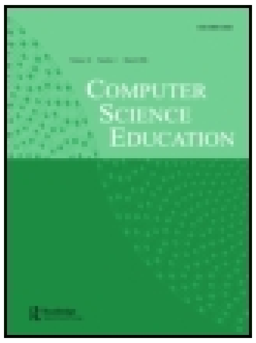

Computer Science Education

ISSN: 0899-3408 (Print) 1744-5175 (Online) Journal homepage: https://www.tandfonline.com/loi/ncse20

Three traditions of computing: what educators should know

Matti Tedre & Erkki Sutinen

To cite this article: Matti Tedre & Erkki Sutinen (2008) Three traditions of computing: what educators should know, Computer Science Education, 18:3, 153-170, DOI: 10.1080/08993400802332332

To link to this article: https://doi.org/10.1080/08993400802332332

Published online: 15 Sep 2008.

Computer Science Education
Vol. 18, No. 3, September 2008, 153–170

Three traditions of computing: what educators should know

Matti Tedre[a]* and Erkki Sutinen[b]

[a]*B.Sc Program in Information Technology, Tumaini University, Iringa, Tanzania;* [b]*Department of Computer Science and Statistics, University of Joensuu, Joensuu, Finland*

Educators in the computing fields are often familiar with the characterization of computing as a combination of theoretical, scientific, and engineering traditions. That distinction is often used to guide the work and disciplinary self-identity of computing professionals. But the distinction is, by no means, an easy one. The three traditions of computing are based on different principles, they have different aims, they employ different methods, and their products are very different. Educators in the field of computing should be aware of the fundamental differences between the traditions of computing so that they can offer their students a truthful and balanced view about computing branches. In this article the three traditions of computing are presented and some of their underlying assumptions, principles, application areas, restrictions, and weaknesses are portrayed. Also, some of the landmark arguments in the debates about the identity of computing disciplines are discussed.

Keywords: philosophy of computer science; traditions of computing; paradigms of computing; computer science education; computer engineering education; information technology education; philosophy

Introduction

One of the most famous characterizations of computing as a discipline is the report *Computing as a discipline* by the Task Force on the Core of Computer Science headed by Peter J. Denning (Denning et al., 1989). In that article the task force characterized the discipline of computing as a combination of three separate but tightly intertwined aspects: theory, abstraction (modeling), and design. Those aspects rely on three different intellectual traditions (the task force called them paradigms): the mathematical (or analytical, theoretical, or formalist) tradition, the scientific (or empirical) tradition, and the engineering (or technological) tradition. That tripartite was by no means a new insight in the field of computing (see, for example, Wegner, 1976), or a unique thing in sciences in general – it can easily be argued that, say, quantum mechanics is also a combination of mathematical, scientific, and engineering aspects. Nonetheless, even today, almost 20 years since the publication of the Denning report, the tripartite above continues to structure the way many computing professionals think about their discipline.

Although the Denning report is mentioned in such high visibility publications as ACM/IEEE's *Computing curricula 2001*, the significance and implications of the tripartite

*Corresponding author. Email: matti.tedre@acm.org

ISSN 0899-3408 print/ISSN 1744-5175 online
© 2008 Taylor & Francis
DOI: 10.1080/08993400802332332
http://www.informaworld.com

above easily go unheeded. At first sight the tripartite may appear like a general description of computing fields, intended to inform computing professionals and students about their disciplinary identity. However, the consequences of that tripartite go deep into the foundations and methodology of work in computing fields. Students in computing disciplines should get a good overview of the terrain of computing. But for that aim, educators must know the landmarks of different computing branches. Students in computing fields need to be given a tour led by educators who know the terrain and the landmarks. Viewing work and research in computing from different perspectives offers the students insight about their field and offers the educators room for new teaching variations and new meaning. Perhaps that could help the students to see the field not as myopic and narrow – "hacking for hacking's sake" (Fisher & Margolis, 2002) – but in all its theoretical, practical, scientific, and philosophical richness.

Most importantly, far from being a trivial description, the above mentioned tripartite of computing disciplines has profound intellectual and practical ramifications that students should be aware of. The three traditions of computing – the mathematical, scientific, and engineering traditions – all provide for unique possibilities in computing but they also have their restrictions. Those traditions entail different epistemological, ontological, and methodological assumptions (Eden, 2007), which also underlie their different aims. For instance, the logico-mathematical tradition and the engineering tradition entail different ontological views about their subjects of study, the empirical tradition and the theoretical tradition employ different methodologies, and the engineering tradition and the empirical tradition have different views about the epistemological status of their research results. It is notoriously difficult to conduct research in the intersection of research traditions without making a mess of it (see, for example, Denzin & Lincoln, 1994, pp. 2–3 & 99–100). And it is a matter of intellectual integrity to not trivialize the matter.

In this article we take a look at each of the three traditions. We briefly present each tradition, some of their underlying assumptions, some of their application areas, some of their restrictions, and some of their weaknesses. We refrain from giving undue emphasis to any of the traditions and we attempt to portray them in a neutral manner. We emphasize the knowledge that computing educators should be aware of, and we try to stick to common computing terminology where possible.

The mathematical tradition

Beginning in Ancient Greece there has been a tight connection between mathematics and other academic disciplines; it has been argued that above the entrance to Plato's academy there was a sign that read "Let none ignorant of geometry enter here." In a similar manner, it has been argued that mathematics is the quintessential knowledge and skill for a computing professional (cf. Davis, 1977; Ralston & Shaw, 1980). Hartmanis (1993) argued that a mathematical reductionist could say, somewhat facetiously, that the discipline of computing is nothing but a paradigm change in mathematics. Denning et al. (1989) wrote "Theory is the bedrock of the mathematical sciences: applied mathematicians share the notion that science advances only on a foundation of sound mathematics." Formal and mathematical methods have indeed proven very successful for a great number of theoretical and practical purposes (Bowen & Hinchey, 2005; Hamming, 1980).

At the turn of the 1970s C.A.R. Hoare was one of the strong proponents of a view that computer science (including programming) is fully reducible to mathematics (Hoare, 1969). He argued that computers are mathematical machines, computer programs are mathematical expressions, programming languages are mathematical theories, and programming is a mathematical activity (Hoare, 1969). In Hoare's view knowledge about programs is a priori knowledge – that is, testing is not necessary to gain knowledge about programs. Although in the 1970s authorities such as Edsger Dijkstra (1972), Niklaus Wirth (1971), and Peter Naur (1966, 1969, 1972) promoted views according to which computing is the study of certain kinds of mathematical expressions – namely algorithms – today it is not commonplace to reduce computing as a discipline to mathematics.

Generally speaking, researchers in mathematically oriented disciplines aim at coherent theoretical structures, and they arrive at their aims by creating and proving hypotheses and theorems. Such an analytical approach does not necessitate the use of empirical studies to test hypotheses, but claims are proven formally. However, the unfortunate phenomenon that researchers occasionally take the analytical research tradition outside its conventional logico-mathematical and philosophical domains has caused some computing authors to argue that the analytical research approach is "seriously flawed" and "alarming" (see, for example, Glass, 1995; Tichy, Lukowicz, Prechelt, & Heinz, 1995). Of the many debates around various branches of the mathematical tradition, computing educators should be aware of at least the formal verification debate.

The formal verification debate was one of the characterizing features of the 1970s and 1980s in computing disciplines. The debate was between a number of hardline formal verificationists and an eclectic bunch of critics of the verificationist position. Early proponents of formal verification argued that formal verification can, in effect, be used to prove that a computer system works correctly. For a long time the proponents of engineering-oriented and empirically-oriented branches of computing had no credible counter-argument, but in the 1980s things changed. Broadly speaking, there were three different objections to hardline verificationist positions. There was the objection that proofs are products of social processes (De Millo, Lipton, & Perlis, 1979), the objection that the physical world, vis-à-vis mathematical abstractions, is uncertain (Fetzer, 1988), and the objection that there is an insurmountable gap between models and the world (Smith, 1985/1996). Those objections are outlined below – for more detailed summaries of the formal verification debate see, for example, Colburn (2000) and Tedre (2007a, 2007b).

The first notable critique of formal verification was put forth by De Millo et al. (1979). They noted that proofs in mathematics are products of social processes and that the proofs that formal verification produces are very different in that respect. In mathematics proofs go through painstaking cycles of peer reviews, editorial processes, scrutiny by colleagues and the mathematical community, and so forth – and proofs can be rewritten, refuted, or reformulated at any stage (Lakatos, 1976). Contrary to that, De Millo et al. (1979) argued that in program verification proofs of program correctness do not create or undergo similar social processes because proofs of program correctness have nothing exciting in them.

De Millo et al. (1979) noted that formal proofs of program correctness are long, complex, and peculiar; they are just too cumbersome and boring to read. Proofs of program correctness fill up thick piles of printouts. De Millo et al. contrasted the joy of mathematical achievement with boring program proofs, writing:

> The verification of even a puny program can run into dozens of pages, and there's not a light moment or a spark of wit on any of those pages. Nobody is going to run into a friend's office with a program verification. Nobody is going to sketch a verification out on a paper napkin. Nobody is going to buttonhole a colleague into listening to a verification. Nobody is ever going to read it. One can feel one's eyes glaze over at the very thought. (De Millo et al., 1979)

The second notable critique of formal verification came from professional philosopher James H. Fetzer (1988), and at the time of its publication the critique raised fervent opposition (see, for example, Ardis et al., 1989). Fetzer noted that mathematicians need not care about the uncertainties of the physical world, but those uncertainties cannot be completely abstracted away from real computers. The crux of the matter is that one cannot establish the running of a computer with mathematical certainty. Simply put, a program run on a real computer can be disrupted by Tony stumbling on the computer's power cord, by a cosmic ray interfering with the circuitry, or by too few electrons being at the right place at the right time. Computers, the machines, are physical objects and although one could prove computer blueprints to be theoretically correct, the physical world does not work with mathematical certainty. One cannot prove how single electrons in computer circuits will behave (although one can make very good predictions of how very large clouds of electrons behave).

The third notable critique of proofs of program correctness, albeit less well known than the previous two, came from the philosopher of computing Brian Cantwell Smith (1985/1996). Smith noted that formal proofs always take place between two formal systems – the program and its formal specification – but that there is a fundamental gap between formal systems and the world. On the one hand, it might be possible to verify a program that is intended for calculating values of a mathematical function. Such a program is correct, in some sense of the term correct, if the program output always corresponds to the results of the mathematical function with the same input. Indeed, if one can rigorously and formally define what the program is intended to do, then one can proceed to prove that the program text is correct (yet cannot prove that the program will always run correctly).

However, most computer programs defy exact, formal specifications. Most characteristically, many programs have some kind of a relationship with the physical world. For instance, some programs model phenomena, some programs measure physical phenomena, some programs are tools for work, and some are for entertainment – and some programs are parts of larger causal systems that affect the physical world to a great extent, like those programs that control nuclear plants and land aeroplanes.

The problem is the definition of what the term correct means with programs. Is a program correct when it does exactly what it was instructed to do? That might not do as the definition, as bug-free programs and erroneous programs alike work exactly as they were instructed to do. Alternatively, is a program correct when it works exactly according to its model (of, say, landing an aeroplane)? That might not be a good definition either, since in that case the correctness of the program may not have anything to do with how well the program actually lands planes (the models can be broken). Alternatively, is a program correct when it frequently succeeds in its task (like landing an aeroplane)? In that case, correctness is a relative concept – any program can be correct in some situations (a proper term there might be efficient, not correct). Alternatively, is a program correct when its execution corresponds exactly to its specifications? But on what level of abstraction should the specifications be? Should they be of the kind "Land the plane safely" or of the kind "If the reading from sensor 22 exceeds 255, increase thrust by 30"? Associating program correctness with the relationship between program code and program specifications just shifts the burden of correctness proof to another formal system. That is to say, then one should prove that the program specifications are correct.

Modern proponents of formal verification do not go to the same lengths that Hoare, Dijkstra, and other early proponents of formal verification went. Today, formal verification is a view of computer program construction that begins with a well-defined formal program specification, and the computer program is constructed so that it corresponds to that specification. Formal verification can, according to modern proponents of formal verification, be used to prove that the program code corresponds to the program specification. If one does not formally verify the correctness of a program, even a thorough, exhaustive testing cannot prove that the program always works correctly (not even in principle, because a buggy program can coincidentally work correctly once but fail when the program is run again).

Many early proponents of formal verification failed to recognize the fundamental difference between programs as abstract things and programs as executable implementations on a computer – and the fundamental difference between pure mathematics and applied (numerical) mathematics. Howevert, the failure to formally verify causal systems (or the workings of actual computers) may not be a very serious weakness of a mathematical approach to computing. Most proponents of formal methods – who, generally speaking, recognize the limited applicability of formal verification – still have a strong argument for a mathematical view of computing (see, for example, Bowen & Hinchey, 2005). And, furthermore, there are large branches of computing where researchers are not concerned with executing programs but with, for instance, analytically studying properties of computations.

In summary, the term proof is problematic outside axiomatic systems and should not be used lightly (especially it should not be mixed with testing terminology). Formal methods and analytical approaches are very powerful in their proper domains, but their power weakens outside axiomatic systems. Whenever computers as physical machinery are in the picture pure mathematics turns out to be inadequate, and some other intellectual frameworks must be utilized. In the mathematical/theoretical tradition of computing, as in all mathematical fields (cf. Pólya, 1957, p. 117), actual work is often inductive and empirical, although the results are reported in a deductive form. Generally speaking, one should appreciate the fact that formal proofs play an important role in most fields of computing yet remember the impossibility of proving facts in science in general (see, for example, Chalmers, 1999; Couvalis, 1997; Rosenberg, 2000).

The scientific tradition

In the debates about the soul of computing disciplines the scientific nature of computing has been most controversial (see, for example, Forsythe, 1967, 1969; Newell, Perlis, & Simon, 1967; Rosenbloom, 2004; for an overview see Tedre, 2007a). The question is often about whether computing is a science in the same sense that natural sciences are. Sometimes it is argued that computing is indeed a proper natural science in the sense that it studies naturally (but also artificially) occurring information processes (Denning, 2005, 2007). Some have even hinted that the study of information processes will qualify as a theory of everything (see, for example, Lloyd, 2007; Seife, 2006; Wolfram, 2002). Donald Knuth (2001, p. 167) called computer science an unnatural science and Herbert A. Simon (1981) called it an artificial science. Most often the pro-science argument is that although computing is not a natural science, it is still an empirical science or experimental science, because researchers in computing follow the scientific method (they explore and observe phenomena, form hypotheses, and empirically test those hypotheses).

When considering whether computing is a science or not one should ask what is it a science of (see, for example, Rapaport, 2005). The spectrum of views about the subject matter of computing disciplines range from computers and related phenomena (Newell et al., 1967) to algorithms and related phenomena (Knuth, 1974a) to complexity (cf. Minsky, 1979; Simon,1981, p. 64) to information representation and processing (Denning et al., 1981; Forsythe, 1967; Hartmanis, 1994) to procedures (Shapiro, 2001, cited in Rapaport, 2005) to computations (Dijkstra, 1972) to programming (Khalil & Levy, 1978). The spectrum of views about the activities in computing disciplines range from representing and processing (Forsythe, 1967) to designing (Denning et al., 1989; Dijkstra, 1972) to mastering complexity (Dijkstra, 1974) to formulating (Dijkstra, 1974) to programming (Khalil & Levy, 1978) to empirical research (Wegner, 1976) to modeling (Denning et al., 1989) and beyond. Computing fields have been conceptualized as mathematics, as engineering and design, as art, as science, as social science, and as an interdisciplinary endeavor (Goldweber et al., 1997). Some of the above-mentioned views are indeed compatible with the view that computing is a science proper.

When Denning et al. (1989) discussed the scientific nature of computing disciplines, they wrote "Abstraction (modeling) is the bedrock of the natural sciences: Scientists share the notion that scientific progress is achieved primarily by formulating hypotheses and systematically following the modeling process to verify and validate them." However, how scientists should, generally speaking, work is not an easy question at all. The first problem is, of course, that there is no consensus of what the term science means. Science can be considered to be a class of specific activities, a sociocultural and historical phenomenon, a kind of knowledge, a societal institution, a world view, a specific style of thinking and acting, or a profession (Tedre, 2007a). Some people have proposed idealized versions of science (see, for example, Ayer, 1936/1971; Hempel, 1965; Popper, 1935/1959), whereas some have argued that scientists do not actually work according to any rigid guidelines, and that there neither is nor should be a fixed set of rules suitable for all branches of science (Feyerabend, 1975/1993). If one considers computing to be a science, there are a plethora of things that one should know about science.

Firstly, one of the less contested ideas connected with science is that the aim of science is to describe, explain, and predict phenomena in the world (cf. Okasha, 2002, p. 1; von Wright, 1971, p. 6). One could argue that having those aims is a necessary condition of being science, yet by the same token many established intellectual fields are excluded – fields such as linguistics, archeology, history, and mathematics. Be that as it may, those three aims are certainly not a sufficient condition of being science. For instance, religions and astrology often aim at describing, explaining, and predicting phenomena, yet they are not considered to be sciences. Aims alone do not define science very well.

Quite a lot of people agree that striving for objectivity is a sine qua non of true science (see, for example, Couvalis, 1997). However, the term objective refers to different things in different contexts. For instance, one can talk about statements, knowledge, reality, methods, standards, or justification as being objective. In the context of scientific observations objectivity often refers to the idea that observations can be confirmed by anyone who carries out the same experimental procedure under similar conditions, i.e. observations in science should not be relative or open to interpretations. In the context of scientific theories objectivity often refers to the idea that all the logical consequences of a scientific theory can be tested regardless of time, place, or other contingent factors (cf. Kemeny, 1959, p. 96). For instance, any logical consequence of atomic theory can be tested by anyone, and those tests can either support or conflict with atomic theory. Few natural scientists would disagree that objectivity is a characteristic of science. Quite a few of them might, however, face a problem if they were asked to define what objectivity, strictly speaking, means.

Some philosophers have argued that the most useful way of defining science is by its method (Kemeny, 1959, p. 174). The scientific method is, however, not a method in any strictly defined manner, but a broad collection of principles and techniques. In its most basic sense the term scientific method refers to a cycle of observations, descriptions, predictions, and experiments that test those predictions (Figure 1).

In the cycle of phases that belong to the scientific method a scientist begins by observing a phenomenon, often instigated by a problem. When enough observational data has accumulated the scientist proceeds to describe the phenomenon by forming hypotheses about the phenomenon, possibly supported by other theories. Scientific hypotheses have logical consequences that can be tested, which allows one to make predictions of phenomena that have not yet come to pass. The next step in the cycle is to design experiments to test some predictions and to carry out those experiments. If the experimental results support the hypothesis, the scientist goes back to design more experiments to test the hypothesis. If the experimental results conflict with the hypothesis, the scientist goes back to rethink the hypothesis (or to think whether the anomalous results might have been caused by other things, such as instrumental flaws or flaws in the test setting). Note that in Figure 1 the applied, empirical (usually inductive) parts of the scientific method are on the left side of the picture and the theoretical, deductive parts are on the right side of the picture.

A similar picture could be drawn about the experimental or empirical tradition in computing: in many branches of computing researchers work by forming hypotheses, constructing models, making predictions of those models, designing experiments and collecting data, analyzing results, and iterating the cycle until the model is accurate enough (Denning et al., 1989). Another view is that computing professionals think hard about a problem, construct programs to solve that problem, run the program and collect data, analyze results, and repeat until the program seems to solve the problem.

The problem with the cycle in Figure 1 is that it is a simplified and idealized version of scientists' work. Many modern philosophers and sociologists of science argue that that is not how scientists actually work (Kuhn, 1962/1996; Pickering, 1995). Some argue that that is not how scientists should always work either (Feyerabend, 1975/1993). The cycle in Figure 1 is also unsuited for some well-established branches of science. Especially, experimentation that relies on the researcher's ability to freely manipulate the subject matter and to freely control the test setting does not seem to be a necessary condition of science. For instance, astronomers have little means of manipulating the subject matter they study; astronomers work on their tools and theories and they try to find a good fit between their observational data, their instruments, their theories, and their theories on

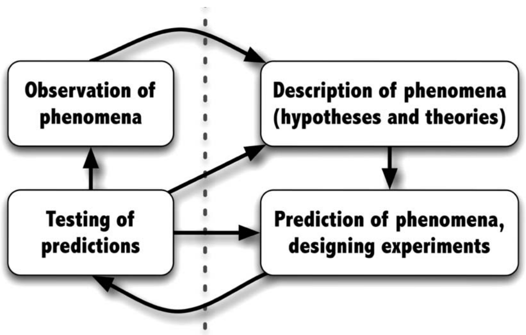
Observation of phenomena
Description of phenomena (hypotheses and theories)
Testing of predictions
Prediction of phenomena, designing experiments

Figure 1. Cycle of scientific research.

how their instruments work (cf. Pickering, 1995, about particle physics). Again, in social sciences it might be considered unethical to conduct experiments on society, and in psychology one cannot test just any hypothesis on the participants. Newell and Simon (1976) indeed noted the distinction between experimental science and empirical science and argued that, similar to astronomy, economics, and geology, computer science is an empirical science that does not rely solely on the traditional experimental method.

It should be noted that modern scientists usually do not claim that their explanations of the world will be unfailing. Generally speaking, modern scientists often claim that (i) their explanations are truer than any non-scientific model of the world, (ii) they are able to test such a truth claim, (iii) science as an institution is able to discover its own shortcomings, and (iv) science is able to correct its own shortcomings (Bunge, 1998, p. 33). The argument that science is self-correcting relies on a cycle of hypotheses and theories, predictions, experiments, corroborations, corrections, or refutations, and new hypotheses and theories. Furthermore, many scientists hold the view that scientists should actively seek to find flaws in scientific theories, try to unearth sources of bias in research settings, question all presuppositions, and so forth.

Finally, one of the very basic concepts of science, the concept of law, is not a straightforward one at all. Authors in the field of computing have used the term law in many different senses, often inconsistently. Frederick P. Brooks Jr (1996) wrote that although a new law is an accomplishment in science, computing researchers should not confuse any of their products with laws. Donald Knuth (1974a) wrote that computing researchers deal with 'man-made' laws. Peter J. Denning (1980) even equated the behavior of algorithms with laws of nature. And there are a number of rules of thumb that are nevertheless called laws, such as Moore's law, Rock's law, and Metcalfe's law (Ross, 2003).

Generally speaking, two of the most common meanings of the term law are (1) regular patterns of (causal) events and (2) descriptions of those regular patterns (see, for example, Bunge, 1998, pp. 391–392). Note the difference: in the former meaning laws are independent of whatever people may think about them – they obtain because that is how the world works; but in the latter meaning laws are descriptions that people have made to describe how the world works – descriptions that can be wrong. An example of the former might be the phenomenon that any two masses attract each other; an example of the latter might be Newton's law of universal gravitation $F = G[(m_1 m_2)/r^2]$. Often, such as in the case of gravity, the description of a regular pattern is called a law even if the actual cause of the phenomenon is unknown. However, many things that computing researchers call laws do not fall into either category, and it should indeed be asked whether computing disciplines deal with laws or theorems, both of them, or something else.

Objections to the scientific view

It would be controversial to say that computing could be a science in all the senses mentioned above. Firstly, of course, there is great uncertainty about what the subject matter of computing actually is, and about how computing professionals' work corresponds to the scientific method. But there are a number of other arguments against the scientific nature of computing. For instance, Peter Fletcher (1995) opposed an undue emphasis of the hypothetico-deductive model in computing fields and noted that without the theoretical principle of Turing equivalence of all computers (i.e. particular characteristics of computers are, in principle, irrelevant) there would be no academic discipline of computing, but just eclectic knowledge about particular machines.

He continued to argue that instead of seeking the best solutions to previously specified problems, often researchers in computing work with problems that are poorly understood and where one major goal is to find and understand problems, delimit them precisely, and convince others that those problems are worth solving (Fletcher, 1995). In other words, exploration and description seem to be valuable aims in computing.

The difference between research of naturally occurring phenomena and artificial, human-made phenomena has been the foremost target of criticism by those who argue that computer science is the wrong name. Juris Hartmanis (1993) noted that unlike in natural sciences, in the field of computing theories do not compete for their validity as explanations of phenomena. Indeed, observations about algorithm behavior, about usability of machinery and software, or about information retrieval are observations of things that researchers have constructed. Those observations do not tell us anything new about the world – those observations tell us only about how well previous computing professionals have done their job. This, some critics argue, is not the task of science. In that critique the term science, of course, refers strictly to studies of naturally occurring phenomena.

George McKee (1995) went on to argue that even if the difference between the subject matters of computing and traditional scientific fields is recognized, people in computing have different goals and methodologies than people in traditional sciences. He argued that in the natural sciences research is based on a wealth of observations (data), which scientists can explain, predict, and replicate. In the field of computing, McKee argued, there is no data beyond the computer and programs, which behave exactly as they were designed to behave. In a similar manner, Frederik Brooks (1996) argued that computing is not a science but a synthetic engineering discipline. However, neither McKee nor Brooks argued that computing as a discipline should change – they argued that the name science just does not reflect what actually happens in the field of computing.

Generally speaking, if one refers to computing as a science one should understand what the term science actually entails. One cannot just call an activity science and start to apply for National Science Foundation grants – science is a multifaceted and complex concept. If one were to call computing a science one should understand the various meanings of the term science as well as the aims, methods, and limitations of science. One should understand the complexity of argumentation, logic, confirmations, concepts, demonstrations, and consensus in the computing disciplines; as well as problems with objectivity and the limits of scientific knowledge. One should appreciate the power of the hypothetico-deductive method, yet know its basic blind spots. Finally, students of computing should be taught the proper use of the vocabulary of science – for instance, today many students know slogans such as "correlation is not causality," but too few of those students might be aware of the characteristics of causal and statistical explanations or differences between them.

The engineering tradition

The third tradition in computing, engineering (design), deals with artifacts, which are tangible things created by people, as opposed to naturally occurring things and abstract things. The origins of modern computing lie equally strongly in engineering as they lie in mathematics. Many of the turning points in the history of computing come from technological breakthroughs, not only theoretical breakthroughs. Many pioneers of modern computing, such as John Atanasoff, John Presper Eckert, and Vannevar Bush, were electrical engineers. Whereas the proponent of a mathematical view of computing could argue that computing disciplines as we know them today might not exist without the work of Church, Gödel, or Turing, the proponent of an engineering view of computing could argue that without engineers the sophistication of computing disciplines would have few consequences outside academia, that without engineers computing would still be a compartment of mathematics, or that without engineers the theories of computing would be just idle speculation.

After decades of belittling engineering, a revival of the engineering characteristic of computing was instigated by the software crisis, i.e. the crisis that was born when computing systems grew so large and so complex that their design and management could not have been done by single individuals anymore. A specific term software engineering was first introduced in 1968 at a conference held to discuss the software crisis (Naur & Randell, 1969). The significance and contribution of the engineering view in academic computing was at the time contested, but in the course of time it gained increasing recognition.

Usually the argument for the engineering character of computing fields relies on the view that the goal of computing is to design and construct useful things (cf. Loui, 1995; Wegner, 1976). Frederick P. Brooks Jr (1996) 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 while the engineer studies in order to build. In Brooks' opinion computing professionals are engineers: computing professionals study in order to build, and the discipline of computing is exactly about making things and not about discovering things. Denning et al. (1989) characterized the engineering part of computing, writing "Design is the bedrock of engineering: Engineers share the notion that progress is achieved primarily by posing problems and systematically following the design process to construct systems that solve them." Those who liken programming to art (see, for example, Knuth, 1974b) generally speaking do not refer to art in the sense of ballet or music, but in the sense of craft, trade, or skill – which clearly are traits of engineering.

Juris Hartmanis (1993) argued that generally speaking computer science, "the engineering of mathematics," is concentrating more on the how than the what. Whereas advancements in the natural sciences are often documented by dramatic experiments, in computing advancements are often documented by dramatic demonstrations (Hartmanis, 1993). In some branches of computing the scientists' slogan "publish or perish" indeed might have turned into the engineers' slogan "demo or die". Hartmanis noted that whereas the physical sciences are focused on what exists, computing is focused on what can exist (Hartmanis, 1981, cited in Traub, 1981). That question is, of course, the driving question in many modern fields that aim at creating physical entities that do not occur naturally. However, the term engineering is vague and can encompass a wide variety of computing professionals' activities. Hence, a discussion about computing's engineering traits cannot properly be discussed without understanding some basic ideas about engineering.

Characteristics of engineering

Michael Davis (1998, pp. 7–8) argued that engineers "invent useful things or, at least, add to our knowledge of how to do it." Engineering utilizes, and contributes to, systematic knowledge of how to make useful things. From Davis' (1998, pp. 15–16) viewpoint what separates engineers from applied scientists is the engineers' primary commitment to human welfare vis-à-vis applied scientists' primary commitment to theoretical or applied knowledge. Although engineers do often embrace the precision and rigor of science and mathematics, engineers neither have nor wish to have the "luxury" of being able to infinitely hone their products (see, for example, Florman, 1994, p. 178; cf. Kidder, 1981).

There is a diversity of descriptions of the engineering profession and the engineering philosophy (see, for example, Koen, 2003; Mitcham, 1994), but most of those descriptions 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 (Hamming, 1969; Wegner, 1976). 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 that are directed towards some social need or desire (Mitcham, 1994, pp. 146–147).

It has been argued that the old idea that 'technology is applied science' might no longer be true (if it ever was). Some argue that the inverse direction might be even stronger: most of the progress in modern science can easily be attributed to technological development (see, for example, Mitcham, 1994, pp. 76 & 86). 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 (Pickering, 1995). Some authors have even begun to use the term technoscience instead of science and technology to emphasize that the two have become inseparable (Haraway, 1999; MacKenzie & Wajcman, 1999). Yet although science and engineering share some similarities – for instance, they both have to conform to the laws of nature, they are both cumulative, and they both share the scaling problem (Vincenti, 1990, pp. 134 & 139) – they still utilize and produce different kinds of knowledge and employ different methodologies (Mitcham, 1994, pp. 193–194 & 197).

Denning et al. (1989) 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. The engineering method, as seen by Denning et al. (1989), is a cycle that consists of defining requirements, defining specifications, designing and implementing, and testing. The engineering method of parameter variation seems like an all-time favorite of many computing professionals. Parameter variation refers to the method whereby engineers 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, p. 139). Another flavor of the engineering approach in computing was described by Timothy R. Colburn (2000, p. 167) in the form of solution engineering. That is, in some branches of computing the usual scenario includes rigorous requirements, and the task of the computing professional is to engineer an algorithmic solution.

Difficult distinctions in computing

Sometimes the line between computer science and computer engineering is drawn following the 'physical vs. nonphysical' lines (cf. Arden,1980, p. 7). In other words, computer engineers work with physical things (like hardware or machinery), whereas computer scientists work with abstract things (like algorithms and program texts). However, that distinction is vague and difficult. Firstly, the dual (physical–abstract) nature of programs makes the distinction difficult. Secondly, and perhaps surprisingly, establishing a strict line between hardware and software is difficult.

The dual nature of programs has been illustrated by many philosophers of computing. For instance, Brian Cantwell Smith (1998, pp. 29–32) illustrated that nature by introducing his readers to two programmers named McMath and McPhysics who disagreed on what computer programs really are. McPhysics claimed that computer programs are causal, physical things: computer programs are swarms of electrons in the circuits of a computer. They are physical phenomena and they can make physical phenomena happen. For instance, computer programs can make monitors blink and printers rattle, computer programs can land aeroplanes and guide missiles to their targets.

In Smith's (1998, pp. 29–32) example McMath contradicted McPhysics and claimed that programs are abstract things in the same way that mathematical objects are abstract. Researchers in computing fields can construct procedures and programs in the same way mathematicians construct functions, theorems, and proofs – in their minds or with a pen and paper. Computers are not necessary: computer programs need not have any executable physical counterparts (programs in computer memory or hard drive or such). The same abstract versus executable distinction has been noted by other authors too. James Fetzer (2000, p. 267) called abstract programs (that are not in an executable form) programs-as-texts and executable programs (that reside in, for example, computer memory) programs-as-causes. Herbert A. Simon (1981, pp. 22–26) noted that computers are both abstract objects and "empirical objects."

Programs-as-causes can be presented as programs-as-texts and programs-as-texts can be transformed into programs-as-causes. However, problems arise if one proposes arguments that assume programs as abstract things (programs-as-texts) or programs as physical things (programs-as-causes) but lets the meaning of the term loosely slide between the two meanings. This equivocation was, in fact, the issue behind the formal verification debate that was discussed earlier in this article. Interestingly, scientists in many fields use mathematical tools and abstractions to study the properties of causal entities: for instance, in particle physics mathematical abstractions are used to model causal entities. In various fields similar debates arise in different forms, for instance, in the philosophy of physics there are debates on whether a physical theory's predictive success justifies belief in the unobservable physical entities the theory postulates (Hitchcock, 2004, pp. 115–148).

The hardware–software distinction also faces some difficulties. Those parts of the computer system that one can touch are often considered to be hardware, while software is often considered to be the "non-physical" parts of a computer system. That distinction is, again, vague and it relies on a dualistic view of the world. Firstly, programs-as-causes (when stored as electrical charges in memory or as blips on a magnetic disc) are physical and causal phenomena. More importantly, most hardware can be implemented as software and most software can be implemented as hardware. James H. Moor (1978) noted that a hardware–software distinction is a pragmatic distinction, and it is a subjective distinction. For the user of a microwave oven the whole thing is hardware. But for the engineer of a microwave oven there are software and hardware components. For the systems programmer circuitry is hardware, but a circuit designer can see microprograms as software. A three-dimensional graphics programmer can rarely tell which parts of a three-dimensional program are going to be hardware accelerated and which will eventually run on software. In the end, the functioning of computer software boils down to states of logic gates, flip-flops, adders, and such.

Resistance to the engineering view

However strong the engineering character of computing might seem today, the status of the engineering tradition in computing has frequently been challenged. One of the most free-spoken opponents of the engineering view of computing, Edsger Dijkstra, opposed both the status and 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, 1972, 1989). 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 technical details or any societal aspects of their discipline (Dijkstra, 1987). In his oft-quoted argument against the technological bent of computing Dijkstra (1987) argued that computer 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'."

In particular, accusations of lack of rigor in software engineering have fueled debates about the academic image of software engineering. For instance, C. Michael Holloway (1995) accused software engineers of basing their work on a combination of anecdotal evidence and human authority. In their study of 600 published articles on software engineering Marvin Zelkowitz and Dolores Wallace (1997, 1998) found that in about one-third of the articles the authors failed to experimentally validate their results (see also Tichy, 1998). In his defence of the scientific tradition of computing Eden (2007) criticized the decline of the scientific method in software engineering textbooks and accused the technocratic orientation in computing of causing the problems in the software industry. Michael Davis (1998, p. 38) argued that the fundamental problems of considering software engineering to be a proper branch of engineering are, firstly, that the software engineering curriculum is very different from curricula in other engineering fields and, secondly, that software engineers embrace a code of ethics very different from that of proper engineers.

Critics of the engineering view often argue that it is difficult to see the theoretical foundations of, for instance, software engineering and that the engineering parts of computing are based on rules of thumb. They also often complain about the absence of mathematically and scientifically rigid methods in engineering-oriented branches of computing. However, the usual argument against the engineering view of computing is not so much about the unimportance of engineering. It is widely accepted that producing useful and reliable computing machinery is a 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 (cf. Abrahams, 1987). Not all important activities need to be nominated as academic disciplines.

If one were to appreciate the engineering aspects of computing, one might ask whether computing professionals build in order to study or study in order to build, or, perhaps, if in computing to study is to build. However, those working in the engineering branches should not confuse novelty with a valuable feature of a product (Brooks, 1996). One should understand the dual nature of programs, and the implications of that dual nature. One should be aware of the different epistemological status of engineering outcomes, scientific outcomes, and theoretical outcomes; and of the different methodological and even ontological bases of the three traditions. Finally, one should understand the very strong engineering roots and traditions in computing – the birth of the discipline of computing owes as much to engineering as it owes to mathematics, and many of the turning points in computing have been engineering achievements. Many of the great accomplishments in computing today are indeed technology led.

Conclusions

On the previous pages the three traditions of computing have been presented along with some central debates concerning each of those traditions. Table 1 draws together the

Table 1. Common characterizations of the three traditions of computing.

	Mathematical tradition	Engineering tradition	Scientific tradition
Assumptions	Programs (algorithms) are abstract objects, they are correct or incorrect, as well as more or less efficient – knowledge is a priori	Programs (processes) affect the world, they are more or less effective and reliable – knowledge is a posteriori	Programs can model information processes, models are more or less accurate – knowledge is a posteriori
Aims	Coherent theoretical structures and systems	Constructing useful, efficient, and reliable systems; solving problems	Investigating and explaining phenomena, solving problems
Strengths	Rigorous, results are certain, utilized in other traditions	Able to work under great uncertainty, flexible, progress is tangible	Combines deduction and induction, cumulative
Weaknesses	Limited to axiomatic systems	Rarely follows rigid, preordained procedures; poor generalizability	Incommensurability of results, uncertainty about what counts as proper science
Methods	Analytic, deductive (and inductive)	Empirical, constructive	Empirical, inductive and deductive

general perceptions about the assumptions, strengths, weaknesses, and typical methods of each tradition. Note that most of the characteristics in Table 1 can be disputed. For instance, mathematicians as well as philosophers of mathematics have debated whether numbers are abstract or concrete objects (see, for example, Gödel, 1983), whether mathematical results are really certain (see, for example, Lakatos, 1976), whether mathematics is a priori or a posteriori (see, for example, Mill, 1973), whether mathematics in making is an inductive experimental science (Pólya, 1957, p. 117), and whether applied mathematics is mathematics proper. In Table 1 those debates are ignored, and it presents a simplified overview of the positions that can be found in the computing literature.

In Table 1 the assumptions row lists some basic ontological and epistemological assumptions generally connected with each of the three traditions; the aims row notes a number of aims most commonly associated with each tradition; the strengths row provides some argued strengths of each tradition; the weaknesses row portrays some weaknesses that are often criticized in each tradition; the methods row shows the methodological views often connected with each tradition. However, it should be noted that the portrayal of each tradition in Table 1 is a stereotypical one: each cell could easily be amended with alternative views, and the items in each cell could also be disputed. After all, similarly to many other sciences today, in computing the three traditions are inseparably intertwined: most practitioners in each tradition utilize results and tools from the other two traditions.

Although this discussion might help one to position oneself within the landscape of computing disciplines, usually one's position in that landscape is not fixed. Those who are ambivalent about their standing will benefit from knowing some of the landmark debates about the disciplinary identity of computing disciplines. In addition, contemplation of one's own educational background, of the courses one teaches, of one's mission, and of one's passion will help one to navigate between the traditions. But no matter where one finds oneself within the computing landscape, those who wish to take students on a fascinating tour of computing need to know the pros and cons, promises and challenges, as well as landmark achievements, of each of the major traditions in computing.

References

Abrahams, P. (1987). What is computer science? *Communications of the ACM*, *30*(6), 472–473.

Arden, B.W. (Ed.). (1980). *What can be automated? Computer science and engineering research study*. Cambridge, MA: MIT Press.

Ardis, M., Basili, V., Gerhart, S., Good, D., Gries, D., Kemmerer, R., Leveson, N., Musser, D., Neumann, P., & von Henke, F. (1989). Editorial process verification. *Communications of the ACM*, *32*(3), 287–288.

Ayer, A.J. (1971). *Language, truth, and logic*. Harmondsworth, UK: Penguin Books. (Original work published 1936).

Bowen, J.P., & Hinchey, M.G. (2005). Ten commandments of formal methods ... ten years later. *IEEE Computer*, *39*(1), 40–48.

Brooks, F.P. Jr. (1996). The computer scientist as toolsmith II. *Communications of the ACM*, *39*(3), 61–68.

Bunge, M. (1998). *Philosophy of science,* Vol 1. *From problem to theory* (Rev. ed.). New Brunswick, NJ: Transaction Publishers. (Original work published 1967).

Chalmers, A.F. (1999). *What is this thing called science?* (3rd ed.). Brisbane, Australia: University of Queensland Press. (Original work published 1976).

Colburn, T.R. (2000). *Philosophy and computer science*. Armonk, NY: M.E. Sharpe.

Couvalis, G. (1997). *The philosophy of science: Science and objectivity*. London: Sage.

Davis, M. (1998). *Thinking like an engineer – Studies in the ethics of a profession*. New York: Oxford University Press.

Davis, R.L. (1977). Recommended mathematical topics for computer science majors. *ACM SIGCSE Bulletin*, *9*(3), 51–55.

De Millo, R.A., Lipton, R.J., & Perlis, A.J. (1979). Social processes and proofs of theorems and programs. *Communications of the ACM*, *22*(5), 271–280.

Denning, P.J. (1980). What is experimental computer science? *Communications of the ACM*, *23*(10), 534–544.

Denning, P.J. (2005). Is computer science science? *Communications of the ACM*, *48*(4), 27–31.

Denning, P.J. (2007). Computing is a natural science. *Communications of the ACM*, *50*(7), 13–18.

Denning, P.J. (Chairman), Comer, D.E., Gries, D., Mulder, M.C., Tucker, A., Turner, A.J., & Young, P.R. (1989). Computing as a discipline. *Communications of the ACM*, *32*(1), 9–23.

Denning, P.J., Feigenbaum, E., Gilmore, P., Hearn, A., Ritchie, R.W., & Traub, J. (1981). A discipline in crisis. *Communications of the ACM*, *24*(6), 370–374.

Denzin, N.K. & Lincoln, Y.S., (Eds.). (1994). *Handbook of qualitative research*. London: Sage.

Dijkstra, E.W. (1972). The humble programmer. *Communications of the ACM*, *15*(10), 859–866.

Dijkstra, E.W. (1974). Programming as a discipline of mathematical nature. *American Mathematical Monthly*, *81*(June/July), 608–612.

Dijkstra, E.W. (1987). Mathematicians and computing scientists: The cultural gap. *Abacus*, *4*(4), 26–31.

Dijkstra, E.W. (1989). On the cruelty of really teaching computer science. *Communications of the ACM*, *32*(12), 1398–1404.

Eden, A.H. (2007). Three paradigms of computer science. *Minds and Machines*, *17*(2), 135–167.

Fetzer, J.H. (1988). Program verification: The very idea. *Communications of the ACM*, *31*(9), 1048–1063.

Fetzer, J.H. (2000). Philosophy and computer science: Reflections on the program verification debate. In T.W. Bynum & J.H. Moor (Eds.), *The digital phoenix: How computers are changing philosophy* (pp. 253–273). Oxford, UK: Blackwell (Original work published 1998).

Feyerabend, P. (1993). *Against method* (3rd ed.). New York: Verso. (Original work published 1975).

Fletcher, P. (1995). Readers' corner: The role of experiments in computer science. *Journal of Systems and Software*, *30*(1–2), 161–163.

Fisher, A., & Margolis, J. (2002). Unlocking the clubhouse: The Carnegie Mellon experience. *ACM SIGCSE Bulletin*, *34*(2), 79–83.

Florman, S.C. (1994). *The existential pleasures of engineering* (2nd ed.). New York: St Martin's Press.

Forsythe, G.E. (1967). A university's educational program in computer science. *Communications of the ACM*, *10*(1), 3–11.

Forsythe, G.E. (1969 August). Computer science and education. In *Proceedings of the IFIP Congress 1968*, Vol. 2 (pp. 92–106). Edinburgh, UK: International Federation for Information Processing.

Glass, R.L. (1995). A structure-based critique of contemporary computing research. *Journal of Systems and Software*, *28*(1), 3–7.

Gödel, K. (1983). Russell's mathematical logic. In P. Benacerraf & H. Putnam (Eds.), *Philosophy of mathematics* (2nd ed., pp. 447–469). Cambridge, UK: Cambridge University Press.

Goldweber, M., Impagliazzo, J., Bogoiavlenski, I.A., Clear, A.G., Davies, G., Flack, H., Myers, J.P., & Rasala, R. (1997). Historical perspectives on the computing curriculum. *ACM SIGCUE Outlook*, *25*(4), 94–111.

Hamming, R.W. (1969). One man's view of computer science (ACM Turing Lecture). *Journal of the Association for Computing Machinery*, *16*(1), 3–12.

Hamming, R.W. (1980). The unreasonable effectiveness of mathematics. *The American Mathematical Monthly*, *87*(2), 81–90.

Haraway, D.J. (1999). Modest_witness@second_millennium. In D. MacKenzie & J. Wajcman (Eds.), *The social shaping of technology* (2nd ed., pp. 41–49). London: Open University Press.

Hartmanis, J. (1981). Nature of computer science and its paradigms (part of a panel discussion). *Communications of the ACM*, *24*(6), 353–354.

Hartmanis, J. (1993). Some observations about the nature of computer science. In R.K. Shyamasundar (Ed.), *Lecture notes in computer science*, Vol. 761 (pp. 1–12). Berlin: Springer.

Hartmanis, J. (1994). Turing award lecture: On computational complexity and the nature of computer science. *Communications of the ACM*, *37*(10), 37–43.

Hempel, C.G. (1965). *Aspects of scientific explanation and other essays in the philosophy of science*. New York: The Free Press.

Hitchcock, C. (2004). *Contemporary debates in philosophy of science*. Oxford, UK: Blackwell.

Hoare, C.A.R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, *12*(10), 576–580.

Holloway, C.M. (1995). Software engineering and epistemology. *ACM SIGSOFT Software Engineering Notes*, *20*(2), 20–21.

Kemeny, J.G. (1959). *A philosopher looks at science*. Princeton, NJ: Van Nost, Reinhold.

Khalil, H., & Levy, L.S. (1978). The academic image of computer science. *ACM SIGCSE Bulletin*, *10*(2), 31–33.

Kidder, J.T. (1981). *The soul of a new machine*. New York: Little, Brown, & Co.

Knuth, D.E. (1974a). Computer science and its relation to mathematics. *American Mathematical Monthly*, *81*(April), 323–343.

Knuth, D.E. (1974b). Computer programming as an art. *Communications of the ACM*, *17*(12), 667–673.

Knuth, D.E. (2001). *Things a computer scientist rarely talks about*. Stanford, CA: CSLI Publications.

Koen, B.V. (2003). *Discussion of the method: Conducting the engineer's approach to problem solving*. Oxford, UK: Oxford University Press.

Kuhn, T. (1996). *The structure of scientific revolutions* (3rd ed.). Chicago: University of Chicago Press (Original work published 1962).

Lakatos, I. (1976). *Proofs and refutations: The logic of mathematical discovery*. In J. Worrall & E. Zahar (Eds.). Cambridge, UK: Cambridge University Press.

Lloyd, S. (2007). *Programming the universe: A quantum computer scientist takes on the cosmos*. London: Vintage Books.

Loui, M.C. (1995). Computer science is a new engineering discipline. *ACM Computing Surveys*, *27*(1), 31–32.

MacKenzie, D. & Wajcman, J., (Eds.). (1999). *The social shaping of technology* (2nd ed.) London: Open University Press.

McKee, G. (1995). Computer science or simply 'computics'? *IEEE Computer*, *28*(12), 136.

Mill, J.S. (1973). *A system of logic: The collected works of John Stuart Mill*. In J.M. Robson (Ed.), Vol. 7. Toronto: University of Toronto Press.

Minsky, M.L. (1979). Computer science and the representation of knowledge. In M.L. Dertouzos & J. Moses (Eds.), *The computer age: A twenty-year view* (pp. 392–421). Cambridge, MA: MIT Press.

Mitcham, C. (1994). *Thinking through technology: The path between engineering and philosophy*. Chicago: University of Chicago Press.

Moor, J.H. (1978). Three myths of computer science. *The British Journal for the Philosophy of Science*, *29*, 213–222.

Naur, P. (1966). Proof of algorithms by general snapshots. *BIT*, *6*(4), 310–316.

Naur, P. (1969). Programming by action clusters. *BIT*, *9*(3), 250–258.

Naur, P. (1972). An experiment on program development. *BIT*, *12*(3), 347–365.

Naur, P. & Randell, B., (Eds.). (1969). *Software engineering: Report on a conference sponsored by the Nato Science Committee; Garmisch, Germany, 7th–11th October 1968*. Brussels: NATO Scientific Affairs Division.

Newell, A., Perlis, A.J., & Simon, H.A. (1967). Computer science. *Science*, *157*(3795), 1373–1374.

Newell, A., & Simon, H.A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, *19*(3), 113–126.

Okasha, S. (2002). *Philosophy of science: A very short introduction*. Oxford, UK: Oxford University Press.

Pickering, A. (1995). *The mangle of practice: Time, agency, and science*. Chicago: University of Chicago Press.

Pólya, G. (1957). *How to solve it* (2nd ed.). London: Penguin Books (Original work published 1945).

Popper, K. (1959). *The logic of scientific discovery*. London: Routledge. (Original work published 1935).

Ralston, A., & Shaw, M. (1980). Curriculum '78 – is computer science really that unmathematical? *Communications of the ACM*, *23*(2), 67–70.

Rapaport, W.J. (2005). Philosophy of computer science: An introductory course. *Teaching Philosophy*, *28*(4), 319–341.

Rosenberg, A. (2000). *Philosophy of science: A contemporary introduction* (2nd ed.). New York: Routledge.

Rosenbloom, P.S. (2004). A new framework for computer science and engineering. *IEEE Computer*, *37*(11), 23–28.

Ross, P.E. (2003). 5 commandments (technology laws and rules of thumbs). *IEEE Spectrum*, *40*(12), 30–35.

Seife, C. (2006). *Decoding the universe. How the new science of information is explaining everything in the cosmos, from our brains to black holes*. London: Penguin Books.

Simon, H.A. (1981). *The sciences of the artificial* (2nd ed.). Cambridge, MA: MIT Press. (Original work published 1969).

Smith, B.C. (1996). Limits of correctness in computers. In R. Kling (Ed.), *Computerization and controversy* (pp. 810–825). San Diego, CA: Academic Press. (Original work published 1985).

Smith, B.C. (1998). *On the origin of objects*. Cambridge, MA: MIT Press.

Tedre, M. (2007a). *Lecture notes in the philosophy of computer science*. Retrieved February 18, 2008, from http://cs.joensuu.fi/~mmeri/teaching/2007/philcs/

Tedre, M. (2007b). Know your discipline: Teaching the philosophy of computer science. *Journal of Information Technology Education*, *6*(1), 105–122.

Tichy, W.E. (1998). Should computer scientists experiment more? *IEEE Computer*, *31*(5), 32–40.

Tichy, W.F., Lukowicz, P., Prechelt, L., & Heinz, E.A. (1995). Experimental evaluation in computer science: A quantitative study. *Journal of Systems and Software*, *28*(1), 9–18.

Traub, J.F. (1981). Quo vadimus: Computer science in a decade. *Communications of the ACM*, *24*(6), 351–369.

Vincenti, W.G. (1990). *What engineers know and how they know it: Analytical studies from seronautical history*. Baltimore, MD: Johns Hopkins University Press.

von Wright, G.H. (1971). *Explanation and understanding*. London: Routledge & Kegan Paul.

Wegner, P. (1976). *Research paradigms in computer science*. In *Proceedings of the 2nd International Conference on Software engineering* (pp. 322–330). San Francisco, CA: IEEE Computer Society Press.

Wirth, N. (1971). Program development by stepwise refinement. *Communications of the ACM*, *14*(4), 221–227.

Wolfram, S. (2002). *A new kind of science*. Champaign, IL: Wolfram Media.

Zelkowitz, M.V., & Wallace, D. (1997). Experimental validation in software engineering. *Information and Software Technology*, *39*(11), 735–743.

Zelkowitz, M.V., & Wallace, D.R. (1998). Experimental models for validating technology. *IEEE Computer*, *31*(5), 23–31.