*Shari Lawrence Pfleeger*
Systems/Software Inc.

# *Albert Einstein and Empirical Software Engineering*

**Too often we tend to view software development the same way nineteenth-century physicists viewed the universe. Taking our cue from Einstein, we should shape our theories and models to fit a more probabilistic reality.**

As scientists, we apply scientific investigative techniques to gain more understanding of what makes software "good" and how to make software well. Often, we adapt investigative techniques from other disciplines to define measures that make sense in the business, technical, and social contexts we use for decision making. However, sometimes we can learn as much from another discipline's failures as from its successes. Examples from nineteenth-century physics show how a change in perspective can lead to explanations for previously misunderstood phenomena.

We must also consider whether our measurements constrict our view of what is really happening in the development process, and we must change or expand them if they are. Such changes may involve looking further afield, drawing on examples from the legal and business communities to improve our own models and theories. Science, and particularly physics, clearly illustrates the limitations of a too-literal approach to building and maintaining software.

## A DECEPTIVELY ORDERED UNIVERSE

At the end of the nineteenth century, several things about physics were well known:

- Newton's laws of mechanics described how particles respond to forces. Thus, physicists knew that things at rest tend to remain at rest, and things in motion tend to remain in motion.
- Newton's laws of gravity explained how the mass of objects is involved in their attraction for one another.
- Thermodynamics theories explained how heat and motion are related.
- Maxwell's equations unified electrical and magnetic phenomena. Consequently, scientists thought of light as a wave of magnetic energy.
- New areas of physics, such as statistical physics and kinetic theory, explained the behavior of gases and fluids on the basis of collisions among atoms and molecules.


0018-9162/99/$ 10.00 © 1999 IEEE

Underlying these theories were several assumptions, including notions about how energy is continuous, matter is particulate, and ether fills air and space and is the medium through which light travels. Most important is that at that time, physicists believed that the world works in a rational way, and if they tried hard enough, they could find the rules by which this behavior happens. This essential notion is the basis of all empirical software engineering: If we look long enough and hard enough, we will find rational rules that show us the best ways to build the best software.

## EINSTEIN'S DISRUPTIVE THEORIES

Then along came Albert Einstein, who pointed out that some physics problems could not be explained by prevailing theories. For example, consider the situation in which you throw a ball to a friend. If you throw the ball at 90 miles per hour, then we can say that the ball leaves your hand at 90 miles per hour. But suppose you are standing on a conveyor belt that is moving at 10 miles per hour toward your friend. According to the physical theories accepted by the scientific community at the turn of the century, if you throw the ball at 90 miles per hour in the direction in which the conveyor belt is traveling, the ball leaves your hand at 100 miles per hour: 90 from your throw, and 10 from the conveyor belt's movement.[1]

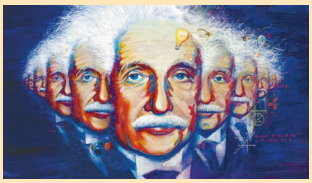
But Einstein uncovered a flaw in this logic. Suppose instead of a ball, you are holding a flashlight that shines on your friend. Light travels at 186,000 miles per second; regardless of whether the conveyor belt is moving at 10, 100, or even 10,000 miles per second, the light is still traveling toward your friend at 186,000 miles per second. How can that be? Does energy really differ from matter and obey different rules?

In fact, no. You experience Einstein's theory of relativity when you are on one train and look out at another train that is traveling in the same direction. If the trains are traveling at the same speed, it appears to you that they are not moving at all. Einstein's theory of relativity also explained that time actually changes. The flashlight paradox is based on faulty intuition, and our intuition about time is based on everyday speeds.

### A flawed model

Nineteenth-century physicists considered matter to be granular, composed of a finite number of atoms. But they thought that energy was indivisible because they had never seen light particles. Basing their measurement on a faulty model was the major reason for their failure to see light particles.

To understand the flaws in this model, consider how meteorologists measure rainfall. Because they are interested only in calculating total rainfall for the day, meteorologists measure the inches of rise in an agreed-upon-sized beaker or tub. They know that rain falls in finite droplets, but they don't measure rain drop by drop. Their gross measurement determines only the daily rainfall. In other words, meteorologists conduct a goal-question-metric analysis of the problem. Their measurement provides information about one characteristic of rain, volume, but reveals no information about rain's other characteristics.



**Even though our approaches to empirical software engineering illuminate the relationships among variables, they may in fact limit what we see.**

### Expanded horizons

Einstein expanded our horizons when he ignored current measurement and theory. By focusing instead solely on the paradox and redefining our notions of simultaneity, he created a theory that explains both well-understood phenomena like throwing a ball while on a conveyor belt and perplexing ones like the flashlight on the conveyor belt. It is for this work that Einstein received the Nobel Prize in 1921.

## EXPERIMENTAL LIMITATIONS AND THE NATURE OF PROCESS

So what does this have to do with empirical software engineering? A great deal. Einstein appreciated the importance of using experiments to assess theories. Indeed, no science can advance without good experimentation and measurement. But Einstein also appreciated the *limitations* of measurement, experiments, and human sensory perception. In the same way, we must recognize that even though our approaches to empirical software engineering illuminate the relationships among variables, they may in fact limit what we see.

For example, because we usually assume that the world works rationally, we seek relationships to help us understand what makes good software. We then apply what we learn so that we get good software more often. Our search is based in large part on the notion of cause and effect. If we can find out what causes good software in terms of process activities, tools, measurements, and the like, we can build an effective software process that will produce good software the next time.

### Natural processes versus social processes

We borrow this approach from the physical sciences, which deal with natural processes. A natural process is a part of the given world, and it occurs separate from our perception of it. Chemical reactions are natural processes, as are bodily functions such as digestion or respiration. Natural processes differ from social processes, which are products of human intention and consciousness. A social process exists because

**We need to devise strategies to help us deal with the imperfect knowledge and uncertainty in our measures and models.**

of our shared conception of it and our assumptions about it. Education and software development are examples of social processes.

The differences between natural and social processes are critical because they require different study methods.[2] One danger of pursuing empirical software engineering as we currently do—treating it as a natural process—is that we are looking at it in the wrong way.

### The nature of causality

The nature of causality is a key difference between the two kinds of processes. We like to think that cause and effect are deterministic: Every time we invoke a particular cause, we get the expected effect. But software development is more stochastic: A probability distribution describes the likelihood of the effect's occurrence.

Accepting this difference means that we must change some of our underlying notions about measurement and empiricism. For example, measurement theory's representation condition requires natural-process causality. But we need to devise strategies to help us deal with the imperfect knowledge and uncertainty in our measures and models. For example, we know that the Capability Maturity Model is imperfect—there is no guarantee that a Level 5 organization will produce good software. However, if we understand the uncertainty inherent in using the CMM, we can feel confident that a Level 5 organization will produce good software a certain percentage of the time under certain conditions.

Similarly, we can't say that inspecting code will guarantee that the code is fault-free. But we may be able to determine that using a certain type of design inspection in a certain way at a particular point in the development process is likely to eliminate a given percentage of the faults.

Our goal, then, is to understand the likelihood that, under certain conditions, a particular tool or technique will lead to improved software.

## THE KNOWLEDGE IN OUR MODELS

Understanding our technology in this way is no easy task. To start, we must catalog the types of knowledge we use to build our measures and models. We can think of knowledge in four ways:

- *Theoretical scientific knowledge* consists of hard facts and probabilistic information about cause and effect, usually obtained by testing theories. This type of knowledge is the most objective, and we often obtain it by doing controlled studies.
- *Engineering knowledge* is based on experience. We capture information about how best to design and operate the things we build. Engineering knowledge includes information about how well tools work together, what kinds of skills are needed for a task, and what has or has not worked well in the past.
- *Biomedical and epidemiological knowledge* is also experiential. It captures evidence about causation, although not necessarily based on an underlying theory. For example, we can observe correlations, but we cannot always distinguish between cause and effect. When we do have causation information, it can be either stochastic or deterministic. This type of knowledge can be obtained from either retrospective or prospective studies. A typical result can be expressed as a dose-response curve, meaning that a little of something does not necessarily have the same effect as a lot.
- *Social, economic, and institutional knowledge* tells us about who and what are involved in what we are observing. For example, social knowledge tells us how best to form teams, and economic knowledge allows us to compare and contrast different resource allocation options.

In empirical software engineering, we gather this knowledge to form theories. Often, we try to build a large body of evidence about a theory, incorporating many replications of the same study until we clearly support or refute the theory. However, such an approach takes a great deal of time and can produce inconclusive or conflicting results. Moreover, the technology under scrutiny may be changing as we study it, making it difficult or even impossible to amalgamate study results over time.

## SEQUENTIAL STUDIES: A BETTER APPROACH

How should we deal with empirical software engineering's shortcomings? Perhaps by extrapolating proven software development techniques to our theory building. When asked to solve a new software problem, we often invoke the "design-a-little, code-a-little, test-a-little" approach, trying this and that to see what works best. In the same way, we can perform sequential studies: study a little, theorize a little, then iterate.

The social sciences apply this technique frequently. For example, an educator proposes a new reading technique and tries it on a group of school children. Based on the results of the initial study, the technique is improved somewhat, and a second, similar study is run. In this way, educators have the advantage of using the most effective techniques known at the time, without having to wait for large numbers of replications.

Likewise, our new model of empirical software engineering should involve three key steps:

- reaching an initial understanding, including identifying likely variables, capturing the magnitude of problems and variables, documenting behaviors, and generating theories to explain perceived behaviors;
- testing theories by matching theory with practice, eliminating variables and identifying new ones, determining the relative importance of variables, and identifying the range of variable values and probabilities; and
- reducing uncertainty, but not always determining cause and effect.

### Software development strategies

As we build our theories, we must consider the various ways to apply our understanding to our software development activities. In particular, we should identify the best strategy for selecting one tool or technique over another. As scientists, we prefer to use the complete dominance strategy: Every value associated with option one is always greater than its counterpart associated with option two. For example, we want to satisfy our managers' concern about whether object-oriented technology is better than procedural technology or if Java is better than C++.

More often than not, we don't find complete dominance. Instead, we find that one option is better than another under certain conditions. In this case, we need to use a strategy of sufficiency, finding an option whose values are all acceptable. In this case, good enough is okay; we don't need to search for the best option. For example, when we use a novice programmer to write a subsystem, his code will provide the required functionality, but it may not be as elegant or as fast as the code our best programmers can write.

Another option is to use a strategy of caution, either minimizing the maximum possible loss or maximizing the minimum possible gain. For example, when a constrained schedule prevents us from testing completely, we can use a cautious strategy to thoroughly test the most critical parts of the system, thereby minimizing our loss from incomplete testing.

### Evidential criteria

Our selection of a particular technique or model is influenced by our criteria for determining which pieces of evidence are more valid than others. We can think of this choice the way American lawyers distinguish civil from criminal trials. In a civil trial, the defendant is guilty if the preponderance of evidence shows him or her to be guilty. But in a criminal trial, the evidence must indicate that the defendant is guilty beyond reasonable doubt. Reliance on software-related evidence is much the same. For a safety-critical system, we may insist on strict causality of reliability, for example. But for less critical systems, we are satisfied when a technique works as advertised most of the time.

## EVIDENCE ABOUT EVIDENCE

We put together pieces of evidence to form an argument, then we evaluate that argument using the strategy we've chosen. But constructing the argument is not a simple matter of the whole being the sum of its parts. For example, David Schum[3] examines a technology argument much as he would a legal argument. In particular, he describes how the legal community determines whether and how evidence supports the degree to which one variable causes a particular effect. In the law, the first step is understanding the type of evidence: Is it testimony, heuristics, or authoritative evidence? In software development, we ask whether the technology is provided by the vendor as testimony, by users who rate it on some kind of scale, or by practitioners or researchers who evaluate it objectively and in some quantitative way.

**Mainstream practitioners prefer to make minor modifications to their current ways of doing things so that the new technology enhances rather than replaces their current process.**

### What makes good evidence?

Marvin Zelkowitz and colleagues[4] suggested that evidential credibility depends on both the receiver and the giver of the information. Their work reveals that practitioners and researchers have very different ideas about what makes good evidence. Practitioners prefer to look at field or case studies performed in context to decide if the study's environment is similar to their own. In contrast, researchers prefer to use controlled experiments, simulations, and other more general studies. Without the tie to what they perceive as a "real" situation, practitioners often ignore the results of these more general studies.

### Visionaries versus pragmatists

Biases about technology and business can also affect the way practitioners choose to believe evidence. Geoffrey Moore[5] asserts that, as adopters of a new technology, some practitioners are visionaries. They are eager to change the existing process, willing to deal with faults and failures, and in general are focused on learning how a new technology works. They are revolutionaries willing to take big risks, and they feel comfortable replacing their old tools and practices with new ones. As shown in Table 1, they give more credence to evidence focused on the novelty and effectiveness of a new technology than to business-related evidence.

In contrast, mainstream practitioners are more pragmatic. They are interested in how technology supports business, so they focus on productivity improvement for their existing processes. They prefer to make minor

**Table 1. Early versus mainstream market preferences for evidence (adapted from Moore, 1991).**

| Market preference | Focus | Evidence | Press coverage | Endorsements |
|---|---|---|---|---|
| Early: visionaries | Technology | Architecture<br>Schematics<br>Demonstrations<br>Trials | Technology | Gurus |
| Mainstream: pragmatists | Product | Benchmarks<br>Product reviews<br>Design wins<br>Initial sales volume | Trade | Visionaries |
| | Market | Market share<br>Third-party support<br>Standards certification<br>Application proliferation | Vertical | Industry analysts |
| | Company | Revenues and profits<br>Strategic partners<br>Top-tier customers<br>Full product line | Business | Financial analysts |

modifications to their current ways of doing things so that the new technology enhances rather than replaces their current process. For them, evidence that the new technology enhances their market or company position is far more compelling than information focused on reducing defects or cyclomatic numbers.

## Generating evidence

David Schum[3] reminds us that we must also look at the strength of the evidence, which is often related to the degree of control we have in the studies we perform. For example, if we can carefully control all other variables, we can say that a change in quality is definitely the result of having used a new technology. But if we can't control all the variables, we can say only that it is probable or possible that the new technology causes the result.

The process that generates the evidence can also affect its credibility. Sometimes this process is iterative: We come to a preliminary conclusion with some degree of confidence, then we revise our conclusions and confidence level as new evidence is generated. Building up a more complete body of evidence can amplify our confidence. However, it can just as easily provide conflicts if one study shows clear benefit from a technology, but the next study shows no difference at all. For example, researchers at the University of Strathclyde recently investigated the effect of inheritance levels on the maintainability of object-oriented programs. But when researchers at the Bournemouth University replicated the experiment, the effect they observed was the opposite of what the Strathclyde researchers found.[6,7] It might seem that this conflict indicates lack of confidence, but larger bodies of evidence actually help to show us what the most important variables are, and they sometimes point out variables we didn't consider in earlier studies. Thus, the evidence-building process helps us determine exactly what situations are best for using a particular technology.

We must also take into account the structure of the argument made from the evidence. Each piece of evidence does not stand on its own. We create the fabric of an argument out of threads of evidence; different evidence plays differing roles in supporting our ultimate conclusion. Some of the evidence may be contradictory or conflicting, and our conclusions about the technology must take these imperfections into account. For example, we can't dismiss object orientation for maintainability simply because the evidence conflicts. Rather, we must evaluate the nature of the conflicts to decide either that the technology won't achieve its goal or that it is sure to work.

The implications of these scientific, business, and legal findings for empirical software engineering studies are clear. It is not enough to develop new theories and provide evidence. The practitioners who are the audience for our evidence must be able to understand our theories and findings in the context of their work and values.

When developing theories and models that produce quality software, keeping an open mind is essential. It is tempting to throw out evi-

dence that doesn't fit our pet theory. But we should do the opposite: We must continually question and improve the theory until it explains the phenomenon we see. At the same time, we must incorporate some degree of probabilistic measurement in our models and theories so that we have some understanding of both what we know and what we don't—yet.

The audience for our evidence should help us decide what kinds of studies to perform and in what context. When our studies are complete, we must present the results in ways our audience understands. It never hurts to explain our study's statistics, describe its limitations and uncertainties, and discuss alternative theories that can explain the perceived behavior.

The social sciences tell us to understand and present our results in the context of the wider world. Some of our measurements may necessarily be fuzzy or probabilistic. Some of our theories may explain only part of what we see. But it is better to have a partial understanding that can serve as a platform for future theories than to discard a result simply because it doesn't explain everything. Or, as Einstein pointed out, "Not everything that counts can be counted; and not everything that can be counted counts." ❖


**References**

1. J.J. Stachel, ed., *Einstein's Miraculous Year: Five Papers That Changed the Face of Physics*, Princeton University Press, Princeton, N.J., 1999.
2. R.N. Giere, "Knowledge, Values and Technological Decisions: A Decision-Theoretic Approach," in *Acceptable Evidence: Science and Values in Risk Management*, D.G. Mayo and R.D. Hollander, eds., Oxford University Press, New York, 1991.
3. D.A. Schum, *Evidential Foundations of Probabilistic Reasoning*, John Wiley & Sons, New York, 1994.
4. M.V. Zelkowitz, D.R. Wallace, and D. Binkley, *Understanding the Culture Clash in Software Engineering Technology Transfer*, Tech. Report No. 2, Univ. of Maryland, College Park, Md., 1998.
5. G.A. Moore, *Crossing the Chasm*, Harper Business, New York, 1991.
6. J. Daly et al., "The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study," *Proc. 1995 Conf. Int'l Software Maintenance (ICSM 95)*, IEEE CS Press, Los Alamitos, Calif., 1995, pp. 20-29.
7. M. Cartwright and M.J. Shepperd, "An Empirical View of Inheritance," *Proc. Empirical Assessment and Evaluation in Software Engineering*, Keele University, Staffordshire, UK, 1998.



*Shari Lawrence Pfleeger is the president of Systems/Software Inc. and a research scientist in the University of Maryland Department of Computer Science. Her research interests include empirical studies of software engineering, the practical problems of software development, and the theoretical underpinnings of software engineering and computer science. Pfleeger received a PhD in information technology and engineering from George Mason University. She is a member of the IEEE, the IEEE Computer Society, and the ACM. Contact her at s.pfleeger@ieee.org.*


## Get Involved

### About the Technical Council on Software Engineering

*Shari Lawrence Pfleeger and Gene F. Hoffnagle*

The Technical Council on Software Engineering taps into the myriad ways that software is created. The TCSE has two overriding aims: to contribute to its members' professional expertise and to help advance software engineering research and practice. We invite you to join us in improving software processes and products.

Our members are practitioners and researchers alike, and many of our events focus on enhancing interaction between these groups. Our two new regional organizations, TCSE Europe and TCSE Asia/Pacific, are designed to better serve members through a more local focus. Our flagship conference, the International Conference on Software Engineering, has excellent company in our dozens of other events that draw specialists from around the globe. Our newsletter reports on software-related activities within and outside of the TCSE.

To address all facets of software engineering, the TCSE contains focused subcommittees. The following listing details the activities and interests of a few of our organizations. For a more complete picture, visit http://tcse.org.

#### Recovering information

The Reverse Engineering and Reengineering Committee promotes technologies for recovering information from existing software systems and describes innovative ways of using this in- formation in system renovation, reuse, and migration. The committee cosponsors the Working Conference on Reverse Engineering and the Reengineering Forum. Committee members are establishing consistent terminology, forming a resource repository for research and education, and disseminating information through newsletters and tutorials.

#### Improving the software process

The activities of the Committee on Software Process speed technology transfer and complement other events in the process community, such as the International Software Process Workshop and the International

Software Process Conference (ISPC). In addition, the committee publishes a newsletter three times a year.

### Global approach to requirements engineering

The newly established Task Force on Requirements Engineering is creating an international umbrella organization for the requirements engineering community, promoting cooperation on research, practice, and education. The task force has strong ties to the International Symposium on Requirements Engineering (RE) and the International Conference on Requirements Engineering.

### Leadership in software engineering education

The Committee on Software Engineering Education is involved in several key activities. Curriculum and professionalism materials are being developed by two joint IEEE-CS/ACM committees. The Computing Curricula 2001 task force is developing a computing curriculum, and the Software Engineering Coordinating Committee (SWECC) is overseeing the development of a software engineering code of ethics, a software engineering body of knowledge (SWEBOK), and procedures for all levels of accreditation.

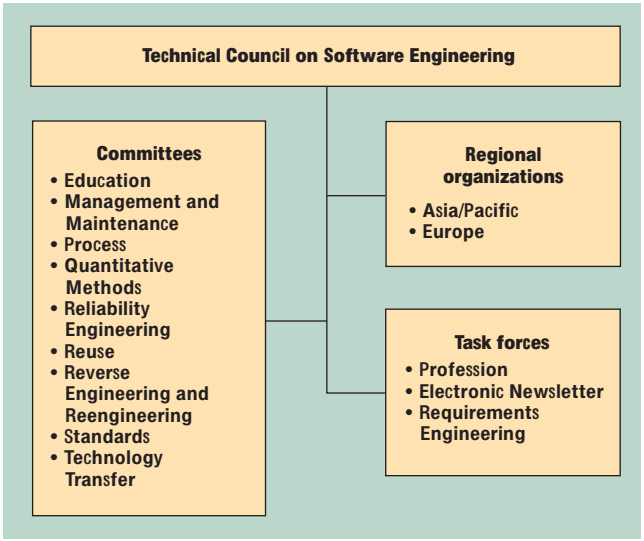



***Figure A. Organization of the Technical Council on Software Engineering.***

### Reliability

The Software Reliability Engineering Committee supports the use of measures and analysis to produce more reliable software. Our annual conference, the International Symposium on Software Reliability Engineering, is supplemented by a newsletter and an electronic discussion group.

### Measurement and prediction

The Quantitative Methods Committee studies measurement and prediction in software engineering. Its members assess and estimate characteristics of software products and processes such as quality, complexity, reliability, and cost. Since 1993, the committee has organized a series of international software metrics symposia. The committee also publishes a newsletter.

### Join 11,000 of your peers

The Technical Council on Software Engineering works to advance our understanding of software engineering, enhance the careers of our members, and create a network that connects members to each other and to related organizations. We encourage you to join us in advancing the state of the art and the practice. For more information, see http://tcse.org or contact TCSE Chair Gene Hoffnagle at g.hoffnagle@computer.org.

*Gene F. Hoffnagle is the chair of the Technical Council on Software Engineering and the director of IBM's technical journals. Contact him at g.hoffnagle@computer.org.*