The Profession of IT
Who Are We—Now?

Considerable progress has been made toward the formation of a computing profession since we started tracking it in this column a decade ago.

In 2001, we launched this column because ACM was interested in tracking the evolution of the computing profession and wanted to explore ideas of potential value to the professional. Let us take stock of where we have come in the past decade and what it means for you.

In the late 1990s, ACM leaders realized the computing field had matured to the point where members were becoming interested in ACM support for their professional activities. ACM wanted to assure itself that there is a deep community need and adequate depth in the existing organization. We asked: Is there a need for a computing profession? What will it be called? What are its components?2

We argued from first principles that professions form when considerable expertise is needed to take care of people’s enduring concerns in a domain. This is plainly true of long-standing professions such as health and law. Computing has become such an integral part of business and everyday life that the reliable operation of ever-advancing computing technologies is an enduring concern. Considerable expertise is needed to properly address this concern. Thus, a profession must inevitably form.

ACM already had considerable depth in professional activities in the various specialties covered by the SIGs. Other professional societies hosted professional groups around other specialties, and corporate universities were offering certifications in some occupational specialties. We counted over 40 organized professional groups and inventoried them in three categories: IT core, IT intensive, and IT occupations (infrastructure). Table 1 is the inventory with a few additions that account for changes since that time. Although the table is not exhaustive, it clearly shows the breadth and depth of the field. Several new specialties that appeared since then are shown in bold type.

At the time, IT (for information technology) seemed to be a broader term than computing, and thus the label “IT profession” seemed to be a good umbrella for all the professional activities listed in Table 1.

Elements of a Profession
In a remarkable 1996 study, Gary Ford and Norman Gibbs laid out a map of the essential elements of a mature profession3, and validated their map against a number of existing professions including health, law, and architecture. Their purpose was to lay out a path to maturity for the software engineering profession. Their map is summarized in Table 2.

Let us define certain terms used in the map and in our subsequent discussions. A profession is a community of practice that forms to take care of people’s enduring concerns in some area of life or work. A professional is a member of that community who renders service to clients of the profes-

---

Table 1. Professional subdivisions of the computing field.

<table>
<thead>
<tr>
<th>Computing-Core Disciplines</th>
<th>Computing-Intensive Disciplines</th>
<th>Computing-Infrastructure Disciplines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Artificial intelligence</td>
<td>Aerospace engineering</td>
<td>Computer technician</td>
</tr>
<tr>
<td>Cloud computing</td>
<td>Bioinformatics</td>
<td>Database administrator</td>
</tr>
<tr>
<td>Computer science</td>
<td>Cognitive science</td>
<td>Help desk technician</td>
</tr>
<tr>
<td>Computer engineering</td>
<td>Digital library science</td>
<td>Network technician</td>
</tr>
<tr>
<td>Computational science</td>
<td>E-commerce</td>
<td>Professional IT trainer</td>
</tr>
<tr>
<td>Database engineering</td>
<td>Genetic engineering</td>
<td>Security specialist</td>
</tr>
<tr>
<td>Computer graphics</td>
<td>Information science</td>
<td>System administrator</td>
</tr>
<tr>
<td>Cyber security</td>
<td>Information systems</td>
<td>Web identity designer</td>
</tr>
<tr>
<td>Human-computer interaction</td>
<td>Public policy and privacy</td>
<td>Web programmer</td>
</tr>
<tr>
<td>Network engineering</td>
<td>Instructional design</td>
<td>Web services designer</td>
</tr>
<tr>
<td>Operating systems</td>
<td>Knowledge engineering</td>
<td></td>
</tr>
<tr>
<td>Performance engineering</td>
<td>Management info systems</td>
<td></td>
</tr>
<tr>
<td>Robotics</td>
<td>Network science</td>
<td></td>
</tr>
<tr>
<td>Scientific computing</td>
<td>Multimedia design</td>
<td></td>
</tr>
<tr>
<td>Software architecture</td>
<td>Telecommunications</td>
<td></td>
</tr>
<tr>
<td>Software engineering</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Knowledge refers to the organized body of knowledge that professionals rely on. Practice refers to the skills displayed by professionals at various levels of competence as they render service. Practicing professionals are often called practitioners.

It is important to note that the elements of the map are not the point of the profession; service to clients is. The map elements are the means.

It is useful to define three other terms. A craft refers to a set of practices shared by a community of practice, but it has no special social status. Wood crafting and programming are examples. A trade refers to an organized group of practitioners, such as a guild or labor union, with restrictions imposed by society in return for freedom to practice for the benefit of society. At this time, there are no computing-specific trades. A discipline refers to a field of study and research that provides knowledge. Chemistry and computer science are examples. Professions include crafts and trades and rely on one or more disciplines. Disciplines are concerned intellectual knowledge, professions with performance and practice.

What Has Changed
The term information technology has not achieved the all-encompassing character we once envisioned. Today, IT refers mainly to technology and business applications of computing. Computing is now the preferred umbrella term.

As a sign of the field’s continued vitality, several important new specialties have appeared. These include cyber security, cloud computing, network science, and Web programming. We expect expansion of the computing profession to continue.

Certification is becoming more common, mainly at the sub-profession or specialty level. A growing number of computing jobs require specialized certifications such as Microsoft or Cisco network engineer. Many organizations, including the U.S. Department of Defense, now require that cyber security personnel have the CISSP certification (certified information systems security professional). The American Society for Quality now certifies Software Quality Engineers via their CSQE program, and the IEEE-CS certifies Software Development Professionals through their CSDP.

Licensing of professionals has proceeded more slowly. Licensing has begun to appear in engineering specialties of computing, where concerns for health and safety are strongest. Computer engineers have been licensed in the U.S. for several years. Software engineers have been licensed in Canada, Australia, and Great Britain for some time. In the U.S., the NCEES (National Council for the Examination of Engineers and Surveyors), with considerable technical help from the IEEE-CS, is developing a licensing examination for software engineers in response to a request from 10 states. ACM has generally not supported licensing and has taken no official position on certification.

Starting in 2002 there was a sharp drop worldwide in the numbers of students enrolling as majors in computing disciplines. This was a great puzzle to computing leaders because demand for computing professionals in the job market was (and continues to be) very high. That concern led government agencies, such as the National Science Foundation in the U.S., to support programs to generate more student interest in computing in colleges and universities. Much of the concern over the educational pipeline has shifted to the K–12 arena, because most high school students have no access to any computing course, serious problems with the unpopular Advanced Placement curriculum have led to its cancellation, and budget shortfalls have blocked computing from being added as a new program.

ACM has built an impressive array of programs that support computing professionals. It spent a lot of money to scan the entire ACM literature, back to its founding in 1947, into the Digital Library. It has entered extensive licensing agreements with businesses and consortia so that almost all employed professionals have access to the Digital Library through their employers. It has enhanced the Digital Library with individual favorite folders and technical reading packs. It has sponsored more conferences for practicing professionals. It created the Profession Board to watch over the interests of professionals in ACM. It created more publications for practicing professions, mostly notably Queue and its new linkage with Communications. It created the online publication Ubiquity to explore the future of computing and the people who are inventing it. It has cooperated with IEEE Computer Society and others in joint programs for professionals, notably curriculum and accreditation guidelines.

Open Issues
We noted previously that most of the certification and licensing activity has been focused on sub-professions of computing. So too, most of the growth in bodies of knowledge, codes of ethics, education, and training programs has been in specialties. This is similar to what has happened in other professions. The health profession is a good example. That profession has established conventions, standards, core knowledge, and common terminology for the whole profession. Most health certifications and training programs focus on specialties. Nurses, doctors, and technicians get accredited degrees in their fields, and then obtain certifications and training for the specific specialties in which they practice.

What can we learn from analogies to medicine, architecture, and other

<table>
<thead>
<tr>
<th>Practitioner Level</th>
<th>Infrastructure Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Professionals</td>
<td>Initial professional education</td>
</tr>
<tr>
<td>Knowledge</td>
<td>Accreditation</td>
</tr>
<tr>
<td>Professional practice</td>
<td>Skills development</td>
</tr>
<tr>
<td></td>
<td>Certification</td>
</tr>
<tr>
<td></td>
<td>Licensing</td>
</tr>
<tr>
<td></td>
<td>Professional Development</td>
</tr>
<tr>
<td></td>
<td>Code of ethics</td>
</tr>
<tr>
<td></td>
<td>Professional societies</td>
</tr>
</tbody>
</table>
professions? It is striking that each profession has its own unique model. The medical profession, the engineering profession (which overlaps with the computing profession), and the architecture profession have very different solutions to the elements in Table 2. Computing has not developed good solutions to all the elements in Table 2 because it has not settled on a model or paradigm for itself. If history is a guide, this will not be an easy path. The interdisciplinary character of computing and its overlap with and use by so many other professions will require a degree of nimbleness that may be unique among professions. As we work to build our profession, we can set our sights on some of the characteristics of the most admired professions, which will surely include: well defined and widely accepted terminology and core knowledge; an infrastructure that keeps the various parts of the profession aligned and reasonably consistent with each other; and, ultimately, a structure that the public—whom we serve—will grow to respect, admire, and count on.

Neville Holmes in his Computer magazine column, the counterpart of this column, has raised a number of issues worth noting. He is concerned that we do not seem to have standard definitions of key terms in computing such as data, information, computation, programming. Further, he postulates, with some grounding, that many practitioners do not care about being identified with a profession. He is concerned about practitioners who believe their programs are the end products of their work; as professionals, they should be concerned with their clients—how the program affects clients goes on long after the program is delivered. Are computing practitioners non-professional if they have such an attitude? Should we be instilling a sense of professionalism in our students, regardless of their specialties?

What This Means for You
All the above looks like it would be interesting to organizational leaders looking to build useful professional society programs. But what about the “ordinary practicing person?” Why should you pay attention to this?

Writing in the Chronicle of Higher Education, Kevin Carey, the director of education for a Washington, D.C., think tank, gave a beautiful answer to this question. He himself started out to join the computing profession and became a good programmer. But his true interest turned out to be bringing computing to the education of young people. The practice of programming gave him a discipline of precision and conciseness that has helped him in his current profession. He says, “I left computer science when I was 17 years old. Thankfully, it never left me.”

A particularly insightful benefit of Carey’s computing knowledge was how it enabled him, as an employee of the Indiana state-budget office, to help rewrite Indiana’s school financing law to make it substantially simpler and easier to administer. Seeing it as similar to a large program that had grown too hoary with age, he “sat down, mostly as an intellectual exercise, to rewrite the formula from first principles.” Many other laws and government structures could benefit from such an approach.

Even if you do not have a professional certification, you are probably operating in an environment where you have customers with various concerns around computing. You have the expertise to help them and you want to help them. You want to conduct yourself ethically and provide benefits to society as well as your clients. In short, you want to be a professional. We hope the map we have outlined here can help you find the services to assist you doing these things.

References

Peter J. Denning (pjd@jnp.scs.edu) is Distinguished Professor of Computer Science and Director of the Cebrowski Institute for Information Innovation at the Naval Postgraduate School in Monterey, CA, and is a past president of ACM.

Dennis J. Frailey (frailey@lyle.smu.edu) recently retired from Raytheon Company as a principal fellow. He is an adjunct professor at Southern Methodist University, a member of the IEEE-Computer Society Board of Governors, and a past vice-president of ACM.

Copyright held by author.
IN THE 1980s, when I first began thinking and writing about ethics and computing, there was much speculation about how computing would and should develop as an occupation or a set of occupations. At that time, of course, one had to turn to the histories of other fields to learn about paths to professionalization. With more than 25 years behind us, the picture remains unclear. What is the state of the field of computing now and where should it go?

Professionalization is of interest not for its own sake, but for what it would do to promote a socially responsible deployment of computing expertise. To get quickly to the heart of the matter, we might use a distinction as a foil: although it oversimplifies a complex situation, the distinction between guns-for-hire and professionals frames the issues of professionalization in stark form. A gun-for-hire is someone who puts his or her expertise up for sale to the highest bidder; he or she will do anything anyone wants as long as it is legal. By contrast, and in what is admittedly an idealized paradigm, professionals have standards; they take responsibility, individually and collectively, for setting standards of practice acknowledging that law is limited and will not adequately protect the values that should guide the field. Typically, professions act collectively through an organization that promulgates and enforces a code of ethics and professional conduct, and that articulates the core values of the profession, for example life (in medicine), safety (in engineering), and accuracy (in auditing). Are computer experts guns-for-hire or professionals?

Sociological accounts of professions have suggested that we think of professions as systems or mechanisms for managing expertise. A group convinces society that restrictions should be placed on who engages in a particular occupation. It convinces society there is a body of knowledge that should be mastered before one practices, for example, before one treats the sick or represents another in a court of law or audits a financial statement. The group convinces society that competence can only be determined by those who have already mastered the relevant body of knowledge. Thus, experts, not outsiders, should be in charge of specifying requirements for the field and deciding who has met the requirements.

When a group successfully makes these claims, society grants the group the power of self-regulation. However, this power is granted in exchange for the group’s commitment to manage its activities to achieve social good, or at least not in ways that are harmful to society. When doctors professionalized, the intention was to distinguish themselves from “charlatans” and “quacks,” those who claimed they could heal patients but who had no scientific understanding of how the human body worked. Once the system of medicine was established, patients could expect that when they went to a “doctor,” they would be treated by someone with a certain level of competence. This serves the interests of those who are sick and, in turn, the broader society.

Professionalization often occurs against a backdrop of concerns about the pressures of the marketplace; that is, professionalization is targeted, in part at least, to take certain issues out of the marketplace. When an occupational group has specified standards and articulated its values, then members will (at least, they are expected to) refuse to do anything inconsistent with those standards and values—no matter how much a client or customer is willing to pay. The standards and values become part of the professional culture.

The distinction between guns-for-hire and professionals doesn’t map neatly onto computing. Rather than a sharp division, there seems to be a
A key feature of any profession—from the perspective of professional ethics—is how it manages the differential in knowledge between its members (experts) and those whom they serve.

In Cyberspace and views beyond, his 1999 book, Code and Other Laws of Cyberspace. The work of computer experts may structure an environment, facilitate and constrain behavior, and materialize social values in one form or another. To characterize this work as that of an agent is to deny the real power that computer experts have. Second, if computer experts operate as if they are agents, their clients and employers don’t get the full benefit of their expertise. Clients, employers, and the public need computer expertise for higher-order decisions, that is, they need help identifying goals and strategies, not just implementation. When you go to a doctor, you don’t tell the doctor what to do, leaving implementation to the doctor’s discretion; you want the doctor to determine what needs to be done and to discuss with you the options that are available; you want the doctor to explain the risks and benefits of alternative approaches.

Second, we might think of the proper role for computer experts as paternalistic. Computer experts, it might be argued, are in the best position to understand needs, comprehend potential risks and benefits, as well as foresee the consequences of implementing a system in various ways. Thus, non-experts need computer experts to act on their behalf. According to this model, a client, employer, or the public should transfer all decision-making authority to the computer expert. This provides what is missing in the first model for clients, employers, and the public to get the full benefit of the expert’s knowledge. The problem is that computer experts aren’t experts with regard to values, interests, and preferences. The model oversteps the expertise of anyone who is competent in computing for no matter how well trained or how much experience a computer expert has, he or she is not an expert on someone else’s needs and values.

The third model of the relationship between non-experts and experts combines elements of the first two models and is best suited to the complexities of decision making in computing. It is a model in which experts and those whom they serve share responsibility. Decisions are made through interaction and iteration. Referred to as the fiduciary model (“fiduciary” means trust), this model calls for a relationship of trust between experts and non-experts. The client/employer/public must trust the expert to use his or her knowledge to pursue their interests and values. The professional must trust that the client/employer/public
Computer experts have power—in virtue of their expertise, in virtue of their occupational roles, and simply because so many non-experts depend on their work.

Deborah G. Johnson is the Anne Shirley Carter Olsson Professor of Applied Ethics at the University of Virginia, Charlottesville, VA.
Inside Risks

Risks of Undisciplined Development

An illustration of the problems caused by a lack of discipline in software development and our failure to apply what is known in the field.

The branches of engineering (such as civil, electrical, and mechanical), are often referred to as disciplines for good reason. Associated with each specialty is a set of rules that specify:

- checks that must be made;
- properties that must be measured, calculated, or specified;
- documentation that must be provided;
- design review procedures;
- tests that must be carried out on the product; and
- product inspection and maintenance procedures.

Like all professional education, engineering education is designed to prepare students to meet the requirements of the authorities that regulate their chosen profession. Consequently, most graduates are taught they must carry out these procedures diligently and are warned they can be deemed guilty of negligence and lose the right to practice their profession if they do not.

Because they are preparing students for a career that can last many decades, good engineering programs teach fundamental principles that will be valid and useful at the end of the graduate’s career. Engineering procedures are based on science and mathematics; and graduates are expected to understand the reasons for the rules, not just blindly apply them.

These procedures are intended to assure that the engineer’s product:

- will be fit for the use for which it is intended;
- will conform to precise stable standards;
- is robust enough to survive all foreseeable circumstances (including incorrect input); and
- is conservatively designed with appropriate allowance for a margin of error.

In some areas, for example building and road construction, the procedures are enforced by law. In other areas, and when engineers work in industry rather than selling their services directly to the public, employers rely on the professionalism of their employees. Professional engineers are expected to know what must be done and to follow the rules even when their employer wants them to take inappropriate shortcuts.

Anyone who observes engineers at work knows that exercising due diligence requires a lot of “dog work.” The dull, but essential, work begins in the
design phase and continues through construction, testing, inspection, commissioning, and maintenance. Licensed engineers are given a unique seal and instructed to use it to signify the acceptability of design documents only after they are sure the required analysis has been completed by qualified persons.

Real-World Experience
Recent experiences reminded me that the activity we (euphemistically) call software engineering does not come close to deserving a place among the traditional engineering disciplines. Replacing an old computer with a newer model of the same brand revealed many careless design errors—errors that in all likelihood could have been avoided if the developers had followed a disciplined design process. None of the problems was safety critical, but the trouble caused was expensive and annoying for all parties.

My “adventure” began when the sales clerk scanned a bar code to initiate the process of creating a receipt and registering my extended warranty. There were three codes on the box; not surprisingly, the sales clerk scanned the wrong one. This is a common occurrence. The number scanned bore no resemblance to a computer serial number but was accepted by the software without any warning to the clerk. The nonsense number was duly printed as the serial number on my receipt. My extended warranty was registered to a nonexistent product. I was billed, and no problem was noted until I phoned the customer care line with a question. When I read the serial number from the receipt, I was told that I had purchased nothing and was not entitled to ask questions. After I found the correct number on the box, I was told that my computer was not yet in their system although a week had passed since the sale.

Correcting the problem required a trip back to the store and tricking the company computer by returning the nonexistent machine and buying it again. In the process, my name was entered incorrectly and I was unable to access the warranty information online. After repeatedly trying to correct their records, the help staff told me it could not be done.

A different problem arose when I used the migration assistant supplied with the new computer to transfer my data and programs to the new machine. Although the description of the migration assistant clearly states that incompatible applications will be moved to a special directory rather than installed, a common software package on the old machine, one that was not usable or needed on the new one, was installed anyway. A process began to consume CPU time at a high rate. Stopping that process required searching the Internet to find an installer for the obsolete product.

These incidents are so petty and so commonplace that readers must wonder why I write about them. It is precisely because such events are commonplace, and so indicative of lack of discipline, that such stories should concern anyone who uses or creates software.

As early as the late 1950s, some compilers came with a complete list of error messages and descriptions of the conditions that caused them. Today, such lists cannot be found. Often,
when reviewing a system, I will pick a random message or output symbol and ask, “When does that happen?” I never get a satisfactory answer.

There are methods of design and documentation that facilitate checking that a programmer has considered all possible cases (including such undesired events as incorrect input or the need to correct an earlier transaction) and provided appropriate mechanisms for responding to them. When such methods are used, people find serious errors in software that has been tested and used for years. When I talk or write about such methods, I am often told by colleagues, experienced students, and reviewers that, “Nobody does that.” They are right—that’s the problem!

Much of the fault lies with our teaching. Computer science students are not taught to work in disciplined ways. In fact, the importance of disciplined analysis is hardly mentioned. Of course, just telling students to be diligent is hardly enough. We need to:

▷ teach them what to do and how to do it—even in the first course;
▷ use those methods ourselves in every example we present;
▷ insist they use a disciplined approach in every assignment in every course where they write programs;
▷ check they have inspected and tested their programs diligently, and
▷ test their ability to check code systematically on examinations.

Many of us preach about the importance of determining the requirements a software product must satisfy, but we do not show students how to organize their work so they can systematically produce a requirements specification that removes all user-visible choices from the province of the programmer.

Some of us advise students to avoid dull work by automating it, but do not explain that this does not relieve an engineer of the responsibility to be sure the work was done correctly.

**Innovation and Disciplined Design**

It has become modish to talk about teaching creativity and innovation. We need to tell students that inventiveness is not a substitute for disciplined attention to the little details that make the difference between a product we like and a product we curse. Students need to be told how to create and use checklists more than they need to hear about the importance of creativity.

It is obviously important to give courses on picking the most efficient algorithms and to make sure that students graduate prepared to understand current technology and use new technology as it comes along, but neither substitutes for teaching them to be disciplined developers.

Disciplined design is both teachable and doable. It requires the use of the most basic logic, nothing as fancy as temporal logic or any of the best-known formal methods. Simple procedures can be remarkably effective at finding flaws and improving trustworthiness. Unfortunately, they are time-consuming and most decidedly not done by senior colleagues and competitors.

Disciplined software design requires three steps:

1. Determine and describe the set of possible inputs to the software.
2. Partition the input set in such a way that the inputs within each partition are all handled according to a simple rule.
3. State that rule.

Each of these steps requires careful review:

1. Those who know the application must confirm that no other inputs can ever occur.
2. Use basic logic to confirm that every input is in one—and only one—of the partitions.
3. Those who know the application, for example, those who will use the program, must confirm the stated rule is correct for every element of the partition.

These rules seem simple, but reality complicates them:

---

**Even sophisticated and experienced purchasers do not demand the documentation that would be evidence of disciplined design and testing.**

1. If the software has internal memory, the input space will comprise event sequences, not just current values. Characterizing the set of possible input sequences, including those that should not, but could, happen is difficult. It is very easy to overlook sequences that should not happen.
2. Function names may appear in the characterization of the input set. Verifying the correctness of the proposed partitioning requires knowing the properties of the functions named.
3. The rule describing the output value for some of the partitions may turn out to be complex. This is generally a sign that the partitioning must be revised, usually by refining a partition into two or more smaller partitions. The description of the required behavior for a partition should always be simple but this may imply having more partitions.

Similar “divide and conquer” approaches are available for inspection and testing.

While our failure to teach students to work in disciplined ways is the primary problem, the low standards of purchasers are also a contributing factor. We accept the many bugs we find when a product is first delivered, and the need for frequent error-correcting updates, as inevitable. Even sophisticated and experienced purchasers do not demand the documentation that would be evidence of disciplined design and testing.

We are caught in a catch-22 situation:

▷ Until customers demand evidence that the designers were qualified and disciplined, they will continue to get sloppy software.
▷ As long as there is no better software, we will buy sloppy software.
▷ As long as we buy sloppy software, developers will continue to use undisciplined development methods.
▷ As long as we fail to demand that developers use disciplined methods, we run the risk—nay, certainty—that we will continue to encounter software full of bugs.

---

David L. Parnas (parnas@mcmaster.ca) is Professor Emeritus at McMaster University and the University of Limerick as well as President of Middle Road Software. He has been looking for, and teaching, better software development methods for more than 40 years. He is still looking!

Copyright held by author.
The Profession of IT
Is Software Engineering Engineering?

Software engineering continues to be dogged by claims it is not engineering. Adopting more of a computer-systems view may help.

It is a time of considerable introspection for the computing field. We recognize the need to transcend the time-honored, but narrow image of, “We are programmers.” That image conveys no hint of our larger responsibilities as software professionals and limits us in our pursuit of an engineering model for software practice.

The search for an alternative to the programmer image is already a generation old. In 1989 we asked: Are we mathematicians? Scientists? Engineers? We concluded that we are all three. We adopted the term “computing,” an analogue to the European “informatics,” to avoid bias toward any one label or description.

Today, we want all three faces to be credible in an expanding world. The cases for computing as mathematics and as science appear to be widely accepted outside the field. However, the case for computing as engineering is still disputed by traditional engineers. Computer engineering (the architecture and design of computing machines) is accepted, but software engineering remains controversial.

In this column, we examine reasons for the persistent questions about software engineering and suggest directions to overcome them.

**Engineering Process**
The dictionary defines engineering as the application of scientific and mathematical principles to achieve the design, manufacture, and operation of efficient and economical structures, machines, processes, and systems.

**Software engineering may suffer from our habit of paying too little attention to how other engineers do engineering.**

When applied to software engineering, this definition calls attention to the importance of science and math principles of computing. Software engineering has also contributed principles for managing complexity in software systems.

Some definitions insist that engineering mobilizes properties of matter and sources of energy in nature. Although software engineering does not directly involve forces of nature, this difference is less important in modern engineering.

The main point of contention is whether the engineering practices for software are able to deliver reliable, dependable, and affordable software. With this in mind, the founders of the software engineering field, at the legendary 1968 NATO conference, proposed that rigorous engineering process in the design and implementation of software would help to overcome the “software crisis.”

In its most general form, the “engineering process” consists of a repeated cycle through requirements, specifications, prototypes, and test-
In software engineering, the process models have evolved into several forms that range from highly structured preplanning (waterfalls, spirals, Vs, and CMM) to relatively unstructured agile (XP, SCRUM, Crystal, and evolutionary). No one process is best for every problem.

Despite long experience with these processes, none consistently delivers reliable, dependable, and affordable software systems. Approximately one-third of software projects fail to deliver anything, and another third deliver something workable but not satisfactory. Often, even successful projects took longer than expected and had significant cost overruns. Large systems, which rely on careful preplanning, are routinely obsolescent by the time of delivery years after the design started. Faithful following of a process, by itself, is not enough to achieve the results sought by engineering.

**Engineering Practice**

Gerald Weinberg once wrote, “If software engineering truly is engineering, then it ought to be able to learn from the evolution of other engineering disciplines.” Robert Glass and his colleagues provocatively evaluated how often software engineering literature does this. They concluded that the literature relies heavily on software anecdotes and draws very lightly from other engineering fields. Walter Tichy found that fewer than 50% of the published software engineering papers tested their hypotheses, compared to 90% in most other fields.

So software engineering may suffer from our habit of paying too little attention to how other engineers do engineering. In a recent extensive study of practices engineers expect explicitly or tacitly, Riehle found six we do not do well.

1. **Predictable outcomes (principle of least surprise).** Engineers believe that unexpected behaviors can be not only costly, but dangerous; consequently, they work hard to build systems whose behavior they can predict. In software engineering, we try to eliminate surprises by deriving rigorous specifications from well-researched requirements, then using tools from program verification and process management to assure that the specifications are met. The ACM Risks Forum documents a seemingly unending series of surprises from systems on which much attention has been lavished. Writing in ACM SIGSOFT in 2005, Riehle suggested a cultural side of this: where researchers and artists have a high tolerance, if not love, for surprises, engineers do everything in their power to eliminate surprises. Many of our software developers have been raised in a research tradition, not an engineering tradition.

2. **Failure tolerance.** Henry Petroski writes, “An idea that unifies all engineering is the concept of failure. Virtually every calculation an engineer performs...is a failure calculation...to provide the limits that cannot be exceeded.” There is probably no more important task in engineering than that of risk management. Software engineers could more thoroughly examine and test their engineering solutions for their failure modes, and calculating the risks of all failures identified.

3. **Separation of design from implementation.** For physical world projects, engineers and architects represent a design with blueprints and hand off implementation to construction specialists. In current practice, software engineers do both, design and build (write the programs). Would separation be a better way?

4. **Reconciliation of conflicting forces and constraints.** Today’s engineers face many trade-offs between conflict-
The System
The problems surrounding the six issues listed here are in large measure the consequence of an overly narrow view of the system for which the software engineer is responsible. Although controlled by software, the system is usually a complex combination of software, hardware, and environment.

Platform independence is an ideal of many software systems. It means that the software should work under a choice of operating systems and computing hardware. To achieve this, all the platform-dependent functions are gathered into a platform interface module; then, porting the system to another platform entails only the building of that module for the new platform. Examples of this are the Basic Input-Output System (BIOS) component of operating systems and the Java Virtual Machine (JVM). When this can be achieved, the software engineer is justified in a software-centric view of the system.

But not all software systems are platform independent. A prominent example is the control system for advanced aircraft. The control system is implemented as a distributed system across many processors throughout the structure where they can be close to sensors and control surfaces. Another example is software in any large system that must constantly adapt in a rapidly changing environment. In these cases the characteristics of the hardware, the interconnections, and the environment continually influence the software design. The software engineer must either know the system well, or must interact well with someone who does. In such cases adding a system engineer to the team will be very important.

Engineering Team
No matter what process engineers use to achieve their system objectives, they must form and manage an engineering team. Much has been written on this topic. Software engineering curricula are getting better at teaching students how to form and work on effective teams, but many have a long way to go.

Every software team has four important roles to fill. These roles can be spread out among several people.

The software architect gathers the requirements and turns them into specifications, seeks an understanding of the entire system and its trade-offs, and develops an architecture plan for the system and its user interfaces.

The software engineer creates a system that best meets the architecture plan. The engineer identifies and addresses conflicts and constraints missed by the architect, and designs controls and feedbacks to address them. The engineer also designs and oversees tests. The engineer must have the experience and knowledge to design an economical and effective solution with a predictable outcome.

The programmer converts the engineering designs into working, tested code. Programmers are problem-solvers in their own right because they must develop efficient, dependable programs for the design. Moreover, anyone who has been a programmer knows how easy it is to make mistakes and how much time and effort are needed to detect and remove mistakes from code. When the software engineer has provided a good specification, with known exceptions predefined and controls clearly delineated, the programmer can work within a model that makes the job of implementation less error-prone.

The project manager is responsible for coordinating all the parts of the team, meeting the schedules, getting the resources, and staying within budgets. The project manager interfaces with the stakeholders, architects, engineers, and programmers to ensure the project produces value for the stakeholders.

In some cases, as noted previously, a systems engineer will also be needed on the team.

Conclusion
We have not arrived at that point in software engineering practice where we can satisfy all the engineering criteria described in this column. We still need more effective tools, better software engineering education, and wider adoption of the most effective practices. Even more, we need to encourage system thinking that embraces hardware and user environment as well as software.

By understanding the fundamental ideas that link all engineering disciplines, we can recognize how those ideas can contribute to better software production. This will help us construct the engineering reference discipline that Glass tells us is missing from our profession. Let us put this controversy to rest.
An anonymously attributed adage states: "With another name, social engineering would not be mistaken for engineering." Approximately 15 years ago, I published a short article in the *Journal of Engineering Education* arguing—among other things—that software engineering was not then engineering. I have now been asked whether enough has changed to make me think software engineering is engineering. My answer is: much has changed—with some changes weakening the separation between engineering and software engineering and some reinforcing it—but, overall, the argument stands. This answer will surprise those who, unaware of that article, think software engineering's status as engineering is obvious. I therefore think it wise to precede any explanation of why software engineering is not engineering by disposing of a few unexamined presumptions that might make software engineering's status as engineering seem obvious.

**Senses of Engineering**
"Engineering" has at least four senses in English. One, the oldest, understands engineering as tending engines (originally, "engines of war"). Casey Jones was an engineer in this sense; so is the custodian of my building, a licensed "boiler engineer"; and so too, the sailor rated "marine engineer." Neither engineers (strictly speaking) nor software engineers are engineers in this sense.
Almost the opposite of this first sense is what we might call the functional sense, engineering-as-invention-of-useful-objects. In this sense, the first engineer may have been the caveman (or cavewoman) who invented the club, cutting stone, or fire pit. Though this sense would certainly make software engineers engineers, there are at least two reasons to reject it here. First, the functional sense is too broad. Architects, industrial designers, and even weekend inventors are all engineers in this sense, making software engineering's claim to be engineering uninteresting. Second, the functional sense is anachronistic. It takes a sense of "engineering" that did not exist much before 1700 and applies it to cavemen, carpenters, tinkerers, and the like, who would have understood themselves quite differently.

The functional sense of engineering nonetheless seems relevant here. Software engineering's official Body of Knowledge offers this definition of software engineering: "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches; that is, the application of engineering to software."\(^2\)

The Body of Knowledge assumes, without argument (a mere "that is"), that engineering is a certain function, any "systematic, disciplined, quantifiable approach to the development, operation, and maintenance [of something]". That assumption must be false. It would force us, for example, to rank accounting—a field no one supposes to be engineering—as "financial-records engineering" (since accounting is a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of financial records).

Closer to our subject is a third sense, engineering-as-discipline. A discipline is a distinctive way of carrying on an activity, some combination of knowledge, skill, and judgment that must be learned. Any craft or trade has its discipline—as do many activities that are not craft or trade, such as meditation or calisthenics. In this sense, neither architects, nor industrial designers, nor weekend inventors are engineers. Architecture and industrial design each have a discipline easily distinguished from engineering's. Weekend inventors have no discipline at all; they may invent any way they like.

Software engineering is not engineering in this third sense. The body of knowledge engineers are supposed to learn differs in important ways from software engineering's body of knowledge. So, for example, engineers have to take courses concerned with the material world, such as chemistry and statistics; software engineers do not. Software engineering's official Body of Knowledge was
in fact an important step in clarifying the distinction between engineering proper and software engineering. It requires software engineers to know things other engineers do not and not to know some things other engineers do know.

Software engineering has, indeed, become a profession. What it has not become is part of the engineering profession.

The last sense of engineering we need to distinguish here is engineering-as-profession. A profession is (we may say) a number of individuals in the same occupation voluntarily organized to earn a living by openly serving a moral ideal in a morally permissible way beyond what law, market, morality, and public opinion would otherwise require. An occupation is a discipline by which one may, and some do, earn a living. Both engineering and software engineering are now occupations but, having (as just noted) different disciplines, must be different occupations. That is one reason why they cannot share a profession. There is another.

The Software Engineering Code of Ethics and Professional Practice differs in significant ways from all the engineering codes I know. Software engineers are, for example, supposed to "[m]oderate the interests of the software engineer, the employer, the client and the users with the public good" (1.02). Engineers do not now have such a duty to moderate.

Software engineering has, indeed, become a profession. What it has not become is part of the engineering profession. Anyone who claims otherwise must find a sense of engineering different from those distinguished here, one that makes software engineering a part of engineering without including as well disciplines, occupations, or professions, such as architecture or accounting, that clearly are not part of engineering.

Professions are voluntary associations. You cannot become a member simply by claiming to be one. You must be admitted (by the profession, not just by a technical society like the ACM). Engineering has a long history of other occupations claiming to be engineering: recent examples include genetic engineering (a kind of tinkering with genes); reengineering (a fad in management); and financial engineering (gambling on Wall Street). Software engineering actually began with an attempt to copy engineering practices, making its claim to be engineering more respectable than most. But the enormous complexity of software has forced software engineering to develop in ways engineering has not—and may never. Many of the very methods that make software engineering useful distinguish it from engineering. Engineers have good reason to continue to treat software engineers as belonging to another profession.
I have, I hope, just explained why I still think software engineering is not engineering in a way that engineers should recognize. I now want to point out four reasons to think that engineering might someday merge with software engineering. All four are, oddly, changes in engineering, not software engineering.

- Electrical and computer engineering (ECE) is often thought to be the field of engineering closest to software engineering. Over the last decade, ECE has become less committed to traditional engineering courses concerned with the material world. So, for example, a number of ECE departments, including the one at the University of Illinois at Urbana-Champaign, have stopped requiring statics, dynamics, and thermodynamics. If that trend continues, then either ECE will split off from the main body of engineering or engineering’s core of required engineering courses will increasingly resemble software engineering’s.

- Since the 1700s, engineers have had to know just two natural sciences: physics and chemistry. Recently, some programs in environmental engineering, biomedical engineering, and agricultural engineering have begun to allow students to substitute biology for physics or chemistry. For engineers, this makes sense, since several of the new frontiers of engineering rely on biology rather than physics and chemistry (as until recently). But, if this trend continues, engineering’s science core will increasingly resemble the science courses software engineers take to satisfy general distribution requirements.

- Engineers are increasingly replacing mechanical systems with software. Not only do most engineers now use software regularly, many write specifications for software, modify existing programs themselves, or even write (simple) programs. Whether or not software engineers do any engineering, engineers increasingly engage in activities that look like software engineering (even if these engineers do not call themselves "software engineers" and do not work the way that software engineers would). Whether some fields of engineering will dissolve into software engineering seems an open question.

- Computer science used to have an accreditation body separate from engineering’s. That is no longer true. All computer science programs, including software engineering, are now under engineering’s accreditation body, ABET. Of course, the accreditation process and standards distinguish between engineering programs and computer science programs. But that distinction does not preclude eventual merger. ABET has always distinguished between various fields (or subdisciplines) of engineering. So, for example, it always sent mechanical engineers to review a mechanical engineering program; electrical engineers, to review an electrical engineering program; and so on. The expansion of ABET’s accreditation powers makes it easier than before for software engineering to merge into engineering, indeed, for all of computer science to do that.
Whether or not software engineers do any engineering, engineers increasingly engage in activities that look like software engineering.

Having pointed out four reasons that seem to point to software engineering's eventual merger with engineering, I now point out three reasons to believe the merger will not happen soon, if at all:

- All engineering is still fundamentally about physical systems; software engineering is not. Even a field so closely allied to software engineering as computer engineering must take into account physical factors in design, for example, heat produced in a microchip or speed of electrical current, to a degree software engineers do not.

- Software engineering is today a large profession, indeed, one of the largest—half the size of engineering, true, but about the same size as medicine or law. With so many practitioners, software engineering is more likely to divide than to join up with another large profession.

- If computer science ever ceased to be the home of software engineering, the most likely new home might well be management information systems or information technology management. These business disciplines resemble software engineering at least as much as engineering does. In practice, most software engineers work more with information systems managers than with engineers.

Conclusion

Whether knowledge of the future is possible is a perennial question in philosophy. What is certain is that prophets are seldom right on any important question. So, I make no claim to know whether software engineering will ever merge with engineering. I claim only to know that—despite the common term "engineering"—software engineering is not now engineering.

References


Author
Michael Davis (davis2643@gmail.com) is a senior fellow at the Center for the Study of Ethics in the Professions and a professor of philosophy at Illinois Institute of Technology, Chicago.

Footnotes

a. For a defense of this definition, see my article "Is Engineering a Profession Everywhere?" *Philosophia* 37 (June 2009), 211–225.


c. Michal Young and Stuart Faulk, "Sharing What We Know About Software Engineering," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (FoSER ’10), ACM, 439–442, argue that engineering has much to learn from software engineering—inadvertently making clear how much engineering’s discipline differs from software engineering’s.

d. For a darker route to this conclusion, see David L. Parnas, "Risks of undisciplined development," *Commun. ACM* 53, 10 (Oct. 2010), 25–27. Note that Parnas, though a star of software engineering, is an electrical engineer—both by discipline and declaration—looking at software engineering the way knowledgeable engineers typically do.

Thanks to Keith Miller and Rachelle Hollander for helping me think through this column.

Copyright held by author.

The Digital Library is published by the Association for Computing Machinery. Copyright © 2011 ACM, Inc.

---

**USER COMMENTS**

Davis mentions four senses of the word "engineering", none of which qualify software engineering as an engineering discipline. A fifth sense appeared, until recently, in the Merriam-Webster dictionary: the application of science and mathematics to the design of useful things. Software engineering is not engineering in this sense, either. It could be, since mathematical logic provides principles on which useful software can be built and guaranteed to have certain behavioral properties. If software engineering were to go in this direction and the other fields of engineering adopted practices akin to present-day software engineering, the graphs would cross at some point and then diverge again. Ironically, software engineering would be going up (in my view) while the others fields descended.

- Rex Page (Univ of Oklahoma)

— Anonymous, November 9, 2011

Like most dictionary definitions, this one seems to be over-broad. Even synthetic chemistry and architecture would count as engineering
under it. Both use science and mathematics to make useful things (new chemical compounds or buildings).

—Michael Davis

CACM Administrator, November 10, 2011

It may only be tangentially related to the topic at hand, but it might be worth considering another thing that differs between the two, on the academic level:

that Software Engineers who learn the trade in college typically study in a field titled "Computer Science" - whereas a Civil Engineer would study "Civil Engineering".

Possibly just a nomenclature issue, but a lot of what people learn in Computer Science isn't really "software engineering" (to the point where some academic institutions have split their CS programs to focus on the science of computers versus the art of software design).

— Mark Laczin, November 16, 2011

If the dictionary definition is too broad, then what passes for software engineering today fails to reside even on the fringes of engineering.

— Rex Page, November 16, 2011

The article seems to be assuming that (real) engineering is about physical systems and processes, whereas software is not concerned about these things, and therefore is not engineering. However I don't know of any definition of modern engineering that specifically or exclusively requires this physical world connection.

My favourite "definition" of engineering stems from an old adage about (civil) engineers, which basically states that while anyone can design a bridge over a river that will stand up in a storm, it takes an engineer to design a bridge that will only just stand up in a storm. In other words, an engineer must constantly care about the costs (i.e., money, time, resources, social, environmental, operational, etc.) involved in a project, as well as making sure the end result does what it was supposed to do. This "cost" focus is probably one reason why so many senior engineers end up in management roles - managing costs and resources is second nature. (We'll ignore the point about whether managing people is second nature as well :).

So what about software engineering as "engineering" then? I guess that may depend partly on whether you define a software engineer as an engineer by training (i.e., studied engineering at school/university) with a focus or specialisation on software, or by "mislabelling" someone who trained as a computer scientist (i.e., studied computer science), which seems to be the most common use of "software engineer" that I see/have seen in the market over the last decade or two.

Training background aside, a real software engineer, like any engineer, needs to be concerned with the costs (all of the costs) of the software in a project, as well as ensuring that the final product is fit-for-purpose. Which brings us back to the Body of Knowledge definition, and specifically to "that is, the application of engineering to software", which makes perfect sense provided you've defined what engineering is...

— Jason Pieloort, November 16, 2011

We have to move forward to 21st Century with new meanings and interpretations. This may include Software Engineering as an Engineering discipline without question.

Thanks
Pradyot Sahu, 3innovate

— Pradyot Sahu, November 17, 2011
Viewpoint

Computer Science Can Use More Science

Software developers should use empirical methods to analyze their designs to predict how working systems will behave.

The so-called LAMP stack (Linux OS/Apache Internet server/MySQL DBMS/Perl PL) consists of 10 million lines of code, interacting in myriad ways to achieve impressive functionality and performance. This approaches the intellectual complexity of the Saturn V rocket (with three million parts) that took humans to the moon. The similarities between these two enormous engineering feats are important: good engineering design practices have been followed; requirements were defined and met; rigorous testing and debugging has taken place. Yet, to date, the LAMP stack is much less well understood than the Saturn V rocket. It is much more difficult to predict how the LAMP stack will perform under varying conditions and where things might go wrong than it is to consider how the Saturn V may behave in different operating environments.

What the engineers at NASA have that the developers and users of the LAMP stack do not is an understanding of how configurations of system components (in NASA’s case, physical materials) will behave in a variety of (physical) contexts and an idea of where the boundary conditions of those behaviors lie. This understanding is the product of physical theories about universal laws of nature—laws that have been identified through a tradition of model construction and empirical testing to produce general principles.

The Saturn V rocket that launched humans to the moon relied on an IBM System/360 for data processing.
Developers of the LAMP stack do not approach this level of understanding.

If computer science can achieve this level, that is, uncover the kinds of scientific theories and laws that physics provides, software developers building artifacts could do what NASA engineers do now: analyze their designs according to underlying theories and predict how a working system will behave. Extracting such laws and principles is the goal of science.

Whether and how computer science is a science has been a topic of discussion since the beginnings of the field. Thinkers including Herbert Simon have offered deep insight into the special nature of computation and how it exists as a phenomenon—in part, it’s artificial, something we create. However, the nature of what computer science is studying, or how best to study it, is by no means a settled topic. Just because we design and build computational systems does not mean we understand them; special care and much more work is needed to correctly characterize computation as a scientific endeavor.

The early challenges of the field were engineering in nature, leading to a de-emphasis of empirical methods. In recent years, with increasing complexity of computational systems, empirical methods are now needed to discover system limits and predict future behavior.

The Methodology of Debugging

Let’s start with the parts of the scientific method we do well. Programmers engage naturally in one of the purest forms of empirical investigation: debugging. Whenever a program exhibits a fault, in which a run over particular input data yields an exception or incorrect output, the programmer generally follows a series of steps. First, the programmer develops a hypothesis of where the fault lies (whether in a particular statement or more generally with the logic of the design) and considers how to correct it. This hypothesis is based on the programmer’s mental model of how the programming environment, referenced libraries, operating system, and environment (such as input data, system events, and so forth) function and interact. This model is generated and refined through additional runs of the program with print statements or through interaction with a debugger in which the program state can be examined as the program is single-stepped. The programmer then tests this hypothesis by making changes believed to fix the problem and rerunning the program, predicting that the change will result in the correct output for that particular test input. If the program still faults, the programmer refines the hypothesis, updating their mental model of how they believe the underlying system is behaving, and further investigates possible causes. Programmers frequently demonstrate they are highly skilled at understanding complex interactions by forming models and testing and revising them.

Why, then, do we not have the same depth of understanding of the LAMP stack as the Saturn V rocket?

Toward General Predictive Models

Why, then, do we not have the same depth of understanding of the LAMP stack as the Saturn V rocket? Formulating hypotheses, devising tests and carrying them out, and then revising hypotheses based on test outcomes are critical components of the basic methodology of empirical science. But these activities are not all that science does. What is missing from the debugging picture is searching for generalizations: attempting to extract general patterns and identify the factors that govern their behavior. This requires looking past individual systems, past simply getting the program to run in a particular context, to characterizing general principles of system behavior. The predictive model underlying the hypothesis testing undertaken by our programmer is specific to the extreme.

Empirical generalization progresses from description of particular instances to prediction (via models) of classes of systems. This progression is advanced to an explanation that connects what is going on in particular observed cases to a general class, in this case of certain kinds of software systems in certain kinds of configurations. The models and the kinds of systems they describe should both be broadly construed to render a true understanding of these systems. Are such general predictive models even possible for software systems? Experience indicates that when such systems are examined experimentally, with an explicit goal of discovering causal models, such models can in fact be found and highlight deep structure.

The articulation of scientific theories as predictive causal models, the methodology of empirical generalization, and the evaluation of such theories via hypothesis testing is prevalent in isolated sub-disciplines of computer science, including human-computer interaction, empirical software engineering, Web science, and data mining. It is, however, rarely used in other sub-disciplines of computer science, especially those concerned with software systems artifacts, such as compilers, databases, networks, operating systems, and programming languages.

Examples of Computational Models

One example of a computational model that is the product of empirical generalization is the Theory of Locality. This model arose from a study of the cost of managing page transfers between main memory and a much slower disk drive, within the general area of operating systems. What Denning found was that data relevant to the current context of a running program tended to be grouped in space and time to local chunks, and this was the product of the way programs are written by humans, rather than due to any underlying constraints of the computing system. When our memory management algorithms respect this general pattern, performance improves dramatically. The resulting theory—that human-constructed information processing systems will exhibit locality—is inherently predictive and has been tested many, many times. This theory has subsequently been generalized to apply to computational systems of all kinds: “in virtual memory to organize caches for address translation and to design the replace-
Good scientific models will generate new questions to be answered and will drive our field to yet deeper understanding.

a threshold value of certain parameters. Subsequent studies have since systematically explored this phenomenon, and have discovered that where these transitions occur may depend on a variety of additional factors, including the kind of problem-solving method used. Different problem-solving methods provide us with different tools with which to study the complexity of these problems; an apt analogy has been made to astronomers using telescopes that operate on different light wavelengths to provide different perspectives on the structure of the cosmos. The jury is still out on how to best characterize the transition behavior, but the key point for our purposes is to again focus on the methodology employed: these researchers conduct experiments, take measurements, and refine their models, while striving for generality.

Realizing the Benefits of More Science in CS
These examples and other extant computer science theories emphasize that by embracing the methodology of developing and evaluating predictive models through experimentation over multiple members of a class of software systems, a more complete understanding of such artifacts will emerge. In addition, this observational and experimental scientific perspective encourages the computer scientist to actively look for relevant phenomena that may have been missed because they aren’t currently described by existing, closed-form analytical descriptions. The developed explanatory model provides better prediction and ultimately more dependable products built on those principles. Finally, good scientific models will generate new questions to be answered and will drive our field to yet deeper understanding.

How can these benefits be realized? How might we change what we do? We can adapt our already very skilled hypothesis testing in debugging and broaden it by asking more general questions, identifying the classes of system properties that contribute to behavior, identifying their boundary conditions, and working to fit them into a unified picture. We can also more broadly adopt additional methodological tools, such as from statistics and dynamic systems; a number of computer science subfields already profitably do so.

The pristine presentations of scientific reasoning and the tremendous successes of such reasoning in other fields may appear to the practicing computer scientist as out of reach. But many of our colleagues have started down this path, the tools are accessible, and the promise is great.

Acknowledgments
We thank the reviewers and editors for an enjoyable and vigorous conversation around the topic of this column. The reviewers will not agree with every detail, but they have certainly helped us refine the presentation.

References

Clayton T. Morrison (clayton@cs.arizona.edu) is a research assistant professor of computer science at the University of Arizona. He has more than a decade of experience building artificial intelligence systems for DARPA, AFOSR and NSF projects.

Richard T. Snodgrass (rts@cs.arizona.edu) is a professor of computer science at the University of Arizona. He is an ACM Fellow, has chaired the ACM Publications Board and SIGMOD, and has served as the editor-in-chief of ACM Transactions on Database Systems.
ONE SCORE AND SEVEN YEARS AGO, KEN THOMPSON BROUGHT FORTH A NEW PROBLEM, CONCEIVED BY THINKING, AND DEDICATED TO THE PROPOSITION THAT THOSE WHO TRUSTED COMPUTERS WERE IN DEEP TROUBLE.

I AM, OF COURSE, TALKING ABOUT THOMPSON’S 1984 ACM A.M. TURING AWARD LECTURE—“REFLECTIONS ON TRUSTING TRUST.” UNLESS YOU REMEMBER THIS PIECE BY HEART, YOU MIGHT WANT TO TAKE A MOMENT TO READ IT IF AT ALL POSSIBLE (HTTP://BIT.LY/NNGH5B).

THE ONE SENTENCE IN THOMPSON’S LECTURE THAT REALLY, REALLY MATTERS IS: “YOU CAN’T TRUST CODE THAT YOU DID NOT TOTALY CREATE YOURSELF.”

THIS STATEMENT IS NOT A MATTER OF POLITICS, OPINION, TASTE, OR IN ANY OTHER WAY A VALUE JUDGMENT; IT IS A FUNDAMENTAL LAW OF NATURE, WHICH FOLLOWS DIRECTLY FROM PURE MATHEMATICS IN THE GENERAL VICINITY OF THE WORKS OF TURING AND GÖDEL. IF YOU DOUBT THIS, PLEASE (AT YOUR CONVENIENCE) READ DOUGLAS HOFSTADTER’S CLASSIC GÖDEL, ESCHER, BACH, AND WHEN YOU GET TO THE PART ABOUT “MR. CRAB’S RECORD PLAYER,” SUBSTITUTE “MR. CRAB’S LAPTOP.”

GODEL, ESCHER, BACH

HOFSTADTER’S BOOK, ORIGINALLY PUBLISHED IN 1979, DOES NOT IN ANY WAY DETRACT FROM KEN THOMPSON’S FAME, IF, INDEED, HIS LECTURE WAS INSPIRED BY IT;
1979 was a long time ago, and it’s possible that not every reader may know of—much less have read—this book. My editor proposed that I summarize or quote from it to make things clearer for such readers.

Considering that Gödel, Escher, and Bach are all known for their intricate multilayered works and that Hofstadter’s book is a well-mixed stew not only of their works, but also of the works of Cantor, Church, Gantòr, Turing, and pretty much any other mathematician or philosopher you care to mention, I will not attempt a summary beyond: “It’s a book about how we think.”

The relevant aspect of the book here is Gödel’s incompleteness theorem, which, broadly speaking, says that no finite mathematical system can resolve, definitively, the truth value of all possible mathematical conjectures expressible in that same
In strict mathematical terms, you cannot trust a house you did not totally create yourself, but in reality, most of us will trust a house built by a suitably skilled professional.

practice

In the book this is illustrated with a fable about Mr. Crab’s “perfect record player,” which, because it can play any and all sounds, can also play sounds that make it resonate and self-destroy—a vulnerability exploited on the carefully constructed records of Mr. Crab’s adversary, Mr. Tortoise.

Mr. Crab tries to protect against this attack by preanalyzing records and rearranging the record player to avoid any vulnerable resonance frequencies, but Mr. Tortoise just crafts the sounds on his records to the resonance frequencies of the part of the record player responsible for the rearrangement. This leaves Mr. Crab no alternative but to restrict his record playing to only his own, preapproved records, thereby severely limiting the utility of his record player.

Malware-scanning programs try to classify executable code into “safe” and “unsafe,” instead of mathematical conjectures into “true” and “false,” but the situation and result are the same: there invariably is a third pile called “cannot decide either way,” and whatever ends up in that pile is either a security or a productivity risk for the computer user.

Amusingly, malware scanners almost unfailingly classify malware-scanner programs, including themselves, as malware, and therefore contain explicit exemptions to suppress these “false” positives. These exemptions are of course exploitable by malware—which means the classification of malware scanners as malware was correct to begin with. “Quis custodiet ipsos custodes?” (Who will guard the guards themselves?)

Back to Thompson

In 1984, the Thompson lecture evoked wry grins and minor sweating for Unix system administrators at universities, because those were the only places where computers were exposed to hostile users who were allowed to compile their own programs. Apart from sporadic and mostly humorous implementations, however, no apocalyptic horsemen materialized in the sky.

In recent years, there have been a number of documented instances where open source projects were broken into and their source code modified to add backdoors. As far as I am aware, none of these attacks so far has reached further than the lowest rung on Ken Thompson’s attack ladder in the form of a hardcoded backdoor, clearly visible in the source code. Considering the value to criminals, however, it is only a matter of time before more advanced attacks, along the line Thompson laid out, will be attempted.

The security situation with commercial closed-source software is anyone’s guess, but there is no reason to think—and no credible factual basis for a claim—that the situation is any different or any better than it is for open source projects.

The now-legendary Stuxnet malware incident has seriously raised the bar for just how sophisticated attacks can be. The idea that a widely deployed implementation of Java is compiled with a compromised compiler is perfectly reasonable. Outsourced software development does not make that scenario any less realistic, likely, or scary.

We Have to Do Something, But What?

We have to do something that actually works, as opposed to accepting a security circus in the form of virus or malware scanners and other mathematically proven insufficient and inefficient efforts. We are approaching the point where people and organizations are falling back to pen and paper for keeping important secrets, because they no longer trust their computers to keep them safe.

Ken Thompson’s statement—“You can’t trust code that you did not totally create yourself”—points out a harsh and inescapable reality. Just as we don’t expect people to build their own cars, mobile phones, or homes, we cannot expect secretaries to create their own text-processing programs nor accountants to create their own accounting systems and spreadsheet software. In strict mathematical terms, you cannot trust a house you did not totally create yourself—but in reality, most of us will trust a house built by a suitably skilled professional. Usually we trust it more than the one we might have built ourselves—even when we may have never met the builder and/or when the builder is
dead. The reason for this trust is that shoddy construction has had negative consequences for builders for more than 3,700 years. “If a builder builds a house for someone, and does not construct it properly, and the house which he built falls in and kills its owner, then the builder shall be put to death.” (Hammurabi’s Code, approx. 1700 BC)

Today the operant legal concept is “product liability,” and the fundamental formula is “if you make money selling something, you’d better do it properly, or you will be held responsible for the trouble it causes.” I want to point out, however, that there are implementations of product liability other than those in force in the U.S. For example, if you burn yourself on hot coffee in Denmark, you burn yourself on hot coffee. You do not become a millionaire or necessitate signs pointing out that the coffee is hot.

Some say the only two products not covered by product liability today are religion and software. For software that has to end; otherwise, we will never get a handle on the security madness unfolding before our eyes almost daily in increasingly dramatic headlines. The question is how to introduce product liability, because just imposing it would instantly shut down any and all software houses with just a hint of a risk management function on their organizational charts.

A Software Liability Law
My straw-man proposal for a software liability law has three clauses:

Clause 0. Consult criminal code to see if any intentionally caused damage is already covered. I am trying to impose a civil liability only for unintentionally caused damage, whether a result of sloppy coding, insufficient testing, cost cutting, incomplete documentation, or just plain incompetence. Intentionally inflicted damage is a criminal matter, and most countries already have laws on the books for this.

Clause 1. If you deliver software with complete and buildable source code and a license that allows disabling any functionality or code by the licensee, then your liability is limited to a refund. This clause addresses how to avoid liability: license your users to inspect and chop off any and all bits of your software they do not trust or do not want to run, and make it practical for them to do so.

The word disabling is chosen very carefully. This clause grants no permission to change or modify how the program works, only to disable the parts of it that the licensee does not want. There is also no requirement that the licensee actually look at the source code, only that it was received.

All other copyrights are still yours to control, and your license can contain any language and restriction you care to include, leaving the situation unchanged with respect to hardware locking, confidentiality, secrets, software piracy, magic numbers, and so on. Free and open source software is obviously covered by this clause, and it does not change its legal situation in any way.

Clause 2. In any other case, you are liable for whatever damage your software causes when used normally. If you do not want to accept the information sharing in Clause 1, you would fall under Clause 2 and have to live with normal product liability, just as manufacturers of cars, blenders, chainsaws, and hot coffee do. How dire the consequences and what constitutes “used normally” are for the legislature and courts to decide.

An example: A salesperson from one of your longtime vendors visits and delivers new product documentation on a USB key. You plug the USB key into your computer and copy the files onto the computer. This is “used normally” and should never cause your computer to become part of a botnet, transmit your credit card number to Elbonia, or send all your design documents to the vendor.

The majority of today’s commercial software would fall under Clause 2. To give software houses a reasonable chance to clean up their acts and/or to fall under Clause 1, a sunrise period would make sense, but it should be no longer than five years, as the laws would be aimed at solving a serious computer security problem.

And that is it, really. Software houses will deliver quality and back it up with product liability guarantees, or their customers will endeavor to protect themselves.

Would it Work?
There is little doubt that my proposal would increase software quality and computer security in the long run, which is exactly what the current situation calls for.

It is also pretty certain there will be some short-term nasty surprises when badly written source code gets a wider audience. When that happens, it is important to remember that today the good guys have neither the technical nor legal ability to know if they should even be worried, as the only people with source-code access are the software houses and the criminals.

The software houses would yell bloody murder if any legislator were to introduce a bill proposing these stipulations, and any pundit and lobbyist they could afford would spew their dire predictions that “this law will mean the end of computing as we all know it!”

To which my considered answer would be: “Yes, please! That was exactly the idea.”

Related articles
on queue.acm.org
CTO Roundtable: Malware Defense
http://queue.acm.org/detail.cfm?id=1731902
All Things Being Equal?
Stan Kelly-Bootle
http://queue.acm.org/detail.cfm?id=1348596
B.Y.O.C. (1,342 Times and Counting)
Poul-Henning Kamp
http://queue.acm.org/detail.cfm?id=1944489

References

Poul-Henning Kamp (phk@FreeBSD.org) has programmed computers for 26 years and is the inspiration behind bikesinhedcorp. His software has been widely adopted as “under the hood” building blocks in both open source and commercial products. His most recent project is the Varnish HTTP accelerator, which is used to speed up large Web sites such as Facebook.

© 2011 ACM 0001-0782/11/11 S10.00
Software Engineering Code of Ethics and Professional Practice


- Short Version
- Full Version

Software Engineering Code of Ethics and Professional Practice (Short Version)

PREAMBLE

The short version of the code summarizes aspirations at a high level of the abstraction; the clauses that are included in the full version give examples and details of how these aspirations change the way we act as software engineering professionals. Without the aspirations, the details can become legalistic and tedious; without the details, the aspirations can become high sounding but empty; together, the aspirations and the details form a cohesive code.

Software engineers shall commit themselves to making the analysis, specification, design, development, testing and maintenance of software a beneficial and respected profession. In accordance with their commitment to the health, safety and welfare of the public, software engineers shall adhere to the following Eight Principles:

1. PUBLIC - Software engineers shall act consistently with the public interest.

2. CLIENT AND EMPLOYER - Software engineers shall act in a manner that is in the best interests of their client and employer consistent with the public interest.

3. PRODUCT - Software engineers shall ensure that their products and related modifications meet the highest professional standards possible.

4. JUDGMENT - Software engineers shall maintain integrity and independence in their professional judgment.

5. MANAGEMENT - Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance.

6. PROFESSION - Software engineers shall advance the integrity and reputation of the profession consistent with the public interest.

7. COLLEAGUES - Software engineers shall be fair to and supportive of their colleagues.

8. SELF - Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.

Software Engineering Code of Ethics and Professional Practice (Full Version)

PREAMBLE
Computers have a central and growing role in commerce, industry, government, medicine, education, entertainment and society at large. Software engineers are those who contribute by direct participation or by teaching, to the analysis, specification, design, development, certification, maintenance and testing of software systems. Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice.

The Code contains eight Principles related to the behavior of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession. The Principles identify the ethically responsible relationships in which individuals, groups, and organizations participate and the primary obligations within these relationships. The Clauses of each Principle are illustrations of some of the obligations included in these relationships. These obligations are founded in the software engineer’s humanity, in special care owed to people affected by the work of software engineers, and the unique elements of the practice of software engineering. The Code prescribes these as obligations of anyone claiming to be or aspiring to be a software engineer.

It is not intended that the individual parts of the Code be used in isolation to justify errors of omission or commission. The list of Principles and Clauses is not exhaustive. The Clauses should not be read as separating the acceptable from the unacceptable in professional conduct in all practical situations. The Code is not a simple ethical algorithm that generates ethical decisions. In some situations standards may be in tension with each other or with standards from other sources. These situations require the software engineer to use ethical judgment to act in a manner which is most consistent with the spirit of the Code of Ethics and Professional Practice, given the circumstances.

Ethical tensions can best be addressed by thoughtful consideration of fundamental principles, rather than blind reliance on detailed regulations. These Principles should influence software engineers to consider broadly who is affected by their work; to examine if they and their colleagues are treating other human beings with due respect; to consider how the public, if reasonably well informed, would view their decisions; to analyze how the least empowered will be affected by their decisions; and to consider whether their acts would be judged worthy of the ideal professional working as a software engineer. In all these judgments concern for the health, safety and welfare of the public is primary; that is, the "Public Interest" is central to this Code.

The dynamic and demanding context of software engineering requires a code that is adaptable and relevant to new situations as they occur. However, even in this generality, the Code provides support for software engineers and managers of software engineers who need to take positive action in a specific case by documenting the ethical stance of the profession. The Code provides an ethical foundation to which individuals within teams and the team as a whole can appeal. The Code helps to define those actions that are ethically improper to request of a software engineer or teams of software engineers.

The Code is not simply for adjudicating the nature of questionable acts; it also has an important educational function. As this Code expresses the consensus of the profession on ethical issues, it is a means to educate both the public and aspiring professionals about the ethical obligations of all software engineers.

**PRINCIPLES**

**Principle 1: PUBLIC**

**Software engineers shall act consistently with the public interest. In particular, software engineers shall, as appropriate:**

1.01. Accept full responsibility for their own work.

1.02. Moderate the interests of the software engineer, the employer, the client and the users with
the public good.

1.03. Approve software only if they have a well-founded belief that it is safe, meets specifications, passes appropriate tests, and does not diminish quality of life, diminish privacy or harm the environment. The ultimate effect of the work should be to the public good.

1.04. Disclose to appropriate persons or authorities any actual or potential danger to the user, the public, or the environment, that they reasonably believe to be associated with software or related documents.

1.05. Cooperate in efforts to address matters of grave public concern caused by software, its installation, maintenance, support or documentation.

1.06. Be fair and avoid deception in all statements, particularly public ones, concerning software or related documents, methods and tools.

1.07. Consider issues of physical disabilities, allocation of resources, economic disadvantage and other factors that can diminish access to the benefits of software.

1.08. Be encouraged to volunteer professional skills to good causes and contribute to public education concerning the discipline.

Principle 2: CLIENT AND EMPLOYER
Software engineers shall act in a manner that is in the best interests of their client and employer, consistent with the public interest. In particular, software engineers shall, as appropriate:

2.01. Provide service in their areas of competence, being honest and forthright about any limitations of their experience and education.

2.02. Not knowingly use software that is obtained or retained either illegally or unethically.

2.03. Use the property of a client or employer only in ways properly authorized, and with the client's or employer's knowledge and consent.

2.04. Ensure that any document upon which they rely has been approved, when required, by someone authorized to approve it.

2.05. Keep private any confidential information gained in their professional work, where such confidentiality is consistent with the public interest and consistent with the law.

2.06. Identify, document, collect evidence and report to the client or the employer promptly if, in their opinion, a project is likely to fail, to prove too expensive, to violate intellectual property law, or otherwise to be problematic.

2.07. Identify, document, and report significant issues of social concern, of which they are aware, in software or related documents, to the employer or the client.

2.08. Accept no outside work detrimental to the work they perform for their primary employer.

2.09. Promote no interest adverse to their employer or client, unless a higher ethical concern is being compromised; in that case, inform the employer or another appropriate authority of the ethical concern.

Principle 3: PRODUCT
Software engineers shall ensure that their products and related modifications meet the highest professional standards possible. In particular, software engineers shall, as appropriate:

3.01. Strive for high quality, acceptable cost and a reasonable schedule, ensuring significant tradeoffs are clear to and accepted by the employer and the client, and are available for consideration by the user and the public.

3.02. Ensure proper and achievable goals and objectives for any project on which they work or propose.

3.03. Identify, define and address ethical, economic, cultural, legal and environmental issues related to work projects.

3.04. Ensure that they are qualified for any project on which they work or propose to work by an appropriate combination of education and training, and experience.

3.05. Ensure an appropriate method is used for any project on which they work or propose to work.

3.06. Work to follow professional standards, when available, that are most appropriate for the task at hand, departing from these only when ethically or technically justified.

3.07. Strive to fully understand the specifications for software on which they work.

3.08. Ensure that specifications for software on which they work have been well documented, satisfy the users’ requirements and have the appropriate approvals.

3.09. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work and provide an uncertainty assessment of these estimates.

3.10. Ensure adequate testing, debugging, and review of software and related documents on which they work.

3.11. Ensure adequate documentation, including significant problems discovered and solutions adopted, for any project on which they work.

3.12. Work to develop software and related documents that respect the privacy of those who will be affected by that software.

3.13. Be careful to use only accurate data derived by ethical and lawful means, and use it only in ways properly authorized.

3.14. Maintain the integrity of data, being sensitive to outdated or flawed occurrences.

3.15 Treat all forms of software maintenance with the same professionalism as new development.

Principle 4: JUDGMENT

Software engineers shall maintain integrity and independence in their professional judgment. In particular, software engineers shall, as appropriate:

4.01. Temper all technical judgments by the need to support and maintain human values.

4.02 Only endorse documents either prepared under their supervision or within their areas of competence and with which they are in agreement.
4.03. Maintain professional objectivity with respect to any software or related documents they are asked to evaluate.

4.04. Not engage in deceptive financial practices such as bribery, double billing, or other improper financial practices.

4.05. Disclose to all concerned parties those conflicts of interest that cannot reasonably be avoided or escaped.

4.06. Refuse to participate, as members or advisors, in a private, governmental or professional body concerned with software related issues, in which they, their employers or their clients have undisclosed potential conflicts of interest.

Principle 5: MANAGEMENT

Software engineering managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance. In particular, those managing or leading software engineers shall, as appropriate:

5.01 Ensure good management for any project on which they work, including effective procedures for promotion of quality and reduction of risk.

5.02. Ensure that software engineers are informed of standards before being held to them.

5.03. Ensure that software engineers know the employer's policies and procedures for protecting passwords, files and information that is confidential to the employer or confidential to others.

5.04. Assign work only after taking into account appropriate contributions of education and experience tempered with a desire to further that education and experience.

5.05. Ensure realistic quantitative estimates of cost, scheduling, personnel, quality and outcomes on any project on which they work or propose to work, and provide an uncertainty assessment of these estimates.

5.06. Attract potential software engineers only by full and accurate description of the conditions of employment.

5.07. Offer fair and just remuneration.

5.08. Not unjustly prevent someone from taking a position for which that person is suitably qualified.

5.09. Ensure that there is a fair agreement concerning ownership of any software, processes, research, writing, or other intellectual property to which a software engineer has contributed.

5.10. Provide for due process in hearing charges of violation of an employer's policy or of this Code.

5.11. Not ask a software engineer to do anything inconsistent with this Code.

5.12. Not punish anyone for expressing ethical concerns about a project.

Principle 6: PROFESSION

Software engineers shall advance the integrity and reputation of the profession consistent with the public interest. In particular, software engineers shall, as appropriate:
6.01. Help develop an organizational environment favorable to acting ethically.

6.02. Promote public knowledge of software engineering.

6.03. Extend software engineering knowledge by appropriate participation in professional organizations, meetings and publications.

6.04. Support, as members of a profession, other software engineers striving to follow this Code.

6.05. Not promote their own interest at the expense of the profession, client or employer.

6.06. Obey all laws governing their work, unless, in exceptional circumstances, such compliance is inconsistent with the public interest.

6.07. Be accurate in stating the characteristics of software on which they work, avoiding not only false claims but also claims that might reasonably be supposed to be speculative, vacuous, deceptive, misleading, or doubtful.

6.08. Take responsibility for detecting, correcting, and reporting errors in software and associated documents on which they work.

6.09. Ensure that clients, employers, and supervisors know of the software engineer's commitment to this Code of ethics, and the subsequent ramifications of such commitment.

6.10. Avoid associations with businesses and organizations which are in conflict with this code.

6.11. Recognize that violations of this Code are inconsistent with being a professional software engineer.

6.12. Express concerns to the people involved when significant violations of this Code are detected unless this is impossible, counter-productive, or dangerous.

6.13. Report significant violations of this Code to appropriate authorities when it is clear that consultation with people involved in these significant violations is impossible, counter-productive or dangerous.

Principle 7: COLLEAGUES

Software engineers shall be fair to and supportive of their colleagues. In particular, software engineers shall, as appropriate:

7.01. Encourage colleagues to adhere to this Code.

7.02. Assist colleagues in professional development.

7.03. Credit fully the work of others and refrain from taking undue credit.

7.04. Review the work of others in an objective, candid, and properly-documented way.

7.05. Give a fair hearing to the opinions, concerns, or complaints of a colleague.

7.06. Assist colleagues in being fully aware of current standard work practices including policies and procedures for protecting passwords, files and other confidential information, and security measures in general.

7.07. Not unfairly intervene in the career of any colleague; however, concern for the employer, the client or public interest may compel software engineers, in good faith, to question the
competence of a colleague.

7.08. In situations outside of their own areas of competence, call upon the opinions of other professionals who have competence in that area.

Principle 8: SELF

Software engineers shall participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession. In particular, software engineers shall continually endeavor to:

8.01. Further their knowledge of developments in the analysis, specification, design, development, maintenance and testing of software and related documents, together with the management of the development process.

8.02. Improve their ability to create safe, reliable, and useful quality software at reasonable cost and within a reasonable time.

8.03. Improve their ability to produce accurate, informative, and well-written documentation.

8.04. Improve their understanding of the software and related documents on which they work and of the environment in which they will be used.

8.05. Improve their knowledge of relevant standards and the law governing the software and related documents on which they work.

8.06 Improve their knowledge of this Code, its interpretation, and its application to their work.

8.07 Not give unfair treatment to anyone because of any irrelevant prejudices.

8.08. Not influence others to undertake any action that involves a breach of this Code.

8.09. Recognize that personal violations of this Code are inconsistent with being a professional software engineer.

This Code was developed by the ACM/IEEE-CS joint task force on Software Engineering Ethics and Professional Practices (SEEPP):

Executive Committee: Donald Gotterbarn (Chair), Keith Miller and Simon Rogerson;


This Code may be published without permission as long as it is not changed in any way and it carries the copyright notice. Copyright (c) 1999 by the Association for Computing Machinery, Inc. and the Institute for Electrical and Electronics Engineers, Inc.