DOI:00.0000/000

Peter J. Denning and Richard D. Riehle

The Profession of IT
Is Software Engineering Engineering?

Software engineering continues to be dogged by claims it is not engineering. Adopting more of a computer-systems view may help.

It is a time of considerable introspection for the computing field. We recognize the need to transcend the time-honored, but narrow image of, "We are programmers." That image conveys no hint of our larger responsibilities as software professionals and limits us in our pursuit of an engineering model for software practice.

The search for an alternative to the programmer image is already a generation old. In 1989 we asked: Are we mathematicians? Scientists? Engineers?[3] We concluded that we are all three. We adopted the term "computing," an analogue to the European "informatics," to avoid bias toward any one label or description.

Today, we want all three faces to be credible in an expanding world. The cases for computing as mathematics and as science appear to be widely accepted outside the field.[1] However, the case for computing as engineering is still disputed by traditional engineers. Computer engineering (the architecture and design of computing machines) is accepted, but software engineering remains controversial.

In this column, we examine reasons for the persistent questions about software engineering and suggest directions to overcome them.

Engineering Process

The dictionary defines engineering as the application of scientific and mathematical principles to achieve the design, manufacture, and operation of efficient and economical structures, machines, processes, and systems. When applied to software engineering, this definition calls attention to the importance of science and math principles of computing. Software engineering has also contributed principles for managing complexity in software systems.

Some definitions insist that engineering mobilizes properties of matter and sources of energy in nature. Although software engineering does not directly involve forces of nature, this difference is less important in modern engineering.

The main point of contention is whether the engineering practices for software are able to deliver reliable, dependable, and affordable software. With this in mind, the founders of the software engineering field, at the legendary 1968 NATO conference, proposed that rigorous engineering process in the design and implementation of software would help to overcome the "software crisis."

In its most general form, the "engineering process" consists of a repeated cycle through requirements, specifications, prototypes, and testing. In software engineering, the process models have evolved into several forms that range from highly structured preplanning (waterfalls, spirals, Vs, and CMM) to relatively unstructured agile (XP, SCRUM, Crystal, and evolutionary). No one process is best for every problem.

Despite long experience with these processes, none consistently delivers reliable, dependable, and affordable software systems. Approximately one-third of software projects fail to deliver anything, and another one-third deliver something workable but not satisfactory. Often, even successful projects took longer than expected and

had significant cost overruns. Large systems, which rely on careful preplanning, are routinely obsolete by the time of delivery years after the design started.[2] Faithful following of a process, by itself, is not enough to achieve the results sought by engineering.

Engineering Practice

Gerald Weinberg once wrote, "If software engineering truly is engineering, then it ought to be able to learn from the evolution of other engineering disciplines." Robert Glass and his colleagues provocatively evaluated how often software engineering literature does this.[4] They concluded that the literature relies heavily on software anecdotes and draws very lightly from other engineering fields. Walter Tichy found that fewer than 50% of the published software engineering papers tested their hypotheses, compared to 90% in most other fields.[8]

So software engineering may suffer from our habit of paying too little attention to how other engineers do engineering. In a recent extensive study of practices engineers expect but do not always write down, Riehle found six we do not do well.[5]

▸ *Predictable outcomes (principle of least surprise).* Engineers believe that unexpected behaviors can be not only costly, but dangerous; consequently, they work hard to build systems whose behavior they can predict. In software engineering, we try to eliminate surprises by deriving rigorous specifications from well-researched requirements, then using tools from program verification and process management to assure that the specifications are met. The ACM Risks Forum documents a seemingly unending series of surprises from systems on which such attention has been lavished. Writing in ACM SIG-

Software engineering may suffer from our habit of paying too little attention to how other engineers do engineering.

The main point of contention is whether the engineering practices for software are able to deliver reliable, dependable, and affordable software.

SOFT in 2005, Riehle suggested a cultural side of this: where researchers and artists have a high tolerance, if not love, for surprises, engineers do everything in their power to eliminate surprises.[6] Many of our software developers have been raised in a research tradition, not an engineering tradition.

▸ *Design metrics, including design to tolerances.* Every branch of modern engineering involves design metrics including allowable stresses, tolerances, performance ranges, structural complexity, and failure probabilities for various conditions. Engineers use these metrics in calculations of risk and in sensitivity analyses. Software engineers do not consistently work with such measures. They tend to use simple retrospective measures such as lines of code or benchmark performance ranges. The challenge is to incorporate more of these traditional engineering design metrics into the software development process. Sangwan gives a successful example.[7]

▸ *Failure tolerance.* Henry Petroski writes, "An idea that unifies all engineering is the concept of failure. Virtually every calculation an engineer performs...is a failure calculation... to provide the limits than cannot be exceeded." There is probably no more important task in engineering than that of risk management. Software engineers could more thoroughly examine and test their engineering solutions for their failure modes, and calculating the risks of all failures identified.

▸ *Separation of design from implementation.* For physical world projects, engineers and architects represent a design with blueprints and hand off implementation to construction specialists. In current practice, software engineers do both, design and build (write the programs). Would separation be a better way?

▸ *Reconciliation of conflicting forces and constraints.* Today's engineers face many trade-offs between conflicting natural forces and a dizzying array of non-technical economic, statutory, societal, and logical constraints. Software engineering is similar except that fewer forces involve the natural world.

▸ *Adapting to changing environments.* Most environments that use computing constantly change and expand. With drawn-out acquisition processes for complex software systems, it is not unusual for the system to be obsolete by the time of delivery. What waste! Mastering evolutionary development is the new challenge.[2]

The System

The problems surrounding the six issues listed here are in large measure the consequence of an overly narrow view of the system for which the software engineer is responsible. Although controlled by software, the system is usually a complex combination of software, hardware, and environment.

Platform independence is an ideal of many software systems. It means that the software should work under a choice of operating systems and computing hardware. To achieve this, all the platform-dependent functions are gathered into a platform interface module; then, porting the system to another platform entails only the building of that module for the new platform. Examples of this are the Basic Input-Output System (BIOS) component of operating systems and the Java Virtual Machine (JVM). When this can be achieved, the software engineer is justified in a software-centric view of the system.

But not all software systems are platform independent. A prominent example is the control system for advanced aircraft. The control system is implemented as a distributed system across many processors throughout the structure where they can be close to sensors and control surfaces. Another example is software in any large system that must constantly adapt in a rapidly changing environment. In these cases the characteristics of the hardware, the interconnections, and the environment

continually influence the software design. The software engineer must either know the system well, or must interact well with someone who does. In such cases adding a system engineer to the team will be very important.

Engineering Team

No matter what process engineers use to achieve their system objectives, they must form and manage an engineering team. Much has been written on this topic. Software engineering curricula are getting better at teaching students how to form and work on effective teams, but many have a long way to go.

Every software team has four important roles to fill. These roles can be spread out among several people.

The software architect gathers the requirements and turns them into specifications, seeks an understanding of the entire system and its trade-offs, and develops an architecture plan for the system and its user interfaces.

The software engineer creates a system that best meets the architecture plan. The engineer identifies and addresses conflicts and constraints missed by the architect, and designs controls and feedbacks to address them. The engineer also designs and oversees tests. The engineer must have the experience and knowledge to design an economical and effective solution with a predictable outcome.

The programmer converts the engineering designs into working, tested code. Programmers are problem-solvers in their own right because they must develop efficient, dependable programs for the design. Moreover, anyone who has been a programmer knows how easy it is to make mistakes and how much time and effort are needed to detect and remove mistakes from code. When the software engineer has provided a good specification, with known exceptions predefined and controls clearly delineated, the programmer can work within a model that makes the job of implementation less error-prone.

Although controlled by software, the system is usually a complex combination of software, hardware, and environment.

The project manager is responsible for coordinating all the parts of the team, meeting the schedules, getting the resources, and staying within budgets. The project manager interfaces with the stakeholders, architects, engineers, and programmers to ensure the project produces value for the stakeholders.

In some cases, as noted previously, a systems engineer will also be needed on the team.

Conclusion

We have not arrived at that point in software engineering practice where we can satisfy all the engineering criteria described in this column. We still need more effective tools, better software engineering education, and wider adoption of the most effective practices. Even more, we need to encourage system thinking that embraces hardware and user environment as well as software.

By understanding the fundamental ideas that link all engineering disciplines, we can recognize how those ideas can contribute to better software production. This will help us construct the engineering reference discipline that Glass tells us is missing from our profession. Let us put this controversy to rest. C

We need to encourage system thinking that embraces hardware and user environment as well as software.

References

1. Denning, P. Computing is a natural science. *Commun. ACM 50*, 7 (July 2007), 13–18.
2. Denning, P., Gunderson, C., and Hayes-Roth, R. Evolutionary system development. *Commun. ACM 51*, 12 (Dec. 2008), 29–31.
3. Denning, P. et al. Computing as a discipline. *Commun. ACM 32*, 1 (Jan. 1989), 9–23.
4. Glass, R., Vessey, I., and Ramesh, V. Research in software engineering: An analysis of the literature. *Information and Software Technology 44*, 8 (2002), 491–506.
5. Riehle, R. An Engineering Context for Software Engineering. Ph.D. thesis, 2008; theses.nps.navy.mil/08Sep_Riehle_PhD.pdf.
6. Riehle, R. Engineering on the surprise continuum: As applied to software practice. *ACM SIGSOFT Software Engineering News 30*, 5 (Sept 2005), 1–6.
7. Sangwan, R., Lin, L-P, and Neill, C. Structural complexity in architecture-centric software. *IEEE Computer* (Mar. 2008), 96–99.
8. Tichy, W. Should computer scientists experiment more? *IEEE Computer* (May 1998), 32–40.

Peter J. Denning (pjd@nps.edu) is the director of the Cebrowski Institute for Information Innovation and Superiority at the Naval Postgraduate School in Monterey, CA, and is a past president of ACM.

Richard Riehle (rdriehle@nps.edu) is a visiting professor at Naval Postgraduate School in Monterey, CA, and is author of numerous articles on software engineering and the popular textbook *Ada Distilled*.