The Software-Research Crisis

ROBERT L. GLASS, *Computing Trends*

◆ *With the advantage of more than 25 years' hindsight, this twenty-first century author looks askance at the "crisis" in software practice and expresses deep concern for a crisis in software research.*

It is the year 2020. We in computing and software research are certainly glad we've been able to put the "research crisis" behind us.

What was this crisis? It was the realization that occurred, right around the turn of the century, that research in computing and software — as it was then focused — was all too often both arrogant and narrow.

It was arrogant because many computing researchers of that era were doing research in a topic they thought they understood, but didn't. It is amazing that, in retrospect, those computer scientists simply didn't know what they didn't know.

It was narrow because, of all the possible research models twentieth century computer scientists might have used, most of them were stuck using only one of them. Even though we now understand how terribly limiting that approach was, it is easy to see, in retrospect, why computer scientists could not see the error in their ways.

Before I continue, I think it is worth spending a moment or two elaborating on the term *research crisis*. There is a fine irony to it, and the story makes good telling.

To understand this research crisis, it is important that we confront and deal with a prior "software crisis."

It has been about 50 years now since the software community coined the term *software crisis*. It was invented at a conference predominantly attended by theorists rather than practitioners, but still, at the outset, almost

0740-7459/94/$04.00 © 1994 IEEE

everyone agreed there was some validity to the term. What did "software crisis" mean? It is hard to believe, now, but what it meant way back then was that the *practice* of software was in crisis — that it was characterized by projects that were "always over budget, behind schedule, and unreliable."

With the advantage of hindsight, we can look back on the last century as a time of wonder. Of all the dramatic changes that took place in the world, computing and software gave rise to the name "computing era" that we now recognize as the proper characterization of the last half of the twentieth century. Oh, there were software project failures, of course. Some of them were even catastrophic — *runaway* described the worst of them. But their existence provided only anecdotal evidence, certainly not solid data, to support the claims of crisis. In fact, the issue of solid data became the shoal on which the claims of crisis foundered, as we will see later in this story.

Software crisis indeed! We may look back on that earlier time as archaic and ignorant, perhaps, but certainly not as a crisis. It was the beginning, in fact, of the Golden Age of Computing Practice that persists today.

Nevertheless, the notion of software crisis reached a fever pitch of intensity in the 1980s and 1990s. Researchers began nearly every paper on software engineering by invoking the software crisis as a reason for listening to whatever new theoretical notion they were advocating. Researchers at the time didn't realize that what many of them were doing would later be characterized, derisively, as *advocacy research*. But the software crisis was in fact the platform on which most of this advocacy research was founded.

SOFTWARE CRISIS DISCREDITED

At the time, researchers frequently cited both the anecdotal evidence referred to earlier and a government study from the US General Accounting Office on the problems of building software, as support for their claims of crisis.

In most cases, this GAO study was the only real, nonanecdotal data cited, and almost every researcher cited it. It seemed to say that most software projects failed, and that most money spent on software was wasted.

Fortunately, cooler heads began to prevail. One researcher, then a voice crying in the wilderness, brought an end to the use of the term software crisis.

CRISIS, INDEED! THE TWENTIETH CENTURY WAS A TIME OF IGNORANCE, BUT ALSO THE DAWN OF THE GOLDEN AGE OF PRACTICE.

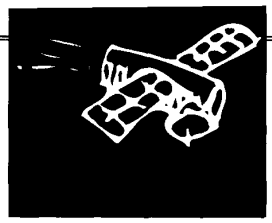
Bruce Blum, arguably one of the few researchers of the time who actually read the study (most researchers apparently copied the data from some other researcher's paper without going back to the source), discovered the data was being misunderstood and misused.[1] The study, an analysis of government software projects, examined only projects that were in trouble when it was conducted. Given that, the study's conclusion that most such projects failed, and most money spent on them was wasted, was interesting — "troubled projects frequently fail" — but hardly a basis for blackening the reputation of all software practice.

Even after the Blum revelation, a few computing researchers continued, for their own self-serving reasons, to use the GAO data and cry "software crisis," knowing that the data was being misused. But eventually common sense and ethics prevailed, and the notion of a software crisis slowly died away. Furthering its demise was Michiel van Genuchten's data, which presented findings from other researchers to the effect that typical software overruns were 33-36 percent over budget and 22 percent behind schedule, clearly a problem but hardly a crisis.[2]

One lingering aftereffect of this so-called crisis, however, was a deep resentment by software practitioners of software theorists and software theory. Most people in a profession would be offended, of course, by the notion that their projects and products were constantly characterized as "behind schedule, over budget, and unreliable." Ironically, computing researchers never understood how offensive their software-crisis campaign had been to practitioners. It is fair to say, in the enlightened year 2020, that those old resentments still linger, and partly because of them, new theory still has a difficult time penetrating current practice.

But back to the 1990s. New government studies and open-minded computing researchers both began pointing out that serious problems existed in the 1990s model of software research. In the early 1990s, for example, a couple of government studies concluded that research was ignoring practice almost entirely and, in addition, ignoring the notion of practical application. There was a flurry of opposition to those reports, as researchers mired in the old ways resisted change (the very thing they accused practitioners of!), but eventually most computing researchers began to realize that theirs was truly a troubled endeavor. A flurry of criticisms stirred the pot: Software-engineering research was described as increasing in quantity but not in quality, lacking in evaluation, "becoming less credible," having a "gaping hole" in its "generally accepted methods," and in need of a "paradigm shift ... from purely theoretical and building-oriented to experimental...."[3]

Later, in an exchange of letters to the editor, both a letter-writer and the authors of the article addressed by the letter-writer agreed that, "software research is in a sad state," and that, "Without a scientific method, tech-



nobabble, lemmingineering, and fads run rampant."[4]

At the time, it seemed natural to shift the crisis from practice to research. This was not done in a spirit of meanness, but rather in the original fundraising spirit of the term. Researchers proclaimed a "research crisis" to obtain funds for newer and better ways of doing research, just as their forerunners had used the practitioner "software crisis" as a way of obtaining funding to investigate the problems *that* crisis implied. For quite a long time, cries of "research crisis" were as prevalent in the computing literature as mentions of its predecessor had been. As I mentioned at the beginning of this essay, that notion, like the earlier crisis that preceded it, has died out. But it was a long time in the dying.

FOCUS ON RESEARCH

Back to those twin notions of arrogant and narrow that characterized twentieth century computing and software research. What was that all about?

Let's start with arrogant. Remember the term advocacy research? Well, in the early — perhaps even primitive — model of computing research employed up to the end of the twentieth century, much research consisted of a model that some characterized as "conceive an idea, analyze the idea, advocate the idea." At the conclusion of many research papers was a discussion of the implications for practice, which usually concluded with the claim that the idea should be transferred to practice as quickly as possible. Or words to that effect. Colin Potts referred to this as the "research-then-transfer" approach, contrasting it with what he called the "industry-as-laboratory" approach.[5]

What made advocacy research arrogant was that the typical researcher had never worked in software practice and had no basis for assuming that his or her idea would really work there. Most researchers had a mental model of software practice as an enterprise in crisis, one that did a bad job of whatever it undertook. There seemed to be an underlying assumption in most research that any change was better than the status quo.

The problem got so bad, in fact, that the software consortia and institutions of the time were sometimes characterized as "arrogant and ignorant." This charge was unfair to many of them, of course, but there was enough truth to it that the notion stuck. It is difficult for a research-and-development institution to be effective when its people are seen as arrogant and ignorant. That problem nearly destroyed the effectiveness of those institutions before they managed to overcome it. But that is another story.

THE ONLY RESEARCH MODEL IN USE WAS AN ANALYTICAL METHOD DEVOID OF EMPIRICAL EVALUATION.

What about the word narrow? Basically, there was only one research model used in most twentieth century computing. I have already characterized that model as advocacy research, but that is not entirely fair. Still, it wasn't until the early 1990s that one software researcher published material on possible research models,[6] identifying these:

- *The scientific method.* Observe the world, propose a model or theory of behavior, measure and analyze, validate hypotheses of the model or theory, and if possible repeat.
- *The engineering method.* Observe existing solutions, propose better solutions, build or develop, measure and analyze, repeat until no further improvements are possible.
- *The empirical method.* Propose a model, develop statistical or other methods, apply to case studies, measure and analyze, validate the model, repeat.
- *The analytical method.* Propose a formal theory or set of axioms, develop a theory, derive results, and if possible compare with empirical observations.

Then, a few years later, another paper also addressed the issue,[7] breaking research down into four phases:

- *The informational phase.* Gather or aggregate information via reflection, literature survey, people/organizational survey, or poll.
- *The propositional phase.* Propose and/or build a hypothesis, method or algorithm, model, theory, or solution.
- *The analytical phase.* Analyze and explore a proposal, leading to a demonstration and/or the formulation of a principle or theory.
- *The evaluative phase.* Evaluate a proposal or analytic finding by means of experimentation (controlled) or observation (uncontrolled, such as a case study or protocol analysis), perhaps leading to a substantiated model, principle, or theory.

Once those papers were absorbed by the field, the conclusion was inevitable. Almost no computing research to that time had used the scientific method (It begins with "observe the world." No one was doing even that first step, let alone formulating and validating hypotheses); there was just as little use of the engineering-research method (almost no one was "observing existing solutions"); and there was a fringe group using the empirical method, but judging by the academic-tenure success of many of its proponents, their research was not admired by their more traditional colleagues.

Furthermore, almost no computing research to that time had an evaluative phase. Information may have been gathered, propositions may have been made, and analysis may have been conducted. But, typically, the research ended there.

In other words, the only research model commonly in use in that unen-

lightened era was the analytical method, the one we have characterized unfavorably as advocacy research. Most research involved proposing formal methods for building software (the propositional phase), fairly deep and often mathematical analysis of those methods (the analytical phase), derivation of a theory of the applicability of those methods (more analytical phase), and no, repeat, *no* "compare with empirical observations" (the evaluative phase). Probably the first paper to publicly notice this deficiency was Norman Fenton's.[8]

Perhaps the analytical method might have been somewhat more acceptable had it employed the last part of its definition (its evaluative phase, "if possible compare with empirical observations"), but it rarely did. It was as if there were a disdain for anything connected with practice. Establishing pilot studies to try out ideas in a realistic setting and evaluate their success, incredibly enough, was not done. Researchers seemed to believe that was a task that practice should do, much like the old "exercise left for the student." One leading computing-research journal of the early 1990s devoted a special issue to researchers giving practitioners advice on how to evaluate new research ideas, as if to say, "That's your job, not ours." No one seemed to notice the irony.

It was the publication of Walter Tichy's position statement on experimental software engineering research that finally provided the solid data to demonstrate how widespread the research crisis was.[9] In that article, Tichy studied several of the leading computing and software-research journals of the time, characterizing the papers contained therein as to how much they involved

- theory,
- design,
- quantitative evaluation, and
- hypothesis testing.

In what the author called "alarming" findings, few if any papers contained any hypothesis testing, and less than 20 percent contained quantitative evaluation. The paper concluded, "Computer scientists may produce too many designs and not enough quantitative results," and, "The balance between theory and experiment in Computer Science seems skewed." It was a combination of the government research committees emphasizing reality-focused research and articles like Tichy's position paper, which showed how prevalent the problem was, that finally caused the research community to address its problems. This, in turn, helped the research crisis begin to abate and, eventually, go away.

THE RESEARCH CRISIS BEGAN TO ABATE WHEN GOVERNMENT COMMITTEES BEGAN TO EMPHASIZE A REAL-WORLD FOCUS.

SOFTWARE RESEARCH'S MATHEMATICS CONNECTION

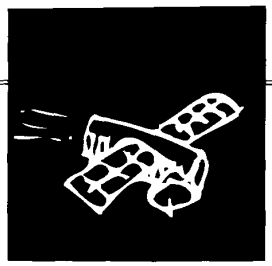
Why did computing and software research start — and continue — to use such a narrow approach? With respect to how it got started, the answer is easy:

Computing and software at most academic institutions were spawned in a mathematics department. Mathematics is a peculiar discipline. There are no physical artifacts for mathematicians to study, as there are in the more scientific disciplines, for example, and thus mathematical research takes on an entirely different flavor from other academic research. Of the research models mentioned above, the only one that makes much sense is the last one. Mathematics, in other words, is rarely into scientific or engineering or empirical research. Analytical research is the only approach that it tends to employ. It is no accident that the origins of computing research were the same as the mathematical research that gave it birth: analytical in nature.

But why did computing research get stuck there? Here, the explanation is somewhat more complicated. There are several reasons:

- Mathematicians at many academic institutions long ago divided themselves into two warring camps: those who did applied research, studying ways of using mathematics in other fields, and those who did pure research, studying mathematics for the sake of its own improvement. Both disciplines are vitally needed, of course, but as often happens when humans and politics get into the act, these two groups began to dislike and, eventually, to disdain one another. Johnny-come-lately computing people, born into this conflict, naturally chose up sides. Most of them favored the "pure" side, perhaps because that was the side that often had more academic "respectability" and political power. Whereas applied mathematicians and computing researchers might have broadened their research approach, pure mathematicians could not and did not need to, and "pure" computer scientists simply emulated that outlook.
- The artifacts of computing and software are expensive and time-consuming to build (much more so than, for example, new mathematical concepts). Software research that involved concept evaluation in a somewhat realistic setting would be very expensive in terms of both time and money. The money to do so was typically unavailable in research settings, and most researchers were not motivated to try to solve this problem.

There is an irony to that reason, of course. At the same time that software practitioners were taking the position that it was too expensive to try out new ideas from research, researchers were



taking the position that it was too expensive to test the value of their new ideas with respect to practice. Software progress was in fact stuck in place by a funding problem that almost no one even recognized!

♦ During the early days of computing and software theory and practice, the field moved forward so rapidly that almost all new ideas were good ideas. For the field to wait until these new ideas were evaluated would have dramatically slowed down that early progress in the field. Progress began to slow in the 1970s, and continued to do so in the 1980s, despite a growth spurt triggered by the advent of the ubiquitous microcomputer. Yet once progress slowed, no one stepped back to reanalyze the field's approaches to it to see that a new model was now needed. Long past the time that it made sense, researchers continued to expect practice to embrace their new ideas with open arms.

NOW, IN THE YEAR 2020, MEMBERS OF THE RESEARCH AND PRACTICE COMMUNITIES EVEN RESPECT EACH OTHER.

2020 VISION

But that was then and this is now. As I mentioned at the outset, we in computing and software research are certainly glad we've been able to put the research crisis behind us.

How did the change happen? The first necessary phase, of course, was acknowledging that there was a problem. Some of the papers mentioned earlier in our story were instrumental in making that happen; the gradual accumulation of enough researchers expressing the same view began to swing the field toward less arrogant and narrow, more realistic approaches. Progress was slow, of course, as it always is when fundamental human viewpoints must be changed. It helped that there had been in existence for nearly two decades an excellent model of what the research field needed to achieve. That model was a consortium involving academe (the University of Maryland Computer Science Department); industry (Computer Sciences Corporation); and government (NASA-Goddard, the sponsoring body), which together had formed the Software Engineering Laboratory. The SEL does engineering and empirical research using all the research phases, including — especially — evaluation.

Researchers the world over began to study that model, and emulate it in other settings and for other application domains (the SEL was focused on flight dynamics problems). It took years, of course, but that is how the research crisis disappeared.

What happens in computing and software research and practice in the year 2020? It took a long time to get here, but we have been able to achieve three things:

Software practice and research work together. Researchers have given up on the "arrogant and narrow" approaches to software research, and have come to realize that the only way to tell if their new ideas have value is to try them out in a practical setting. There is a real "development" focus as well as a research one in most "research-and-development" organizations and projects, both in academe and in industry. The "industry-as-laboratory" research concept advocated so long ago[5] has finally come to fruition!

In fact, researchers and practitioners tend to move back and forth freely between their previously isolated turfs. Top practitioners move to the world of academe and help ensure that research and pedagogy there reflect reality. Top researchers move to the world of industry to try out their ideas in a setting that can be truly evaluative. Sabbaticals and leaves of absence are often exchanged. Perhaps best of all, practitioners and researchers tend to even like and respect each other!

Good research results make it into practice. There is, of course, the lingering mistrust of theory that built up during the era of the "software crisis," but most of those effects can be overcome when a researcher is working alongside a practitioner, being open to adjusting and improving ideas in order to make them useful in practice. Both researchers and practitioners have come to understand the time and money cost of the learning curve, realizing that the adoption of any new idea has an initial price to be paid before any payoff is achieved. Modifying or getting rid of the schedule-driven approach to building software has freed practitioners to try new ideas, to undertake the risks necessary to making progress. The era of boastful claims of "breakthroughs" has long since disappeared, replaced by a healthy understanding of the realities of technology transfer. Slowly but surely, ways of building software are improving. We still have a lot to learn, of course, but at least we understand what that really means now.

Bad research ideas get discarded fairly quickly. Now that research ideas must meet the test of practical usage, we no longer have the situation that prevailed in the twentieth century when a new research idea would be conceived, analyzed by researcher after researcher, advocated thoroughly, and never used in practice. The early feedback that researchers now get as to the value of their new ideas is invaluable. Research, as a result, tends to move forward to embrace and study new ideas, not get stuck regurgitating old ones. For example, nearly 50 years ago, the research topic of formal verification (also called proof of correctness) was conceived. Its advocates persisted for over 30 years in pushing that particular wheelbarrow uphill, paying no attention either to the disinterest

and disclaimers of practitioners who saw more cost than benefit in its use or to the warnings of fellow researchers, published every five years or so in the literature, and angrily refuted each time they appeared! (For example, Fenton proclaimed "There is no hard evidence to show that:

- formal methods have been used cost-effectively on a realistic, safety-critical system development,
- the use of formal methods can deliver reliability more cost-effectively than, say, traditional structured methods with enhanced testing,
- either developers or users can ever be trained in sufficient numbers to make proper use of formal methods)." [10]

It is still possible to pursue a research idea past its point of value even today, of course, but at least we now have a mechanism in place for shedding bad ideas.

It is often true that we humans make more progress out of our failures than our successes. Perhaps, in spite of the pain of the research crisis, in the long term something good has come of it. Certainly we are glad now that we understand the value of practice and theory moving forward hand in hand. Both researchers and practitioners, working together, can see a future in which the wisdom of each group is understood and appreciated by the other. And that, in human terms, may be the biggest success of all, in the year 2020. ●

REFERENCES:

1. B.I. Blum, "Some Very Famous Statistics," *The Software Practitioner*, March 1991.
2. M. van Genuchten, "Why is Software Late? An Empirical Study of Reasons for Delay in Software Development," *IEEE Trans. Software Eng.*, June 1991, pp. 582-590.
3. W. F. Tichy, N. Haberman, and L. Prechelt, "Future Directions in Software Engineering, Summary of the 1992 Dagstuhl Workshop," *Software Eng. Notes*, Jan. 1993.
4. E.V. LaBudde, "Why is Requirements Engineering Underused?" *IEEE Software*, Mar.1994, pp.6-9; response by P. Hsia, A. Davis, and D. Kung.
5. C. Potts, "Software Engineering Research Revisited," *IEEE Software*, Sept. 1993, pp. 19-28.
6. W. R. Adrion, "Research Methodology in Software Engineering, Summary of the Dagstuhl Workshop on Future Directions in Software Engineering," *Software Eng. Notes*, Jan. 1993.
7. R. L. Glass, "A Structure-Based Critique of Contemporary Computing Research," *J. Systems and Software*, Jan. 1995, to appear.
8. N. Fenton, "How Effective Are Software Engineering Methods?" *J. Systems and Software*, Aug. 1993.
9. W. F. Tichy, et al., "Experimental Evaluation in Computer Science: A Quantitative Study," (draft), *J. Systems and Software*, Jan. 1995, to appear.
10. N. Fenton, S. L. Pfleeger, and R.L. Glass, "Science and Substance: A Challenge to the Software Engineering Community," *IEEE Software*, July 1994, pp. 86-95.

Robert L. Glass is publisher of *The Software Practitioner*, editor of the *Journal of Systems and Software*, a regular columnist for *Managing System Development*, and a sometime visiting professor of software engineering at Linköping University. He is interested in all facets of software engineering, especially in quality and maintenance. He has written 17 books and more than 30 papers on computing and software.

Glass received an MSc in mathematics from the University of Wisconsin at Madison. He is a member of the IEEE, the IEEE Computer Society, and ACM.

Address questions about this article to Glass at *Computing Trends*, 350 Dalkeith Ave., Los Angeles, CA 90049.