developed every few years; developers can't commit to technologies unless they know they will be there on future platforms. Many users prefer not to use performance tools because they involve complications scaling to large processors, have difficult-to-use interfaces, involve steep learning curves, and provide too much detail. Moreover, codes are multi-language and run on remote machines. Many software tools simply don't work in this environment. There is extensive reuse of libraries but little or no reuse of frameworks [8].

Developers have much to learn about the development of high-end codes. The folklore is often inconsistent, because context is assumed rather than made explicit. What is true for marquee users is not also true for novices. What is true for one subapplication area is not necessarily also true for another area. Organizations and domains have different characteristics, goals, and cultures, and stakeholders have different needs and profiles. Computer scientists must understand when certain relationships hold and when they break down. Exploring these environments and experimenting with a variety of strategies is the only way to fully understand these issues.

Conclusion

Experimentation is fundamental to any engineering or science discipline. The interplay between theorists and experimentalists is the way we learn, building, testing, and evolving models. It is the most dependable way physics, medicine, and manufacturing have evolved as disciplines. The learning process is continuous and evolutionary.

Computer science involves people solving problems, so computer scientists must perform empirical studies that involve developers and users alike. They must understand products, processes, and the relationships among them. They must experiment (human-based studies), analyze, and synthesize the resulting knowledge. They must package (model) that knowledge for further development. Empirical studies involve many individuals and require laboratories to do good experimentation. Interaction among industrial, government, and academic organizations is essential. C

References

1. Arisholm, E., Sjoberg, D., Carelius, G., and Lindsjom, Y. A Web-based support environment for software engineering experiments. *Nordic Journal of Computing 9,* 3 (Sept. 2002), 231–247.
2. Basili, V., Caldiera, G., McGarry, F., Pajersky, R., Page, G., and Waligora, S. The Software Engineering Laboratory: An operational software experience factory. In *Proceedings of the 14th ACM/IEEE International Conference on Software Engineering* (Melbourne, Australia, May 11–15, 1992), 370–381.
3. Basili, V. and Green, S. Software process evolution at the SEL. *IEEE Software 11,* 4 (July 1994), 58–66.
4. Basili, V. and Selby, R. Comparing the effectiveness of software testing strategies. *IEEE Transactions on Software Engineering 13,* 12 (Dec. 1987), 1278–1296.
5. Basili, V. and Weiss, D. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering 10,* 3 (Nov. 1984), 728–738.
6. Basili, V., Zelkowitz, M., McGarry, F., Page, J., Waligora, S., and Pajerski, R. Special report: SEL's software process-improvement program. *IEEE Software 12,* 6 (Nov. 1995), 83–87.
7. Boehm, B. *Software Engineering Economics.* Prentice Hall, Englewood Cliffs, NJ, 1981.
8. Carver, J., Hochstein, L., Kendall, R., Nakamura, T., Zelkowitz, M., Basili, V., and Post, D. Observations about software development for high-end computing. *Cyberinfrastructure Technology Watch Quarterly 2,* 4A (Nov. 2006), 33–38.
9. Hochstein, L., Carver, J., Shull, F., Asgari, A., Basili, V., Hollingsworth, J., and Zelkowitz, M. Parallel programmer productivity: A case study of novice HPC programmers. In *Proceedings of SC05* (Seattle, Nov. 12–18). ACM Press, New York, 2005, 1–9.
10. Johnson, P., Kou, H., Agustin, J., Zhang, Q., Kagawa, A., and Yamashita, T. Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering* (Los Angeles, Aug. 19–29, 2004), 136–144.
11. Porter, A., Siy, H., and Votta, L. A survey of software inspections. *Advances in Computers, Vol. 42,* M. Zelkowitz, Ed. Academic Press, San Diego, 1996, 39–76.
12. Zelkowitz, M. Data-sharing enabling technologies. In *Empirical Software Engineering Issues: Critical Assessment and Future Directions, LNCS-4336.* Springer-Verlag, Berlin, 2007, 108–110.

Victor Basili (basili@cs.umd.edu) is a professor of computer science at the University of Maryland, College Park.
Marvin V. Zelkowitz (mvz@cs.umd.edu) is a professor of computer science at the University of Maryland, College Park.

This research was supported in part by Department of Energy contract DE-FG02-04ER25633 and Air Force grant FA8750-05-1-0100 to the University of Maryland.

The SEL study team includes Frank McGarry, Rose Pajerski, Jerry Paige, and Sharon Walagora. The HPCS study team includes Jeff Hollingsworth, Taiga Nakamura, Sima Asgari, Forrest Shull, Nico Zazworka, Rola Alameh, Daniela Suares Cruzes, Lorin Hochstein, Jeff Carver, Philip Johnson, Nicole Wolter, and Michael McCracken. Professors who allowed us to use their classes include Alan Edelman, John Gilbert, Mary Hall, Aiichiro Nakano, Jackie Chame, Allan Snavely, Alan Sussman, Uzi Vishkin, Ed Luke, Henri Casanova, and Glenn Luecke.

© 2007 ACM 0001-0782/07/1100 $5.00