Lecture Notes in
The Philosophy of Computer Science

Matti Tedre

Department of Computer Science and Statistics, University of Joensuu, Finland

This .pdf file was created February 26, 2007.
Available at cs.joensuu.fi/~mmeri/teaching/2007/philcs/

3.2 Is Computer Science an Engineering Discipline?

It can be argued that the origins of computer science lie equally strongly in engineering as they lie in mathematics or natural sciences. Many pioneers of modern computing, such as John Atanasoff and John Presper Eckert, were electrical engineers. One of the early computing pioneers, Vannevar Bush, explicitly exclaimed, "*I'm no scientist, I'm an engineer*" (Kevles, 1987:293). In the early days of modern computing it was indeed vague whether computing was a theoretical subject like mathematics or a practical, design-oriented subject like engineering. Still today many computer science departments are together with the schools of electrical engineering in which they were born (e.g., MIT and University of California at Berkeley). But there has been quite some resistance towards linking computer science with engineering: especially around the 1960s and 1970s there was a strong movement to distance computer science from engineering and to liken it with mathematics or natural sciences.

3.2.1 The Field of Engineering

The argument that computer science is an engineering discipline relies on the view that the goal of computer science is to design and construct useful things (cf. Wegner, 1976; Loui, 1995). The themes of engineering are indeed clearly visible in the construction of computer systems. However, the term *engineering* in computing disciplines is a broad and ambiguous term: it can be considered to encompass things such as requirements engineering, software development, interface design, and computer engineering; things such as robotics, operating systems, and signal processing; and things such as software / hardware testing, maintenance, and project management.

Sometimes the difference between computer science and computer engineering is drawn following the "physical vs. nonphysical" lines (cf. Arden, 1980:7). In other words, computer engineers work with physical things (like hardware or machinery), whereas computer scientists work with abstract things (like algorithms and programs). But that distinction is vague and difficult. Firstly, programs have the dual nature (physical-abstract) that was discussed earlier (see Section 3.1.2 of these lecture notes). And secondly, establishing a strict line between *hardware* and *software* is also difficult (see page 76 of these lecture notes).

There is a diversity of descriptions of the engineering profession and the engineering philosophy, but most of those share some similar features. For instance, they commonly share the view that unlike mathematicians, engineers, who design working computer systems, have to cater to material resources, human constraints, and laws of nature. Engineers design complex, cost-effective systems with minimal resource consumption. Unlike natural scientists who deal with naturally occurring phenomena, engineers deal with artifacts, which are created by people. What seems to be common to all the different engineering branches is that they all aim at *producing things*. Carl Mitcham, who is a prominent philosopher of technology[16], wrote:

16 For those who are interested in the philosophy of technology there are a number of good anthologies available. See, e.g., Scharff & Dusek, 2003; Kaplan, 2004. For those who are fluent in Finnish, Timo Airaksinen's texts offer great insight into the philosophy of technology. See, e.g., Airaksinen, 2003.

> *Engineering as a profession is identified with the systematic knowledge of how to design useful artifacts or processes, a discipline that (as the standard engineering educational curriculum illustrates) includes some pure science and mathematics, the "applied" or "engineering sciences" (e.g., strength of materials, thermodynamics, electronics), and is directed toward some social need or desire. But while engineering involves a relationship to these other elements, artifact design is what constitutes the essence of engineering, because it is design that establishes and orders the unique engineering framework that integrates other elements.*
>
> (Mitcham, 1994:146-147)

Similar to Mitcham, who associated engineering with artifact design, Peter Wegner wrote that the *aim* of engineering differs from mathematics and science:

> *Research in engineering is directed towards the efficient accomplishment of specific tasks and towards the development of tools that will enable classes of tasks to be accomplishment more efficiently.*
>
> (Wegner, 1976)

It might be warranted to argue that researcher's interests are a necessary condition of activity A being science or engineering (that is, to argue that an activity cannot be considered to be engineering unless the researcher's aim or goal is to produce useful things). But certainly the researcher's interests (goals or aims) are not a sufficient condition of A being science or engineering. In most accounts of engineering, not all activities that aim at building things are considered to be engineering. And in most accounts of science not all activities that aim at explaining the world are considered to be science. Most accounts of science and engineering impose many necessary conditions, such as how the work is done (methodology) or what the outcomes of the work are (e.g., theories or products).

Many modern philosophers and sociologists have argued that the old idea that "technology is applied science" is no longer true (if it ever was). Many argue that the inverse direction might be even stronger: most of the progress in modern science could be attributed to technological development (see Hacking, 1983[17] for an argument against an undue emphasis on theories). In the *idealist attitude* towards technology, technology is considered to be applied science, whereas in the *materialist attitude* towards technology, science is considered to be theoretical technology (Mitcham, 1994:76).

For example, astronomy took giant leaps after the invention of the telescope. The theoretical progress in particle physics is inextricably linked with the development of instruments such as different kinds of particle detectors and particle accelerators (cf. Mitcham, 1994:204; Pickering, 1995). Some authors, such as Donna Haraway, have even begun to use the term *technoscience* instead of *science* and *technology* because they believe that the two have become inseparable (Haraway, 1999; MacKenzie & Wajcman, 1999; the idea can be seen already in the Marxist term *scientific-technological revolution*; see Mitcham, 1994:84).

17 Reprinted in Kaplan, 2004:435-447

Mitcham noted that technology can be seen as a new cognitive method for science, whereas science can be seen to offer new principles for technology (Mitcham, 1994:86). But although science and engineering share some similarities—for instance, they both have to conform to the laws of nature, they both are cumulative, and they both share the scaling problem[18]—they still utilize and produce different kinds of knowledge and employ different methodologies. According to Mitcham, whereas scientific knowledge consists of a set of observations, laws, and theories; technological knowledge consists of actions, rules, and theories (Mitcham, 1994:193-194,197).

Mitcham divided technological knowledge into four kinds of knowledge: (1) *sensorimotoric skills* of making ("know-how"), (2) *technical maxims* ("rules of thumb" or "recipes", which offer heuristic strategies for successfully completing tasks), (3) *descriptive laws* (that is, "If A then B"-kind of rules, based on experience—yet those rules do not go into explaining *why* A and B seem to be connected in some way), and (4) *technological theories* (applications of scientific theories to practice; e.g., the theory of flight is an application of fluid dynamics).

Also the meaning and status of knowledge differ between science and engineering. Scientific knowledge is about description, explanation, prediction, and understanding of natural (or artificial) phenomena, and scientists are concerned of whether their knowledge is *true*. Technological, engineering knowledge is about heuristic prescriptions (best practices) of how things should be done, and engineers are concerned of whether their knowledge *works* (cf. Mitcham, 1994:197). Engineers have to often work relying on information that scientists would not consider adequate for scientific purposes (cf. Vincenti, 1990).

Denning et al. wrote that engineers share the *methodological* notion that progress is achieved primarily by posing problems and systematically following the *design process* to construct systems that solve them (Denning et al., 1989). The engineering method, as seen by Denning et al., is a cycle that consists of defining requirements, defining specifications, designing and implementing, and testing. In their work, engineers often follow the method of *parameter variation*—that is, they repeatedly measure the performance of a device or process, while they systematically adjust the parameters of the device or its conditions of operation (Vincenti, 1990:139). This method surely seems like an all-time favorite of quite a few computer scientists.

Peter Wegner divided the engineering part of computer science into two parts; *practical engineering* and *research-based engineering* (Wegner, 1976). He wrote that the problem-solving paradigm of practicing engineers involves systematical selections between design decisions, selections which progressively narrow down alternative options for accomplishing the task, and finally lead to a unique realization of the task. According to Wegner, research engineers may use mathematics and physics when they develop the tools for practicing engineers, yet research engineers are much more concerned with the practical implications of their work than empirical scientists and mathematicians are.

Philosopher of computing Timothy R. Colburn sketched another flavor of the engineering approach in computer science in form of *solution engineering* (Colburn, 2000:167). In some branches of computer science the usual scenario includes rigorous requirements, and the task of the computer

18 Scaling problem refers to the phenomenon that how things work in small scale or in small quantities might be different from how things work in large scale or in very large quantities (see Vincenti, 1990:134,139).

scientist is to engineer an algorithmic solution. Colburn portrayed an analogy between the scientific method and the problem-solving approach in computer science (Table 1).

The scientific method	***Problem-solving in computer science***
1. Formulate a *hypothesis* for explaining a phenomenon 2. *Test* the hypothesis by conducting an *experiment* 3. *Confirm* or *disconfirm* the hypothesis by evaluating the results of the experiment	1. Formulate an *algorithm* for solving a problem 2. *Test* the algorithm by writing and running a *program* 3. *Accept* or *reject* the algorithm by evaluating the results of running the program

Table 2: Analogy Between the Scientific Method and Problem-Solving in Computer Science (near-verbatim from Colburn, 2000:168)

In Colburn's analogy above, what is being tested in "the scientific method" is not the experiment, but the hypothesis. The experiment is a tool for testing the hypothesis. Similar, what is being tested in problem-solving in computer science is not the program, but the algorithm. The program is written in order to test the algorithm. In this analogy, writing a program is analogous to con – structing a test situation. (Interestingly, also Khalil and Levy made a similar analogy, yet in a dif – ferent context: they wrote, "programming is to computer science what the laboratory is to the phys – ical sciences"; Khalil & Levy, 1978). Although Colburn noted that his analogy does not hold very far, that analogy displays another view of problem-solving, which is indeed a characteristic of com – puting disciplines.

3.2.2 Proponents of the Engineering Argument

Frederick P. Brooks Jr. is one of the outspoken proponents of an engineering view of computer sci – ence. In his ACM Allen Newell Award lecture, titled *Computer Scientist as a Toolsmith II* (Brooks, 1996), Brooks argued that although scientists and engineers both may spend most of their time building and refining their apparatus, the distinction between a scientist and an engineer is that the *scientist builds in order to study*, and the *engineer studies in order to build*. In Brooks' opinion, computer scientists are engineers: computer scientists study in order to build. He wrote that com – puter science is very different from natural sciences—computer science is a *synthetic*, engineering discipline. According to Brooks, science is concerned with the *discovery* of facts and laws (take the terms *fact* and *law* with some qualifications here), whereas engineering is concerned with *making* things, be they computers, algorithms, or software systems. Brooks argued that computer science is exactly about *making* things and not about *discovering* things.

Juris Hartmanis argued that generally speaking, computer science is concentrating more on the *how* than the *what* (Hartmanis, 1993). He wrote that natural sciences concentrate more on questions of *what*, and that computer science, with its bias on *how*, reveals its engineering concerns and consid – erations. Hartmanis wrote that whereas the advancements of natural sciences are often documented by dramatic *experiments*, in computer science the advancements are often documented by dramatic *demonstrations*. In some branches of computer science the scientists' slogan "publish or perish" in – deed might have turned into the engineers' slogan "demo or die". Hartmanis argued that computer science is the "*engineering of mathematics*". Even more interesting is Hartmanis' argument that whereas the physical sciences are focused on *what exists*, computer science is focused on *what can exist* (Hartmanis, 1981 in Traub, 1981).

The field of *systems engineering* began developing around and after the second world war when new technical systems grew so complex that their design and management could not have been done by single individuals anymore. In computer science, a more specific term *software engineering* was first introduced in 1968 at a conference held to discuss the software crisis (Naur, 1969). Although the status of software engineering as a part of computer science and as an intellectually respectable endeavor altogether was disputed in its beginning at the 1970s and 1980s, nowadays software engineering is largely considered to be a legitimate part of academic computer science. Software engineering definitely emphasizes the engineering side of computer science—production of things, the operationality and usability of those things, maintenance of things, time frames, budgets, project management, and so forth.

A side note: It was mentioned earlier that one of the main activities of engineering is to compare solutions and select alternatives. In engineering, those comparisons are often made in terms of costs and efficiency. Interestingly, a lot of theoretical computer science, which one might consider to be one of the least engineering oriented branches of computer science, is focused on the cost and efficiency of algorithms (the costs are expressed in resources such as time and storage) (cf. Arden, 1980:7). A hasty reader might note that emphasizing optimization of resources is an engineering concern, and that the cost/efficiency concerns reveal an engineering strand of theoretical computer science. But quite some theoretical computer scientists might object and argue that the *goal* of theoretical computer science is not to produce useful, cost-effective things, but to understand properties of algorithms—properties such as cost and efficiency. (This takes the discussion back to the question of whether researcher's interests can actually define if something is engineering or science.)

Whereas the proponent of a mathematical view of computer science could argue that computer science as we know it today would not exist without the work of Church, Gödel, or Turing; the proponent of an engineering view of computer science could argue that without engineers computer science would have no consequences outside the academia, that without engineers computer science would still be a compartment of mathematics, or that without engineers computer science would be just idle speculation. Indeed, many of the turning points in the history of computing come from technological breakthroughs, not only theoretical breakthroughs.

For instance, John Mauchly, who was one of the designers of ENIAC, noted that it was not the *idea* or *theory* of ENIAC but the *construction* of ENIAC that convinced many institutions and people—scientists, military officials, and industrialists alike—to commit to the rapid development of electronic computing (Mauchly, 1979). Similar, the development of high-level programming languages, such as FORTRAN, was not a theory-driven move, but it was a response to practical and economic issues (Campbell-Kelly & Aspray, 2004:168; Backus, 1981:26-27). What is more, those technological breakthroughs were made despite the opposition of the academic establishment. It can be argued that many of the most visible developments of computer science have been technology-driven instead of theory-driven.

3.2.3 Opposition to the Engineering Argument

Generally the opposition to the engineering approach to computer science is not so much that engineering were considered to be unimportant. Usually the opposition is against the inclusion of engin-

eering-like activities and aims under the umbrella of *academic* computer science. But although computer science and computer engineering could be considered to be separate disciplines, the dividing lines are vague. There are plenty of examples of computing topics that are difficult to be classified as either computer science or computer engineering; topics such as software engineering, computer architecture, parallel computing, embedded systems, and many programming-intensive topics. In addition, the vagueness of the software/hardware distinction makes it hard to draw the line between science and engineering by distinguishing between non-physical and physical subjects in computer science.

> SOFTWARE AND HARDWARE
>
> Those parts of the computer system that you can touch are often considered to be *hardware*, and respectively, *software* is often considered to be the "non-physical" parts of a computer system. This line is, however, vague. Firstly, programs (when stored as electrical charges in memory, or as blips on a magnetic disc) *are* physical phenomena. More importantly, most hardware can be implemented as software and most software can be implemented as hardware. For instance, codecs are sometimes implemented as hardware, sometimes as software.
>
> A hardware-software distinction is a *pragmatic* distinction, and it is a *subjective* distinction (Moor, 1978). For the user of a microwave oven, the whole thing is hardware. But for the engineer of a microwave oven there is often software and hardware. For the systems programmer, circuitry is hardware, but a circuit designer can see microprograms as software. A graphics programmer may not even known which parts of his or her program are going to be hardware-accelerated and which run on software.

There have certainly been also opponents of the engineering view in general. Edsger Dijkstra was one of the most free-spoken opponents of the engineering view of computer science. Dijkstra opposed the inclusion of software engineering under the umbrella of academic computer science and he opposed also the methods of software engineering. He wrote that software engineering, "*The Doomed Discipline*", had accepted as its charter, "*how to program if you cannot*" (Dijkstra, 1989). Dijkstra wrote that comput*ing* scientists should not bother to make programs, but they should focus on designing classes of computations that display desired behaviors (Dijkstra, 1972).

Dijkstra not only opposed software engineering, but he opposed the connection of computing science with *any kind* of specific technological solutions. In 1987 he wrote that the "incoherent bunch of disciplines" that began computer science, hardly appealed to the "intellectually-discerning palate" of mathematicians. Computing science, in Dijkstra's opinion, is about what is common to the use of any computer in any application, and computing scientists should not be concerned with any technological details or any societal aspects of their discipline (Dijkstra, 1987). In his argument against the technological bent of computer science, Edsger Dijkstra argued that comput*er* science is an entirely wrong term: "*Primarily in the U.S., the topic became prematurely known as 'computer science'—which actually is like referring to surgery as 'knife science'*." (Dijkstra, 1987).

The opponents of the engineering view of computer science also argue that if computer science is a *science*, and if scientists adhere to the scientific method but engineers do not, then engineering is not a proper part of computer science. The opponents argue that it is difficult to see the theoretical foundations of, for instance, software engineering, and that the engineering parts of computer sci – ence are based on rules of thumb. That is, they argue that there is nothing *scientific* in engineering.

Especially the accusations of a lack of rigor in software engineering have arisen debates of the aca – demic image of software engineering. For instance, C. Michael Holloway accused software engin – eers of basing their work on a combination of anecdotal evidence and human authority (Holloway, 1995). In their study of 600 published articles on software engineering, Marvin Zelkowitz and Dolores Wallace found that about one third of articles failed to experimentally validate their results (Zelkowitz & Wallace, 1997; Zelkowitz & Wallace, 1998; see also Tichy, 1998).

The usual argument against the engineering view of computer science is not that much about the un – importance of engineering. It is widely accepted that producing useful and working computational tools is a well-justified aim that is societally and intellectually important. Instead, the opponents ask whether engineering can contribute anything to the *common knowledge about computing* and whether engineering should be considered to be a part of the *academic* discipline of computing. Not all important activities need to be nominated as academic disciplines.