Cybersquare

Walter F. Tichy
University of Karlsruhe

Should Computer Scientists Experiment More?

Computer scientists and practitioners defend their lack of experimentation with a wide range of arguments. Some arguments suggest that experimentation is inappropriate, too difficult, useless, and even harmful. This article discusses several such arguments to illustrate the importance of experimentation for computer science.

Do computer scientists need to experiment at all? Only if we answer "yes" does it make sense to ask whether there is enough of it.

In his Allen Newell Award lecture, Fred Brooks suggests that computer science is "not a science, but a synthetic, an engineering discipline."[1] In an engineering field, testing theories by experiments would be misplaced. Brooks and others seem troubled by the fact that the phenomena studied in computer science appear manufactured. Computers and programs are human creations, so we could conclude that computer science is not a natural science in the traditional sense.

The engineering view of computer science is too narrow, too computer-myopic. The primary subjects of inquiry in computer science are not merely computers, but information structures and information processes.[2] Computers play a dominant role because they make information processes easier to model and observe. However, by no means are computers the only place where information processes occur. In fact, computer models compare poorly with information processes found in nature, say, in nervous systems, in immune systems, in genetic processes, or, if you will, in the brains of programmers and computer users. The phenomena studied in computer science are much broader than those arising around computers.

Regarding the manufactured nature of computer phenomena (its "syntheticness"), I prefer to think about computers and programs as models. Modeling is in the best tradition of science, because it helps us study phenomena closely. For example, for studying lasing, one needs to build a laser. Regardless of whether lasers occur in nature, building a laser does not make the phenomenon of massive stimulated emission artificial. Superheavy elements must be synthesized in the lab for study, because they are unstable and do not occur naturally, yet nobody assumes that particle physics is synthetic.

Similarly, computers and software don't occur naturally, but they help us model and study information processes. Using these devices does not render information processes artificial.

A major difference from traditional sciences is that information is neither energy nor matter. Could this difference be the reason we see little experimentation in computer science? To answer this questions, let's look at the purpose of experiments.

0018-9162/98/$10.00 © 1998 IEEE

WHY SHOULD WE EXPERIMENT?

When I discuss the purpose of experiments with mathematicians, they often exclaim that experiments don't prove a thing. It is true that no amount of experimentation provides proof with absolute certainty. What then are experiments good for? We use experiments for theory testing and for exploration.

Experimentalists test theoretical predictions against reality. A community gradually accepts a theory if all known facts within its domain can be deduced from the theory, if it has withstood numerous experimental tests, and if it correctly predicts new phenomena.

Nevertheless, there is always an element of suspense: To paraphrase Edsger Dijkstra, an experiment can only show the presence of bugs in a theory, not their absence. Scientists are keenly aware of this uncertainty and are therefore ready to shoot down a theory if contradicting evidence comes to light.

A good example of theory falsification in computer science is the famous Knight and Leveson experiment,[3] which analyzed the failure probabilities of multiversion programs. Conventional theory predicted that the failure probability of a multiversion program was the product of the failure probabilities of the individual versions. However, John Knight and Nancy Leveson observed that real multiversion programs had significantly higher failure probabilities. In essence, the experiment falsified the basic assumption of the conventional theory, namely that faults in program versions are statistically independent.

Experiments are also used where theory and deductive analysis do not reach. Experiments probe the influence of assumptions, eliminate alternative explanations of phenomena, and unearth new phenomena in need of explanation. In this mode, experiments help with induction: deriving theories from observation.

Artificial neural networks are a good example of the explorative mode of experimentation. After having been discarded on theoretical grounds, experiments demonstrated properties better than predicted. Researchers are now developing better theories to account for these properties.

Traditional scientific method isn't applicable

The fact that—in the field of computer science—the subject of inquiry is information rather than energy or matter makes no difference in the applicability of the traditional scientific method. To understand the nature of information processes, computer scientists must observe phenomena, formulate explanations and theories, and test them.

There are plenty of computer science theories that haven't been tested. For instance, functional programming, object-oriented programming, and formal methods are all thought to improve programmer productivity, program quality, or both. It is surprising that none of these obviously important claims have ever been tested systematically, even though they are all 30 years old and a lot of effort has gone into developing programming languages and formal techniques.

Fallacy 1. Traditional scientific method isn't applicable.

Rebuttal: To understand information processes, computer scientists must observe phenomena, formulate explanations, and test them. This *is* the scientific method.

Traditional sciences use theory test and exploration iteratively because observations help formulate new theories that can be validated later. An important requirement for any experiment, however, is repeatability. Repeatability ensures that results can be checked independently and thus raises confidence in the results. It helps eliminate errors, hoaxes, and frauds.

The current level of experimentation is good enough

Suggesting that the current level of experimentation doesn't need to change is based on the assumption that computer scientists, as a group, know what they are doing. This argument maintains that if we need more experiments, we'll simply do them.

But this argument is tenuous; let's look at the data. In "Experimental Evaluation in Computer Science: A Quantitative Study,"[4] my coauthors and I classified 400 papers. We then continued considering those papers whose claims required empirical evaluation. For example, we excluded papers that proved mathematical theorems, because mathematical theory can't be proven by experiment.

In a random sample of all the papers ACM published in 1993, the study found that 40 percent of the papers with claims that needed empirical support had none at all. In software-related journals, this fraction was 50 percent. The same study also analyzed a non-computer-science journal, *Optical Engineering*, and found that the fraction of papers lacking quantitative evaluation was merely 15 percent.

Fallacy 2. The current level of experimentation is good enough.

Rebuttal: Relative to other sciences, the data shows that computer scientists validate a smaller percentage of their claims.

The study by Marvin Zelkowitz and Dolores Wallace (in this month's *Computer*) found similar results. When applying consistent classification schemes, both studies report that between 40 and 50 percent of software engineering papers were unvalidated. Zelkowitz and Wallace also surveyed journals in physics, psychology, and anthropology and again found much smaller percentages of unvalidated papers than in computer science.

Relative to other sciences, the data shows that computer scientists validate a smaller percentage of their claims. Some would argue that computer science at age 50 is still young and hence comparing it to other sciences is of limited value. I disagree, largely because 50 years seems like plenty of time for two to three generations of scientists to establish solid principles. But even on an absolute scale, I think it is scary when half of the nonmathematical papers make unvalidated claims.

Assume that each idea published without validation would have to be followed by at least two validation studies (which is a very mild requirement). It follows that no more than one-third of the papers published could contain unvalidated claims. The data suggests that computer scientists publish a lot of untested ideas or that the ideas published are not worth testing.

I'm not advocating replacing theory and engineering with experiment, but I am advocating a better balance. I advocate balance not because it would be desirable for computer science to appear more scientific, but because of the following principal benefits:

- Experimentation can help build a reliable base of knowledge and thus reduce uncertainty about which theories, methods, and tools are adequate.
- Observation and experimentation can lead to new, useful, and unexpected insights and open whole new areas of investigation. Experimentation can push into unknown areas where engineering progresses slowly, if at all.
- Experimentation can accelerate progress by quickly eliminating fruitless approaches, erroneous assumptions, and fads. It also helps orient engineering and theory in promising directions.

Conversely, when we ignore experimentation and avoid contact with reality, we hamper progress.

Fallacy 3. Experiments cost too much.

Rebuttal: Meaningful experiments can fit into small budgets; expensive experiments can be worth more than their cost.

Experiments cost too much

Experimentation clearly requires more resources than theory does. The first line of defense against experimentation is typically, "Doing an experiment would be incredibly expensive" or "To do this right, I would need hundreds of subjects, work for years without publishing, and spend an enormous amount of money." A hard-nosed scientist might respond, "So what?"

Instead of being paralyzed by cost considerations, such a scientist would first probe the importance of the research question. When convinced that the research addresses a fundamental problem, an experienced experimentalist would then plan an appropriate research program, actively looking for affordable experimental techniques and suggesting intermediate steps with partial results along the way.

For a scientist, funding potential should not be the only or primary criterion in deciding what questions to ask. In the traditional sciences, there is a complex social process in which important questions crystallize. These become the focuses of research, the breakthrough goals that open new areas.

For instance, the first experimental validation of general relativity—performed by Issac Eddington in 1919—was tremendously expensive and barely showed the effect. Eddington used a total solar eclipse to check Einstein's theory that gravity bends light when it passes near a massive star. This was a truly expensive experiment because it involved an expedition to Principe Island, West Africa, and also because the experiment pushed the limits of photographic emulsion technology. But it was important to test whether Einstein was correct.

Not many investigations are of a scope comparable to that for general relativity, but there are many smaller, still-important questions to answer. Experiments can indeed be expensive, but not all are prohibitively expensive. Meaningful experiments can fit in the budget of small laboratories. On the other hand, expensive experiments can be worth much more than their cost.

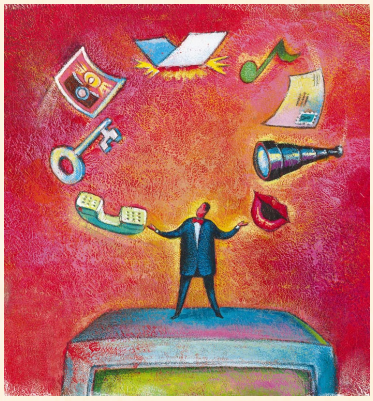
When human subjects are involved in an experiment, the cost often rises dramatically while the significance drops. When are expensive experiments justified? When the implications of the gained insights outweigh the costs.

A significant segment of the software industry converted from C to C++ at a substantial cost in retraining. We might ask how solidly grounded the decision to switch to C++ was. Other than case studies (which are questionable because they don't generalize easily and may be under pressure to demonstrate desired outcomes), I'm not aware of any solid evidence showing that C++ is superior to C with respect to programmer productivity or software quality.

Nor am I aware of any independent confirmation of such evidence. However, while training students in improving their personal software processes, my research group has recently observed that C++ programmers appear to make many more mistakes and take much longer than C programmers of comparable training—both during initial development *and* maintenance.

Suppose this observation is not a fluke. (Just as this article went to press, we learned that a paper by Les Hatton, "Does OO Really Match the Way We Think?" will appear in the May issue of *IEEE Software*, reporting strong evidence about the negative effects of C++.) Then running experiments to test the fundamental tenets of object-oriented programming would be truly valuable. These experiments might save resources far in excess of their cost. The experiments might also have a lasting and positive effect on the direction of programming language research. They may save not only industry money, but also research effort.

Fallacy 4. Demonstrations will suffice.



Rebuttal: Demos can provide incentives to study a question further. Too often, however, these demos merely illustrate a potential.

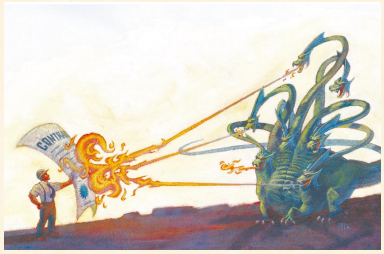
Interestingly, the software industry is beginning to value experiments, because results may give a company a three- to five-year lead over the competition. For instance, according to Larry Votta in a personal communication, Lucent Technologies estimates that it is presently benefiting from a five-year lead in software inspections based on a series of in-house experiments.

It is useful to check what scientists in other disciplines spend on experimentation. Testing pharmaceuticals is extremely expensive, but only desperate patients accept poorly tested drugs and therapies. In aeronautics, engineers test airfoils extensively and build expensive wind tunnels to do so. Numerical simulation has reduced the number of such tests, but it hasn't eliminated them.

In many sciences, simulation has become a useful form of experimentation; computer science might also benefit from good simulation techniques. In biology, for example, Edward Wilson calls the Forest Fragmentation Project in Brasil the most expensive biological experiment ever.[5] While clearing a large tract of the Amazon jungle, the researchers left standing isolated patches of various sizes (1 to 1,000 hectares). The purpose was to test hypotheses regarding the relationship between habitat size and number of species remaining.

Experimentation is widely used in physics, chemistry, ecology, geology, climatology, and on and on. *Scientific American* publishes experiments in every issue. Computer scientists need not be afraid or ashamed of conducting large experiments to explore important questions.

Fallacy 5. There's too much noise in the way.



Rebuttal: Fortunately, benchmarking can be used to simplify variables and answer questions.

Demonstrations will suffice

In his 1994 Turing Award lecture, Juris Hartmanis argues that computer science differs sufficiently from other sciences to permit different standards in experimentation, and that demonstrations can take the place of experiments.[6] I couldn't disagree more. Demos can provide proof of concepts (in the engineering sense) or incentives to study a question further. Too often, however, these demos merely illustrate a potential. Demonstrations critically depend on the observers' imagination and their willingness to extrapolate; they do not normally produce solid evidence. To obtain such evidence, we need careful analysis involving experiments, data, and replication.

For example, because the programming process is poorly understood, computer scientists could introduce different theories about how to build programs from requirements. These theories could then be tested experimentally. We could do the same for perception, human-machine interfaces, or human-computer interaction in general.

Also, computer science cannot accurately predict the behavior of algorithms on typical problems or on computers with storage hierarchies. We need better algorithm theories, and we need to test them in the lab. Research in parallel systems can generate machine models, but their relative merits can only be explored experimentally. The examples I've mentioned are certainly not exhaustive, but they all involve experiments in the traditional sense. They require a clear question, an experimental apparatus to test the question, data collection, interpretation, and sharing of the results.

There's too much noise in the way

Another line of defense against experimentation is: "There are too many variables to control and the results would be meaningless because the effects I'm looking for are swamped by noise." Researchers invoking this excuse are looking for an easy way out.

An effective way to simplify repeated experiments is by benchmarking. Fortunately, benchmarking can be used to answer many questions in computer science. The most subjective and therefore weakest part of a benchmark test is the benchmark's composition. Everything else, if properly documented, can be checked by the skeptic. Hence, benchmark composition is always hotly debated.

Though often criticized, benchmarks are an effective and affordable way of conducting experiments. Essentially, a benchmark is a task domain sample executed by a computer or by a human and computer. During execution, the human or computer records well-defined performance measurements.

Benchmarks have been used successfully in widely differing areas, including speech understanding, information retrieval, pattern recognition, software reuse, computer architecture, performance evaluation, applied numerical analysis, algorithms, data compression, logic synthesis, and robotics. A benchmark provides a level playing field for competing ideas, and (assuming the benchmark is sufficiently representative) allows repeatable and objective comparisons. At the very least, a benchmark can quickly eliminate unpromising approaches and exaggerated claims.

Constructing a benchmark is usually intense work, but several laboratories can share the burden. Once defined, a benchmark can be executed repeatedly at moderate cost. In practice, it is necessary to evolve benchmarks to prevent overfitting.

Regarding benchmark tests in speech recognition, Raj Reddy writes, "Using common databases, competing models are evaluated within operational systems. The successful ideas then seem to appear magically in other systems within a few months, leading to a validation or refutation of specific mechanisms for modeling speech."[7] In many of the examples I cited earlier, benchmarks cause an area to blossom suddenly because they make it easy to identify promising approaches and to discard poor ones. I agree with Reddy that "all of experimental computer science could benefit from such disciplined experiments."

Experiments with human subjects involve additional challenges. Several fields, notably medicine and psychology, have found techniques for dealing with human variability. We've all heard about control groups, random assignments, placebos, pre- and post-testing, balancing, blocking, blind and double-blind studies, and batteries of statistical tests. The fact that a drug influences different people in different ways doesn't stop medical researchers from testing.

When control is impossible, researchers will use case studies, observational studies, and other investigative techniques. Indeed, medicine offers many important

lessons on experimental design, such as how to control variables and minimize errors. Eschewing experimentation because of difficulties is not acceptable.

In so-called soft science, experimental results cannot be reproduced. The fear is that computer science will fall into this trap, especially with human subject testing. But experiments with human subjects are not necessarily soft. There are stacks of books on how to conduct experiments with humans. Experimental computer scientists can learn the relevant techniques or ask for help. The "How to Experiment" sidebar provides some starting points.

Progress will slow

Some argue that if everything must be experimentally supported before publication, then the flow of ideas would be throttled and progress would slow.

This is not an argument to be taken lightly. In a fast-paced field like computer science, the number of ideas under discussion is obviously important. However, experimentation need not have an adverse effect:

- Increasing the ratio of papers with meaningful validation has a good chance of actually accelerating progress. Questionable ideas would be weeded out more quickly, and scientists would concentrate their energies on more promising approaches.

Fallacy 6. Experimentation will slow progress.

Rebuttal: Increasing the ratio of papers with meaningful validation has a good chance of actually accelerating progress.

How to Experiment

For the reader eager to learn about the role of experimentation in general, I suggest the following literature:

Chalmers, A.F., *What Is This Thing Called Science?* The Open University Press, Buckingham, England, 1988. Addresses the philosophical underpinnings of the scientific process, including inductivism, Popper's falsificationism, Kuhn's paradigms, objectivism, and the theory dependence of observation.

Latour, B., *Science in Action: How to Follow Scientists and Engineers through Society*, Harvard University Press, Cambridge, Mass., 1987. Describes the social processes of science-in-the-making as opposed to ready-made science. Latour illustrates the fact-building and convincing power of laboratories with fascinating examples.

Basili, V.R., "The Role of Experimentation in Software Engineering: Past, Current, and Future." *Proc. 18th Int. Conf. Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., March 1996.

Frankl, P.G., and S.N. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow Testing," *IEEE Trans. Software Eng.*, Aug. 1993, pp. 774-787.

Brett, B., "Comments on The Cost of Selective Recompilation and Environment Processing," *ACM Trans. Software Eng. and Methodology*, 1995, pp. 214-216. A good example of a repeated experiment in compiling.

Denning, P.J., "Performance Evaluation: Experimental Computer Science at Its Best," *ACM Performance Evaluation Review*, ACM Press, New York, 1981, pp. 106-109. Argues that performance evaluation is an excellent form of experimentation in computer science.

Hennessy, J.L., and D.A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, San Mateo, Calif., 1990. A landmark in making computer architecture research quantitative.

Cohen, P.R., *Empirical Methods for Artificial Intelligence*, MIT Press, Cambridge, Mass., 1995. Covers empirical methods in AI, but a large part applies to all of computer science.

Fenton, N.E., and S.L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach* (2nd edition), Thomson Computer Press, New York, 1997. Excellent discussion of experimental designs as well as a wealth of material on experimentation with software.

Christensen, L.B., *Experimental Methodology*, Allyn and Bacon, New York, 1994. Judd, C.M., E.R. Smith, and L.H. Kidder, *Research Methods in Social Relations*, Holt, Rinehart, and Winston, 1991. General experimental methods.

Moore, D.S., and G.P. McCabe, *Introduction to the Practice of Statistics*, W.H. Freeman and Co., New York, 1993. Excellent introductory text on statistics.

Venables, W.N. and B.D. Ripley, *Modern Applied Statistics with S-PLUS*, Springer Verlag, New York, 1997. One of the best statistical packages available today is S-Plus. Venables and Ripley's book is both a guide to using S-Plus and a course in modern statistical methods. Keep in mind, however, that sophisticated statistical analysis is no substitute for good experimental design.

Fallacy 7. Technology changes too fast.

Rebuttal: If a question becomes irrelevant quickly, it is too narrowly defined and not worth spending a lot of effort on.

- I'm confident that readers would continue to value good conceptual papers and papers formulating new hypotheses, so such papers would still be published. Experimental testing would come later.

It is a matter of balance. Presently, nontheory research rarely moves beyond the assertive state, characterized by such weak justification as "it seems intuitively obvious," or "it looks like a good idea," or "I tried it on a small example and it worked." We need to reach a ground firmer than assertion.

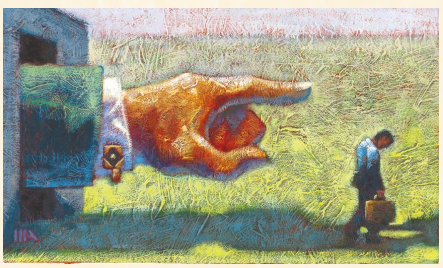
Technology changes too fast

Concerns about technology changing too rapidly frequently arise in computer architecture. Trevor Mudge summarizes it nicely: "The rate of change in computing is so great that by the time results are confirmed they may no longer be of any relevance."[8] We can say the same about software. What good is an experiment when its duration exceeds the useful life of the experimental subject—a software product or tool?

If a question becomes irrelevant quickly, it is perhaps too narrowly defined and not worth spending a lot of effort on. But behind many questions with a short lifetime lurks a fundamental problem with a long lifetime; scientists should probe for the fundamental and not the ephemeral, learning to tell the difference. Also, technological change often shifts or eliminates assumptions that were once taken for granted.

Scientists should therefore anticipate changes in assumptions and proactively employ experiments to explore the consequences of such changes. This type of work is much more demanding and can have much higher long-term value than merely comparing software products.

Fallacy 8. You'll never get it published.



Rebuttal: Smaller steps are still worth publishing because they improve our understanding and raise new questions.

You'll never get it published

Some established computer science journals have difficulty finding editors and reviewers capable of evaluating empirical work. Theorists may dominate their editorial boards, and experimenters are often confronted with reviewers who expect perfection and absolute certainty. However, experiments are conducted in the real world and are therefore always flawed in some way. Even so, I've seen publications demand that experiments be conducted with hundreds of subjects over a span of many years and several industrial projects before publication. We need to realize that smaller steps are still worth publishing because they improve our understanding and raise new questions.

In my experience, publishing experimental results is not difficult if one chooses the right outlet. I'm on the editorial board of three journals. I review for several additional journals and have served on numerous conference committees. All nontheoretical journals and conferences that I know of would greatly welcome papers describing solid experimentation. The occasional rejection of high-quality papers notwithstanding, I'm convinced that the low number of good experimental papers is a supply problem.

I fear, however, that the systems researcher of old will face difficulties. Just building systems is not enough unless the system demonstrates some kind of a first, a breakthrough. Computer science continues to be favored with such breakthroughs, and we should continue to strive for them. The majority of systems researchers, however, work on incremental improvements of existing ideas. These researchers should try to become respectable experimentalists, and they must articulate how their systems contribute to our knowledge. Systems come and go. We need insights about the concepts and phenomena underlying such systems.

WHY SUBSTITUTES WON'T WORK

Can we get by with forms of validation that are weaker than experimentation? It depends on what question we're asking. A conventional model for a scientific paper includes the following elements:

- The work describes a new idea, prototyped perhaps in a small system.
- The work claims its place in "science" by making feature comparisons. That is, the report sets out a list of features and qualitatively compares older approaches with the new one, feature by feature.

I find this method satisfactory when someone presents a radically new idea or a significant breakthrough, such as when researchers presented the first compiler for a block-structured language, time-sharing system, object-oriented language, or Web browser. Unfortunately, the majority of papers published take much smaller steps forward. As computer science becomes a harder science, mere discussions of advantages and disadvantages or long feature comparisons will no longer suffice; any PC magazine can provide those. Science, on the other hand, cannot live off such weak inferences in the long run. Instead, scientists should create models, formulate hypotheses, and test them using experiments.

Trust your intuition

In a March 1996 column, Al Davis, the editor-in-chief of *IEEE Software*, suggested that gut feeling is enough when adopting new software technology; experimentation and data are superfluous. He even suggested ignoring evidence that contradicts one's intuition.[9]

Instinct and personal experience occasionally lead down the wrong path, and computer science is no exception to this truism, as several examples illustrate:

- For about 20 years, it was thought that meetings were essential for software reviews. Recently, however, Porter and Johnson found that reviews without meetings are substantially no more or less effective than those with meetings.[10] Meetingless reviews also cost less and cause fewer delays, which can lead to a more effective inspection process overall.
- Another example is when small software components are proportionally *less* reliable than larger ones. This observation was first reported by Victor R. Basili[11] and confirmed by several disparate sources. (Les Hatton offers summaries and an explanatory theory.[12])
- As mentioned, the failure probabilities of multiversion programs were incorrectly believed to be the product of the failure probabilities of the component versions.
- Type checking is thought to reveal programming errors, but there are contexts in which it does not help.[13]

What we can learn from these examples is that intuition may provide a starting point, but must be backed up by empirical evidence. Without proper grounding, intuition is questionable. Shari Lawrence Pfleeger provides further discussion of the pitfalls of intuition.[14]

The fact that the subject of inquiry in computer science is information rather than energy or matter makes no difference in the applicability of the traditional scientific method.

Trust the experts

During a recent talk at a US university, I was about to present my data when a colleague interrupted and suggested that I skip that part and go on to the conclusions. "We trust you" was the explanation. Flattering as this was, it demonstrates a disturbing misunderstanding of the scientific process (or indicates someone in a hurry). Any scientific claim is initially suspect and must be examined closely. Imagine what would have happened if physicists hadn't been skeptical about the claims by Stanley Ponds and Martin Fleischman regarding cold fusion.

Frankly, I'm continually surprised by how much the computer industry and sometimes even university teaching relies on so-called experts who fail to support their assertions with evidence. Science, on the other hand, is built on healthy skepticism. It is a good system to check results carefully and to accept them only provisionally until they have been independently confirmed.

PROBLEMS DO EXIST

There are always problems with experimentation. Experiments may be based on unrealistic assumptions, researchers may manipulate the data, or it might be impossible to quantify the variable of interest. There are plenty of potential flaws. Good examples of solid experimentation in computer science are rare, but we should not discard the concept of experimentation because of this. Other scientific fields have been faced with bad experiments, even frauds, but—on the whole—the scientific process has been self-correcting.

Competing theories

A science is most exciting when there are two or more strong, competing theories. There are a few competing theories in computer science, none of them earth-shaking. The physical symbol system theory and the knowledge processing theory in AI are two competing theories that attempt to explain intelligence. The weak reasoning methods of the first theory have gradually given way or have coupled with knowledge bases.[15]

Another important example is algorithm theory. The present theory has many drawbacks. In particular, it does not account for the behavior of algorithms on typical problems.[16] A theory that more accurately

applies to modern computers would be valuable.

A prerequisite for competition among theories, however, is falsifiability. Unfortunately, computer science theorists rarely produce falsifiable theories. They tend to pursue mathematical theories that are disconnected from the real world. While computer science is perhaps too young to have brought forth grand theories, my greatest fear is that the lack of such theories might be caused by a lack of experimentation. If scientists neglect experiment and observation, they'll have difficulties discovering new and interesting phenomena worthy of better theories.

Unbiased results

Another argument against experimentation takes the following direction: "Give the managers or funding agencies a single figure of merit, and they will use it blindly to promote or eliminate the wrong research."

This argument is a red herring. Good managers, scientists, and engineers all know better than to rely on a single figure of merit. Also, there is a much greater danger in relying on intuition and expert assertion alone. Keeping decision-makers in the dark has an overwhelmingly higher damage potential than informing them to the best of one's abilities.

Experimentation is central to the scientific process. Only experiments test theories. Only experiments can explore critical factors and bring new phenomena to light so that theories can be formulated and corrected. Without experiments, computer science is in danger of drying up and becoming an auxiliary discipline. The current pressure to concentrate on application is the writing on the wall.

I don't doubt that computer science is a fundamental science of great intellectual depth and importance. Much has already been achieved. Computer technology has changed society, and computer science is in the process of deeply affecting the world view of the general public. There is also much evidence suggesting that the scientific method does apply. As computer science leaves adolescence behind, I hope to see the experimental branch of this discipline flourish. ❖

Acknowledgments

I'm grateful for thought-provoking comments from Les Hatton, Ernst Heinz, James Hunt, Paul Lukowicz, Anneliese von Mayrhauser, David Notkin, Shari L. Pfleeger, Adam Porter, Lutz Prechelt, and Larry Votta.

References

1. F.P. Brooks, "Toolsmith," *Comm. ACM*, Mar. 1996, pp. 61-68.
2. A. Ralston and E.D. Reilly, *Encyclopedia of Computer Science*, Third Edition, Van Nostrand Reinhold, 1993.
3. J.C. Knight and N.G. Leveson, "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming," *IEEE Trans. Software Eng.*, Jan. 1986, pp. 96-109.
4. W.F. Tichy et al., "Experimental Evaluation in Computer Science: A Quantitative Study," *J. Systems and Software*, Jan. 1995, pp. 1-18.
5. E.O. Wilson, *The Diversity of Life*, Harvard Univ. Press, Cambridge, Mass., 1992.
6. J. Hartmanis, "Turing Award Lecture: On Computational Complexity and the Nature of Computer Science," *Comm. ACM*, Oct. 1994, pp. 37-43.
7. R. Reddy, "To Dream the Possible Dream," *Comm. ACM*, May 1996, pp. 105-112.
8. T. Mudge, "Report on the Panel: How Can Computer Architecture Researchers Avoid Becoming the Society for Irreproducible Results?" *Computer Architecture News*, Mar. 1996, pp. 1-5.
9. A. Davis, "From the Editor," *IEEE Software*, Mar. 1996, pp. 4-7.
10. A.A. Porter and P.M. Johnson, "Assessing Software Review Meetings: Results of a Comparative Analysis of Two Experimental Studies," *IEEE Trans. Software Eng.*, Mar. 1997, pp. 129-145.
11. V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM*, Jan. 1984, pp. 42-52.
12. L. Hatton. "Reexamining the Fault Density: Component Size Connection," *IEEE Software*, Apr. 1997, pp. 89-97.
13. L. Prechelt and W.F. Tichy, "An Experiment to Assess the Benefits of Inter-Module Type Checking," *IEEE Trans. Software Eng.*, Apr. 1998.
14. S.L. Pfleeger et al., "Rebuttal to March 96 editorial," *IEEE Software*, July 1996.
15. E.A. Feigenbaum, "How the What becomes the How," *Comm. ACM*, May 1996, pp. 97-104.
16. J.N. Hooker, "Needed: An Empirical Science of Algorithms," *Operations Research*, Mar. 1994, pp. 201-212.

Walter F. Tichy is professor of computer science at the University Karlsruhe, Germany. Previously, he was senior scientist at the Carnegie Group in Pittsburgh, Pennsylvania, and on the faculty of computer science at Purdue University in West Lafayette, Indiana. His research interests include software engineering and parallelism, specifically software architecture, design patterns, configuration management, workstation clusters, optoelectronic interconnects for parallel computers, and program optimization tools for parallel computers. He received an MS and a PhD in computer science from Carnegie Mellon University.

Contact Tichy at tichy@ira.uka.de or http://wwwipd.ira.uka.de/~tichy