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 January 29, 2007.
Available at cs.joensuu.fi/~mmeri/teaching/2007/philcs/

2 The History of the Discipline

- How has the term *computer science* been used in different dec – ades?
- What kinds of arguments have been used to back up different in – terpretations of *computer science*?
- What kinds of controversies have there been in the disciplinary history of computer science?
- What kinds of "schools" have there been in computer science?
- What kinds of things affected the growing interdisciplinarity of computer science?
- How has computer science been characterized recently?
- Why is the name of the field so important?

The history of computing as a discipline and profession is not much shorter than the history of elec – tronic computing itself. The first societies for computing professionals were founded at the same time that the first fully electronic, digital, Turing-complete computer ENIAC was unveiled. It is difficult to trace the earliest discussions of computing as a distinct discipline, but in the April 1958 issue of *Communications of the ACM* (*CACM*) the editors of DATA-LINK asked the editor of CACM:

> [1958] *What is your reply when someone asks your profession? Computing Engineer? Numerical Analyst? Data Processing Specialist? To say "Computer" sounds like a machine, and "Programmer" has been confused with "Coder" in the public mind (if your particular segment of the public knows what you are talking about at all!)*
>
> (The editors of the DATA-LINK (Los Angeles ACM Chapter Newsletter) in Letters to the Editor, April 1958 issue of *Communications of the ACM*)

The editors of DATA-LINK then asked for suggestions for the *name of the discipline*, noting that a brief, definitive, and distinctive name would help the profession to be widely recognized (yet the editors of DATA-LINK did not further specify what "the profession" actually was).

After the early attempts to describe the academic field of computing, such as the one above, there has been a dizzying amount of debate about the form and content of the field. "*What is computer science?*" is certainly one of the central questions of the philosophy of computer science. Computer science seems to defy strict characterizations, yet not just *any* activity is computer science. Philo – sophical questions might include, for instance, if there are *necessary conditions* that an activity must meet in order to be computer science, and if there are *sufficient conditions* that guarantee that an activity is certainly computer science. This week a look at the changes between 1946 and 1978 in the views of computing as a discipline is taken. During this era there grew an agreement that com – puter science is indeed an academic discipline of its own, but computer scientists really had to struggle for an acknowledgment of the status of their science.

NECESSARY AND SUFFICIENT CONDITIONS

If condition C must be satisfied in order for a statement S to be true, then condition C is a *necessary condition* for statement S. For instance, a set of rules cannot be considered to be *an algorithm* unless the execution of those rules will end after a finite number of steps (Knuth, 1997:4-5). That is, *finiteness* is a *necessary condition* for being an algorithm. Meeting one necessary condition does not, however, mean that the statement is true. There can be many necessary conditions that all need to be fulfilled—for instance, algorithms must also be definite and effective (see p.85 of these lecture notes), and have output (Knuth, 1997:5-6).

If condition C is enough to *guarantee* that a statement S is true, then condition C is a *sufficient condition* for statement S. Condition C can be both a necessary condition and a sufficient condition at the same time. For instance, a function cannot be considered to be *computable* unless it is computable with a Turing Machine (a necessary condition); but there again, if a function is computable with a Turing Machine, it is, by definition, considered to be computable (a sufficient condition). (See Brennan, 2003 or Swartz, 1997 for further discussion on the topic).

Most of the sources in this chapter are from U.S.-based publications, which inevitably biases the discussion to the English-speaking world and to the American scene. The approach in this chapter is *historical* and *descriptive*: the aim is to demonstrate the diversity of opinions on what *computer science* is, by presenting a wide variety of characterizations of computer science or computing as a discipline. In this chapter, all long verbatim quotations begin with the year of the quotation in brackets (e.g., [1976]). This convention is adopted to help the reader with the timeline of this section.

This chapter follows the historical development of computer science. The chapter begins with some early attempts to define computing as a discipline, and proceeds to the official birth and recognition of computer science. Then the topics shift to the different foci that divided computer scientists into coteries, and to the different viewpoints into the dispute between theoretically oriented and practically oriented computer scientists. Finally, this chapter portrays the growth pains that arose from the developing diversification of computer science.

2.1 Struggling for Status: 1940s-1970s

The same year ENIAC was unveiled, 1946, the first society for computing professionals, *Subcommittee on Large-Scale Computing* of the *American Institute of Electrical Engineers* (AIEE) was founded. Five years later, the *Institute of Radio Engineers* (IRE) formed its *Professional Group on Electronic Computers*. Then, in 1963 the AIEE and IRE merged, forming today's *Institute of Electrical and Electronics Engineers* (IEEE). This merging also led to the *IEEE Computer Society*. The *Association for Computing Machinery* (ACM) was founded 1947, and the specialization of IEEE and ACM emerged early in their history: The IEEE Computer Society focuses on standards and hardware, whereas the ACM focuses on theoretical computer science and applications.

Although the professional societies of AIEE, IRE, and ACM were clearly focused on computing, it is difficult to say if the professionals in those societies considered themselves primarily as comput-

ing professionals or perhaps engineers or mathematicians who focus on computing. For instance, in some of the articles in the early issues of *Journal of the ACM* (JACM) the identity of computing professionals is still unclear (see Williams, 1954; Householder, 1956; Householder, 1957; Carr, 1957).

Journal of the ACM was first published in 1954, and in the inaugural issue the president of the ACM at the time, Samuel B. Williams, sketched the development of ACM (Williams, 1954; see also Alt, 1962). Williams noted that the membership of the *Eastern Association for Computing Machinery*, renamed to ACM in January 1948, had grown in seven years from a 78-member in – formal group of people interested in computing (May 1947) into a 1200-member international group of people (January 1954). In the early years of the ACM there were debates about whether the ACM should concern itself with hardware or theory (Householder, 1957). Eventually, *Commu – nications of the ACM* was established in 1958 to be a forum for timely information in the field, whereas *Journal of the ACM* was for articles less temporal by nature (Bauer et al., 1959).

During the last years of the 1950s, the terminology in the field of computing was discussed in the Communications of the ACM, and a number of terms for the practitioners of the field of computing were suggested: *turingineer*, *turologist*, *flow-charts-man*, *applied meta-mathematician*, *applied epi – stemologist* (Weiss & Corley, 1958), *comptologist* (Correll, 1958), *hypologist* (Zaphyr, 1959), and *computologist*[1]. The corresponding names of the discipline were, for instance, *comptology*, *hypo – logy*, and *computology*. Later John A. Berenberg suggested the term *metaphrast* to describe a com – puter scientist who works in the area of compilers (Berenberg, 1971); Peter Naur suggested the terms *datalogy*, *datamatics*, and *datamaton* for the names of the field, its practitioners, and the ma – chine (Naur, 1966); and recently George McKee suggested the term *computics* (McKee, 1995). None of these terms stuck, but the discussion about the name of the field was a clear indicator that by the turn of the 1960s the search for a disciplinary identity had begun.

2.1.1 Early Characterizations

As the field matured, and as model curricula and textbooks were developed, the programs at various universities increasingly came to resemble one another (Aspray, 2000). Although a common agree – ment of the field developed, throughout the 1960s computer specialists continued to wonder at the "*almost universal contempt*" (C.J.A., 1967) or at least "*cautious bewilderment and misinterpreta – tion*" (Datamation, 1962) with which programmers were regarded by the general public (Ens – menger, 2001). If there was a public image of computing as a profession, it seems to have been somewhat negative. In the November-December 1959 issue of *Datamation*, the leading periodical of the era, Herb Grosch was concerned that information processing was being defined too narrowly because in Grosch's opinion it is "*as broad as our culture and as deep as interplanetary space*" (Grosch, 1959).

In 1967 computer science as a term was still quite new—for instance, the world's first department of computer science was established at Purdue University in 1962 (Rice & Rosen, 2004). Nonethe – less, in the 1960s people from fields other than computer science often asked, "What exactly is

1 The December 1958 issue of DATA-LINK attributes the term *computology* to Edmund C. Berkeley. The editors wrote, "*We like the name "Computology" for our profession, as suggested by E.C. Berkeley in the November issue of COMPUTERS AND AUTOMATION. A member of the profession would therefore be a "Computologist.*"".

computer science?". Three computer scientists—Allen Newell, Alan J. Perlis and Herbert A. Simon—gave a public answer. They elaborately defended the disciplinary uniqueness of computer science in an article published in the September 1967 issue of *Science*:

> [1967] *Wherever there are phenomena, there can be a science to describe and explain those phenomena. Thus, the simplest (and correct) answer to "What is botany?" is "Botany is the study of plants". And zoology is the study of animals, astronomy the study of stars, and so on. Phenomena breed sciences.*
>
> *There are computers. Ergo, computer science is the study of computers. The phenomena surrounding computers are varied, complex, rich.* [...] *Computer science is the study of the phenomena surrounding computers.*
>
> (Newell et al., 1967)

As it was noted earlier, Newell et al.'s definition is a *descriptive* one (i.e., it describes what professionals, researchers, and teachers actually do), and it leans towards the *empiricist* research tradition because it emphasizes knowledge creation from description and explanation (in contrast to the rationalist research tradition; see p.25). (Note that at the time of Newell et al.'s writing, terms that today might be called "computer engineering" and "computer science" did not exist in the same manner they do today.)

> EMPIRICISM
>
> The British empiricists of the 17th and 18th centuries held that all knowledge should be derived from ideas that senses have implanted in the mind. That is, knowledge should be derived from *experience*. Hence the name *empiricism*: empiricists believe that all knowledge must be obtained *empirically*—via experience and observation. Empirical knowledge, or knowledge that is derived from experience, experiments, and observation, is also called *a posteriori* knowledge (Latin for "*from after*", because knowledge comes *after* one experiences or observes something) (see, e.g., Markie, 2004 in Stanford Encyclopedia of Philosophy).
>
> In computer science, the *empiricist tradition* can be seen when computer scientists make hypotheses; experiment; and, based on their experiments, make arguments about computation, computers, or other topics of computer science. For example, computer scientists are doing empirical research when they implement two algorithms; compare their running times; and, based on that comparison, argue that at least in certain cases one of the two algorithms is faster than the other one. Their argument rests on *empirical results*.

Despite its eloquence, Newell et al.'s definition has been criticized as being a circular definition that seems flippant to outsiders (Denning et al., 1989). Nonetheless, the definition has been widely quoted ever since its publication. Newell et al. wrote that unlike tools like the electron microscope and the spectrometer, the computer cannot be said to be subsumed under any other science as an instrument. (Note that there actually was an academic field that studied microscopes and their applications, that is, *microscopical science* or *microscopy*, but it is no longer considered to be a distinct

discipline.) Newell et al. argued that the study of computers *does not lead to user sciences, but to the further study of computers*. They continued that the computer by this definition is not just an in – strument but a phenomenon as well, requiring description and explanation—and phenomena define the focus of the science, not its boundaries. Note that in Newell et al.'s definition, boundaries form around the machine called *the computer*, not around *computing* or *users*.

Although the computer is not an instrument of one *single* field, it does work as an instrument for numerous sciences. The development of computer hardware and software—especially nowadays—is often not an end in itself. Thus, from the academic view of computer science, a utilitarian ques – tion has been raised: "*Of what use is computer science in the real world?*" (MacKinnon, 1988). One can, for the time being, elide the issues with the term *real world* and give one answer to the question.

MacKinnon's question above could be answered in the spirit of Newell et al.'s definition: If com – puter science is the study of the phenomena surrounding computers then the more instrumental uses *the computer* has, the more diverse *computer science* is. Consequently, the more instrumental uses the computer has, the more use computer science has in the real world. Be that as it may, Newell et al.'s definition inevitably includes many cases that most people would not consider to be computer science (McGuffee, 2000). If one sticks to Newell et al.'s definition and reads it loosely, in today's Western society where computers are ubiquitous, the "phenomena surrounding computers" can be almost anything, and computer science can extend to concern almost everything.

Although Newell et al.'s article may not characterize computer science very well, in their article, Newell et al. listed six objections that skeptics of computer science often pose and that are relevant also today: (1) "Only natural phenomena breed sciences, but computers are artificial—hence com – puters are whatever they are made to be, hence obey no invariable laws, hence cannot be described and explained"; (2) "The term 'computer' is not well defined, and its meaning will change with new developments, hence computer science does not have a well-defined subject matter"; (3) "Computer science is the study of algorithms (or programs), not computers"; (4) "Computers, like thermomet – ers, are instruments, not phenomena"; (5) "Computer science is a branch of electronics (or mathem – atics, psychology, and so forth)"; and (6) "Computers belong to engineering, not science" (Newell et al., 1967, verbatim quotes).

Those six objections appear, in different forms, throughout the history of computer science, and a number of variants of those objections are presented in this and following chapters. If those objec – tions are formulated in the form of questions, one gets a number of good questions about the nature of computer science:

- Is computer science a *science* in the same sense *natural sciences* are? Are there *laws* in computer science?
- What is the *subject matter* of computer science? Is it *computers*, *programs*, *algorithms*, *users*, *uses*, or something else?
- Are computers only *instruments*, or are they a worthy subject of study as such?
- Does computer science have a *disciplinary identity* distinct from the disciplines that it ori – ginated from?

2.1.2 The Art and Science of Processing Information

One of the power figures of early computer science, George Forsythe, wrote in the January 1967 issue of *CACM*:

> [1967] *I consider computer science, in general, to be the art and science of representing and processing information and, in particular, processing information with the logical engines called automatic digital computers.* [... A] *central theme of computer science is analogous to a central theme of engineering science—namely, the design of complex systems to optimize the value of resources.*
>
> (Forsythe, 1967)

Forsythe brought a number of important aspects of computing to light. Those aspects are the dichotomy between arts and science, the activities in computer science, the significance of real-world constraints, and the juxtaposition of practice and theory.

First, Forsythe noted the oft-used dichotomy between *art* and *science*. Although Forsythe has not defined *art* in the context of computing, another famous computer scientist, Donald Knuth, contemplated on the term *art* in the context of *programming* in his 1974 Turing Award lecture. In Knuth's words, "*Science is knowledge which we understand so well that we can teach it to a computer; and if we don't fully understand something, it is an art to deal with it*." (Knuth, 1974c). It must be noted here that many of those who refer to *art* of computer programming and quote Knuth, consider art in the sense of music, ballet, or painting; whereas Knuth used the term *art* more in the sense of a craft, trade, or skill. See Knuth's original article (Knuth, 1974c) for an enlightening discussion on the topic.

Referring to C.P. Snow (1905-1980) Knuth described the scientific approach with terms such as logical, systematic, impersonal, calm, and rational; and described the artistic approach with terms such as æsthetic, creative, humanitarian, anxious, and irrational (see Snow, 1964). Knuth wrote, "*It seems to me that both of these apparently contradictory approaches have great value with respect to computer programming*.". Also Richard Hamming noted, in his article in the January 1969 issue of the Journal of the ACM, the lack of insight into the artistic aspects of computing: "*To parody our current methods of teaching programming, we give beginners a grammar and a dictionary and tell them that they are now great writers. We seldom, if ever, give them any serious training in style*." (Hamming, 1969:10).

ART

Art is a notoriously ambiguous term. In this context, as contrasted with *science*, the following definitions of *art* from the American Heritage Dictionary of the English Language might be relevant: [...] **5.** A nonscientific branch of learning; [...] **6a.** A system of principles and methods employed in the performance of a set of activities: the art of building. **6b.** A trade or craft that applies such a system of principles and methods: the art of the lexicographer [...] **7b.** Skill arising from the exercise of intuitive faculties: "Self-criticism is an art not many are qualified to practice" (Joyce Carol Oates).

The question of whether computer science is an art intertwines with the questions of whether programming is an art and whether programming is a part of computer science. There is indeed a problem with Forsythe's definition of computer science above: If art is different from science, and if computer science is "*art and science of representing and processing information*", then computer science inevitably includes nonscientific aspects and perhaps the *science* part of computer science is misleading. On the other hand, many intellectually challenging activities—including mathematics—have been argued to include artistic or artistic-like aspects and methods (e.g., Pólya, 1957).

Now, if programming is an *art* and if computer science is strictly *science*, then programming cannot be a part of computer science proper. In that case programming is something else, perhaps application of scientific and non-scientific knowledge into production of computer programs. But some people argue that programming is indeed science. And on the other hand, some people argue that computer science is not only a science but also, at the same time, a sort of *engineering* and a sort of *design activity*.

SCIENCE

As it was noted earlier (p. 4), *science* is an excessively vague term. In this context, as contrasted with *art*, the term *science* refers to means of acquiring knowledge that is based on methodological observation, identification, description, experimental investigation, and theoretical explanation of phenomena. Most scientists maintain that science is done following the scientific method. See, for instance, American Heritage Dictionary of the English Language, Wikipedia, and Encyclopædia Britannica for dictionary entries for *science*.

One must remember that neither science nor art are solely virtuous categories. Paul Feyerabend noted that terms such as *science* and *art* are temporary collecting-bags containing a great variety of products, some excellent, others rotten, and all of them characterized by a single label (Feyerabend, 1994). But collecting-bags and labels, Feyerabend wrote, do not affect reality. They can be omitted without changing what they are supposed to organize. Of course, Knuth never implied that all science and all art would be equally valuable. Knuth's notions that art can be transformed into science and that theory improves artistry are important (Knuth, 1974c; Knuth, 1991). Art and science are not rigid labels or collecting-bags—ideas can move between art and science.

Second, Forsythe noted that *representing* and *processing* information are activities of computer science. He also distinguished between a broad sense (*general*) and a narrow sense (*particular*) of those two activities. In the broad sense, *representing* can be considered to refer to *abstract* data representations or data structures, be they trivial or complex, or be they intuitive (art) or structural-categorical (science). Furthermore, *processing* can be considered to refer to informal (art) or formal (science) ways of manipulating those representations of information (algorithms) (see Table 1). Forsythe's narrow sense of representing and processing seems to confine these actions to those achievable with a digital computer. That is, Forsythe noted that in computer science there is the study of abstract aspects of computing, but there is also the study of concrete implementations and applications of computing.

Table 1: Activities of Computer Science

	Broad sense	***Narrow sense***
Representing	*Abstract representations*. Intuitive or structural-categorical representations of information.	*Data structures*. Concrete implementations on a digital computer.
Processing	*Algorithms*. Informal or formal approaches to manipulation of information—no need to be tractable or even mechanically realizable.	*Computer programs*. Executable and tractable implementations and applications on a digital computer.

Third, Forsythe foregrounded *design* and *resources*. Forsythe's computer science is not situated in the ideal, infinite world of mathematics, but it is situated within the finite boundaries of available resources. Design is rooted in engineering and deals with constructing systems or devices to solve a given problem (Denning et al., 1989). Design, as an engineering activity, has to cater not only to material resources but also to human constraints. In addition, whereas the Turing Machine does not have space constraints (Turing, 1936; Turing, 1950), Forsythe's version of computing takes into account the limits of computing resources, that is, actual computers.

On the whole, Forsythe offered an early definition of computer science that tries to accommodate both the pragmatic and the abstract sides of computer science under the same umbrella term, instead of dividing computing into separate fields of science and engineering (or into abstract and concrete, or into pure and applied).

2.1.3 The Official Birth of Computer Science

At the turn of the 1970s neither the theoretical camp nor the practical camp in computer science were undivided. Judith Gal-Ezer and David Harel wrote that there existed a clear dichotomy within the ranks of mathematically oriented computer scientists because “the mathematical aspects of computer science include not only *computability* and *complexity*, which touch upon logic, combinatorics, and probability theory—but the mathematical aspects also include *numerical analysis*, which can be viewed as a direct outcome of the need for extremely heavy, yet accurate, computations” (Gal-Ezer & Harel, 1998). The practitioners' side was even more scattered—the practical sides of computer science included, for instance, architectural design, coding, computer engineering, and program design.

Lotfi A. Zadeh, who has been credited as being the father of fuzzy logic, wrote in 1968 that com – puter science consists of subjects which belong to computer science to different degrees (Zadeh, 1968). In Zadeh's opinion, fuzzy sets of topics that play a central role in computer science, such as programming languages, operating systems, and data structures, have almost full containment in the fuzzy set of computer science. Those topics that are more peripheral, such as mathematical logic, have "less containment in computer science". Zadeh's (sets of) topics of computer science are grouped in Figure 1 according to their containment in (the set of) computer science (adapted from Zadeh, 1968, Table 1—*Containment Table for Computer Science*).

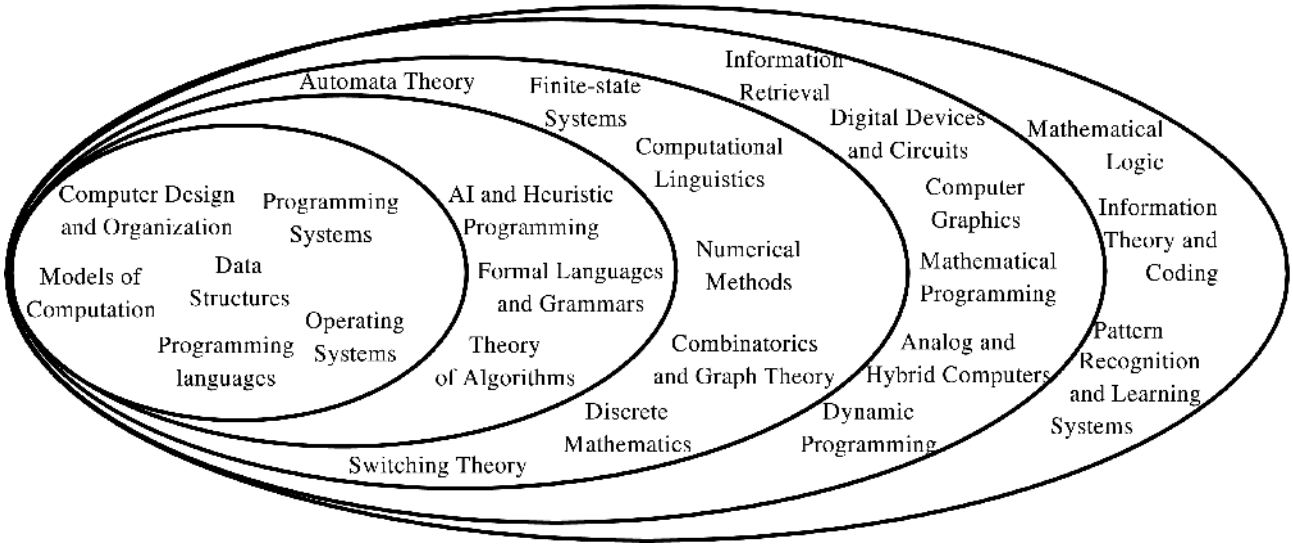

Figure 1: Topics of Computer Science Grouped According to Their Containment in CS

The topics of computer science on the innermost circle of Figure 1 are those that Zadeh wrote to have a "degree of containment equal to unity". Zadeh graded these topics a degree of containment equal to 1 (on a scale from 0 to 1). The topics that Zadeh graded to have a degree of containment in computer science equal to 0.9 are grouped on the second innermost circle in Figure 1. The topics on the third, fourth, and fifth circles have degrees of containment equal to 0.8, 0.7, and 0.6, respect – ively.

Official Steps in Legitimating Computer Science

The ACM had begun working on a recommendation for academic programs in computer science as early as 1962 (the same year as Purdue University launched the first study program actually called computer science; see Rice & Rosen, 2004). The *ACM Curriculum Committee* became an inde – pendent committee of the ACM in 1964, and released their first draft in 1965 (Conte et al., 1965). The final version of the first ACM Curriculum was completed in 1968, and it characterized the sub – ject areas of computer science as follows:

> [1968] *The subject areas of computer science are grouped into three major divisions: "information structures and processes", "information processing systems" and "meth – odologies".*
>
> (Atchison et al., 1968)

Atchison et al.'s description of computer science can be seen either as a normative account of com – puter science (i.e., aiming at defining the discipline by listing the topics computers scientists *should*

be doing) or a descriptive account of computer science (i.e., describing what is *actually* being done in computer science). Computer science, according to this view, is the study of *information structures and processes*. String processing is, after all, what computers do. Atchison et al.'s report, *Curriculum '68*, describes and discusses twenty-two courses that the committee found to be a part of the computer science curriculum at universities in the U.S., and the report uses two dozen pages of suggested readings for those courses. In the '68 report, the ACM people clearly took some distance to subjects and institutions they believed should not belong to computer science:

> [1968] *...these recommendations are not directed to the training of computer operators, coders, and other service personnel. Training for such positions, as well as many programming positions, can probably be supplied best by applied technology programs, vocational institutes, or junior colleges.*
>
> (Atchison et al., 1968)

ACM's 1968 definition reflects a rationalistic (p. 25), mathematical research tradition, insomuch as algorithms and information structures are *abstractions* of the phenomena that computer science is concerned with (Wegner, 1976). Note that the ACM definition excludes computer-related work that is not considered academic, such as programming. That is, *programming does not belong to computer science*, or at least programmers are not computer scientists. At the time Atchison et al.'s position was already criticized for being too academic, theoretical, and narrow. The critics demanded a more practitioner oriented view, hands-on laboratory work, and inclusion of other computer-related areas into computer science education.

In the June 1974 issue of CACM, the president of the ACM at the time, Bernard A. Galler, announced that the National Science Foundation (NSF) had passed the following resolution:

> [1974] *Resolved that the Computer Science and Engineering Advisory Panel of NSF affirms the distinction of Computer Science from all other science or engineering disciplines and recommends that the National Science Foundation make this manifest in its statistical and programmatic activities.*
>
> (Galler, 1974)

This resolution was greeted as an important step, because it was expected that the distinction of computer science as an autonomous discipline would increase funding, give computer science its own student and research fellow quotas, and grant computer science its own representation on the National Science Board and similar policy-making committees and boards (Galler, 1974). Those expectations mirror the importance of a disciplinary identity.

Shaping the Public Image of Computing

In his 1968 Turing Award lecture, Richard Hamming stated his occasional frustration with the debate around "*what is computer science, what is it currently, what can it develop into, what should it develop into, and what will it develop into*" (Hamming, 1969). Still he considered it very important not to ignore the discussion over definitions of computing to just "get on with doing it", and he brought to light a very important point—the point of societal influence in how science is formed:

> [1969] *the picture which people have of a subject can significantly affect its subsequent development. Therefore, although we cannot hope to settle the question definitively, we need frequently to examine and to air our views on what our subject is and should become.*
>
> (Hamming, 1969)

In Hamming's sense of the term computer science, at the heart of computer science lies a technological device, the computing machine. Without it, Hamming argued, almost everything that computer scientists do would become idle speculation, hardly different from that of the notorious Scholastics of the Middle Ages[2]. Hamming noted that the work of computer scientists is guided by expenses, workload, and the popular image of computer science (which also affects the amount of money granted to science):

> [1969] *So much of what we do is not a question of can it be done as it is a question of finding a practical way. It is not usually a question of can there exist a monitor system, algorithm, scheduler, or compiler, rather it is a question of finding a working one with a reasonable expenditure and effort.*
>
> (Hamming, 1969)

That is, in Hamming's opinion, theoretician's question "*Can there be x?*" is less frequent than practitioner's question "*What is the most cost-effective way of building x?*". Therefore the focus should be not computing or computations but the computer.

2.1.4 A Concern Over Dogmatism in Computing

In his 1970 Turing Award lecture, published in the April 1970 issue of Journal of the ACM, Marvin Minsky expressed his concern about computer science having an obsession with form instead of content (Minsky, 1970). For instance, programming languages and the theory of computation, Minsky argued, had an excessive preoccupation with formalism:

> [1970] *To build a theory, one needs to know a lot about the basic phenomena of the subject matter. We simply do not know enough about these, in the theory of computation, to teach the subject very abstractly. Instead, we ought to teach more about the particular examples we understand thoroughly, and hope that from this we will be able to guess and prove more general principles.*
>
> [... The] *syntax* [of programming languages] *is often unnecessary. One can survive with much less syntax than is generally realized.* [...] *What is compiler for? The usual answers resemble "to translate from one language to another".* [...] *For the future, a more ambitious view is required. Most compilers will be systems that "produce an algorithm, given a description of its effect."*
>
> (Minsky, 1970)

2 Scholastics relied on books by renowned scholars, studying them thoroughly, learning the theories of the authorities to the letter. Scholastics' work relied solely on the books, not on developing their own theories.

Minsky's concern was related to the fact that "*there are many ways to formulate things and it is risky to become too attached to one particular form or law and come to believe that it is the real basic principle*" (Minsky, 1970). The underdetermination thesis in the philosophy of science states that there are indeed infinitely many ways to formulate a theory about a phenomenon. Minsky's concern is close to what I have decided to call *underrepresentation problem*: Given a number of models that all model and predict well different aspects of a phenomenon, but that all are flawed in some ways, how does one determine which model to choose? (Underdetermination and underrep – resentation are discussed in the second half of these lecture notes.)

A caution about detrimental dogmatism is reflected in Minsky's text. Minsky is anxious that too strong of an adherence to normal science (or too strong of a belief that the normal science of a par – ticular era is the culmination of progress) may blindfold the researcher from seeing alternative paths. In Minsky's opinion, concentrating on content instead of form would free people to "*build, in their heads, various kinds of computational models*" instead of wasting their resources on the form (Minsky, 1970). Rather than being a call for formality or preordained forms, Minsky's argu – ment is a call for intuitiveness, creativity, and ad hocness. The *content vs. form*-juxtaposition might be interpreted as a juxtaposition between *actively shaping* new ideas vs. *adapting* existing ideas into different contexts.

> NORMAL SCIENCE
>
> Normal science is one of the terms that Thomas Kuhn's book *The Structure of Sci – entific Revolutions* brought to light. The term refers to science that is done accord – ing to a commonly agreed set of scientific practices and theories that underlie those practices (a scientific paradigm). Scientists conducting normal science do not act – ively question the commonly accepted foundations of science but accommodate their findings to the dominant paradigm[3].

The concern Minsky had about programming languages is also quite an unorthodox one, given the year he stated it. Some researchers were quite happy with the state of programming at the time, and Peter Wegner, the former editor-in-chief of one of the major ACM publications, *Computing Sur – veys*, even wrote in *IEEE Transactions on Computers* 1975, that "*It may well be that programming language professionals did their work so well in the 1950's and 1960's that most of the important concepts have already been developed*" (Wegner, 1976b). Note that afterwards there have been de – velopments in web design languages, virtual machines, visual programming, design patterns, and so forth. Although some of those were developed as early as 1960s, they had not been included in Wegner's article. It is unlikely that by *most important concepts* Wegner referred to the *stored-pro – gram-paradigm* because the stored-program-paradigm was at place already in the late 1940s. (The stored-program paradigm *is* arguably the single most important concept defining programming today—see page 59 of these lecture notes)

It is difficult to see that programming languages would cease to change. The different styles of pro – gramming languages such as SQL, Java, Perl, and C++ arise from the application domains that they

3 See more about Kuhn's theory of scientific revolutions in Kuhn's book or, for instance, Alexander Bird's (2004) entry *Thomas Kuhn* in Stanford Encyclopedia of Philosophy, Wikipedia on The Structure of Scientific Revolutions, Frank Pajares' synopsis of Kuhn's book, or the whole chapter 9 of Kuhn's book online.

were created for (Denning, 2003). For instance, some application domains require languages that are capable of compiling small and fast executables, some require interpreted languages, some application domains require full platform-independence, and some require ease-of-programming. Insofar as programming languages are application-dependent, their change would cease when application domains would cease to change. Minsky was far-sighted enough to understand that the application domains of computing are not yet exhausted. Today the number of calls for new approaches to computing, such as empirical modeling, resilient systems, stochastic computation, and quantum computation, is as large as ever, and programming languages will have to keep track of the new developments.

2.1.5 Is the Focus Concrete or Abstract—Computer or Information?

The *International Federation for Information Processing* (IFIP) *World Conference on Computer Education* convened in 1970 in Amsterdam, the Netherlands, and accepted the term computer science as "*the study of computing machines (actual or potential)*" (Finerman, 1970). This definition could not have been even expected to make much difference. The definition does not comment on what subfields are included in the science, neither does it explain in a sufficiently exact way what is it that computer scientists do except for "study" (which is a verb that can be read very loosely). From almost any point of view, this definition suffers from excessive generalization.

Much more interesting than the IFIP recommendation committee's definition of computer science, is that the committee also defined a parallel term, *informatics*, for which the French had championed. Aaron Finerman wrote that inherent in the definition of informatics is that information processing *by computer* is *a syntactic process* involving symbol strings (much in the same manner Shannon defined information—see Shannon, 1948; Shannon, 1950), while information processing *by humans* is *a semantic process* involving word and phrase images (Finerman, 1970). Finerman wrote in the November 1970 issue of CACM:

> [1970] *Informatics is the science of the systematic and effective treatment (especially by automatic machines) of information seen as a medium for human knowledge and for communication in the technical, economic, and social fields.*
>
> (Finerman, 1970)

That is, informatics studies how information can be processed automatically (note the resemblance to the characterization of computer science by Atchison et al., 1968). It is unclear, however, whether the committee implied that computers might, at some point, be able to process information semantically (as humans do), or whether there is some sort of a fundamental dissimilarity between information as human knowledge and information as a machine-processable phenomenon. For instance, the father of information theory, Claude E. Shannon, noted that it is hardly to be expected that a single concept of information would satisfactorily account for the numerous possible applications of the theory of information (Shannon, 1950). In fact, Shannon's use of the term *information* is very specific—from Shannon's engineering viewpoint, the *meaning* of information was not relevant at all (Shannon, 1948).

The relationships between information, meaning, knowledge, signs, concepts, cognition, information processing, computing, and such, have deep philosophical roots and ramifications studied in the

field called *Philosophy of Information*, which "is concerned with (a) the critical investigation of the conceptual nature and basic principles of information, including its dynamics, utilization, and sciences, and (b) the elaboration and application of information-theoretical and computational methodologies to philosophical problems" (Floridi, 2002). In this course the philosophy of information is not discussed further, although some topics in this course may well be topics in the philosophy of information, too. For a collection of good introductory articles to the philosophy of information, see Floridi, 2004.

Even today, computer scientists from different regions of the world name and define the field of computing in different ways. Gal-Ezer and Harel wrote in the September 1998 issue of CACM:

> [1998] *In fact, there is no clear agreement even on the name of the field. In European universities, the titles of many of the relevant departments revolve around the word 'informatics', whereas in the U.S. most departments are 'computer science'. To avoid using the name of the machine in the title,* [...] *some use the word 'computing' instead. Other department names contain 'information systems' or 'computer studies'.*
>
> (Gal-Ezer & Harel, 1998)

There is indeed a variety of names for the field in languages other than English. For example, Finnish universities use the term *computer science* in texts in English, but the Finnish word for the discipline, *tietojenkäsittelytiede*, is translated word for word as "knowledge processing science". In Swedish the discipline is *datavetenskap* or *informationsteknologi*: "data science" or "information technology", respectively. In Dutch the term is *informatica*, which refers to the junction between information and automatic (processing). Similarly, in Russian, the basics of computer science is информатика—"informatics" or "information science"—but the field is divided into subfields and characterized in more specific terms such as вычислительная техника ("computer techniques" or "computer engineering"); the Russian term информатика seems to be used as broadly as the English term *computer science*.[4]

Gordana Dodic-Crnkovic has noted that interestingly, the British term *computer science* with its empirical orientation and the German and French terms *informatics* with their abstract orientation, correspond to the eighteenth- and nineteenth-century characters of British Empiricism and Continental Rationalism (Dodig-Crnkovic, 2002).

4 I wish to thank Dr. Alexander Kolesnikov, Dr. Piet Kommers, mr. Marcus Duveskog, and ms. Evgenia Chernenko for their help with computer science terminology in different languages.

RATIONALISM

Empiricists (see p.14) stand in contrast to *rationalists*, who believe in *reasoning* as a source of knowledge. Plato and Descartes, among others, held that what people know by reason alone is superior to experiential knowledge in a number of ways—it is unchanging, eternal, perfect, certain, and universal (e.g. Markie, 2004). Hence the name *rationalism*: rationalists believe that knowledge should be obtained by rational reasoning. Rationally deduced knowledge is also called *a priori* knowledge (Latin for "*from former*", because reasoned knowledge comes *before* having any experiences). Rationalists have often, however, noted that only the mathematical kinds of knowledge can be deducted using reason alone, but other sorts of knowledge need observation.

In computer science, the *rationalist tradition* can be seen when computer scientists do formal or theoretical analysis, based on mathematical techniques. For example, computer scientists are doing such research when they take two algorithms, analyze their time complexity functions, and, based on a comparison of the two functions, argue that one of the two algorithms is faster than the other one (with input sizes larger than n). Their argument rests on the mathematical *rigor of* their *analysis*.

In 1966, when Peter Naur suggested the term *datalogy* to replace the term *computer science* (Naur, 1966), he made a distinction between *data* and *information.* (One sense in which data and information can be understood in computing is that the term *data* refers to any symbols that can be processed with a computer, whereas information actually describes something (concrete or abstract).) Afterwards, Edsger Dijkstra 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'*." (Dijkstra, 1987). Dijkstra's suggestion for the name of the field was *computing science*. Instead of the computer, or computing technology, Dijkstra wanted to emphasize the abstract mechanisms that comput*ing* scientists use to master complexity. (Another decade later, contrary to Dijkstra's opinion, Frederick Brooks Jr. wrote that "*our namers got the 'computer' part exactly right*" (Brooks, 1996).)

In the 1970s there was a group of proponents of the algorithmic, rationalistic research tradition, and Edsger W. Dijkstra was among them. In the October 1972 issue of CACM, Dijkstra wrote:

> [1972] *We must not forget that it is not our* [computing scientists'] *business to make programs; it is our business to design classes of computations that will display a desired behavior.*
>
> (Dijkstra, 1972)

Dijkstra's viewpoint was a well-founded and original one (see Dijkstra's early writings in, e.g., Dijkstra, 1968b), and top-down methods similar to Dijkstra's were promoted by, for instance, Niklaus Wirth and Peter Naur (Wirth, 1971; Naur, 1969; Naur, 1972). In the 1970s proofs of program correctness were usually written *a posteriori*, that is, after the program had been written (Dijkstra, 1975). Instead of a posteriori proofs, Dijkstra suggested that one should start designing classes of computations by formally (mathematically) specifying what the program should do, and then gradu-

ally *deriving* the actual program from these specifications. This resembles the mathematical tech – nique of *proof by construction*. Since derivations are mathematical transformations between two systems, if the rules for transformations are correct, the resulting program was argued to be neces – sarily correct (bug-free). From a mathematical point of view, Dijkstra's approach is durable, but it never gained much support outside minor coteries in academic circles.

> SOURCES OF BUGS
>
> Donald Knuth often has been said to have written a memo that ended with a sen – tence "*Beware of bugs in the above code; I have only proved it correct, not tried it.*" (See Knuth's web page). This seemingly innocent quotation raises an important point—human errors can occur in any part of the program-construction process. Proving a program correct means neither that the proof was correct, that the actual program corresponds to the proven function (that is, that the translation from proof to program was done correctly), nor that the formal specifications correspond to the intended purpose of the program.

It has been argued that the name of the discipline is important because the name of the field situates the field among other disciplines, creates the expectations set for it, guides the areas in which it can be applied, and affects the choice of problems for the field (Brooks, 1996). Certainly names like *knowledge processing science*, *computer engineering*, *informatics*, *datalogy*, and *computer science* arouse different connotations.

2.1.6 Relating Computer Science With Mathematics

Physics is often called the king and mathematics the queen of sciences (sometimes the other way round), but it is uncertain whether computer science belongs to the royal family in this portrait. In the 1970s, many mathematicians who saw *infinity* as a prerequisite for mathematical depth, saw computers, which are finite and discrete machines, only as number crunchers, or perhaps as tools in numerical analysis (cf. Dijkstra, 1987). Hence, many mathematically oriented computer scientists thought that the best legitimation of the field of computing as an academic discipline would have been to gain the recognition of the mathematical community. Donald Knuth, who is not only a computer scientist but also a recognized mathematician, saw the theory of computing as a subject worthy of study as such. In the April 1974 issue of *American Mathematical Monthly* he wrote:

> [1974] *Like mathematics, computer science will be somewhat different from the other sciences, in that it deals with man-made* [sic] *laws which can be proved, instead of nat – ural laws which are never known with certainty.* [...] *The difference* [between mathem – atics and computer science] *is in the subject matter and approach—mathematics deal – ing with more or less with theorems, infinite processes, static relationships and com – puter science dealing more or less with algorithms, finitary constructions, dynamic re – lationships.*
>
> (Knuth, 1974)

Knuth's article can be divided into two parts: In the first part he noted the similarities between com – puter science and mathematics. In the second part he sketched the differences between the two.

When Knuth sketched the similarities between computer science and mathematics, Knuth's position towards the mode of existence of *laws of computer science* is not clear. The wording "*deals with* [hu]*man-made laws*" implies that Knuth believed that the laws in computer science exist because there are intelligent beings that create and maintain them. That is, laws in computer science are human creations. On the other hand, *human-made* may also refer to a human-made symbol-theory system that describes universal and timeless (not human-made) facts, but in this case Knuth might not *contrast* computer science with natural sciences. After all, that is exactly what natural scientists do: they create symbol-theory systems that describe universal and timeless laws of nature. Note also that Knuth used the term *laws* instead of *constructions*, *theories*, or *hypotheses*.

Knuth also wrote that laws in computer science "can be proved". That laws can be proved indicates that Knuth took there to be a universal logic, following which, any independent thinker would proceed from the same axioms and premises to the same conclusions. Underlying Knuth's comparison of computer science with empirical sciences there is an important claim: natural sciences are uncertain compared to computer science. From this point of view, the laws of computer science are facts that researchers have constructed to perfection within the bounds that the researchers have set—hence the factuality of the laws of computer science compared to the imperfect models and theories of the unbounded external world made by natural scientists.

It seems that the mode of existence of Knuth's computer science is not fixed—the facts that computer science deals with exist because there are thinking creatures that make them exist. Yet it seems that Knuth took these facts to be objective facts in the sense that they are not matters of preferences, evaluations, or moral attitudes (*cf.* Searle, 1996:1). Questions such as this are dealt with later in this course, in the context of epistemology and ontology in the field of computing.

Knuth distinguished computer science from mathematics also by subject matters and approaches. Computer science deals with finite constructions that are characterized by dynamic relationships, and uses algorithms to deal with these dynamic relationships. Knuth listed three characteristics of computer science that differentiate it from mathematics.

- First, *finiteness* (of space, but not of time!) is a prerequisite of realizability. That is to say, infinitely large constructions cannot be realized with computational instruments (yet, although infinitely long computations can be set up, they most probably do not live up to the expectation and run forever).
- Second, *dynamic relationships* are a prerequisite for modeling the dynamic world.
- Third, *algorithms* shift the focus from static models towards processes or automation.

These three characteristics of computer science imply that computer science cannot escape the limits of the actual computational instruments and cannot abstract away the rich dynamism of the real world. In Knuth's version of computer science, algorithms and models link real-world processes and phenomena with computational instruments.

Only two months after Knuth's article was published, Dijkstra also wrote, in the same journal, about the differences and similarities between mathematics and programming; Dijkstra called programming "an activity of mathematical nature" (Dijkstra, 1974). Dijkstra pointed out three characterist-

ics of the mathematics curriculum that are possibly detrimental to the cognitive skills needed by programmers:

(1) standard collection of concepts vs. concept-creating skills;

> [1974] *In the standard mathematical curriculum the student becomes familiar (sometimes even very familiar!) with a standard collection of mathematical concepts, he* [sic] *is less trained in introducing new concepts himself.*

(2) learning standard notation vs. ad-hoc notation (note that ad hoc is considered positive in this sense, unlike the falsificationist sense);

> [1974] *In the standard mathematical curriculum the student becomes familiar (sometimes even very familiar!) with a standard set of notational techniques, he* [sic] *is less trained in inventing his own notation when the need arises.*

(3) and shallow hierarchy vs. deep hierarchy;

> [1974] *In the standard mathematical curriculum the student often only sees problems so "small" that they are dealt with at a single semantic level. As a result many students see mathematics rather as the art of organizing symbols on their piece of paper than as an art of organizing their thoughts.*
>
> (Dijkstra, 1974)

Dijkstra's first observation was that programmers are expected to be able to express themselves in both natural language *and* in formal systems—which, together, should lead to skills in concept creation. In the process of problem solving in computing, Dijkstra wrote, programmers often come up with intermediate concepts and may need to construct their own formalisms. This might contradict the impression of problem solving as straightforward puzzle-solving. Dijkstra held that although concept creation can be found in mathematics, it is especially characteristic of computer science. It seems that concept creation in computing is in effect the continuous (re-)creation of conceptual and modeling schemes.

It also seems that when programmers approach problems through computational tools, they are not only (re-)defining relationships between a number of semantic levels within computing, but also between the semantics of computing and the obscure semantics of the world that the computers are a part of. Computer scientists semantically *relate* computers with their surroundings, and they do this via *models* of the world.

In connection with his third comment, Dijkstra noted that programmers have to be able to move between time grains of nanoseconds (one clock cycle) to time grains of hours (whole computations), and all the levels between. The ratio between these time grains can easily be 10^{10}—being able to move and operate between this many orders of magnitude helps in mastering complexity.

Knuth noted that computer science deals with finitary constructions and dynamic relationships (Knuth, 1974), and Dijkstra's second comment implies that programmers (dynamically) re-create, or at least re-model, these relationships and concepts as they work. Dijkstra wrote that programmers

often need to manipulate a given formal syntax in order to formulate a theory to justify their al – gorithm—which leads to the ability to invent one's own formalisms.

That is, Dijkstra argued that often programming is, in effect, formulation of theories. Of course, Dijkstra's approach to programming was different than many other approaches at the time; it was not trial-and-error, or program-first-proofs-then but it was a mathematical "developing program and proof hand-in-hand" activity. Taking this approach into account, computer science, as seen by Dijkstra and Knuth, still seems like an extraordinarily dynamic field.

If programmers truly are constantly reshaping the *conceptual framework* (constructions) of com – puter science, if they are looking for new ways of *applying computing* (relationships), and if they are often developing *new theories* (tools) for the field, dynamism is not only an integral part of the *constructions* of computer science but also an integral part of the autopoïetic *construction* of com – puter science. Dynamism is not an integral part of *relationships* of computations but it is also an in – tegral part of *relating* computer science to its surroundings.

2.1.7 Tug of War Between the Theoretical and the Practical

Dijkstra's and Knuth's writings were addressed to the mathematical community, which at the time often did not recognize the disciplinary identity of computer science. Elsewhere, the pressures of the business world were building up on computer science. At the same time as some leading com – puter scientists worked to gain the recognition of the mathematical community, proponents of the practical, "real-life" aspects of computer science worked to close the gap between the theoretical and practical sides of computer science. They claimed that the academic computer science had de – tached from the needs of the "real world", and that academic computer science had very little to do with actual computers and computer systems (Kandel, 1972). Even more, some questioned the the – ories of computer science. Abraham Kandel wrote in the June 1972 issue of CACM:

> [1972] *Industry gets graduates from computer science departments with a bag full of the latest technical jargon but no depth of understanding the real computer systems and no concept of the problems they will be asked to solve.*
>
> [...] *It is quite obvious that there is no effective theory of computer science as such. In fact, there are no effective models of computers.*
>
> (Kandel, 1972)

Naturally, "effective" is an ambiguous concept, and without stating what *would* be an effective the – ory of computer science, opinions such as this remain as remarks. Be that as it may, comments such as the one above became increasingly common. The demands of adding practical training and lab work to the university curriculum led to an increased interest in software oriented or commer – cially oriented computer science programs (Pitts & Bateman, 1974), which began to shape what computer science was in practice. That is, the needs of the software industry guided the interests of researchers and teachers, which showed in what was taught in universities.

George Glaser, the president of AFIPS (*American Federation of Information Processing Societies, Inc.*) addressed the 10th annual meeting and conference of the *Inter-University Communications Council* (Educom):

> [1974] *A formal education in computer science is not an adequate—not even appropriate—background for those who must design and install large-scale computer systems in business environment.* [...] *The educational system is providing nicely for a body of competent computer researchers and teachers but has done little to provide for the needs of those who must apply computer technology, particularly in a business environment.*
>
> (Glaser, 1974)

Since computer systems by the 1970s had grown—and kept on growing—in size, they were increasingly unmanageable without new approaches (Sommerville, 1982:*v*; Campbell-Kelly & Aspray, 2004:176-180). This was due to the fact that large computational systems are unities of structured elements, irreducible to the sum of component elements and structure (Spier, 1974). (Because the complexity in a system arises not from the parts of the system, but from the semantical and functional connections between these parts, a system is more complex than the sum of its parts.) The resulting problem was called the *software crisis* (Campbell-Kelly & Aspray, 2004:176-180). Academic computer science of the time was accused of being unable to respond to the software crisis—for instance, Michael J. Spier wrote in the *ACM SIGOPS Operating Systems Review*:

> [1974] [...] *our current computer science, as it applies to operating systems, does not provide us the necessary foundation in terms of which we would understand and control programs of significant complexity.* [...] *The applicability* [of formal computer science disciplines] *to the definition and global design of operating systems is close to nil.*
>
> (Spier, 1974)

This ineptitude led to a lack of university graduates who were able to "*work effectively in all phases of the development of large software systems*" (Egan, 1976)—the interests and expertise at industry did not meet the interests and expertise at academia. This problem concerned both industry and computer centers of academia because neither was able to maintain their ever-growing computing systems. Although there had been studies of complex systems, such as Herbert A. Simon's *The Sciences of the Artificial* (Simon, 1981, orig. 1969), there was a lack of rigorous methodology for mastering complex systems. As a response to growing complexity of computer systems, around the mid-1970s the s*ystems approach* (systems analysis or systems engineering), which had began emerging as a field around 1950s, began to gain popularity among computer scientists.

2.1.8 Software Engineering: A Response to the Software Crisis

Systems engineering had started developing when new tools and machines had become so complex that it was no longer possible for a single individual to design and maintain them (Sage, 1992:6). In systems engineering the life cycle process, high complexity, and the application of system engineering techniques are recognized throughout the project life cycle (Shemer, 1987). In computer science, the systems approach relies on scientific principles of, for instance, life-cycle management, performance evaluation, quality assurance, and rigorous testing, rather than on algorithmic solutions (Spier, 1974; Egan, 1976). A more specific term *software engineering* was first introduced in 1968 at a conference held to discuss the software crisis (Naur & Randell, 1969). Although software en-

gineering can be considered to be a sub-area of systems engineering in general, it has also contrib – uted to the development of systems engineering.

From a sociological point of view, note that systems engineering was a response to the problems that arose when the complexity of systems exceeded the skills of a single person. Now, if one wishes to explain the assumptions, beliefs, preferences, and decisions that the designs of a system may incorporate, the emergence of systems engineering signifies a shift from explaining individuals to explaining groups. That is, when one wants to explicate the intentions behind building a complex system, collective intentions and perhaps multiple intentions need to be catered for. The intentions and motivations for constructing the system may be heterogeneous and conflicting.

Ian Sommerville's characterization of software engineering, in his 1982 book *Software Engineering*, reveals the crux of the heated 1970s debate. The debate was about whether software engineering should be considered to be a branch of computer science:

> [1982] *Software engineering is a practical subject. It is concerned with building us – able systems economically and, to this end, utilises appropriate, rather than fashion – able, techniques. The software engineer should be conservative—he* [sic] *cannot af – ford to experiment with each and every new technique put forward by research scient – ists. His principal responsibility is to produce a working system to specification, on time and within budget, and the use of untested methods might compromise this inten – tion. On the other hand, he or she should not ignore new developments nor should they be rejected simply because they are new.*
>
> (Sommerville, 1982:2-3)

Sommerville's quote helps to explain why software engineering was rejected by theoretical com – puter scientists, programmers, and hardware specialists. The quote begins with a notion that soft – ware engineering is a practical subject that contrasts with the theoretical, mathematically oriented aspects of computing. The focus on practical goals probably made it harder for software engineers to gain the acceptance of mathematically oriented theoreticians. According to Sommerville, soft – ware engineering also concerns human issues, namely usability—which juxtaposes software engin – eering with the (practical) electronic engineering and programming aspects of computer science. Sommerville also emphasized the economic motivation, which distances software engineering even further from the mathematical sciences, towards business goals.

Sommerville continued with a relativist argument about appropriate techniques. Substantially, any – thing goes if it gets the job done. Sommerville's notion of conservativeness and the disassociation from untested methods differentiates techniques of software engineering from techniques of *re – search*. The key words in Sommerville's definition of software engineering are production, opera – tionality, time frame, and budget—not, for instance, theory-creation or fact-finding.

The emphasis of Sommerville's definition suggests that a software engineer should be a *bricoleur*, a jack-of-all-trades, an opportunist who has a toolbox full of different tools that help him or her to ac – commodate to a variety of situations with a variety of methods (see, e.g., Denzin & Lincoln, 1994:2-3; Horgan, 1996:52; Lévi-Strauss, 1966:16-17). The computer scientists of the 1970s scarcely would have appreciated the bricolage approach to research. (Ironically, since the term *bri* –

coleur was originally conceived to refer to the opposite of an engineer, the term engineer-as-a-bri – coleur seems to be an oxymoron.)

> RESEARCHER-AS-A-BRICOLEUR
>
> Bricoleur: from French *bricoler*: trifle, tinker. Some qualitative researchers have used the term *researcher-as-bricoleur* to refer to a researcher who is familiar with a variety of schools of research (or paradigms), but also knows that paradigms cannot be easily synthesized. The term bricoleur was originally used to describe the op - posite of an engineer—an engineer creates and uses specialized tools for special - ized purposes, whereas the *researcher-as-bricoleur* is knowledgeable with and works between and within competing and overlapping paradigms (Denzin & Lin - coln, 1994:2-4). The term comes originally from anthropologist Claude Lévi-Strauss, although he used the term in a negative sense (see Lévi-Strauss, 1966:16-17). The term has been used also in a somewhat negative connection with com - puter science: "*the manifestation of bricolage in computer science is endless de - bugging: Try it and see what happens*" (Ben-Ari, 2001; see also Ben-Ari, 1998).

If software engineering is a practical, opportunistic, business oriented, and close-to-human work, exclusive of research aspect, as it is described by Sommerville, then it is easy to see why the com – puter scientists of the era wanted to dissociate from software engineering. After all, in the early his – tory of computing machinery, computer science had been considered to be a second-class science, mainly because of its practical bent and the impression of computing as a service operation (Aspray, 2000). Computer scientists might have been afraid that acknowledging software engineering as an integral part of computer science could have undermined whatever status computer science had achieved as a scientific discipline.

Sommerville's characterization may not be representative of the era's software engineering in gener – al. (Note that there was no consensus about the definition of software engineering at the time). Nonetheless, Sommerville's definition offers a good example of the different levels of confrontation between what was at the time considered to be computer science and what was considered to be software engineering. This confrontation was sometimes heated and it did not die out easily; for in – stance, in 1989 Dijkstra wrote that software engineering, "*The Doomed Discipline*", had accepted as its charter, "*how to program if you cannot*" (Dijkstra, 1989).

2.1.9 The Focus Turns to Programming

In the previous sections it was noted that in the early days of computing, programming was not con – sidered worthy of the university stamp—neither by mathematicians (Aspray, 2000) nor by serious computer scientists (Atchison et al., 1968). But before the beginning of the 1980s programming had become established as an integral part of the discipline of computing. Consider the following definition of computer science from a B.Sc program in computer science, introduced by Khalil and Levy in *ACM SIGCSE Bulletin*:

> [1978] *Our (first order) definition is that computer science is the study of the theory and practice of programming computers. This differs from the most widely used definition by emphasizing programming as the central notion and algorithms as a main theoretical notion supporting programming.*
>
> (Khalil & Levy, 1978)

Although Khalil and Levy are mathematically oriented computer scientists, their definition directly contradicts Knuth's view that algorithms are "*the central core of the subject and the common denominator which underlies and unifies the different branches*" (Knuth, 1974). (Remember that similar to Khalil and Levy, also Knuth's background is in mathematics.)

Khalil and Levy's foregrounding of programming derives, first, from their idea that programming is to computer science what the laboratory is to the physical sciences, and, second, from the context of their definition—in their article Khalil and Levy introduced a graduate program in computer science. Very much contrary to Dijkstra's opinion, they believed that one cannot conceive of an understanding of computer science without having experience in programming (Khalil & Levy, 1978). Khalil and Levy's position clearly reflects an empiricist *a posteriori* tradition rather than a rationalist *a priori* tradition and it focuses on *techniques and instruments* rather than on *discovering and proving laws* or designing classes of computations (like Dijkstra's computing science; see Dijkstra, 1972). Khalil and Levy's focus on programming may come from the needs of software engineering during and after the software crisis: Khalil and Levy wrote that the graduates of their program are expected to have an "acceptable" level of *professional expertise*. Professional expertise in Khalil and Levy's definition is mainly programming expertise.

In 1968, the ACM had published a recommendation for a four-year program in computer science, curriculum guidelines which later came to be known as *Curriculum '68* (Atchison et al., 1968; the report in which the curriculum was published is known as the *'68 report*). The curriculum guidelines encouraged university computer science departments to drop electronics and hardware courses in favor of mathematics and algorithms courses (Ensmenger, 2001). It has been argued that Curriculum '68 served as a "*fundamental source document for the establishment of computer science education in the United States*" (Austing et al., 1977). However, it has been also claimed that Curriculum '68 included little of interest to employers and business practitioners, particularly when compared to alternative curricula advanced by the IEEE or the DPMA (Ensmenger, 2001; Wishner, 1968; Hamming, 1969). Critics such as Raymond Wishner and Richard Hamming wanted to see more practical topics than theoretical topics included in Curriculum '68—they wrote in CACM and JACM:

> [1968] *Good education should not be solely directed towards academicians whose only economic justification is to teach in order to turn out recursively new generations of academicians.*
>
> (Wishner, 1968)

> [1969] *Were I setting up a computer science program, I would give relatively more emphasis to laboratory work than does Curriculum '68, and in particular I would require every computer science major, undergraduate or graduate, to take a laboratory course in which he* [sic] *designs, builds, debugs, and documents a reasonably sized program.*
>
> (Hamming, 1969)

Both comments criticize the lack of emphasis on practical issues of computing. In addition, Wishner argued that the Curriculum '68 addresses the needs of physical scientists and engineers, but that it does not address the needs of business-systems designers and information technologists. Even though the ACM did recognize the growing importance of meeting business needs to the future of computing, the emphasis of the ACM was on research and education (Ensmenger, 2001). It was argued, for instance, that the ACM wanted to see fundamental research in the field of data processing before the ACM would recognize business data processing as a topic of research (Postley, 1960).

About ten years after the ACM had published Curriculum '68, the ACM Curriculum Committee on Computer Science published an update to Curriculum '68: *Curriculum '78: Recommendations for the Undergraduate Program in Computer Science* (Austing et al., 1979; the report in which the curriculum was published is known as the *'78 report*). During the ten years between 1968 and 1978, the common understanding of computer science had expanded from a mathematically oriented discipline to a diverse, interdisciplinary discipline. During that decade there were major advances also in the theoretical aspects of computer science, such as the theory of computation, algorithm analysis, and in principles and theories for the design and verification of algorithms and programs. At the turn of the 1980s many computer scientists, like Anthony Ralston and Mary Shaw, believed that "*there is nothing laughable about calling computer science a science* [anymore]" (Ralston & Shaw, 1980).

The '68 report and '78 report seem to have both normative and descriptive characteristics in the sense that they recommend what a computer scientist should know by listing what computer scientists do. However, Goldweber et al. argued that soon after the '78 report, the curriculum recommendations became descriptive (Goldweber et al. called it "reactive"; see Goldweber et al., 1997). Similar to the '68 report, the '78 report defines the discipline by listing the topics included in the discipline. The '78 report, however, does not define the discipline as strictly as the '68 report, because the authors of the '78 report recognize that the report "*is a set of guidelines, prepared by a group of individuals working in a committee mode*" (Austing et al., 1979). In the report, the committee remarked that they did not expect the report to satisfy everyone or intend it to be appropriate for all institutions.

The '78 report is not just *any* curriculum proposition—it was directed at the whole academic field of computing. The '78 report was an effort of a large number of recognized individuals and institutions (Austing et al., 1979), and, as such, it had certain authority. However, the authors of the '78 report were not able to create a strict definition of computing as a discipline, but they had to leave a lot to interpretation, and the authors explicitly noted the subjectivity of the curriculum committee and the different aims of each educational institution. Whereas the aim of the '68 committee was to specify a number of course combinations that entitle a student to receive a degree in computer sci-

ence (Atchison et al., 1968), the objective of the '78 report committee was to "*stimulate computer science educators to think about their programs*" (Austing et al., 1979). The difference between the motivations and the content of the '68 report and the '78 report is a sign of the dispersion of the discipline—it was no longer possible to define what a properly recognized computer scientist actually knows and does.

In both the '68 report and the '78 report, computer science is divided into subareas. The '68 report divides computer science into three subareas;

(1) information structures and processes,

(2) information processing systems, and

(3) methodologies.

The '78 report report divides computer science into four subareas;

(1) programming topics,

(2) software organization,

(3) hardware organization, and

(4) data structures and file processing.

Note the lack of a distinct subarea, *theoretical foundations*, that some computer scientists might consider important. In the '68 report, models of computation is a subtopic of its own, but not in the '78 report. Theoretical topics, such as grammars, automata, and complexity, in the '78 report are scattered amongst other topics. Other subsequent curricula, that have been published in about ten-year intervals, follow the same convention (see, for instance, the nine subject areas of Curriculum '91 in Tucker et al., 1991). Curriculum developers probably have considered theoretical topics so central to computing that they need to pervade the whole curriculum.

What is also noticeable in the '78 report—compared to the '68 report—is the emphasis on hands-on work. The authors of the '78 report wrote, "*throughout the presentation of the elementary level material, programming projects should be assigned*" (Austing et al., 1979). The emphasis on programming is even more visible in the description of *philosophy of the discipline* in the '78 report:

> [1978] *A specific course on structured programming or on programming style, is not intended at the elementary level. The topics are of such importance that they should be considered a common thread throughout the entire curriculum and, as such, should be totally integrated into the curriculum. They provide a philosophy of discipline which pervades all of the course work.*
>
> (Austing et al., 1979)

Over the course of ten years, computer science in the ACM curriculum turned from a theoretical, mathematically-based discipline that studies information structures into programming and applications-centered discipline. Although the topics in the '68 report and '78 report are quite similar, the focus had definitely shifted between the '68 report and the '78 report. Even the "philosophy of discipline" had changed from information structures into structured programming and programming

style. It is not a coincidence that the computer science curriculum, including the philosophy of the discipline, shifted towards the needs of the software industry during the years of software crisis. But because the '78 report is more of a descriptive than a normative account of the computing field, it seems that the curriculum committee was just trying to keep pace with the changes in computing practices.

2.1.10 Separation from Mathematics

Whereas the critics of the '68 report criticized the '68 report for being too academic, too theoretical, too narrow, and too impractical, the critics of '78 report criticized the '78 report for lacking mathem - atics and for implicitly stating that "*computer science = programming*" (Ralston & Shaw, 1980). The difference between the emphasis on mathematics in the two reports is indeed notable. Whereas the authors of the '68 report stood firmly behind the mathematical viewpoint of computing, the au - thors of the '78 report did not see mathematics anymore as the cornerstone of computer science. The following two quotes from '68 report and '78 *preliminary* report exemplify the shift of focus well:

> [1968] *The committee feels that an academic program in computer science must be well based in mathematics since computer science draws so heavily upon mathematical ideas and methods.*
>
> (Atchison et al., 1968)
>
> [1977] [...] *no mathematical background beyond the ability to perform simple algebra - ic manipulation is a prerequisite to an understanding of the topics* [...] *As was men - tioned in the section on the core curriculum, mathematics is not required as a pre - requisite for any of that material.*
>
> (Austing et al., 1977b)

However, the advocates of mathematically based computer science (see, e.g., Ralston & Shaw, 1980; Davis, 1977) succeeded to change the above mentioned part of the '78 preliminary report, and the final '78 report included a more conventional wording:

> [1978] *An understanding of and the capability to use a number of mathematical con - cepts and techniques are vitally important for computer scientist.*
>
> (Austing et al., 1979)

The leap from a science that is "*well based in mathematics*" to a science where "*no mathematical background beyond the ability to perform simple algebraic manipulation is needed*" would have marked a complete detachment from the mathematical history of computing. The final wording was much more conventional, mentioning the *vital importance* of a number of mathematical con - cepts and techniques. As computer science began to achieve a a disciplinary identity, the institu - tional ties between mathematics and computer science weakened steadily. In the August 1981 issue of *American Mathematical Monthly*, Anthony Ralston explained the reason for the divergence of the two disciplines during the 1970s with four arguments (Ralston, 1981).

First, the importance of some of the traditional mathematical areas, such as numerical analysis, de – creased in computer science. (Note that the history of numerical analysis is hundreds of years old and that computers are mostly tools for numerical analysis and not a topic of research in numerical analysis.) This is perhaps best described as a simple shifting of research interests.

Second, Ralston argued that the difficulties that computer science had faced in being recognized as a separate discipline from mathematics urged many computer scientists to fight for the formation of Departments of Computer Science separate from Departments of Mathematics. This implies that the separation between mathematics and computer science was partly a result of professional pride.

Third, Ralston noted that beginning from the early 1970s, the composition of computer science fac – ulties begun to shift from predominantly mathematicians to predominantly computer scientists. This shift is not surprising. If the first PhD-granting department of computer science was founded 1962 and the first PhD was awarded 1966, there could not have been many computer science PhDs around to fill the faculty positions during the early 1970s. As the number of PhDs in computer sci – ence grew, it seems logical that the departments of computer science were increasingly able to em – ploy computer scientists instead of mathematicians.

Fourth, Ralston argued that people at mathematics departments were not very hospitable to the ideas and techniques of computer science. Ralston criticized the "computer science = program – ming"-outlook, and spoke up for mathematics-based computer science. With Mary Shaw he wrote, "*Inevitably, for any science or any engineering discipline, the fundamental principles and theories can only be understood through the medium of mathematics*" (Ralston & Shaw, 1980).

In short, Ralston gave four reasons for the field of computer science's divergence from the field of mathematics: Disciplinary changes, an insecure disciplinary identity, a growing number of people in the field of computing, and disciplinary disagreements. Only the first one of Ralston's four argu – ments actually concerns academic and intellectual aspects of computer science. The other three ar – guments are perhaps better explained in psychological, sociological, or anthropological terms. For instance, Ralston's second argument can be explained as a resistance towards an old adversary; Ral – ston's third argument can be explained as a growth of the number of computer scientists and their placing in working life; and Ralston's fourth argument can be explained as a difference between the cultures of abstractly oriented mathematicians and practically oriented computer scientists.

Throughout the 1970s, descriptions and definitions of computer science increasingly came to in – clude practical issues. Computer science, as such, was claimed to be a legitimized, mature discip – line (Ralston & Shaw, 1980), and the legitimization pressures moved increasingly to subareas of computer science (such as software engineering; see, e.g., Pour et al., 2000; Holloway, 1995). Yet, during the years between the early 1960s and the early 1970s the ones who worked with computers were mostly professionals in computer programming. Jonathan Grudin wrote that through the 60s and mid-70s, improving *usability* still meant improving *programmer efficiency* (Grudin, 1990). But during the 1970s, a significant change in computing took place. This change, which had a signific – ant impact on both the form and the content of the field, was due to changes in the user base: The computer broke out of the laboratory; it became available to Western office workers and to the gen – eral public.