Position Paper for the Software Engineering Education Workshop at the 13th International Conference on Software Engineering, Austin, Texas, May 13, 1991.

Putting Engineering into Software Engineering Education

Mary Shaw
Carnegie Mellon University
Pittsburgh, PA

April 1991

Abstract

The current practice of software engineering bears only slight resemblance to the usual standards of engineering practice. This is in part a consequence of the immaturity of the field, but it also results from the failure of software engineering education to instill an engineering mindset in students. This position paper discusses the character of engineering practice with special emphasis on the routine use of reference materials; based on this characterization, it suggests ways software engineering education could better support good engineering practice.

I. Engineering Practice

Traditionally, engineering is defined as "creating cost-effective solutions to practical problems by applying scientific knowledge to building things in the service of mankind" [Shaw 90b]. Against that standard, the phrase "software engineering" is a statement of aspiration, not a description of accomplishment. Nevertheless, we face a desperate need to get better control over the software development task--and to do so in the short term.

Engineering design problems come in a number of forms. One of the most significant distinctions separates *routine* from *innovative* design. Routine design involves solving problems that are not significantly different from previously-solved problems; it relies on reusing large portions of those prior solutions. Innovative design, on the other hand, involves finding new ways to solve unfamiliar problems. The need for innovative design is generally much rarer than the need for routine design, so routine design is the bread and butter of engineering practice. Most engineering disciplines capture, organize, and share design knowledge in order to make routine design simpler. Handbooks, catalogs, and manuals are often the carriers of this organized information [Marks 87, Perry 84].

Alas, most software development is anything but routine. Software developers are not much inclined to draw strongly on prior experience, nor do they have reference materials that allow them to do so. As a result, software is more often created *as if* it were innovative than would be necessary if we concentrated on capturing and organizing what is already known. One path to increased productivity is identifying applications that should be made routine and developing appropriate support. The current emphasis on reuse [Biggerstaff 89] emphasizes capturing and organizing existing knowledge. Indeed, subroutine libraries—especially libraries of operating system calls and general-purpose mathematical routines—have been a staple of programming for decades. But this knowledge cannot be useful if programmers don't know about it or aren't encouraged to use it, and library components require more care in design, implementation, and documentation than similar components that are simply embedded in systems

Several kinds of incentives encourage innovative development of software that probably could and should be routine. These range from the mindset taught in introductory programming courses to government contracting policies. The problem has been documented in various forums: most recently by a National Research Council study on software [CSTB 89], earlier by a Defense

This work was sponsored by the US Department of Defense. Author contact information: e-mail (Internet) shaw@sei.cmu.edu; telephone: (412) 268-7731; fax: (412) 268-5758.

Science Board study [SWTF 87]; a recent World Bank study of the world software industry identifies some of the same problems [World Bank 89].

Parnas argues forcefully that the education of a computing professional should resemble the education of any other sort of engineer, including emphasis on the cooperation between theory and practice, engineering discipline, design for reliability and safety, finding good (not merely workable) solutions, design presentation, and practical use of logic and communication concepts [Parnas 90]. The specific curriculum Parnas proposes is, to my mind, too strongly dominated by classical engineering and by requirements of engineering accreditation. However, his analysis of the problem is substantially on target.

II. Proficiency and Reference Materials

Proficiency in any field requires not only higher-order reasoning skills but also a large store of facts, together with a certain amount of context about their implications and appropriate use. This is true across a wide range of problem domains; studies demonstrate it for medical diagnosis, physics, chess, financial analysis, architecture, scientific research, policy decision making, and others [Reddy 88 pp 13-14; Simon 89 p.1]. An expert in a field must know around 50,000 chunks of information, where a chunk is any cluster of knowledge sufficiently familiar that it can be remembered rather than derived. Furthermore, in domains where there are full-time professionals, it takes no less than ten years for a world-class expert to achieve that level of proficiency [Simon 89 pp.2-4].

Thus, fluency in a domain requires content and context as well as skills. In the case of natural language fluency, for example, Hirsch argues that abstract skills have driven out content; students are expected (unrealistically) to learn general skills from a few typical examples rather than by "piling up of information"; intellectual and social skills are supposed to develop naturally without regard to the specific content [Hirsch 88]. However, says Hirsch, specific information is important at all stages. Not only are the specific facts important in their own right, but they serve as carriers of shared culture and shared values.

Hirsch provides a list of some 5,000 words and concepts that represent the information actually possessed by literate Americans. The list goes beyond simple vocabulary to enumerate objects, concepts, titles, and phrases that implicitly invoke cultural context beyond their dictionary definitions. Whether or not you agree in detail with its composition, the list and accompanying argument demonstrate the need for connotations as well as denotations of the vocabulary.

In addition to general knowledge about computer science and the application domain, a software engineer's expertise includes, of course, knowledge of software design elements, programming idioms, representations, protocols, and analysis techniques. It also includes skill with tools: the language, environment, and support software with which this system is implemented. Required programming skills include not only a language but also the system calls supported by the environment, the general-purpose libraries, the application-specific libraries, and how to combine innovations of these definitions effectively. More subtly, the software developer must understand the intended design strategy and the unwritten rules about "good citizenship" in the environment; for example, the rules about how globally-defined entities of a system should be used are as important as their signatures and representations.

A library is of little use if its user doesn't know what definitions are available. The usual response to this problem is improved indexing, classification, and search mechanisms. These, like English dictionaries, are useful for recording little-used portions of the vocabulary and precise definitions of the vocabulary. However, just as natural language fluency requires instant recognition of a sizable vocabulary, programming fluency requires an extensive vocabulary of definitions that the programmer can use familiarly, without regular recourse to documentation. Furthermore, expert skill relies on concepts and practices that are unlikely to be found as discrete entries in dictionaries.

Given that a large body of knowledge is important to a working professional, we turn now to the question of how software engineers should acquire facts, either as students or as working professionals. Generally speaking, there are three ways to obtain a piece of information you need: you can remember it, you can look it up, or you can derive it. These have different distributions of costs:

	Infrastructure Cost	*Initial Learning Cost*	*Cost of Use in Practice*
Memory	low	high	low
Reference	high	low	medium
Derivation	medium-high	medium	high

Memorization requires a relatively large initial investment in learning the material, which is then available for instant use. *Reference materials* require a large investment by the profession for developing both the organization and the content; each individual student must then learn how to use the reference materials and take the time to do so as a working professional. *Deriving information* may involve ad hoc creation from scratch, it may involve instantiation of a formal model, or it may involve inferring meaning from other available information; to the extent that formal models are available their formulation requires a substantial initial investment. Students first learn the models, then apply them in practice; since each new application requires the model to be applied anew, the cost in use may be quite high [SGR 89].

Each professional's allocation of effort among these alternatives is driven by what he or she has already learned, by habits developed during that education, and by the reference materials available. At present, general-purpose reference material for software is scarce, though documentation for specific computer systems, programming languages, and applications may be quite extensive. Even when extensive documentation is available, however, it may be under-used because it is poorly indexed or because software developers have learned to prefer fresh derivation to use of existing solutions. The same is true of subroutine libraries, though incorporation of a library in the programming language (as in Common Lisp) provided better documentation and visibility to the routines that are added to the language [Shaw 90a].

The engineering of software would be better supported if we knew better what specific detailed content a software engineer should know. We could then organize the teaching of this material so that useful subsets are learned first, followed by progressively more sophisticated subsets. We could also develop standard reference materials as carriers of the content.

III. Software Engineering Education

Personnel shortages in computing, and especially in software, are well documented. A study by the Office of Technology Assessment estimates a current shortfall of 50,000 to 100,000 software professionals in the US alone. A World Bank study identifies personnel shortfall as the number one software risk item [World Bank 89].

The software content of the computer science curriculum needs to be re-examined, both for topic selection and for presentation. Both our current understanding of engineering design and examination of the knowledge that programmers actually use suggest flaws in the current software offerings. Undergraduates are being prepared to think about algorithms and to write small programs, but they do not learn enough about existing software, about the incorporation of small program elements into large systems, or about the problems of long-lived software. They do not learn about engineering issues such as matching the software to the user's needs, resolving conflicting design constraints, predicting system performance from the design, choosing among design alternatives, or building for reliability. Students are currently taught to develop their own programs from scratch with little reliance on the historical body of programming experience. This may be a good start for innovative programming (though that's debatable), but it's certainly a poor start for routine programming.

As noted above, engineers distinguish innovative from creative design. Similarly, they distinguish designs created from scratch (so-called green-field designs) from enhancements of existing systems; the latter are regarded as more constrained and hence more difficult (contrast this with the software developer's view of maintenance). Designs intended to be implemented once are also distinguished from those with replicated instantiations, and designs done by individuals are distinguished from large team designs. Computer science teaching and research emphasizes innovative, green-field, once-used designs created by individuals. Software development and software engineering practice, however, involves work that frequently should be routine, that is most often enhancement of previous work, that is normally instantiated in many variants, and that involves large teams.

Software Development Skills

One of the major skills required of a software engineer is the ability to create software. Accordingly, the way we teach programming strongly affects the quality of software engineering education. Examination of the knowledge actually used by programmers (e.g., in [SGR 89] suggests shortcomings in the current software curriculum [Shaw 90a]:

Programming from scratch: Most courses teach program construction by fresh creation, rather than by modifying existing programs or by working from models of good solutions. Moreover, students rarely read good programs; it's as if we asked students to write good English without reading good prose.

Equating program text with software: A complete software product includes not only the code, but also the analysis that led to the design, the user documentation, the test suites, and records of design decisions that will be important to the maintainer. Students too often focus on the code, do ad hoc testing, neglect the user documentation, and ignore everything else.

Learning abstract skills at the expense of specific content: Our curricula are very strong in techniques for formulating solutions from first principles. They present too few well-known examples of good solutions for study and emulation.

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

Some ways to remedy these flaws and improve the programming curriculum include:

Study good examples of software systems: To do this properly requires the development of case studies intended for presentation. However, careful guided reading of good code would be an improvement, as would assignments that start from code distributed by the instructor.

Learn more facts: Software developers won't use resources they don't know about. We should teach more specific substance such as the available subroutine libraries and interface standards; we should reinforce this with assignments that require their use.

Incorporate reference material as it becomes available: There is currently a dearth of good reference material to help software developers avoid re-invention. As such material becomes available, it should be incorporated. In the meanwhile, students should be taught to use reference manuals, library documentation, and the like effectively. Plan for continuing improvement in this area.

Present theory and models in the context of practice: The curriculum should emphasize durable ideas that will transcend a major shift of technology. These ideas are often learned best when coupled with concrete examples of their application; if the examples are selected well they may themselves be worth remembering for reuse.

Engineering Skills

Software engineers also need to develop software that is useful in practice. Some of the engineering shortcomings of the curriculum are similar to the programming shortcomings: failure to study good systems, failure to exercise reasoning skills, failure to understand maintenance and support issues. Some others are:

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

Designing for the implementor: Implementations are often chosen because they match the taste of the software developer, 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, there is no clear incentive for creating comprehensible, well-documented, maintainable software.

Ignoring reliability and safety: Class assignments usually focus on getting correct results for correct inputs. some rudimentary checking of inputs may be required, and performance is occasionally measured. However, systematic analysis of reliability and safety is rarely addressed.

Some ways to remedy these flaws and improve the software engineering curriculum include:

Require consideration of at least two serious designs: In addition to forcing students to think about alternatives, the analysis to support a choice should force them to address customer needs.

Require consultation with end users: Unless end users have a voice in reviewing a design, students won't understand that their needs and preferences are different from the students'. Projects with actual clients are a good way to address this.

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

Modify and combine programs as well as creating them: Students should learn 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.

IV. References

Biggerstaff 89

Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability*. Two volumes, ACM Press 1989.

CSTB 89

Computer Science and Technology Board, National Research Council. *Scaling Up: A Research Agenda for Software Engineering*. National Academy Press 1989.

Hirsch 88

E. D. Hirsch, Jr. *Cultural Literacy: What Every American Needs to Know*. Vintage Books 1988.

Marks 87

Lionel S. Marks. *Marks' Standard handbook for Mechanical Engineers.* McGraw-Hill 1987.

Parnas 90

David Lorge Parnas. Education for Computing Professionals. *IEEE Computer* 23,1 (Jan 1990) pp. 17-22.

Perry 84

Robert H. Perry. *Perry's Chemical Engineers' Handbook.* McGraw-Hill 1984.

Reddy 88

Raj Reddy. Foundations and Grand Challenges of Artificial Intelligence. *AI Magazine*, vol 9, no 4 (Winter 1988), pp. 9-21. (1988 presidential address, American Association for Artificial Intelligence)

SGR 89

Mary Shaw, Dario Giuse, Raj Reddy. What A Software Engineer Needs to Know I: Vocabulary. CMU-CS and CMU-SEI Technical reports, August 1989.

Shaw 90a

Mary Shaw. Informatics for a New Century: Computing Education for the 1990s and Beyond. CMU-CS and CMU-SEI Technical reports, July 1990; to appear in *Education and Computing*.

Shaw 90b

Mary Shaw. Prospects for an Engineering Discipline of Software. *IEEE Software*, November 1990.

Simon 89

Herbert A. Simon. Human Experts and Knowledge-Based Systems. Talk given at IFIP WG 10.1 Workshop on Concepts and Characteristics of Knowledge-Based Systems, Mt Fuji Japan, November 9-12, 1987

SWTF 87

Defense Science Board Task Force on Military Software. *Report of the Defense Science Board Task Force on Military Software*. Office of the Under Secretary of Defense for Acquisition, Washington DC, September 1987.

World Bank 89

Robert Schware. *The World Software Industry and Software Engineering: Opportunities and Constraints for Newly Industrialized Economies.* World Bank Technical Paper 104, August 1989.