

Will the Real Software Engineer Please Stand up?

Frank D. Anger
Division of Computing & Communications Foundations
National Science Foundation
Arlington, Virginia, USA
fanger@nsf.gov

Three significant and peculiar characteristics of software engineering profoundly impact the way in which we should think about software engineering education: the effectiveness of current practice, the precipitous rate of change in the meaning of “software,” and the gulf between software engineering research and practice.

Effectiveness: According to the wags, SE has been in constant crisis ever since computer users had to depend on others to create their software. More recently, the PITAC Report claimed (in slightly different language) that we cannot produce the needed software at the pace we need it, and the software produced breaks too easily. Just as 30 years ago, the slogan “faster, better, cheaper” still sums up today the basic needs in software production. While a doom and gloom attitude about the sorry state of software often predominates, the overwhelming accomplishments achieved by software intensive systems appear to tell a quite different story: software pervades the technologically advanced societies; software controls our cars, our planes, our telephones, and our washing machines; software underpins and drives the economy.

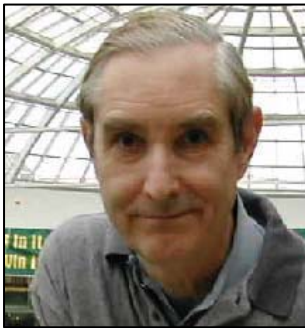
In such a schizophrenic atmosphere, the question of the effectiveness of software engineering as a profession invites discussion, and consequently the question of what constitutes a proper software engineering curriculum always raises controversy. If we consider software engineers as national heroes for their accomplishments, we should pass along the tools and techniques used by them to the next generation; if, however, we believe that their inability to end the software crisis has its roots in the lack of systematic knowledge about how to build large software systems, then we must demand a different emphasis for educating the next generation.

Change: The nation, and much of the world, has moved in just a few decades from a software-poor to a software-rich environment. Software systems have grown from 1000 lines of code to 10 million lines of code. Applications have spread from scientific and financial computations to control and communication protocols. Software production teams have increased from groups of 2 to groups of 2000 and from co-located to globally distributed. Platforms have graduated from single CPU to distributed. Programming paradigms undergo a complete transformation every decade. In a word, the tasks faced by a software developer in the middle of the twentieth century relate only remotely to those at the beginning of the twenty-first.

What has happened to software engineering education while these vast changes have taken place? In the 60s, not even computer science formed part of the curriculum; in the 70s interest and courses grew in systems and data analysis; in the 80s software engineering courses appeared, but few majors; while in the 90s, many educational institutions finally advertised graduate degrees in software engineering and some limited number of undergraduate degrees. In modern parlance, emphasis has gone from implementation to requirements to project management to life-cycle and product-family approaches. While this progression appears reasonable, one conclusion stands out: the profession changes so rapidly that even if universities “get it right” at some point, graduates will soon find their preparation dated.

Research versus practice: Software engineering education can play three roles: provide a broad class of *IT professionals* with an appreciation of the difficulty of going from problem to software solution, provide *software developers* with the techniques and experience they need to approach systematically the creation or maintenance of large software systems, and provide *software researchers* with the understanding and research methodologies that will allow them to recognize important issues and to conduct effective research projects. When teaching science, the distinction between practitioner and researcher need not be a concern; when teaching engineering the distinction becomes more pronounced; and when teaching software engineering, this difference takes on great importance. Why? Unlike most engineering disciplines, the underlying science of software remains unknown or ignored. Whereas a civil engineer can learn the physics behind stress testing and a chemical engineer can learn the chemistry behind quantitative analysis techniques, what can be taught, or is being taught, to a software engineer about the science behind UML or test coverage? What research does one do on a technique taught as “best practice” rather than as a logical outgrowth of scientific principles?

Several current activities at the National Science Foundation and other federal agencies indicate a deep concern for the problems of software engineering in particular and more generally the issues of creation, management, and understanding of software and software-intensive systems. This presentation looks at the three characteristics of software engineering laid out above in the context of these federal initiatives and the implications for software engineering education.



Brief Bio: Frank Anger currently serves NSF as Deputy Division Director for the Division of Computing and Communications Foundations. He holds a B.A. in Mathematics from Princeton University, a Ph.D. degree in Mathematics from Cornell University and a Ph.D. in Computer Science from the University of Florida. Dr. Anger has been Professor of Computer Science at University of W. Florida and Florida Tech, and on the mathematics faculty of the University of Puerto Rico, the University of Auckland (New Zealand), University of Kansas, and Massachusetts Institute of Technology. He has published over 60 papers covering a wide range of topics, although his latest work, in collaboration with Rita Rodriguez, has focused on the use of temporal modeling and analysis of distributed systems.

Anger was a founder of the *Software Engineering Research Forum* (SERF) and of the *Florida Artificial Intelligence Research Symposium* (FLAIRS), and has served as Vice President and as Treasurer of the International Society of Applied Intelligence (ISAI). He has been the principal organizer of a series of workshops on Spatial and Temporal Reasoning held at the world's major AI conferences (IJCAI, AAI, and ECAI) each year since 1993, and was Program Chair of *IEA/AIE-94*. He has also served on the editorial board of the *Applied Intelligence Journal*.

As Program Director at NSF, his duties have included the direction of research funding programs, organization of workshops, planning of new initiatives, and coordinating with other government agencies, both in DC and abroad. He has, during the past few years, played a principal role in defining a number of directions for the software research enterprise, both through internal NSF initiatives in response to the PITAC Report and also as a member of three federal interagency committees: High Confidence Software and Systems, Critical Infrastructure Protection, and (Co-chair of) Software Design and Productivity. In 2001-2002, he led NSF's largest research program, Information Technology Research, in addition to acting as Deputy Division Director.