

Half a Century of Software Engineering Education

The CMU Exemplar

Nancy R. Mead, David Garlan, and Mary Shaw,
Carnegie Mellon University

// Now, more than ever, the software profession needs rigorous software engineering education based on enduring principles. This must reach the increasing diversity of the developer community and the diverse paths the developer takes into the profession. //



THE TERM “software engineering” first appeared at the NATO Software Engineering Conference in 1968.¹ We use this definition: Software engineering is the branch of computer science that creates practical, cost-effective solutions to computing and information-processing problems, preferentially by applying scientific

knowledge, developing software systems in the service of mankind.²

In the Carnegie Mellon University (CMU) view, software engineering rests on three principal intellectual foundations:

- The technical foundation is a body of *core computer science*

concepts relating to data structures, algorithms, programming languages and their semantics, analysis, computability, computational models, etc. This provides the essential technical content of the discipline.

- This technical knowledge is applied through a body of *engineering knowledge* related to architecture, the process of engineering, tradeoffs and costs, conventionalization and standards, quality and assurance, etc. This provides the approach to design and problem solving that respects the pragmatic issues of the applications.
- These are complemented by the *social and economic context* of the engineering effort, which includes the process of creating and evolving artifacts, as well as issues related to policy, markets, usability, and socioeconomic impacts. This provides a basis for shaping the engineered artifacts to be fit for their intended use.

This view stands in contrast to the view, lingering from the 1980s and still implicit in some textbooks, that software engineering should be concerned principally with software project management and software development processes.

Software Engineering Education History

As educators in the 1960s, we drew on early publications about structured programming, including the “Go To Considered Harmful” discussion,³ and on algorithms, data structures, and programming languages. It’s worth noting that at that time there were no software engineering degree programs at universities, so much of the discourse

was aimed at industrial software engineering practice.⁴ There were, nevertheless, courses on software development principles. At CMU at the time of the NATO report, even the introductory programming course emphasized problem solving, with programming as a means to that end. Other common courses covered data structures, compilers, and OSs.

In the late 1960s, some software development organizations started to offer extensive training programs for their employees. At the IBM Federal Systems Division in the 1970s and 1980s, a series of courses entitled Structured Programming Workshop, Structured Design Workshop, Advanced Design Workshop, and Software Management Workshop were developed. Every programmer in the division was required to attend some of the courses and pass the associated exams. Although it doesn't seem like much now, taking programmers off the job for two to three weeks and housing them at a central education location was a big commitment at the time.

The CMU curriculum design philosophy is based on the Carnegie Plan for education, which calls for students to learn concepts, critical thinking, and problem-solving skills that will serve them both as citizens and as professionals, together with the interest and ability to learn new technologies as they emerge. This has guided, and continues to guide, CMU curricula.

In keeping with this philosophy, in the 1970s CMU replaced the then-popular data structures course with a course centered on abstraction and verification of abstract data structures.⁵ In the 1980s, the so-called compiler course was recognized as not a vocational course in compiler building but a course in advanced algorithms and

data structures, language-processing techniques, and the structure of multi-component systems.

In the 1980s, IBM created its own Software Engineering Institute dedicated to software engineering education, with an extensive in-house instructor-training program. Thousands of software developers and managers attended these workshops, the completion of which was tracked by upper management. The early workshops resulted in the Linger-Mills-Witt book on structured programming,⁶ which is still in use. IBM's in-house course offerings were supplemented by short computer science courses taught by university faculty at IBM facilities. For example, a version of the CMU course on abstraction and verification of abstract data structures was offered for the IBM Information Systems Division around 1980.

In the late 1970s, master's degree programs in software engineering started to emerge. Some of the early programs were offered at the Wang Institute of Graduate Studies, Seattle University, and Texas Christian University. There were, however, no standard software engineering curricula at that time, so each institute developed its own. Software engineering textbooks began to appear in the 1980s. At that time, CMU was holding off on an undergraduate computer science degree, let alone a software engineering degree, because the field did not yet have a substantial, mature body of principled content.

MSE Programs, Conferences, and Workshops

During the mid 1980s, the Software Engineering Institute (SEI) was founded at CMU. One of the key

elements of the original SEI charter was to advance software engineering education. Norm Gibbs led the creation of *curriculum modules* as resource material on specific topics that encapsulated knowledge in a form useful for instructors, especially instructors with little direct software engineering experience.⁷ The SEI also led the creation of the Conference on Software Engineering Education (CSEE), of which the first instance was held in 1987.

As a consequence of these early workshops and work at the SEI, a new master's in software engineering (MSE) model curriculum was developed by the SEI's Education Program for the education of professional software engineers, and an MSE program was started at CMU in 1989. The courses in the initial curriculum focused on traditional software lifecycle activities, such as project management, requirements, design, and testing. An important element of the MSE was the Studio project,⁸ a software development project for real customers that was carried out in teams, ran the length of the program, and emulated the idea of an architectural studio. The studio-based MSE program at CMU was recently recognized by the 2017 IEEE Conference on Software Engineering Education and Training's Hall of Fame award.

The MSE graduated its first class of 10 students in 1991. During the 1990s, the number of MSE programs grew rapidly. Recognizing the need for a body of educational material to assist such programs, the SEI developed and distributed a variety of instructional modules in the form of videotaped courses, curriculum modules, and other educational materials. Many universities used these as a springboard, while

others developed programs that were uniquely their own. In the late 1990s, the various programs came to reflect the specialties and emphases of their universities.⁹

At CMU, over time it was decided to revise the CMU MSE curriculum, refactoring it so that the major structure emphasized concepts rather than waterfall lifecycle activities. This yielded a set of courses that provided a different partitioning of major curriculum topics, with courses on architecture, analysis, methods, and models, as well as management.¹⁰ Continuing the practice of designing new courses around concepts rather than activities, this redesign was supported by the first textbook on software architecture¹¹ and a course that replaced the customary requirements course with one on methods for high-level design.¹²

One of the hallmarks of the MSE program was that it was aimed at software practitioners with substantial experience. As a consequence, in the late 1980s, the SEI started a continuing-education group to focus on the needs of practitioners and executive managers. This group initially developed a set of courses that were similar to the MSE courses.¹³ Later, it branched out into courses that were unique to practitioners in the field but not necessarily part of a degree program, and offered a series of courses aimed at software executives.

The CSEE conference series broadened its focus to include training, and in 1997 became known as CSEE&T. A number of additional educational conferences, and tracks in major conferences such as the International Conference on Software Engineering (ICSE), now exist.

In 1995, the Working Group on Software Engineering Education and Training was formed, with a goal to

advance software engineering education. It continued for a number of years, publishing papers and reports on topics such as curriculum development, industry–university collaboration, and professionalism.

One important trend in software engineering education has been the recognition of the importance of collaborating with industry, as well as influencing it. Early MSE programs were among the first to explore deep industry–university collaborations, in several cases establishing industry advisory boards so that their degree programs would remain relevant to their industry partners. Although comprehensive data is not available, at CMU and some other universities, the majority of MSE students were industry-sponsored.

A joint committee was formed by ACM and the IEEE Computer Society in 1993 to promote software engineering as a profession. The Software Engineering Coordinating Committee supported development of the first *Software Engineering Body of Knowledge* (SWEBOK). The Computer Society started to offer a certification program that was initially based on SWEBOK but later evolved to include several levels of certification. The exams were professionally developed and tested extensively.

After several years of work, in 2009, an independent working group published a model curriculum for professional degree programs in software engineering, driven strongly by the CMU model of software engineering education.² As a model curriculum, it was designed to be adapted to the needs of individual institutions. It was accepted or endorsed by the Computer Society, ACM, and INCOSE (International Council on Systems Engineering).¹⁴

Core Principles for Software Engineering Curricula

The turn of the 21st century prompted CMU to take stock of the field. One of the outcomes was an explicit statement of the curriculum design principles that had guided most curriculum design since the undergraduate offerings of the 1980s, including the major redesign of the MSE program in the 1990s. These principles (see the “Core Principles” sidebar) flesh out the intellectual foundations identified above. They transcend technology evolution, so they can be expected to guide future curricula.

Beyond general principles, which endure, a well-designed curriculum should identify program outcomes in terms of the capabilities of its graduates, using, for example, the Bloom taxonomy to establish the desired level of proficiency. Furthermore, the curriculum should support traceability from these capabilities through its courses and other activities back to its principles. The Bloom level of proficiency for each capability will, of course, differ for baccalaureate, master’s, and doctoral programs.

These outcomes evolve over time. In 2005, the CMU set included about a dozen capabilities, including these:

- *Computer science proficiency.* Be able to program effectively; use current tools and frameworks; and apply abstract concepts, theories, and models such as algorithms and architectures. In addition, be able to learn new concepts and technologies as they emerge.
- *Engineering proficiency.* Be able to translate client needs to system requirements, handle

CORE PRINCIPLES

The following principles flesh out the intellectual foundations discussed in the main article.

COMPUTER SCIENCE FUNDAMENTALS

- Abstraction enables the control of complexity.
- Imposing structure on problems often makes them more tractable, and a number of common structures are available.
- Symbolic representations are necessary and sufficient for solving information-based problems.
- Precise models support analysis and prediction.
- Common problem structures lead to canonical solutions.

ENGINEERING FUNDAMENTALS

- Engineering quality resides in engineering judgment.
- The quality of the software product depends on the engineer's faithfulness to the engineered artifact.
- Engineering requires reconciling conflicting constraints.
- Engineering skills improve as a result of careful systematic reflection on experience.

SOCIAL AND ECONOMIC FUNDAMENTALS

- Costs and time constraints matter, not just capability.
- Technology improves exponentially, but human capability does not.
- Successful software development depends on teamwork by creative people.
- Business and policy objectives constrain software design and development decisions as much as technical considerations do.
- Software functionality is often so deeply embedded in institutional, social, and organizational arrangements that observational methods with roots in anthropology, sociology, psychology, and other disciplines are required.
- Customers and users usually don't know precisely what they want, and it is the developer's responsibility to facilitate the discovery of the requirements.

("wicked") problems, work within budgets, and communicate effectively.

Over the past decade, additional capabilities have been recognized as important. Computer science proficiency was augmented to include the use of quality assurance tools (model checkers, static analyzers, testing tools, etc.) and the use of data analytics and machine learning to improve both products and processes. Engineering proficiency now includes Internet-scale systems engineering and a stronger focus on qualities such as availability, security, and privacy. Social and economic proficiency has come to include the role of open source ecosystems and the fundamentals of product management. The Studio also has evolved to provide a more structured set of guidelines for evaluation and incremental milestones, although the focus on real-world, industrially sponsored projects remains unchanged.

CMU additionally established a PhD program in software engineering, in which the expected capabilities of graduates emphasize the ability to perform research in software engineering, rather than proficiency in professional practice. Graduates of the program are positioned to enter academic and industrial research positions in software engineering.

A number of international locations now offer variants on the CMU degree programs, including locations in Australia, Greece, Japan, Portugal, and Qatar. There are also international partner universities, offering their own degree programs in software engineering, while benefiting from collaborative arrangements with CMU.

tradeoffs among conflicting objectives, evaluate alternatives, apply responsible engineering approaches to design, and work effectively within existing systems.

- *Social and economic proficiency.* Be able to apply software development processes, organize and lead teams, work effectively in interdisciplinary contexts, handle unstructured open-ended



UPCOMING CHALLENGES

The challenges we see for software engineering education include the following:

- Revise the curricula for university degree programs to address current needs such as highly distributed adaptive systems, X as a service, continuous deployment and DevOps, software systems that interact with networked physical devices, autonomy and machine learning, and privacy and security—while still rooting the curriculum in durable principles. These revisions should emphasize engineering aspects of development, especially reliability and security.
- Find ways to reach the practitioners who are entering the field in ad hoc ways and provide education that will serve them well as they proceed beyond entry-level programming tasks. Clearer definitions of job roles and responsibilities would help to clarify the skills needed and provide a career path.
- Work with educators in other fields to incorporate elements of End-User Software Engineering¹⁶ in the curricula of professionals who will be dealing with software. At the same time, raise the design standards for the software and tools that they use, to make it easier for people who are not highly trained computer professionals to achieve quality results.
- Since so many students have degrees in computer science and not software engineering, infuse engineering attitudes and techniques in all undergraduate courses. Similarly, teach the appropriate level of software engineering to students in the many specialized master's degree programs in computing fields, such as computer vision, machine learning, computational biology, and language translation.
- Incorporate social, ethical, and policy implications of computing technologies as an integral part of the curriculum. Professional societies have had ethics standards for years, and many universities offer ethics courses in their programs. But isolated courses don't reach all students, and transfer to other courses is challenging. As technology enables new applications, software engineers need to be able to reason not only about how to create those applications but also about how design choices affect their social impact. Software engineers should also be able to contribute to the public discourse about policies that govern the deployment and use of technology.

Relevance of Software Engineering Education for Practitioners

In the current software development world, the number of developers with software engineering (or, more broadly, computer science) degrees is small compared to the number of people developing or modifying software. There seem to be two reasons for this. First, most professions now involve programming-like activities, from using pivot tables in spreadsheets, to improving text-formatting macros, to developing websites in high-level tools, to composing existing components that perform complex data analysis tasks.

Second, a surprising number of full-time software developers have

come into the practice through routes other than formal education in computer science or software engineering. In many companies, most programmers have learned their software development skills on the job (often self-taught). A 2016 survey found that only 19.7% of software developers had master's degrees in computer science or related fields.¹⁵ Furthermore, the number of respondents who reported being self-taught was higher than the number with bachelor degrees in computer science or related fields.

The risk posed by this pattern is that while informally educated practitioners may be able to create small pieces of code, they may be neither aware of nor able to deal with the

extended implications, side effects, and maintenance requirements that are intrinsic to the integrity of real-world software systems.

Another source of risk for software engineering education is that even when students do have some formal education in programming, they often have little exposure to the software engineering capabilities outlined above. This trend is exacerbated by the view of many companies that programming skills are a sufficient background for new hires, who they believe will learn software engineering after employment, even assuming the companies recognize that need. But once on the job, few employees have the time or financial incentives to acquire the software

ABOUT THE AUTHORS



NANCY R. MEAD is a Fellow Emeritus of the Software Engineering Institute and an adjunct professor of software engineering at Carnegie Mellon University's Institute for Software Research. Her interests are in software security and software requirements engineering research, and the development of software engineering and software assurance curricula. Mead received a PhD in mathematics from the Polytechnic Institute of New York. She's a Life Fellow of IEEE and a Distinguished Educator of ACM. Contact her at nrmcmu@gmail.com.



DAVID GARLAN is a professor of computer science and an associate dean at Carnegie Mellon University's Institute for Software Research. His interests include software architecture, self-adaptive systems, formal methods, and cyber-physical systems. Garlan received a PhD in computer science from Carnegie Mellon University. He's a Life Fellow of ACM and IEEE. Contact him at garlan@cs.cmu.edu.



MARY SHAW is the Alan J. Perlis University Professor of Computer Science at Carnegie Mellon University's Institute for Software Research. Her research focuses on software engineering and software design, particularly software architecture and the design of systems used by real people. Shaw received a PhD in computer science from Carnegie Mellon. She's a Life Fellow of ACM and IEEE. Contact her at mary.shaw@cs.cmu.edu.


engineering skills that are taught in degree programs.

A third area of risk for software engineering education is the ability to remain current in areas where technology is changing rapidly and where industry is often at the vanguard of new approaches to software development. Although degree programs may stress foundations and lifelong skills, they also need to show how these are relevant to current industrial contexts. Furthermore, as computing penetrates ever more deeply into everyday life, the

ability to help the public make informed choices about how to manage that technology becomes ever more important.

As a result, we see several upcoming challenges for software engineering educators and, indeed, the software engineering profession (see the "Upcoming Challenges" sidebar).

We find that thoughtful practitioners still see the value of advanced degrees that will enable them

to acquire habits and practices that will serve them well over the course of their careers. Our belief at CMU is that software engineering education needs to consider the long-term benefits to practitioners and to help them to excel in our profession. Our challenge as educators is to continue to support practitioners in developing deep software engineering expertise that will last a lifetime. 

Acknowledgments

We acknowledge the organizations, especially Carnegie Mellon University, that supported our work in software engineering education and the many colleagues with whom we've collaborated on curriculum development, course design, and teaching. We thank past students who continue to remind us how formative and valuable their professional education has been and to educate us on the things we should be teaching.

References

1. P. Naur and B. Randell, eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, NATO, 1968; <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/index.html>.
2. M. Shaw, ed., *Software Engineering for the 21st Century: A Basis for Rethinking the Curriculum*, tech. report CMU-ISRI-05-108, Feb. 2005; <http://reports-archive.adm.cs.cmu.edu/anon/anon/usr/ftp/isri2005/abstracts/05-108.html>.
3. E.W. Dijkstra, "Go To Statement Considered Harmful," *Comm. ACM*, vol. 11, no. 3, 1968, pp. 147–148; doi:10.1145/362929.362947.
4. N.R. Mead, "Software Engineering Education: How Far We've Come and How Far We Have to Go," *J. Systems and Software*, vol. 82, no. 4, 2009,

- pp. 571–575; doi:10.1016/j.jss.2008.12.038.
5. W.A. Wulf et al., *Fundamental Structures of Computer Science*, Addison-Wesley, 1981.
 6. R.C. Linger, H.D. Mills, and B.I. Witt, *Structured Programming: Theory and Practice*, Addison-Wesley, 1979.
 7. D. Budgen and J.E. Tomayko, “The SEI Curriculum Modules and Their Influence: Norm Gibbs’ Legacy to Software Engineering Education,” *J. Systems and Software*, vol. 75, nos. 1–2, 2005, pp. 55–62; <http://dx.doi.org/10.1016/j.jss.2004.02.027>.
 8. J.E. Tomayko, “Carnegie Mellon’s Software Development Studio: A Five-Year Retrospective,” *Proc. 9th Conf. Software Eng. Education*, 1996, pp. 119–129.
 9. D.J. Bagert and X. Mu, “Current State of Software Engineering Master’s Degree Programs in the United States,” *Proc. 35th Ann. Conf. Frontiers in Education*, 2005; <http://ieeexplore.ieee.org/iel5/10731/33854/01612026.pdf>.
 10. D. Garlan, D. Gluch, and J.E. Tomayko, “Agents of Change: Educating Future Leaders in Software Engineering,” *Computer*, vol. 30, no. 11, 1997, pp. 59–65; <http://ieeexplore.ieee.org/document/634865>.
 11. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
 12. M. Shaw et al., “Deciding What to Design: Closing a Gap in Software Engineering Education,” *Proc. 2005 Int’l Conf. Software Eng. (ICSE 05)*, 2005, pp. 28–58.
 13. N.R. Mead and P. Lawlis, “Software Engineering: Graduate-Level Courses for AFIT Professional Continuing Education,” *Proc. 5th SEI Conf. Software Eng. Education*, 1991, pp. 114–126.
 14. M. Ardis et al., “Advancing Software Engineering Professional Education,” *IEEE Software*, vol. 28, no. 4, 2011, pp. 58–63; doi:10.1109/MS.2010.133.
 15. *Developer Survey Results 2016*, Stack Overflow, 2016; <https://insights.stackoverflow.com/survey/2016>.
 16. A.J. Ko et al., “The State of the Art in End-User Software Engineering,” *ACM Computing Surveys*, vol. 43, no. 3, 2011, article 21; <http://dx.doi.org/10.1145/1922649.1922658>.

**SUBMIT
TODAY**

IEEE TRANSACTIONS ON MULTI-SCALE COMPUTING SYSTEMS

**SUBSCRIBE
AND SUBMIT**

For more information
on paper submission,
featured articles, calls for
papers, and subscription
links visit:

www.computer.org/tmscs

TMSCS is financially cosponsored
by IEEE Computer Society, IEEE
Communications Society, and
IEEE Nanotechnology Council

TMSCS is technically cosponsored
by IEEE Council on Electronic
Design Automation



IEEE
computer
society

myCS

Read your subscriptions
through the myCS
publications portal at

<http://mycs.computer.org>