Association for Computing Machinery
Advancing Computing as a Science & Profession

Ubiquity Symposium

The Science In Computer Science

Where's the Science in Software Engineering?

by Walter F. Tichy

Editor's Introduction

The article is a personal account of the methodological evolution of software engineering research from the '70s to the present. In the '70s, technology development dominated. An attitude of "seeing is believing" prevailed and arguing the pros and cons of a new technology was considered to be enough. With time, this advocacy style of research began to bother me. I thought that something was missing from a scientific approach, and that was the experiment. Controlled experiments and other empirical methods were indeed rare in software research. Fortunately, the situation began to change in the nineties, when journals and conferences began to demand evidence showing that a new tool or method would indeed improve software development. This evolution was gradual, but the situation today is that software research is now by and large following the scientific paradigm. However, the cost of experiments can be staggering. It is difficult to find professional programmers willing to participate in controlled studies and it is time-consuming and expensive to train them in the methods to be studied. I suggest an alternative to experiments with human subjects that can help produce results in certain situations more quickly.

Ubiquity, an ACM publication
March 2014

Ubiquity Symposium

The Science In Computer Science

Where's the Science in Software Engineering?

by Walter F. Tichy

"The fundamental principle of science, the definition almost, is this: the sole test of the validity of any idea is experiment." —Richard P. Feynman.

When I started studying in 1974 at Carnegie Mellon University (CMU), the question of whether computer science was a science never even entered my mind. During the '70s, the primary question in CS was "What can be automated"? It was an exciting time—almost weekly, something new and unexpected was taken on by computers. The first type setting programs appeared at the same time as computers began talking to each other via the Arpanet, the precursor of the Internet. As graduate students we experimented with electronic mail for the first time. Computer games became interesting enough so that we wasted lots of time on Dungeons and Dragons, even though the graphics were lousy. Parallel computers appeared (ILLIAC, Cray) and one was even built where I was studying (C.mmp). New programming languages appeared—Pascal, FP, Smalltalk, Ada, others. Could you get garbage collection to run in parallel with the program generating the garbage? The answer turned out to be yes. The Unix operating system showed a computer could help with programming, an insight that led to my life-long interest in software tools. Someone out on the "Cut" (CMU's lawn) was showing off with a magically sounding instrument, one of the first digital synthesizers. The AI folks were thinking they could make computers not only musical, but also intelligent. I needed a lot of luck to beat Berliner's Backgammon program. My wife trained with that program (she played a lot while I was working on my thesis), so after a while, I couldn't beat her either. CMU's computerized cheese co-op predated Groupon by three decades. Screen editors and raster displays turned heads while workers at Xerox invented icons, menus, mice, and personal computing. All in all, everything was new and computer science was charging ahead at a dizzying speed. It seemed there was not much use in arguing whether this was science or not.

But the question about the scientific method did come up, when we students were writing our thesis proposals. My adviser suggested: "We don't need experiments, because we're building everything, and therefore we understand exactly what it does." Indeed, processors and compilers at the time were easy to understand. I wrote a thesis that proposed a language for describing software architectures—how the modules fit together and how to handle multi-versioned software product lines. It was called a module interconnection language because fancy terms such as "software architecture" and "software product line" hadn't been invented yet. Looking back, the striking thing is how I evaluated the language: I used a few small examples (a compiler!) and a prototype to demonstrate that the language could be implemented. But mostly I argued. I argued how the language would meet the goal of describing software structures. If there were choices for language constructs, I stated the pros and cons and selected the best one based on examples and introspection. Because of my adviser's insistence, there was a whole chapter called "Design Rational," which contained all the arguments. A paper about this work later received the IEEE award for most influential paper. But nobody ever tried the language seriously and usability experiments weren't invented yet.

I went on to teach at Purdue University in Indiana and later at the University of Karlsruhe in Germany. Software research expanded and more and more papers were published. Over time, something began to bother me: How could you convince anyone that the tools invented by researchers helped the practicing software engineer? Would all these tools really reduce programming cost and eliminate bugs? Every researcher argued his or her invention would do just that, but I was less and less convinced. When you had only a few tools to choose from, you could try everything out. But the choices were becoming too numerous and the work involved in making up one's mind became overwhelming. So most of the ideas in the research papers were soon forgotten. I started to look for an alternative to advocacy.

I stumbled across the book by A. Chalmers, *What Is This Thing Called Science?*; my students and I devoured it. We organized a seminar around this book, along with B. Latour's *Science in Action: How to Follow Scientists and Engineers Through Society*. We learned about induction, deduction, Poper's falsificationism, and Kuhn's revolutions, and realized how central experimentation was to science. Then we decided to figure out just how good or bad the situation in software actually was: We did a bibliographic study [1]. We chose 400 CS research articles and carefully looked at those that made claims that would require experimental validation.[1] We found in our sample of the CS papers that should have had some form of validation, 40 percent had none at all, and in software engineering, the situation was worse. We compared with other sciences and found their fraction of papers with unvalidated claims was much, much lower. Also, the space devoted to validation in CS articles was comparatively small—20 to 30 percent. Obviously, something was wrong. How can you take a science seriously if it tests less than half of its claims?

Fortunately, the situation has improved tremendously since the 1990s. Other software researchers had the same qualms and slowly, research standards improved. Today, it is hardly possible to publish a research paper without some form of empirical validation. Journals and conferences have been started that are devoted to empirical work and methods. In software engineering as well as in other areas, such as in experimental algorithms or usability research, the scientific method is now being practiced.

If one wants to generate research results that say something about professional programmers (as all research in this area should), one should be using professional programmers as subjects. However, many software researchers use students instead. While student subjects are acceptable for preliminary studies, they differ from professional programmers in a number of crucial aspects such as experience and education. A notable exception is the paper by Arisholm et al [2]. The authors hired almost 300 professional programmers to test claims about pair programming, a technique whereby every line of production code is written by two programmers sitting side-by-side at the same terminal. The question was under which conditions this technique improved software reliability and programmer productivity. This is one of the best experiments I know about in software research and easily the one with the highest number of professional participants. The cost of hiring the professionals was about $330,000. While this sounds high for computer science, think about how much medical studies with thousands of participants cost, or the billions being spent on chasing the Higgs boson.

Unfortunately, large experiments like Arisholm's will remain the exception. Not only is the funding required substantial. It is also difficult to recruit that many professionals, even if they are paid. When developing software tools that automate part of software development, there are numerous choices to be made. It is out of the question to run large experiments for every one of these choices, because it would take too long. Big experiments like Arisholm's will be

[1] It makes no sense to require experimental validation of mathematical proofs, for example.

reserved only for crucial questions. If this is so, how should software researcher proceed? I have a suggestion for this dilemma: benchmarks.

Benchmarks consist of one or more sample problems with a metric for success. Benchmarks have been remarkably successful in several areas of Computer Science. For instance, any microprocessor today is tested on a set of benchmark programs. Nobody argues about the speed of a microprocessor. Instead, one runs a benchmark on the processor or a simulator. The set of programs in the benchmark is extended over time, to keep designers from overfitting their processors to the benchmark. Benchmarks also drive database management systems. In speech understanding and machine translation, everyone uses large databases of speech samples to determine the error rate of various techniques. The error rate is what is compared among research teams, and magically, the best techniques appear in the recognizers by many teams. Rapid progress is the result. Recall the Darpa Grand Challenge. The first benchmark was to drive a robot car along a desert road; the last benchmark was driving automatically in a simulated city environment. Progress was so rapid; in a space of about a decade reliable self-driving cars appeared and are now permitted to drive in real traffic. The RoboCup is another competition of this sort; it led to soccer playing, humanoid robots. Progress was surprisingly fast, with the rules being tightened every year. Here you can see how well they play.

Software research could benefit tremendously from benchmarks. Benchmarks can be tested repeatedly and quickly without requiring human subjects. They help weed out poor techniques quickly and direct attention to the successful ones. There is a certain upfront cost for constructing the benchmark, but that effort could be shared among many researchers. For instance, we have constructed a benchmark, NLRPbench, for software requirements, consisting of set of specifications for software. The benchmark is intended for comparing requirements analysis tools. Such tools detect and correct ambiguities or extract models out of the text. Concurrency benchmarks contain tricky concurrency bugs and have led to highly accurate detectors for race conditions. Suitable benchmarks could be developed for many areas in software research and substitute for controlled experiments with humans. In fact, benchmarking is a from of controlled experimentation.

In summary, there is no question now that software research is following the scientific paradigm. Empirical methods such as case studies, controlled experiments, analysis of real project data, meta studies, and others are being used to test claims and theories and explore the unknown. Benchmarking should be added to the arsenal of empirical methods in order to speed up progress.

References

[1] Tichy, Lukowicz, Prechelt, and Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software* 28, 1 (1995).

[2] Arisholm et al. Evaluating Pair Programming with Respect to System Complexity and Programmer Expertise. *Trans. On Software Engineering* 33, 2 (2007).

About the Author

Walter F. Tichy has been professor of Software Engineering at the Karlsruhe Institute of Technology (formerly University Karlsruhe), Germany, since 1986, and was dean of the faculty of computer science from 2002 to 2004. Previously, he was senior scientist at Carnegie Group, Inc., in Pittsburgh, Pennsylvania and served six years on the faculty of Computer Science at Purdue University in West Lafayette, Indiana. His primary research interests are software engineering and parallelism. He is currently concentrating on empirical software engineering, tools and languages for multicore computers, and making programming more accessible by using natural language for programming.

He earned an M.S. and a PhD in computer science from Carnegie Mellon University in 1976 and 1980, resp. He is director at the Forschungszentrum Informatik, a technology transfer institute in Karlsruhe. He is co-founder of ParTec, a company specializing in cluster computing. He has helped organize numerous conferences and workshops; among others, he was program co-chair for the 25th International Conference on Software Engineering (2003). He received the Intel Award for the Advancement of Parallel Computing in 2009. Dr. Tichy is a fellow of the ACM and a member of GI and the IEEE Computer Society.

DOI: 10.1145/2590528.2590529