Software Engineering 2016, 5(1): 1-7
DOI: 10.5923/j.se.20160501.01

Practitioners Beware: The New Model for Software Engineering May Turn out to be the next 'Holy Roman Empire'

Harvey Hyman

National Library Service, Library of Congress, Washington, DC. USA

Abstract An interesting phenomenon has evolved over the past 30-years, as several titles have been suggested for attempt to anthropomorphize the construct of 'professionalism' in Software practice, these titles include: programmer, engineer, architect and developer. The most recent iteration in the evolution of professionalizing the practice of Software (with a capital S) has been in the form of the professional engineer (PE) license. The on-going trend for definitively narrating the discipline and practice of Software manufacture motivates the discussion presented in this paper. The paper makes the argument that the constant struggle to define Software practice with a descriptive title, may in fact be as elusive as the historical failed attempt to define the ancient Holy Roman Empire. The 18th Century author Voltaire is credited with the famous phrase about the Holy Roman Empire, that it was "Neither holy, nor Roman, nor an empire." This paper explores whether Software Engineering might in fact be, neither software, nor engineering, nor architecture – nor holy, nor Roman for that matter. The paper concludes with a proposed visual model to clarify thinking in this area.

Keywords Software Engineering, Software Development, Software Architecture, Professional Engineer, Models

1. Introduction

The discussion begins by challenging the assumption that Engineering is in fact the correct discipline to define Software and its production: How is it that "Software Development" is not enough to serve to define the foundations and principles of Software production – and how is it that computer science lost its grip on determining the defining characteristics of Software and its production?

What is the value added distinction being claimed by Software Engineering that is not served by Software Development or Computer Science? For that matter, is there a distinction with a difference, among the paradigms of Engineering, Architecture, or Development as applied to the construct of Software itself, and to its production and use.

The National Council of Examiners for Engineering and Surveying (NCEES) has laid claim to Software as Engineering by serving as its licensing body and therefore, de facto gatekeeper for any individual wishing to claim the title of Software Engineer. But, what in fact is the justification for staking such a claim, or for requiring such a license to practice Software production? This of course begs the question of whether software production or development is in fact "engineering" at all. Couldn't we just called it "software manufacturing," and apply the standards of products liability to software production as we do with, say, car manufacturers?

* Corresponding author:
hymanphd@gmail.com (Harvey Hyman)
Published online at http://journal.sapub.org/se

2. Research Question Stated

The introduction questions stated above are re-stated as two core, fundamental research questions serving as the guiding themes for the discussion presented in this essay:

(1) Which discipline should "own" the domain of Software for the purpose of setting standards for the knowledge base and for establishing guiding principles for practice?
(2) What paradigm might best serve as the conceptual and representational framework for defining the characteristics and descriptive attributes of Software as an applied art, science, or professional practice?

3. Motivation

In light of the recent adoption of licensing for Professional Software Engineering (PE), it is important to consider the evolution of the drive toward professionalizing Software production, the current trends defining the discipline of Software, the impacts this will have on the future of Software policy, and the changing attitudes about what constitutes

computer programming and control of hardware. For example, will a licensed software engineer be required to "sign off" on requirements documents or testing reports prior to a "legal" public software release, similar to other engineering projects requiring PE approvals or certifications, or say, architectural blue prints to construct a building?

4. Foundation

If we are to think about the existential definition of "Software" without regard to its application, we begin with lexicon descriptions as a narrative foundation for our discussion: "The programs used to direct the operation of a computer"[1], "Something used or associated with and usually contrasted with hardware: as the entire set of programs, procedures, and related documentation associated with a system and especially a computer system; specifically: computer programs" [2] , "The programs, routines, and symbolic languages that control the functioning of the hardware and direct its operation." [3]

So, if our starting point is to existentially define software, particularly in contrast to hardware, and/or as a control method for the operation of hardware as a system or computer, we next ask: How do we apply this definition operationally, to software development, software engineering, software architecture, or simply, computer science; and what are the distinctions and differences among these reference disciplines that changes our experience of Software as a reference discipline itself, whether as an art or science – or is it merely a support discipline to the aforementioned?

4.1. Evolution and Trends: 1984 - 2015

This discussion, is not new by any means, and in fact, dates back over 30 years, with 10 year intermediate revisions and milestone reports (McKeenan, 1984; Shaw and Garlan, 1996; Kruchten et al., 2006). At this time, we are on the cusp of yet another 10-year milestone, and have recently seen a significant shift in the paradigm of software production with the advent of PE licensing, and therefore, the topic is yet again, highly relevant and worthy of a current resurgence in discussion and debate.

Some might suggest that the watershed moment in the drive towards professionalism never really began, but definitely occurred, in 1984, by William McKeenan, in his address to the graduates at the Wang Institute. McKeenan's artful and elegant description of what it means to be a profession and whether Software (in his address he defined it as the discipline of Engineering) is sufficiently mature as a practice and cohesive as a constituency to don the brand of "profession." This watershed moment of McKeenan's address bares similarity to another discipline's watershed moment, Management Information Systems (MIS), when in 1980, Keen (ironic similarity in root) laid out the core questions MIS had to ask of itself to determine if it was in fact a reference discipline or merely a support for other reference disciplines (ICIS Conference, 1980).

An equally pivotal series of events occurred during the period between 1985 and 1987, (conveniently within a short time of McKeenan's remarks), in what has mainly become known as the Therac-25 incidents. A now infamous exemplar and prototypical citation that has come to justify the hue and cry for the need to "professionalize" software design and development, especially when health or safety are at risk.

A good reference point is the 1996, Shaw and Garlan book, Software Architecture: Perspectives on an Emerging Discipline. In it they "examine architectures for software systems as well as better ways to support software development" (emphasis added). If we take them at their words, their 1996 work seeks to redefine software development under a new (emerging) paradigm, using architectural design as the reference discipline. Their declared target audience is "professional software developers looking for new patterns for system organization." Once again, a good insight into, who and what, they are attempting to influence. But, without a licensing authority or controlling body, how can we expect textbooks and graduation addresses alone, to determine the boundaries of professionalism for a practice?

In 2000, Lethbridge asked, "What knowledge is important to a software professional?" A good probing question perhaps, because it encourages an exploratory discussion of the attributes we would want in the profession of Software, then proceeds to review the available reference disciplines to determine which ones might be the most closely aligned with these attributes. Lethbridge effectively points out in his article that, "Efforts to develop licensing requirements, curricula, or training programs for software professionals should consider the experience of the practitioners who actually perform the work."

In 2006, Kruchten et al., conducted the proverbial "10-year review" of the state of the practice since Shaw and Garlan, describing the rich and storied foundation of software architecture in history, going back as far as, "the 1969 conference on software engineering techniques organized by NATO" (also cited by another Mary Shaw paper later in this essay). An in-depth reading of Kruchten et al., might leave the reader with the impression that McKeenan was irrelevant, or that his address failed to gain traction in the drive toward professionalizing the software practitioner. However, this is not the case. Of particular note is Kruchten et al's declared theme to, "address the latest thinking about creating, capturing, and using software architecture throughout a software system's life" – an interesting metaphor for software practice in terms of design and development, especially if we consider this with respect to the SDLC. In fact, their article appeared in the IEEE Special Issue on the self-declared 10-year history of

1 Dictionary.com.

2 Merriam-Webster On-line Version.

3 The Free Dictionary.

Software Engineering conferences and workshops, anchored by IEEE's first issue dedicated to the subject matter.

Kruchten et al's "address of the latest thinking" in 2006, evidently is not to be undone by yet some more "latest thinking." In 2015, we now find ourselves "back to the future" – 1984 McKeenan, with a resurgence of Software as "Engineering," but going "all in" as a licensed profession, courtesy of the PE exam for Software Engineering. During the 30-year period narrating proposed working definitions for the discipline of Software, what in fact, is the underlying discipline that is being supported, and who has made the best arguments for one view over another?

We take a pause in the discussion to briefly insert Mary Shaw's (the same Shaw as in Shaw and Garlan, 1996) article released in 1990 under the CEI flag at Carnegie Mellon. In it she surveyed the "prospects" for Software Engineering (yes, you read that right, Engineering) as a discipline. An interesting juxtaposition, compared to her later work with Garlan on the "emerging discipline of software architecture."

Shaw's work in 1990 reflected the thinking at the time that, "software engineering is not yet a true engineering discipline, but it has the potential to become one." It is particularly interesting to note here that when we read Shaw's work in 1990, followed by Shaw and Garlan in 1996, Lethbridge in 2000, and Kruchten et al., in 2006 we can see the evolutionary trend of back and forth, between engineering and architecture, grappling with the question of what best defines software practice, and who we should be asking to gain insight into the answer.

We jump back one more time to 1996. As Sinatra would say, "it was a very good year." But, it was not a year of small town girls. Instead it was a year of both engineering and architecture. That is right, the very same year that Shaw and Garlan released their book on Software Architecture, Anthony Wasserman was still making the case for Software Engineering as a discipline. In fact, it was a very passionate argument laid out in his 1996 paper, "Toward a Discipline of Software Engineering," Wasserman makes a convincing argument that "some fundamental ideas have remained constant" in software practice. He goes on to walk the reader through eight "concepts that together constitute a viable foundation for a software engineering discipline." Wasserman's article is so convincing, that if one never read Shaw and Garlan, one would think that engineering would "travel half-way around the world while architecture is putting on its shoes."[4]

4.2. What ever happened to Computer Science?

Contrasting with the evolving claims that engineering or architecture should be the paradigms for defining software, several claims have been advanced that Computer Science should be the reference discipline. In fact, some claims reported in this section, go back to the 1960s and 1970s.

An article cited by many, but fairly obscure is "Computer Science as a Discipline," authored by Lofti Zadeh. It appeared in a 1968 issue of the Journal of Engineering Education. It will show up in any literature search on the subject of computer science as a reference discipline. In 1975, Newell and Simon made the case for Computer Science as "empirical inquiry."

In the modern era, few works that have explored the paradigm of Computer Science as a reference discipline. A series of works by Matti Tedre presented a broad view of the discipline: "Ethnocomputing A Multicultural View on Computer Science", "Computing as a Science: A Survey of Competing Viewpoints", and "The Science of Computing: Shaping a Discipline" (Tedre, 2014). Another good examples came from an interesting survey done by, a then PhD candidate, Brent Jesiek, "Between Discipline and Profession: A History of Persistent Instability in the Field of Computer Engineering, circa 1951-2006," (Jesiek, 2006).

However, it seems that since the 1970s really, Computer Science as the paradigm for Software seems to have lost momentum in light of the increasing traction Engineering has established (as laid out the previous section). In fact, for about the past 20 years, most works exploring Computer Science as a discipline categorize it as an academic discipline and are dedicated to discussing it from an educational perspective – effective teaching of computer science (Ramamurthy, 1976; Austing, 1977; Tedre, 2002).

The proverbial, most recent nail in the coffin, for Computer Science is exemplified in last year's, Raymond Turner work on the "Philosophy of Computer Science" (Turner, 2014). If there is ever a sign that a domain is becoming irrelevant, it has to be the banishing of it to philosophical analysis.

So, if not Computer Science, then what about the paradigm of Computer Programming as a reference discipline for Software? After all, the existential, consensus definition for Software is to describe it as a set of programs or computer programs.

4.3. Computer Programming as a Reference Discipline for Software?

In 1974, Donald Knuth presented a robust review of the classification of computer programming as an art or a science, motivated by the preceding years in which, many in the ACM community seeking to "transition computer programming from an art to a disciplined science" (Knuth, 1974). It seems from a reading of his work here, that there was a feeling in the community that programming was not being taken seriously, and that it needed to stake its claim, so to speak, perhaps supplanting Computer Science as the reference discipline for Software.

This sentiment remained relevant in the zeitgeist for some time. In 1985, Wirth in his book, Algorithms and Data Structures effectively states the trend at the time for programming to lay claim to Software practice which was:

4 Some readers will recognize the Mark Twain reference. Not to suggest that engineering is a lie or architecture is the truth.

"In recent years the subject of computer programming has been recognized as a discipline whose mastery is fundamental and crucial to the success of many engineering projects and which is amenable to scientific treatement [sic] and presentation. It has advanced from a craft to an academic discipline" (Wirth, 1985).

In 1990, Banatre and Le Metayer published work on software development using the reference discipline of Computer Programming. This is an interesting paper to consider because it describes a model to improve software development, and the authors describe it as such, yet they choose to classify their work as referenced to programming rather than to development.

Software Development has been a main stream term describing design and delivery of software for some time. Thus far, this essay has presented a narrative of the competing views of choosing between paradigms that best "tell the story" of Software: Should it be Engineering or Architecture? In light of Banatre and Le Metayer, why shouldn't Development serve as the reference discipline for Software?

4.4. Development as the Discipline

A good place to start a discussion on Development as a paradigm would the 1970 paper by Royce, in which he lays out the waterfall metaphor for what he describes in the paper as "computer program developments," and in which he mentions the term development no less than 24 times. Here we see the use of development to describe the action of creating programs, but not programming as a reference to itself (Royce, 1970).

Fast forward to 1995, when Kraut and Streeter announced that Software Development had been in "crisis for 20 years." That seems to be as good a measure as any, since Royce's article. During this period, we observe the evolving and raging debate of "Agile versus Disciplined," culminating in the Agile Manifesto announced in 2001 (MILSTD, 1986; IEEE 12207-1985; Agile Manifesto, 2001; Boehm and Turner, 2003). During this period, we find references to Software as Development gaining momentum, and also leading to several industry certifications such as Sun Certified Java Programmer (now Oracle Certified Associate), and IEEE Certified Software Development Professional (begun in 2002).

4.5. Architecture as the Discipline

Mathew McBride wrote an interesting piece on the implications of bestowing the "title of architect" upon a software practitioner. He makes a convincing argument for an indictment on the efficacy of granting such a title: "Even experienced software practitioners are often unable to define what exactly the architect does or adds to the software development process" (McBride, 2007).

Leslie Lamport uses the metaphor of the blue print (Lamport, 2015). In fact, Lamport treats the subject with delicate balance. Lamport channels the best representation for the proponents of architecture as the dominant title for the software practitioner: "Architects draw detailed plans before a brick is laid or a nail is hammered. But few programmers write even a rough sketch of what their programs will do before they start coding. We can learn from architects." This statement is tempered with the following: "However, metaphors can be misleading, and I do not claim that we should write specifications just because architects draw blueprints" (Lamport, 2015).

Capilla et al., provide a robust report describing how a software architect impacts software, from two different perspectives. First, they describe the architect from an early period of evolution, as that of "purely technical and examined architecture [of software design] in terms of system structure and behavior." The second description is from a contemporary perspective as "socio-technical and consider [ed] architecture from the point of view of its stakeholders, looking at how they reason and make decisions" (Capilla et al., 2015).

In their work "Variability in Software Architecture: Current Practice and Challenges," Galster et al., 2011, provide a survey of the landscape of the skills and responsibilities of a software architect. An in-depth reading of their paper might leave the audience convinced that every software production effort must have at a minimum: an architect and an engineer, as well as various additional developers and ancillary producers of the software, such as coders and testers for example.

Where would this end? Is a professional software engineer merely the beginning of a new paradigm for the identification of the division of labor within a software project? Is software production set to go the way of Adam Smith's Pin Factory? We already have independent certifications available for coders and testers wishing to distinguish themselves in the labor market, should we require their licensure as well? If the public safety is at risk, shouldn't all participants in a software production effort have to undergo some degree of professionalization?

4.6. Software as Its Own Discipline

Lest we forget to ask: Why can't Software practice be its own discipline? And, what is the justification for it to be a profession anyway, rather than a stand-alone science or art? Several of the articles and works reported in this paper would make a convincing case that Software is perhaps best defined as "Software Science" similar to its ancestor Computer Science.

Or, perhaps, Software should be narratively described as the "art of software." After all, there are numerous ways to design and build a computer program or application; and what makes one program design better than another is often a qualitative judgement, rather than a quantitative one. Halstead, in 1977, introduced the Elements of Software Science. In it, he set out a foundation of concepts and principles that he believed could be used as empirical metrics to definitively measure identifiable properties of

software. Extending his theory forward, we could use the internal disciplinary metrics and measurements of software itself to define the characteristics of a Software Scientist, rather than rely on indirect proxies of architect or engineer to define the Software practitioner.

Lieberman and Fry ask in their 2001 article, "Will Software Ever Work?" Of particular note is Fry's Law, in which he claims that programming performance doubles once every 18 years (vice Moore's Law). The reader will appreciate the irony, that in fact, Lieberman identifies as a scientist, and Fry as an engineer.

Recently, Sandy Payette provided an interesting perspective on Hopper and Dykstra and the "crisis, revolution, and the future of programming." Interestingly, Payette goes back the roots of the discipline through the paradigm of Computer Science, as the representative constituency of programming and software. It is also interesting to note Payette's inclusion of Kuhn and his description of "paradigm shifts" (Payette, 2014). Could it be that Software is undergoing a "paradigm shift" away from science and toward engineering?

5. Making the Case for Engineering (SE) as the Paradigm for Software Practice

So, how do we make a case for Engineering as the reference discipline for Software practice? What is engineering, exactly, and what is an engineer, really? Now, what does an engineer and engineering, have to do with designing and developing software?

Perhaps the most convincing argument is that advanced by Phillip LaPlante in his 2013 article, "The Misconceptions about Licensing Software Engineers," in which he systematically knocks down many, if not all, of the straw-man arguments against the PE licensing exam for Software practitioners. He effectively points out that, at the time of his writing, "thirty states currently require licensure for software engineers working on systems that affect the health, safety, and welfare of the public." He also contends that states license many types of professions "including accountants, doctors, lawyers, and nurses, as well as trades people such as beauticians, electricians, and plumbers – through a combination of requisite education, experience, and the passing of one or more competency exams."

What LaPlante does not address, and in fact, is a lingering gap in the current debate of "To SE or not to SE?" as the question of the control of Software practice, remains the determination of whether the practice shall live within, and be confined and be controlled by, Software Engineering, or whether Software Engineering is merely one of many identifiable occupations that exist within the larger field of Software practice. This is an important distinction to consider, because such a large gap can relegate SE irrelevant if it does not enjoy compulsory enforcement such as that of other professions.

For example, someone who is not a licensed architect cannot design a building under the guise of "I am not an architect, I am a building designer" type of claim. Likewise, what is to prevent a software developer, programmer, designer, or other para-professional from making the same claim, and thus working around the SE license?

Some other lingering questions remain as well. For example, what is the relevancy of the "Software Engineering Body of Knowledge" (SWEBOK), given that IEEE has abandoned its certification for software professionals? In light of the PE as a standard for professionalism in Software practice, what about ACM's, "Software Engineering Code of Ethics and Professional Practice?" Is it still relevant, and if so, how?

5.1. A Proposed Model

During the course of the narrative of this paper, we have discussed the various proposed titles and descriptions for software production. This variety of descriptors has been categorized into the four major classifications of programming, developing, engineering, and architecting. These classifications reflect the competing views of how software practitioners wish to define themselves, but also how outside academic and industry stakeholders view the production of software and the people who do it.

This paper makes the argument that each of these titles legitimately offer elements describing some parts of software production, but non completely describe software production entirely. Therefore, this paper suggests the graphic visualization in Figure 1. The figure is a proposed model to depict this competing tension, with each classification as its own eco-system, but with overlapping portions to reflect the commonalities shared among them.

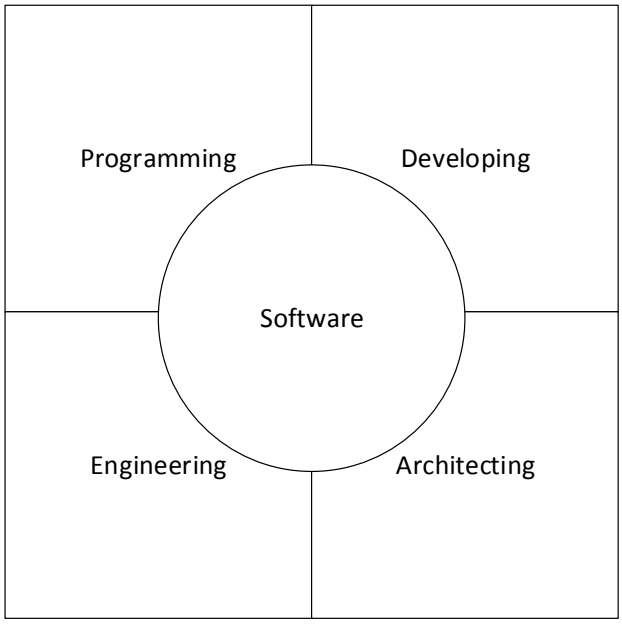

Figure 1.

6. Conclusions

The goal of this essay was to provide a robust discussion of the how the reader might think of the practice of Software in terms of the competing paradigms and reference disciplines of: Computer Science, Development, Engineering, and Architecture. To that end questions were posed, but never really answered. Possibly, there is no answer here; nor does this essay attempt to offer one. Perhaps the answer to the questions posed in this essay will continue to be worked through by the Software community, as it finds itself and defines for itself: what it is and what it wants to be, and who are its members and who do they want to be.

So what is Software? Maybe it is design and development. What is development? Maybe it is good programming – following good analysis, good design, adequate vetting and sufficient testing. What is an example of good analysis and good design? Perhaps we find it in the SDLC –planning, analysis, design, and implementation. Notice how the process is not called the SELC.

Often, when we think of software development practices, we often focus on managerial skill sets such as process, design and project management – some suggest that none of these have anything to do with good coding or good programming. If you doubt this claim, go talk to a business analyst and see how well they can generate a simple computer program, and we are not talking about mocking up some web site pages using HTML and JavaScript.

ACKNOWLEDGEMENTS

The content and opinions expressed in this article are solely that of the author and do not in any way claim any endorsement from the National Library Service (NLS) or the Library of Congress (LOC). No resources of the NLS or LOC were utilized in the production of this article.

REFERENCES

[1] Austing, R. H., Barnes, B. H., & Engel, G. L. (1977). A survey of the literature in computer science education since curriculum'68. Communications of the ACM, 20(1), 13-21.

[2] Banatre, J. P., Le Metayer, D. (1990). "The Gamma Model and its Discipline of Computer Programming." Science of Computer Programming. Volume 15, pp. 55-77.

[3] Boehm, B. W., Turner, R. (2003). Balancing agility and discipline: A guide for the perplexed. Addison-Wesley Professional.

[4] Capilla, R., Jansen, A., Tang, A., Avgeriou, P., Ali Baba, M. (2015). "10 years of software architecture knowledge management: Practice and Future" Journal of Systems and Software. October pp. 1-15.

[5] Galstar, M., Avgeriou, P., Weyns, D., Mannisto, T. (2011). "Variability in Software Architecture: Current Practice and Challenges." ACM SIGSOFT, Software Engineering Notes. Volume 36, Number 5, p. 30.

[6] Halstead, M. H. (1977). Elements of software science. New York: Elsevier.

[7] Jesiek, B. K. (2006). Between Discipline and Profession: A History of Persistent Instability in the Field of Computer Engineering, circa 1951-2006.

[8] Kruchten, P. Obbink, H., Stafford, J. (2006). "The Past Present, and Future of Software Architecture." IEEE Software. March/April, p. 22.

[9] Knuth, D. E. (1974). "Computer Programming as an Art." Communications of the ACM. Volume 17, Number 12.

[10] Lamport, L. (2015). "Who Builds a House without Drawing Blueprints?" Communications of the ACM. Volume 58, Number 4.

[11] Lethbridge, T. C. (2000). "What knowledge is important to a software professional?" Computer. (5) 44.

[12] Lieberman, H., Fry, C. (2001) "Will Software Ever Work?" Communications of the ACM. Volume 44, Number 3.

[13] McBride. M. R. (2007). "The Software Architect." Communications of the ACM. Volume 50, Number 5.

[14] McKeenan, W. M. (1984). "Professional Software Engineering." IEEE Software. October, 1984, p. 112.

[15] Newell, A., Simon, H. A. (1975). "Computer Science as Empirical Inquiry: Symbols and Search." Communications of the ACM. Volume 19, Number 3.

[16] Payette, S. (2014). "Hopper and Dijkstra: Crisis, Revolution, and the Future of Programming." IEEE Annals of the History of Computing, Volume 36, Number 4, pp. 64-73.

[17] Ramamoorthy, C. V. (1976). Computer Science and engineering education. IEEE Transactions on Computers, (12), 1200-1206.

[18] Royce, W. W. (1970). "Managing the Development of Large Software Systems." Proceedings, IEEE WESCON. August, pp. 1-9.

[19] Shaw, M. (1990). "Prospects for an engineering discipline of software." Carnegie Mellon University School of Computer Science at Research Showcase @ CMU.

[20] Shaw, M., Garlan, D. (1996). Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.

[21] Tedre, M. (2002). A Multicultural View on Computer Science. IEEE (Institute of Electricaland Electronics Engineers) ICALT (International Conference on Advanced Learning Technologies), Russia, September, 9-12.

[22] Tedre, M. (2014). The Science of Computing: Shaping a Discipline. CRC Press.

[23] Turner, R. (2014). "The Philosophy of Computer Science." The Stanford Encyclopedia of Philosophy (Fall Edition), Edward N. Zalta (ed.).

[24] Wasserman, A. I. (1996). “Toward a Discipline of Software Engineering.” IEEE Software. November, 1996.

[25] Wirth, N. (1985). Algorithms and Data Structures. Prentice Hall.

[26] Zadeh, L. A. (1968). “Computer Science as a Discipline.” Journal of Engineering Education 58.8: 913.