Software Engineering—Missing in Action: A Personal Perspective

David Lorge Parnas, *Middle Road Software*

Although a huge number of articles have been written about software development and many interesting ideas have been proposed, researchers and practitioners have failed to create a new engineering discipline focused on building software-intensive systems.

My professional career began about the time the term "Software Engineering" came into use. Before that, people were using terms like "programming" or "software development" to talk about work that today might be labeled software engineering.

The new terminology was introduced because some people noticed two things:

- Software was, even in the 1960s, beginning to be a major bottleneck. It was common to find the hardware ready and working when the software was still incomplete and unreliable. Software was usually over budget, behind schedule, and not fit for its intended use.
- Many of those who were developing software began to realize that they were doing something quite different from what they had been taught to do. They had been trained to be scientists or mathematicians and to add to our knowledge. Now, they were creating artifacts for others to use.

The founders of the "Software Engineering" movement noted that Engineers had been taught how to do their job; in contrast, people who were developing software had learned to do very different jobs. Engineers, although far from perfect, did not have as many problem projects as software developers had.

Those who founded the movement hoped that many of the software development problems that we were experiencing would go away if software development became a new field of engineering. This was not a jurisdictional dispute; the issues raised were about education and regulation.

WHAT MAKES ENGINEERING WORK?

Engineers, and their work, are not perfect, but they often succeed in building functional, reliable products within a fairly predictable timeframe and close to estimated cost. This, at least, is what the founders of the movement believed when they decided to use older engineering disciplines as a model for the rapidly growing software development profession.

The secret of the success and good reputation that engineers enjoy is simple:

- They have been taught how to do their job.
- They have been taught the basic mathematics (theory) and science they need to perform their work.
- They have been taught to work according to strict rules and to understand that if they do not follow those rules, they might be found to have been negligent and lose their license to work as Engineers.

Published by the IEEE Computer Society

It is important to note that Engineers are not just taught rote procedures tied to current technology; they are taught the basic mathematics and science that will allow them to understand and use new technology when it becomes available. Their education permits them to understand the assumptions behind standard procedures and, therefore, to know when new developments justify new procedures.

EDUCATION AND LICENSING

There is a "core body of knowledge" associated with each engineering discipline that comprises mathematics, scientific knowledge, guidelines, and regulations that all who practice that discipline are expected to know. In most jurisdictions, this body of knowledge is agreed upon at a state or national level, and academic programs are periodically evaluated to make sure that all elements of the core are taught effectively. Exams after the end of the university program are used to confirm that graduates have learned the core principles and are qualified to work in the profession.

In some jurisdictions, those who did not receive a degree from an accredited engineering program can be licensed if they pass an extensive set of exams that test their comprehension of the core body of knowledge. In most jurisdictions, graduates must work as an apprentice to a licensed engineer for several years before they are finally licensed to work on their own. License holders elect a body that maintains and enforces the regulations. Legislation gives the body the power to license practitioners and to enforce the rules.

This approach, which is also used in other disciplines such as law and medicine, has led to the success that the founders of the software engineering movement envisioned. In most jurisdictions, these professions are self-regulating.

In spite of all the positive things we can say about the engineering profession, it is important not to have an unrealistic view of it. Not all good designers are Engineers or behave like Engineers, and not all Engineers are good designers. The licensing bodies enforce *minimal* standards and discipline practitioners who do not meet those standards .

Professional Engineers have been taught the scientific principles, mathematics, procedures, rules, and regulations appropriate to their discipline. They have been taught to use this information when designing. They also have undergone a period of supervised internship or apprenticeship before they are fully licensed as a professional. However, many things can go wrong:

- Some Engineers never deeply understood what they were taught; they understood it just well enough to pass the exams.
- Some forget or ignore what they did learn.
- Some do not understand the principles behind what they learned and do not apply them correctly in unusual situations.
- Some deliberately take shortcuts or neglect the rules.

Further, engineering science and principles do not tightly constrain designers. Using the established principles of the discipline, Engineers can produce either routine, uncompetitive designs or brilliant designs. Engineering principles allow creating a product that does exactly what is required or one that does more. For example, the product can meet only current needs or it can grow as user needs change.

In spite of all the positive things we can say about software engineering, it is important not to have an unrealistic view of it.

Some brilliant designers never had a formal education in their area of expertise, but they are intuitive and have a good artistic sense. They also are likely to be careful people who pay attention to detail. Often they are not formally recognized as Engineers, and perhaps they do not need to be. However, such brilliant, intuitive designers are rare. In most cases, we cannot rely on them to provide the products we use every day. Some work must be done by less gifted people who benefit from having an engineering education and working in a regulated profession.

A few of the useful products that we depend on today are beautiful, highly functional designs. Far more began as poor designs but were improved through a lengthy period of prerelease testing, followed by beta testing and post-delivery revisions. Almost every piece of software that I encounter contains evidence of oversights either caused or exacerbated by a lack of discipline.[1]

Engineer or technologist?

Engineering educators must clearly distinguish between technology, which changes rapidly and has many arbitrary facts, characteristics, and scientific principles, which remain usable throughout the Engineer's career. A good education teaches future Engineers how to use fundamental science to understand new technologies.

Engineer or application specialist?

Engineers are usually educated to work in broad disciplines and are not restricted to narrow fields of practice. Graduates are civil engineers, not road engineers or bridge engineers. Engineers usually become specialists through experience, but their education allows them to change.

Engineer or scientist?

Engineering curricula often share some content with science and mathematics programs, but the educational goals are different. The engineering student must learn how to use science and mathematics to build things, while the science student must learn how to add new knowledge to previously known information.

When a new engineering field is developing, it can sometimes be difficult to distinguish it from an existing science field. For example, when electrical engineering was new, some universities tried to keep it in physics. Physics departments claimed that physicists learned everything anyone needed to know to design electrical systems. Nonetheless, these fields are now clearly distinguished. Some people educated as engineers might end up doing research and working as scientists, but their focus is usually on applicable science. Often, they work together with pure scientists to develop recent scientific advances into useable technologies.

Engineers are usually educated to work in broad disciplines and are not restricted to narrow fields of practice.

THE STRUCTURE OF THE PROFESSIONS

Scientists have attempted to partition the body of knowledge accumulated about the world into distinct areas such as physics, biology, and chemistry; these are further divided into narrower areas such as hydraulics, thermodynamics, human physiology, and organic chemistry. The borders between these areas are not always clear, but the basic distinctions are. This structuring is essential to the work of scientists who often confine their research to a very narrow area. Specialization allows a scientist to know an area well and to use that knowledge to extend our understanding of the area.

Engineering is not partitioned in the same way. Engineers from a specific discipline are expected to be responsible for developing a class of products; to do that, they are required to know material from several areas of science. At the start of a career, it is not easy to predict what knowledge they will require, so an engineering education must be broad; even the required core is broad. There is significant overlap in the requirements for the various disciplines. An engineering discipline is characterized by a selection of topics from science and mathematics, but it does not have an exclusive claim to those topics.

PROGRAMMER VERSUS SOFTWARE ENGINEER

When the term "Software Engineering" was introduced, many asked a simple question, "How is software engineering different from programming?" Some of those who asked that question were skeptical and wondered if the term had been invented to attract more attention and funding. Others were asking the question rhetorically to suggest that there was no such field.

The best response to this question was provided by British computer scientist Brian Randell, who described software engineering as "the multiperson development of multiversion programs." This pithy phrase implies everything that educators should be teaching to future software developers. It should go without saying that a software engineer must be able to program, but that is not enough.

However, Randell's description is not sufficient to determine the core body of knowledge from the point of view of those who license Engineers. Pure software knowledge is not enough for the development of many software products. Just as those who grant licenses require extensive overlap between mechanical and chemical engineering, they would expect a licensed software engineer to know much more than software design.[2]

WHY BOTH MISSING IN ACTION AND LOST?

The goal of those who introduced the term "Software Engineering" has not been achieved, and in fact we seem to have lost sight of it. The gap between the computer science research world and software development practices continues to grow:

- Industry is aware of the need for improvement and sporadically forms new groups and initiatives that attempt to bring about change re what practitioners do. Most mainstream academics do not get involved. Often, they are too busy playing the publication numbers game.[3]
- On the academic side, we see new notations, formalisms, proof methods, and design approaches. These gain little traction with industry because they do not appear to address the practitioner's problems. Rather than show a better, more efficient way to do things, they call for additional work that has no obvious benefit.[4]

The gap between research and practice is not strictly between academia and industry. Larger companies have in-house research groups whose work looks much like academic research—they interact at least as much with external researchers as with internal developers. Other companies have their own internal methods specialists, but the developers often view them as theoreticians. It is this gap between academic research and developer problems that leads me to say that the "Software Engineering" discipline is "missing in action."

University researchers and educators, even those who claim to be in "Software Engineering," are rarely involved in establishing a regulated profession. In fact, sometimes they actively resist it.[5]

SOFTWARE ENGINEERING MEETS ...?

Each of the other articles in this special issue focuses on a specific area of research or application area and discusses its relation to software engineering. We must consider an obvious question: "If software development has not become an engineering discipline, how can they talk about it meeting another area?" In fact, none of these articles do that; each one confirms my position that software engineering, as originally envisioned, does not yet exist.

Theory

Manfred Broy's message in "Can Practitioners Neglect Theory and Theoreticians Neglect Practice?" is that we cannot have an engineering discipline without "theory." In traditional engineering, theory refers to a set of assumptions about the physics of the situation and a mathematical analysis of the implications of those assumptions. In computer science, there is no physics involved; theory is all mathematics.

Today, more than half a century after the term was coined, an article arguing that software engineering needs mathematics is evidence that we do not yet have such a field. In the traditional engineering disciplines, professors do sometimes discuss the mathematics to be included in a curriculum, but they do not discuss whether they need mathematics—they discuss which mathematics and how much mathematics. In software, it is also necessary to explain how mathematics can be used to improve product quality. In fact, some doubt that it has any relevance at all.[6]

If we want to establish software development as an engineering profession, we definitely need to discuss the role that mathematics can play in the field and exactly what mathematics must be taught. There have been some proposals, but there has not been enough discussion to reach any agreement.[7,8]

Open source software

Brian Fitzgerald's "Open Source Software: Lessons from and for Software Engineering" offers an interesting cautionary tale for software developers. Proponents have advanced OSS as a "silver bullet" that can ameliorate many of the problems that software developers encounter. However, OSS is a business model, not a design or engineering method. OSS is a way to motivate and control developers; it offers a different method for recouping investments. Consequently, an OSS product can be reliable or unreliable, changeable or difficult to change, and so on. A software development effort can use OSS ideas or reject them independently based on whether the work is done by professional engineers or not.

Evolutionary computation

In "Software Engineering Meets Evolutionary Computation," Mark Harman begins by reminding us that software evolves and refers to early research that investigated how software grew and changed as a result of modifications and additions. The bulk of this article deals with techniques that use the idea of evolution in a different way: the application of algorithms that mimic the genetic mutation process. These are interesting algorithms that can be useful in many situations, including their application to some software project management problems. Such algorithms could be a part of the core of software engineering knowledge, but they are never mentioned in many educational programs. Currently, it is not clear whether this approach would be accepted as a part of the core knowledge required of software engineers. However, if we had established an engineering discipline, it would be clear.

University researchers and educators, even those who claim to be in software engineering, are rarely involved in establishing a regulated profession.

This particular programming approach does pose one problem for professional engineers, who are responsible for assuring that their product is fit for its intended use. Providing such assurance is difficult, though not impossible, if the product's performance depends on future evolution.

Space applications

In "Software Engineering for Space Exploration," Robyn Lutz provides an excellent introduction to the role of software in space exploration and describes the problems that programmers and engineers have had to solve in that application area.

This article illustrates why we need to develop a software engineering discipline in general rather than granting degrees in narrower fields such as space software engineering, aircraft software development, or game design. The problems that Lutz describes arise in many software applications. The software field has developed into a set of cliques, each with its own terminology and technologies. These differences in terminology conceal the common principles and lead to duplication, confusion, and some unnecessary disagreements.

Establishing a core body of knowledge would enable and enhance the transfer of ideas and technology between various application areas. It would also reduce the unnecessary differences in terminology between suppliers.

Service-oriented software and cloud computing

In "Software Engineering Meets Services and Cloud Computing," Stephen S. Yau and Ho G. An describe an architecture for an important class of Web-based systems. Although the applications are new and modern high-speed communication networks make the distributed structure practical, the architectural approach is reminiscent of others that work well in other applications. The reader might want to compare the service-oriented approach with the output-oriented approach described in numerous reports.[9-11]

Although a huge number of articles have been written about software engineering and many interesting ideas have been proposed, researchers and practitioners have failed to create a new engineering discipline focused on building software-intensive systems.

While each of the other articles in this special issue claims to discuss the intersection of software engineering with some other research area, they support my position:

- If we are still writing papers arguing that we need some theory or mathematics to be a profession, rather than arguing about specific areas of theory that a software engineer should know, we have not established a profession.
- If we find that individual application areas are discovering common problems and solving them with their own terminology and specialized techniques, we have not yet established a profession.
- If we do not consistently distinguish between engineering problems, management problems, technology problems, and business-plan issues, we have not yet established a profession.
- If we are, as many others have noted, a field that is dominated by fads with clever acronyms that cause a flurry of interest and then fade away, we have not yet established a profession.
- If we continue to treat software engineering as a "grab bag" research area rather than as a regulated profession, we have lost sight of the original goal.

Those who began the software engineering movement were prescient. They seem to have anticipated today's heavy dependence on software; they must have recognized that software engineering should become one of the many established engineering disciplines. Unfortunately, we who came after them underestimated the difficulty of achieving that goal and have lost sight of it.

If we want to establish a discipline of engineering that specializes in software-intensive systems, the first step is to agree on a core body of knowledge. Thus far, the various efforts to establish a body of knowledge have been too inclusive. They have tried to collect every belief and fact about software development rather than identify a small core of solid knowledge that all software engineers must master.

In my experience, establishing a core will not be easy. Unless we are vigilant, the core will continue to expand and, consequently, lose relevance because it is too large. C

Acknowledgments

I thank Carl Chang and David Weiss for their thought-provoking comments on an earlier version of this paper.

References

1. D.L. Parnas, "Risks of Undisciplined Development," *Comm. ACM*, Oct. 2010, pp. 25-27.
2. D.L. Parnas, "Software Engineering Programmes Are Not Computer Science Programmes," *Ann. Software Eng.*, vol. 6, 1998, pp. 19-37.
3. D.L. Parnas, "Stop the Numbers Game." *Comm. ACM*, Nov. 2007, pp. 19-21.
4. D.L. Parnas, "Really Rethinking 'Formal Methods,'" *Computer*, Jan. 2010, pp. 28-34.
5. D.L. Parnas, "Licensing Software Engineers in Canada," *Comm. ACM*, Nov. 2002, pp. 96-98.
6. D.L. Parnas, "How Engineering Mathematics Can Improve Software," *Proc. Int'l Conf. Eng. Reconfigurable Systems and Algorithms* (ERSA 2011); http://ersaconf.org/ersa11/#hdisplay.
7. D.L. Parnas and M. *Soltys, Basic Science for Software Developers*, SQRL report no. 7, Software Quality Research Laboratory, Dept. of Computing and Software, McMaster University, 2002; www.cas.mcmaster.ca/sqrl/sqrl_reports.html.
8. D.L. Parnas, "Mathematics of Computation for (Software and Other) Engineers," *Bull. European Assoc. Theoretical Computer Science*, Oct. 1993, pp. 249-259.
9. K.L. Heninger, "Specifying Software Requirements for Complex Systems: New Techniques and Their Application," *IEEE Trans. Software Eng.*, Jan. 1980, pp. 2-13.
10. D.L. Parnas, "Precise Documentation: The Key to Better Software," *The Future of Software Engineering*, S. Nanz, ed., Springer, 2010, pp. 125-148.
11. Z. Liu, D.L. Parnas, and B. Trancón y Widemann, "Documenting and Verifying Systems Assembled from Components," *Frontiers in Computing Science in China*, June 2010, pp. 151-161.

David Lorge Parnas *is professor emeritus at McMaster University, Canada, and the University of Limerick, Ireland, as well as president of Middle Road Software. Parnas received a PhD in electrical engineering from Carnegie Mellon University and honorary degrees form ETH Zurich, the University of Louvain, the University of Italian Switzerland, and the Technical University of Vienna. Contact him at parnas@mcmaster.ca.*