The Development of Computer Science: A Sociocultural Perspective

Matti Tedre
University of Joensuu
Department of Computer Science and Statistics
P.O.Box 111, 80101 Joensuu, Finland
matti.tedre@cs.joensuu.fi

ABSTRACT

Computer science is a broad discipline, and computer scientists often disagree about the content, form, and practices of the discipline. The processes through which computer scientists create, maintain, and modify knowledge in computer science—processes which often are eclectic and anarchistic—are well researched, but knowledge of those processes is generally not considered to be a part of computer science. On the contrary, I argue that understanding of how computer science works is an important part of the knowledge of an educated computer scientist. In this paper I discuss some characteristics of computer science that are central to understanding how computer science works.

Keywords

Social studies of computer science; social issues; meta-knowledge in computer science

1. INTRODUCTION

Computer science is often taught in modules that dissect the discipline neatly into separate segments. Many of those segments seem quite unconnected—think about such ACM/IEEE computing curriculum's core topics as discrete structures, operating systems, human-computer interaction, and programming languages [8]. Although each core topic has a history of its own, if one takes a closer look at the history of computing it seems that each of the core topics in computing has gone through a number of stages before gaining a status of a core topic. In many cases those stages have included disdain and outright rejection, gradual maturing and academic approval, and finally a recognition as an important and integral part of computing field. In fact, every single core topic in the ACM/IEEE curriculum has been contested by someone, at some moment of time. For instance, pioneers of computing went to great lengths arguing that the theoretical parts of computer science are not a branch of mathematics but a central part of a new discipline called computer science [10,15]. The academic status of many technical things, such as programming, was still rejected in the ACM curriculum '68 [2].

Portraying a holistic view about the intellectual origins and about the development of branches of computing should offer computer science students a better understanding of the discipline than teaching the discipline as a collection of loosely connected islands. A disciplinary understanding requires understanding not only *what* computer science is and *how* its subjects are researched, but also *why* computer science is what it is and how did it get its current form. Learning that there have been methodological, epistemological, and other kinds of intellectual disagreements throughout the whole existence of computer science might give computer science students a perspective into the debates about computer science today. In this paper I outline a number of features in the development of computing as a discipline; features that may shed light on current debates, too: growth of technological momentum, the mangle of practice, and an anarchistic view of science. It does not matter whether one likes those characteristic of computer science or not, understanding those characteristics is an important part of understanding computer science [23].

2. TECHNOLOGICAL MOMENTUM

Historians of computing quite unanimously agree that the circumstances in which the stored-program paradigm was born were highly contingent. Those circumstances were brought together by the political situation, the war effort, advancements in science and engineering, new innovations in instrumentation, interdisciplinarity, influential individuals, coincidences, a disregard for costs, and a number of other sociocultural factors [5,7,20]. Many other development steps in modern computer science—such as the birth of high-level programming languages—have also been attributed to a number of influential sociocultural factors [3,4,22]. Computer science, as we know it, is a sociocultural construction that has been born, nurtured, and raised in a certain society. Today many concepts and developments in computer science, such as the stored-program paradigm and high-level programming languages, have become a part of the relatively stable core of computer science. At the same time, they have lost their human character and they are even sometimes perceived as something other than human constructs.

Thomas Hughes, who is a historian of science, has used the term *technological momentum* to refer to the tendency that young technological systems exhibit characteristics of social construction, but the more widespread and more established technological systems become, the more characteristics of technological determinism they exhibit [14]. In other words, the creation and recognition of newly-born innovations is a collective and subjective process, but usually the older and more prevalent innovations become, the more rigid, less responsive to outside influences, and thus more deterministic by nature those innovations become.

For instance, the stored-program paradigm (the constellation of innovations that surround the stored-program architecture) has gained enough technological momentum that it is today largely taken as an unquestioned foundation—even a sine qua non—of successful automatic computation (which might not be what the pioneers had in mind—for instance, the objective of Turing in his famous 1936 paper [25] was not to explain the limits of machine computation, but to specify the simplest machine that can perform any calculation that can be performed by a human mathematician who has unlimited time, and who works with paper and pencil in accordance with some “rule-of-thumb” or rote method [6]).

The growth of technological momentum can be seen throughout the history of modern computing technology, too. In the early days of electronic digital computing, most computing machines of the time differed from each other in their architecture, design, constraints, and working principles. Over the course of time knowledge about the directions of computing accumulated, systems became more interdependent, and computing researchers and computer manufacturers increasingly followed traditional paths which others had tread. The first united architecture, the IBM System/360, marked a definite turning point in the change from social constructionism to technological determinism in computing machinery. Similar, the early construction of FORTRAN was directed by sociocultural and personal motivations, as well as economical and institutional considerations; but the more FORTRAN developed and the wider it spread, the more it institutionalized and the more rigid it became [21].

No matter how monumental some things in computing (such as the stored-program paradigm and high-level programming languages) seem today, there was a time when they were opposed by the scientific community. For instance, many authorities in the scientific establishment resisted ENIAC, which is now considered to be one of the most important milestones in the history of electronic computing [5,11,16,27]. After the construction of ENIAC and EDVAC there was still great uncertainty about the research directions and paradigms of electronic computing, and the first twenty years of electronic computing saw a great variety of fundamentally different competing technologies [27]. Similar, the first programming languages were shunned by the computing establishment but eventually some people, most notably Grace Hopper and John Backus, succeeded in selling the concept of programming languages to administrative, managerial, and technical people by arguing that programming languages would result in economic savings [3,4,22].

3. THE MANGLE OF PRACTICE

Although the growth of technological momentum in computing fields is quite visible in a number of examples, nothing has yet been said about the mechanisms of that growth. The growth of technological momentum in computer science does not work in any straightforwardly deterministic manner, and the growth of knowledge in computer science does not work according to any single philosophy of science. There are neither rigid rules that computer scientists would always stick to, nor a commonly held understanding of what computer science properly is. Rather, computer science is a logico-mathematical and technological enterprise driven by a diversity of needs, aims, agendas, and purposes; it is constructed and maintained in a complex mesh of institutions, social milieux, human practices, economical concerns, agenda, ideologies, cultures, politics, arts, and other technological, theoretical, and human aspects of the world.

Technological and intellectual changes in computer science can be reflected against Andrew Pickering's concept *mangle of practice* [18], which describes the development of science and technology as an ongoing cycle of development and revision of (1) theories and models, (2) the design and theory of instruments and how they work, and (3) the instruments themselves. When a computer scientist works, usually things do not go as planned—the world resists. He or she accommodates to this resistance by revamping some or all parts of the theoretical-technological structure, and tries again. In the end, the computer scientist hopes to get a robust fit between the theoretical and technological elements of a research study. In addition, researchers often need to accommodate for sociocultural factors, too—technoscience is not developed in a vacuum but within a dynamic network of societal, economic, cultural, institutional, ideological, political, philosophical, and ethical factors. That is, what we build and develop is in numerous ways guided by non-technical considerations and influences.

The birth of the stored-program paradigm is a prime example of the mangle in computing. The researchers who finally came up with the stored-program paradigm had their theories of computation, their prototype machines, and their theories of how their machines should work. They had to deal with numerous dead-ends; had to devote considerable effort to developing peripheral components, test equipment, and component technologies; had to revise their theories and concepts often; and had to spend a great deal of effort convincing other stakeholders about the value of computing [1,5,27]. Through numerous accommodations; such as revisions of theories, modifications of components, and rebuilding of instruments; the researchers at Moore School gradually arrived at the stored-program concept [5]. They certainly did not follow any pre-determined, scripted, or rigid approach but flexibly adapted to new situations, problems, and opposition; and changed their premises and approaches accordingly. Practical computer science combines receptivity and tenacity—it seems that practicing computer scientists can, at the same time, choose freely from a smörgåsbord of disciplines and approaches, yet be stubborn in the face of multidisciplinary opposition.

The mangle of practice in computer science is a continuous cycle of proposed theories, techniques, or mechanisms; corrections to theories, honing of techniques, improvements of mechanisms; negotiations between stakeholders, debates, power struggles; and other kinds of social and technical processes. In the course of time—and in the mangle—many theories are discredited and forgotten, many techniques become outdated, and many mechanisms turn out obsolete, but the knowledge that researchers in the computing science community gain through the mangle is gradually crystallized into a relatively stable core of computer science.

4. AN EFFICIENT ANARCHY

Methodological concerns have never played a leading role in computer scientists' work. Computer scientists publish relatively few papers with experimentally validated results [24], and research reports in computing rarely include an explanation of the research approach in the abstract, key word, or research report itself [26]. Computer science students usually learn their research skills from mentoring by their professors and from imitating previous research [23]. Typical computer science curricula, such as CC2001 [8], do not include courses on research methodology—yet in their work computer scientists do utilize a wide array of research methods, and often they combine methods in order to gain a wider perspective on the topic.

In a sense, many computer scientists might be characterized as *bricoleurs*—as researchers who work between competing paradigms. But from a disciplinary point of view, the diversity of research approaches that are utilized in computer science renders computer science a methodologically and epistemologically eclectic discipline. Based on the methodological and epistemological nonconformity of computer scientists, computer scientists can perhaps be best characterized as opportunists. Their breaches of methodological norms and epistemological concerns are not an act of ignorance, though; there are calls for methodological regimentation in every single computer science forum (e.g., [13,17,24,26]). The best way to characterize the eclecticism and opportunism of computer science, as well as its conscious breaches of methodological and epistemological norms, is that computer science is an anarchistic enterprise.

Eclecticism, opportunism, and interdisciplinarity have not been detrimental to computer science, though. Eclecticism, opportunism, and interdisciplinary work were crucial in the shift from electromechanical computation to electronic computation; and interdisciplinarity also spurred new ideas within traditional disciplines [19]. An eclectic combination of incommensurable crafts and sciences creates an ontological, epistemological, and methodological anarchy, which inhibits detrimental dogmatism because no ontology, epistemology, or methodology can claim superiority over others. It may well be that superficial knowledge about powerful ideas actually enables researchers to utilize concepts or innovations without getting mired in field-specific debates. (Of course it also risks doing research in superficial, incorrect, and contradictory ways.)

Throughout the short history of computer science, computer scientists have quickly deepened the theoretical and technical knowledge about computing, and computer science has also worked as a catalyst in the creation of new research fields and spurred research in other disciplines. It seems that computer science has been efficient because of anarchism, not despite it. For example, the stored-program-paradigm was born as a result of a successful combination of a number of epistemologically and methodologically incompatible disciplines such as logic, electrical engineering, mathematics, physics, and radio technology. And especially after 1945 computer science has been influenced by a large and eclectic bunch of disciplines and approaches.

Anarchism can also be seen in that many innovations in computer science have been spurred despite the lack of support by the academic establishment, and sometimes even despite strong opposition by the establishment. In addition, without anarchism in computer science, the innofusion of many innovations in computer science could have been much slower, and some innovations might have not been introduced at all. For instance, in the 1970s computer scientists widely adopted Dijkstra's famous "GOTO statement considered harmful" hypothesis [9] without ever testing it empirically [12]. Whether computer scientists like it or not, anarchistic strands are thickly woven in computer science.

5. CONCLUSIONS

The processes of the creation, maintenance, and modification of computer science are complex and rich in nuances. Just teaching the topics of computer science and the basics of research in computer science does not yet create an educated academic computer scientist. Courses on the history of computing often focus on the milestones of computing, which might reinforce the idea of deterministic progress instead of portraying a living computer science. It is important to understand that many "milestone" concepts and events in computer science have in reality been far from discrete steps—the milestone concepts and events have often been multifaceted issues, and they have formed as a result of controversies, debates, and power struggles. The dynamics of computer science might be best revealed through social studies of computer science.

Sociohistorical understanding offers important "lessons learned"; it can be used to trace concepts and technologies to their origins in challenges, controversies, and discussions; and it enables one to discover parallels and analogies to modern technology that can be used for reassess current and future developments. Social studies of computer science can shed light on the very foundations of knowledge creation and maintenance in the computing disciplines—for instance, expose how the philosophical, theoretical, conceptual, and methodological frameworks of computer science are created, maintained, and managed. Social studies can explain how computer scientists and other stakeholders create computer science and computing technology. Social studies of computer science can reveal the human origins and human character of our science. Meta-knowledge is an inherent and important part of any academic discipline, including computer science.

6. ACKNOWLEDGMENTS

I wish to thank professor Erkki Sutinen, Dr. Esko Kähkönen, and professor Piet Kommers for their support and numerous comments on my work.

7. REFERENCES

[1] Aspray, W. Was Early Entry a Competitive Advantage? US Universities That Entered Computing in the 1940s. *IEEE Annals of the History of Computing* 22(3) (2000), 42-87.

[2] Atchison, W. F., Conte, S. D., Hamblen, J. W., Hull, T- E., Keenan, T. A., Kehl, W. B., McCluskey, E. J., Navarro, S. O., Rheinboldt, W. C., Schweppe, E. J., Viavant, W., and Young, D. M. Curriculum '68, Recommendations for Academic Programs in Computer Science. *Communications of the ACM*, 11(3) (March 1968), 151-197.

[3] Backus, J. The History of FORTRAN I, II, and III. In *History of Programming Languages* (ed. Wexelblat, R. L.). Academic Press: London, UK, 1981, 25-45.

[4] Bright, H. S. Early FORTRAN User Experience. *IEEE Annals of the History of Computing* 6(1) (1984), 28-30.

[5] Campbell-Kelly, M., Aspray, W. *Computer: A History of the Information Machine (2nd ed.).* Westview Press: Oxford, UK, 2004.

[6] Copeland, B. J., Proudfoot, D. What Turing Did after He Invented the Universal Turing Machine. *Journal of Logic, Language, and Information* 9(4) (2000), 491-509.

[7] Croarken, M. G. The Emergence of Computing Science Research and Teaching at Cambridge, 1936-1949. *IEEE Annals of the History of Computing* 14(4) (1992), 10-15.

[8] Denning, P. J., Chang, C. (chairmen) and the Joint Task Force on Computing Curricula. Computing Curricula 2001. *ACM Journal of Educational Resources in Computing*, 1(3) (Fall 2001), Article #1.

[9] Dijkstra, E. W. Go To Statement Considered Harmful. *Communications of the ACM* 11(3) (1968), 147-148.

[10] Dijkstra, E. W. Programming as a Discipline of Mathematical Nature. *American Mathematical Monthly* 81(June-July 1974), 608-612.

[11] Flamm, K. *Creating the Computer: Government, Industry, and High Technology*. Brookings Institution: Washington, D.C., USA, 1988.

[12] Glass, R. L. "Silver bullet" Milestones in Software History. *Communications of the ACM* 48(8) (2005), 15-18.

[13] Glass, R. L., Ramesh, V., Vessey, I., An Analysis of Research in Computing Disciplines. *Communications of the ACM* 47(6) (2004), 89-94.

[14] Hughes, T. P. Technological Momentum. In *Does Technology Drive History? The Dilemma of Technological Determinism* (eds. Smith, M. R. & Marx, L.). The MIT Press: Cambridge, Mass., USA, 1994, 101-114.

[15] Knuth, D. E. Computer Science and its Relation to Mathematics. *American Mathematical Monthly* 81(April 1974), 323-343.

[16] Marcus, M. and Akera, A. Exploring the Architecture of an Early Machine: The Historical Relevance of the ENIAC Machine Architecture. *IEEE Annals of the History of Computing* 18(1) (1996), 17-24.

[17] Palvia, P., Mao, E., Salam, A.F., Soliman, K. Management Information Systems Research: What's There in a Methodology? *Communications of the AIS* 11(16) (2003), 1-33 (Article 16).

[18] Pickering, A. *The Mangle of Practice: Time, Agency, and Science*. The University of Chicago Press: Chicago, USA, 1995.

[19] Puchta, S. On the Role of Mathematics and Mathematical Knowledge in the Invention of Vannevar Bush's Early Analog Computers. *IEEE Annals in the History of Computing* 18(4) (1996), 49-59.

[20] Pugh, E. W., Aspray, W. Creating the Computer Industry. *IEEE Annals of the History of Computing* 18(2) (1996), 7-17.

[21] Rosenblatt, B. The Successors to FORTRAN: Why Does FORTRAN Survive? *Annals of the History of Computing* 6(1) (1984), 39-40.

[22] Sammet, J. E. *Programming Languages: History and Fundamentals*. Prentice-Hall, Inc.: Englewood Cliffs, New Jersey, 1969.

[23] Tedre, M. *The Development of Computer Science: A Sociocultural Perspective.* Ph.D. Thesis, University of Joensuu, Finland, 2006.

[24] Tichy, W. F., Lukowicz, P., Prechelt, L., Heinz, E. A. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software* 28(1995), 9-18.

[25] Turing, A. M. On Computable Numbers, With an Application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society, Series 2* 42(1936), 230-265.

[26] Vessey, I., Ramesh, V., Glass, R. L. Research in Information Systems: An Empirical Study of Diversity in the Discipline and Its Journals. *Journal of Management Information Systems* 19(2) (2002), 129-174.

[27] Williams, M. R. *A History of Computing Technology*. Prentice-Hall: New Jersey, USA, 1985.