Software Engineering Education: A Roadmap

Mary Shaw
Institute for Software Research, International
Carnegie Mellon University
Pittsburgh Pa 15213
+1 412 268 2589
mary.shaw@cs.cmu.edu

ABSTRACT
Software's increasingly critical role in systems of widespread significance presents new challenges for the education of software engineers. Not only is our dependence on software increasing, but the character of software production is itself changing – and with it the demands on the software developers. Four challenges for educators of software developers help identify aspirations for software engineering education.

Keywords
Software engineering, education, software profession, credentials

1 INTRODUCTION

As we enter the new millennium, software-intensive systems have become essential parts of everyday activity and of business in the global economy. The quality of this software depends on an adequate supply of proficient and up-to-date software developers.

Currently, software developers are educated in the traditional ways. Unfortunately, this has not produced the supply and quality of developers needed to satisfy the growing demand. In addition, traditional education makes scant provision for helping students keep their knowledge current. Since the software field does not distinguish well among different development roles, education for software engineers is confounded with education for programmers and other non-engineers.

Over the next decade, education for software developers should prepare students differently for different roles, infuse a stronger engineering attitude in curricula, help students stay current in the face of rapid change, and establish credentials that accurately reflect ability.

The essential challenges are world-wide problems. Although I describe them in terms of specific examples from the United States, the overall implications are global. Future-looking papers often make predictions. Such predictions consider possible events, good or bad, and try to select the most likely. Instead, I state *aspirations* – projections of desirable outcomes that might come to pass with good luck, good judgment, and good taste.

2 CURRENT STATUS

Software developers are now educated in much the same way as they have been for years, with the recent addition of on-line training for computing skills. However, pressures arising from the changing character of software and from external pressures on educational institutions will require changes in what we teach software developers and how we teach it.

Current status of software education

Over the past three decades, software developers have been educated in traditional ways: undergraduate and graduate programs in colleges and universities, vocational courses and in-house training, and personal initiative in learning new techniques.

Tomayko [14] identifies three periods in the history of software engineering education: the era of single free-standing courses (prior to 1978), the early graduate programs (1978-88), and the rapid spread of graduate programs influenced by the Software Engineering Institute's efforts (since 1988).

The 1998 FASE survey of graduate software engineering programs [5], although incomplete, identifies graduate programs at 77 institutions worldwide. Most of these institutions offer a masters program in some software-related area; nine offer a PhD with software engineering electives. Software engineering PhD programs are also beginning to appear, for example at Carnegie Mellon University [2]. These programs differ in content emphasis: for example, some masters-level programs are principally concerned with management of software activities, whereas others are chiefly technical. They also differ in career emphasis: PhD programs, by their nature, prepare graduates for research and college teaching positions – though many PhD graduates choose to work in industrial development instead. Some of the masters programs are academic programs, preparation for PhD programs. Many of the

masters programs are designed to prepare their graduates for professional practice at a high level of technical or management responsibility (and not for entry to a PhD program).

Most universities now offer undergraduate degrees in computer science, and most provide an extensive selection of software-related courses. These programs typically allow a student to study software design and implementation topics, and they provide a common educational base for entry-level programming positions. For a decade or more, some members of the software education community have advocated undergraduate software engineering degrees separate from computer science. Such programs are intended to provide a better base for a software development career than would a traditional computer science program; the prospects that this will be the case are discussed below. Tomayko [14] notes that we are now entering a new era with the introduction of these undergraduate software engineering programs, but they are not yet widespread.

Specific software development skills are taught outside the university system, in vocational schools, in-house courses, or short courses. They differ in length, in cost, and in the degree to which skills are transferable to other tasks. Some of these lead to vendor certifications of proficiency with specific products.

Notwithstanding all these opportunities, we hear regular complaints of severe shortfalls in the numbers of available software developers.

Current forces on software development:
As software becomes ubiquitous, the relation between end users and software development is undergoing fundamental changes. Some of these changes have to do with the evolving character of software; others result from increasing pressure for recognized professional credentials

Evolving software development models
The prevailing model of software development, on which most educational programs is based, involves a team of professional software developers in a single institution working under a well-defined process and product cycle to produce software for a known client and deliver it on known schedule. This *closed-shop software development model* is increasingly at odds with actual practice.

Some of the discrepancies between the closed-shop model and modern software include:

- System requirements emerge as the clients understand better both the technology and the opportunities in their own settings, and the clients are intimately involved in this progressive development. This often requires software development to be done concurrently with business re-engineering.
- The systems must be designed and fielded under complex economic and legal constraints that affect system design, and they are often distributed hardware/software systems, not pure software. Most educational programs underplay the significance of these additional constraints.
- Software, especially low-level system software, is now being developed by communities of cooperating volunteers [8]. In open-source software, the code is published freely and interested users critique it and propose changes. Quality arises by an intense, highly parallel social process with rapid feedback rather than by a carefully managed process.
- Software is often developed by creating coalitions of existing resources that are not under control of the software developer [12]. The resources include calculation, communication, control, information, and services; they are often distributed, dynamic, autonomous, and independently managed. They may be modified or decommissioned without notice to users. This *open-shop development model* is a major departure from the usual closed-shop model, and the uncertainties associated with externally-managed resources require correspondingly more sophisticated analysis.
- Software development is increasingly disintermediated – software is adapted, tailored, composed, or created by its end users rather than by professional software developers. These end users need to understand software development in their own terms; they particularly need ways to decide how much faith to have in their creations.

To respond to these forces, educational institutions must prepare professional software developers to construct and analyze systems that are heavily constrained by non-technical considerations and that depend on independent distributed resources. In addition, professional software developers must learn to create resources that are sufficiently trustworthy to be used and tailored by non-professionals.

Professional credentials
Software is increasingly of public importance, both as an essential element in engineered systems and as the principal embodiment of capabilities whose failure is of nontrivial consequence to the public at large or individual members of the public. The public wants and deserves assurances about the quality of both systems with embedded software and systems that are principally embodied in software. We can gain confidence in the quality of the product directly – through product validation – or by having prior confidence in either the people who produce the software or the organization that manages its production. Many technologies – most notably testing, design and code reviews, and formal analysis – support product validation. The Capability Maturity Model and ISO 9000 certification address organizational quality. But credentials for professionals are still in their infancy.

There is currently considerable pressure worldwide for professionalization of software engineering. In the United States, this currently takes the form of a debate over the merits of professional licensing of software engineers. The argument in favor of licensing is that we, like other engineering disciplines, should set standards for the

practice, that there is substantial demand for a way an employer or client can easily establish the competence of a software developer, and that licensing would improve the quality of practice. The argument against licensing at this time is that professional licensing carries a commitment to the public that we can achieve a level of practice that provides certain safety and utility properties of the product, but such a level of practice is not yet routinely achieved; that a licensable practice of software engineering has not been distinguished from other aspects of software development, and that licensing has a narrow range of applicability (to matters of pubic interest). A task force chartered by the ACM and IEEE is attempting to define the "body of knowledge" that a software engineer should master [13]. Interestingly, there is little effort to distinguish engineering responsibilities from other development tasks.

In addition, a number of software vendors certify proficiency in the use of specific products. The diversity and specificity of these credentials are evident from some examples: Certified Novell Engineer or Administrator, IBM's Application Development Certifications in XML and VisualAge, Microsoft Certified Systems Engineer or Database Administrator, Oracle Certified Professional tracks, Sun Certified Programmer or Developer for Java, Sun Certified System or Network Administrator, the multi-vendor Certified Internet Professional. These certifications are often specific to a particular version of the application, making them even more narrow.

Credentials that are less broad than professional licensing but broader than product proficiency do exist; however, they are not widely issued and recognized.

Current forces on educational institutions

Incentives for changing the way we educate software developers arise not only from changes in the way software is developed but also from institutional pressures.

Universities have long felt the tension between an internal value system that emphasizes education in enduring principles and the demands of employers who want focused training in current technology. Different schools strike the balance in different places, with general agreement that neither extreme is appropriate. Several recent developments intensify the tension, though.

First, the educational community itself is increasingly moving from lecture-format courses to team projects, problem-solving, direct involvement with actual development, and other formats that require students to exercise the ideas they are learning.

Second, the shortfall of software developers is so dire that students themselves often face the choice between a well-paid programming job and completing their degrees; this is particularly severe in PhD programs, but it is also an issue for undergraduates. The faculty may find it hard to convince the students that choosing the programming job now limits the students' career paths later – and they may not even be correct.

Third, the institutional structure of universities is increasingly challenged by for-profit schools (calling themselves universities as well as vocational schools, now) that emphasize immediately useful skills, by external critics arguing for increased accountability and efficiency, and by on-line training and education.

3 CHALLENGES AND ASPIRATIONS

This discussion lays the groundwork for identifying four challenges for the software engineering education community and selecting some specific aspirations as targets of progress.

Engineering entails creating cost-effective solutions to practical problems by applying scientific knowledge, building things in the service of mankind [9]. Engineers preferentially apply scientific and mathematical knowledge when it's available and rely on less systematic knowledge at other times. Engineers work under limitations of both time and knowledge. They are responsible for reconciling conflicting constraints, especially cost constraints. Engineers make deliberate choices among alternative designs for both technical and nontechnical reasons [1]. Their judgments are based on deep knowledge of the discipline in which they design, and they assume personal responsibility for the safety and quality of the systems they design. (This view of engineering differs from Maibaum's [6], in that his suggests a largely linear process of creation, with iterative refinement but not revision. Maibaum's view lacks a sense of drawing on accumulated disciplinary experience, of reconciling conflicting constraints, and of the need to generate candidate alternatives at various stages and choose among them.)

I interpret "software engineering" in this sense of engineering, and I'll focus principally on education of these engineers, which should prepare them for technical design and decision-making and for assuming responsibility for the success of their products. I'll refer to the entire community of people involved in software development as "software developers".

Identifying Distinct Roles in Software Development and Providing Appropriate Education for Each

Software development and support requires many skills, including design, management, programming, validation, analysis, user studies, documentation, system integration, and property-specific techniques such as design for security and reliability. While engineers apply most of these skills, not everyone who has any of the skills is an engineer. Despite intermittent attempts to identify specific roles, the distinctions remain unclear. Indeed, a wide variety of software developers, including many with no engineering responsibilities, aspire to the title, "software engineer."

Currently, the ambiguity among software development

roles is mirrored in the educational programs. Universities may offer software development materials in different departments, and these programs may distinguish a software focus from other areas of the respective fields. However, there is rarely a sense of specialization *within* software.

Aspiration 1: Discriminate among different software development roles

Available knowledge about software development far exceeds what any one person can know. Other fields responded to such growth in knowledge by specializing roles. The specialization may be vertical (specialist in an application area such as scientific computation), horizontal (specialist in system security), or by level of responsibility (programmer vs engineer). As fields mature, these divisions become the recognized structure of the field, allowing business as well as personnel specialization. For historical reasons, some distinctions are already well-established in software – for example, database administration and more recently web site development.

It is not yet clear whether vertical or horizontal specialization will serve us better. Progress toward identifying the knowledge required for specific functions will help us understand how to align specialties.

Aspiration 2: Make undergraduate software education a valuable long-term investment

The chief responsibility of universities, especially in undergraduate programs, is to teach essential, durable content that will serve the student for several decades. For both practical and pedagogical reasons, it is appropriate to teach the material with examples from current practice. However, courses with a primary emphasis on current technology in which most of the knowledge will become obsolete when the technology does are better taught in other institutions.

Curriculum design is at heart a resource allocation problem, with curriculum space (as measured by courses, hours of study, number of homework problems and projects, …) as the scarce resource. Courses must earn their places in the curriculum with enough compact, durable content to justify the curriculum space they use. Universities regularly face pressure from potential employers to sacrifice systematic understanding for immediately useful skills. (I first encountered this in the 1960's, when employers asked "teach them more JCL".) Each university must select its own balance between immediate and long-term knowledge.

We should therefore resist the temptation to start up new bachelor's degree programs in software engineering, let alone set up new academic departments. Software engineering does not yet have an independent curriculum with enough durable, codified content to justify a separate undergraduate curriculum. Most of the meaty content overlaps substantially with good computer science content. Undergraduate computer science programs would themselves benefit from adding a stronger engineering sense through most of the curriculum, and the energy required for administering separate programs or departments would be better invested in improving the discipline and the courses.

Further, the professional societies should refrain from dictating curricula. The evidence of the past 30 years is that creative, innovative curricula come from individual colleges and universities, not from large committees whose members have diverse and conflicting interests.

Aspiration 3: Provide for specialization through training and graduate education

As specializations emerge, educational institutions must provide opportunities to master them. The character of these opportunities should depend on the level of responsibility the student will assume. Prospective engineers can begin specialization with undergraduate concentrations and electives, but at our present state of maturity they should expect to spend at least a year of graduate study (or comparable time while working) becoming proficient in the specialty. At the other end of the spectrum, vocational schools, proprietary schools, and in-house training already provides a path to product-specific skills.

Preparation for research, of course, is different from preparation for engineering practice. A researcher needs deeper preparation in underlying principles, in problem formulation, and in validation of results [7] as well as a special kind of inquisitiveness and creativity. PhD programs rely heavily on direct mentorship to develop these skills and talents.

Instilling an Engineering Attitude in Educational Programs

Any student who claims an education in any area of software development must be good at developing software. This requires proficiency in both design and programming; both of these proficiencies require an engineering point of view: resolving constraints, considering users, comparing alternatives, etc. Software development should be treated this way not only for prospective software engineers, but for all students.

We currently include software development courses in undergraduate computer science and information technology curricula. All these students, including the software engineers, should learn the material with the engineering point of view.

We regularly hear complaints about the undergraduate computer science curriculum failing to educate engineers. In many respects, the problem lies with failure of the software development courses to address practical considerations of real software. These problems should be

addressed in the courses for all students; the improvements do not require separate software engineering courses, let alone separate curricula. Moreover, they will improve the curriculum for all students who learn about software, not just the prospective software engineers.

In particular, engineers must consider numerous alternatives and choose the appropriate one for the task at hand. Jackson and Rinard, for example, says "Engineers need different degrees of precision in different situations, at different points in the program, and for different data structures" [4] and goes on to emphasize the need for an engineer to exercise judgment in selecting appropriate analyses in light of cost and need. Boehm and Sullivan emphasize that software engineering has a business side, and economic as well as technical considerations should affect decisions [1].

Aspiration 4: Integrate an engineering point of view in undergraduate computer science and other information technology curricula

Practical, useful software doesn't happen by accident. It requires design skills not unrelated to traditional engineering design. Even a cursory look at what engineers know and do reveals problems in the current software curriculum. Shortcomings include:

Programming from scratch: Most courses teach students to code from scratch, rather than by modifying existing programs or by working from model solutions. Moreover, students rarely read good programs. It's as if we asked students to write good prose without first reading good prose.

Programming before reasoning: Although the situation is improving, coding and debugging still seems to win out over specification, analysis, and careful construction or derivation.

Implementing the first design: Problems often admit of more than one solution. The best solution in a given setting often depends heavily on facts about the user or the intended use of the system.

Designing for the implementer: Implementers often chose solutions that match their own tastes, not the needs of the customer.

Failing to understand problem scale: Class assignments usually emphasize functionality but neglect performance requirements, especially scale requirements such as size and throughput.

Writing throwaway exercises: When assignments are discarded as soon as they are graded, students have no incentive for creating comprehensible, well-documented, maintainable software.

Ignoring reliability, safety, economic, and other system requirements: Class assignments usually focus on getting correct results for correct inputs. They occasionally require rudimentary checking of inputs, and they occasionally require performance measurement. Students rarely do systematic analyses of reliability and safety. Similarly, class assignments address asymptotic performance of algorithms and sometimes speedy code, but many students never confront a requirement for practical real-time response. It's also rare for a student to encounter nontechnical issues that drive decisions.

We can address these problems without major disruption to our course structure by changing the emphasis within individual courses. The result would improve the quality of the courses for all students, not just for prospective software engineers:

Study good examples of software systems: Doing this properly requires case studies organized for presentation. Meanwhile, do careful guided reading of good code and make assignments that start from running code provided with the assignment.

Present theory and models in the context of practice: Emphasize durable ideas that will transcend a major shift of technology. Students often learn them best when they appear in concrete examples; good examples will themselves be worth remembering for reuse.

Require consideration of at least two serious designs: Make students choose between design alternatives. Require these choices to address customer needs.

Require consultation with end users: Use projects with actual clients. Unless end users have a voice in reviewing a design, students won't understand that their needs and preferences are different from the students' own.

Teach back-of-the-envelope estimation: Students often believe that they can't do any analysis until all the facts are in hand. Teach them to do quick estimates of usage levels, throughputs, sizes, bandwidths. Show them how this can provide early guidance about scale and performance.

Modify and combine programs as well as creating them: Teach students to work with program structures devised by others, to reuse components, to adhere to standards, and to value good documentation.

Test student implementations with bad data: Run test cases chosen by the instructor, not just demonstration data from the student. Include not only correct inputs, but also erroneous and even malicious inputs. Do this not only for isolated assignments, but as a matter of course.

Make assignments with embedded system requirements: Bad data isn't the only source of real-world demands. Make assignments that expose students to end-to-end time requirements, nondeterminism, race conditions,

and nontechnical constraints.

Keeping Education Current in the Face of Rapid Change

Changes in software technology and models for software development require commensurate change in the education of software developers. First, the educational institutions themselves must be able to adapt quickly, both in the content of their offerings and in their ability to exploit new technology in support of education. Second, the educational institutions must prepare their graduates to assume responsibility for upgrading their own skills throughout their careers.

Aspiration 5: Make curricula flexible and responsive to change

The enduring principles and models at the center of the curriculum will change more slowly than the examples of current practice. Nevertheless, compared to other fields, even the core of the software development curriculum must change rapidly.

For example, most curricula have not kept up with practice in recognizing the role of good abstractions for software architectures in software design [3]. As another example, within the past few years the conversion of the Internet from an email/telnet/ftp service for professionals to an information-distribution system embedded in popular culture has introduced new techniques and models for design and development:

- Open-source software development
- Large-scale, highly distributed information systems, including local caching, automatic updating, push and pull service, event-style control and other features
- Security for transactions between parties who have not pre-arranged passwords or keys
- Software that is platform-independent and trusted not to interfere with the computer on which it executes
- Computation carried out through coalitions of independently-managed resources
- Large-scale information collection and data mining of personal information, with attendant privacy concerns

The curriculum of even five years ago does not cover the concepts required to understand these phenomena, let alone to control them.

Educational institutions need the flexibility and the resources to react to these changes. They should not be constrained by internal fragmentation in the form of multiple competing programs or departments. They should not be constrained externally by standards that constrain the subject matter of the curriculum – as curricula and accreditation standards developed in professional society committees often do. If the professional societies are to be involved, it should be to establish levels of quality and a forum for sharing curriculum examples, not to govern specifics of content.

Aspiration 6: Exploit our own technology in support of education

Computer science and information technology curricula have always been aggressive about making assignments involving actual programming; in this respect we are ahead of many other fields. We can do better, though, at exploiting technology to support the learning process itself.

In local classrooms, we could make better use of simulations and game-playing exercises. We could take better advantage of tutorials embedded in systems that provide information as it's needed; since these facilities would benefit all users, their development cost could be amortized across a large user community.

The internet is already used to support courses. Often it's used simply as an easy way to distribute course materials to resident students, but we are beginning to see courses offered to remote students. Most of the distance courses are skills courses in the use of specific applications or programming languages, but university courses are increasingly coming on-line. Most of the functions of the classroom can be supported through some combination of the web, advance distribution of readings or CD-ROMs, and chat rooms or teleconferencing. The major exception is spontaneous interaction between instructor and students and among students. When this technology shortfall is overcome (perhaps through advances in technology for computer-supported cooperative work), we should be prepared to exploit it.

Unfortunately, the initial investment in preparing a electronic support for a course can be very large, as can the cost of regular revisions to reflect technology change. The cost and faculty load models appropriate to conventional subject areas do not take these factors into account.

Aspiration 7: Provide effective means for software engineers to keep their skills current

The objective of education is learning. Even in the classroom, the objective of teaching is to create a fertile setting for the student to learn. After graduation, though, the student becomes responsible for his or her own further education. Even with the best undergraduate education, software developers – especially software engineers – will need to periodically update their skills and their mastery of new technology. So one of the responsibilities of the formal education is to prepare the student with skills for independent lifelong learning.

Individual learning skills need to be complemented with materials for independent study. Occasional efforts by professional societies to provide self-assessment and independent study materials haven't reached critical mass. Short intensive courses from commercial providers tend to be very concrete (and expensive). Remote offerings of

university courses require a substantial commitment, and the size of a full semester course, or even a half-course may make it poorly matched to the needs of individual professionals.

We can aspire to providing opportunities for ongoing, on-demand, on-location education and training. Eventually, we should provide support that tracks each student's prior knowledge and current objectives, then provides a sequence of content that brings together the content for the current objective with any prerequisites required for that student.

Mid-career students need not be locked into the academic calendar. This provides an opportunity for individually-scheduled competency-based education, where the student studies however long it takes to master the material. In this setting the only grade is "A", but the grade isn't awarded until the student demonstrates competence.

Establishing Credentials that Accurately Represent Ability

As noted above, there are at least three ways to gain confidence in software: direct validation of the product, confidence in the development organization, and confidence in the developer. Our concern here is with ways for individual software practitioners, especially software engineers, to assure clients of their competence.

Credentialling of practitioners can be done (indeed is done) by both public and private bodies. The consequences – the rights, restrictions, privileges, and responsibilities – of these credentials differ for public and private credentials.

Public-interest credentialling of practitioners is generally done in the name of public interest. It is intended to ensure adherence to a minimum standard of practice, both technical and ethical. These credentials can address both professional (engineer, lawyer, doctor) proficiency and nonprofessional (truck driver, electrician, hairdresser) skills.

Private credentialling can be done for many reasons. The most common in software at present are academic degrees (intended to assure depth of understanding and the ability to grow with the field as well as current competence) and vendor-specific skills certification (intended to assure proficiency with a specific set of tools).

Public credentialling for individuals engaged in the practice of an engineering discipline requires

- an achievable level of practice that ensures quality consistent with public safety (i.e., reasonable intuitive expectations, but not perfection),
- an assessment instrument that can be confidently expected to predict that an individual will practice at that level in the future,
- in a field evolving as rapidly as software engineering, a means of ensuring that the practitioner will maintain his/her skills as the level of practice improves

For engineering licensing, in particular, this standard has not yet been achieved. For other, narrower or lower-level skills, it has been: consider, for example, the vendor certifications that are associated with particular versions of systems.

Credentialling, especially public credentialling, resembles software specification: it makes commitments about the capabilities of the practitioner. We have an obligation to ensure that the credentials make assurances that reflect demonstrated skills and address the concerns about competency that are of concern to clients who are laymen with respect to computing.

Aspiration 8: Establish distinct and appropriate credentials for distinct software development roles when possible

As a follow-up to Aspiration 1 (discriminating among different software development roles), we should establish credentials that match the roles, or at least those roles for which the field is sufficiently mature. This will entail both identification of content and clear separation of roles.

The role separation must be done not only to separate professional from non-professional roles, but also to identify professional roles more specialized than the role implied by professional engineering registration.

There are certainly some technical areas in which a useful level of expertise can be achieved and demonstrated. Some are skills, such as administering a particular brand of system software. Others are higher-level, such as database administration or (perhaps) certain aspects of reliability. We should continue existing activities in establishing appropriate credentials for these skills. This can set reasonable expectations, give us experience with certification, and provide discrimination between software developers with audited competence and those without. Certifications can be added as warranted; by making it clear what's being certified, they can avoid misleading the public or the clients.

Aspiration 9: Establish credentials that accurately reflect achievable practice

The question "should there be a profession of software engineering" is often asked in the form "isn't it time we started licensing software engineers through the usual mechanisms of professional engineering registration?" There's a problem with using engineering registration as a surrogate for the activity of raising professional standards, though: The purpose of professional engineering registration is to protect the public by providing some external assurance that a particular engineer will produce safe systems; by signing off on a project, the engineer assumes personal responsibility. The level of performance required for this assurance isn't "the best we can do now"; it's "good enough". Unfortunately, we don't yet have an established, widely achievable level of practice in software engineering that meets this standard. Proposals that we

certify engineers on the basis of current best practice, even the proposals are accompanied by promises to raise the standard as we get better, simply don't address the overriding criterion.

Two things are required before adding software engineers to the pantheon of engineers. First, we need a widely achievable level of practice that provides reasonable protection for the public. Second, we need a testing instrument that can make a reasonable prediction about whether a given person will practice at that level. There's no point in pursuing the second until we have the first.

4 SUMMARY AND CONCLUSIONS

Education for software developers currently emphasizes content inspired by closed-shop mainframe development. It is offered largely in traditional classroom formats. Training also follows traditional lines, teaching specific skills in short-course, hands-on, and independent study formats.

We can aspire to improvements over the next decade, including clarification of the roles involved in software development and appropriate credentialling for those roles; improved treatment of engineering issues; faster response of educational content to changes in technology and fundamental understanding; and better use of information technology in our own education and training.

Realizing these aspirations will require imagination and flexibility. Most important will be providing encouragement, resources, and opportunities to interested faculty – and challenging them to set their own standards high enough to raise the standards of the field.

ACKNOWLEDGEMENTS

Thanks to colleagues who have taught me about education, showed me new alternatives, and otherwise stimulated my appreciation of the problems and opportunities: Jim Tomayko and other members of the CMU Institute for Software Research, International and its Master of Software Engineering program; colleagues in the CMU Center for Innovation in Learning; participants in the ACM/IEEE discussions on professional licensing; Herb Simon, Frank Bruns, and Roger Dannenberg. Portions of this paper are derived from [10] and [11].

REFERENCES

1. Barry W. Boehm and Kevin J. Sullivan. Software Economics. In this volume.
2. Carnegie Mellon University Institute for Software Research, International. *PhD Program in Software Engineering*. http://www.isri.cs.cmu.edu/, then follow link to PhD program.
3. David Garlan. Software Architecture: A Roadmap. In this volume.
4. Daniel Jackson and Martin Rinard. The Future of Software Analysis. In this volume
5. Peter Knoke via Don Bagert. Graduate Software Engineering Program Survey Results & Evaluation. *Forum for Advancing Software engineering Education (FASE)*, vol 8 no 9, September 15, 1998, http://www.cs.ttu.edu/fase/v8n09.txt .
6. TSE Maibaum. Mathematical Foundations of Software Engineering: A Roadmap. In this volume.
7. Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical Studies of Software Engineering: A Roadmap. In this volume.
8. Eric S. Raymond. *The Cathedral and the Bazaar*. http://www.tuxedo.org/~esr/writings/cathedral-paper.html, 1998.
9. Mary Shaw. Prospects for an engineering discipline of software. *IEEE Software*, pages 15-24, November 1990.
10. Mary Shaw. We Can Teach Software Better. *Computing Research News*, 4,4 September 1992 (pp. 2, 3, 4, 12).
11. Mary Shaw. A Profession of Software Engineering: Is There a Need? YES; Are We Ready? NO. *Proc. ACM SIGSOFT Sixth Int'l Symposium on the Foundations of Software Engineering, FSE-6*, Nov 1998, pp. 207-208.
12. Mary Shaw. Architectural Requirements for Computing with Coalitions of Resources. Position paper for *First Working IFIP Conference on Software Architecture*, http://www.cs.cmu.edu/~Vit/paper_abstracts/Shaw-Coalitions_paper.html, 1999.
13. Software Engineering Coordinating Committee of ACM and IEEE. *Guide to the Software Engineering Body of Knowledge*. http://www.swebok.org/ .
14. James E. Tomayko. Forging a discipline: An outline history of software engineering education. *Annals of Software Engineering* 6 (1998) 3-18.