Perspectives on the Past, Present, and Future in Computer-Related Areas as They Impact Academia, Business, and Other Areas

Robin M Snyder
RobinSnyder.com
robin@robinsnyder.com
http://www.robinsnyder.com

ABSTRACT

The author has attended and presented at most ASCUE meetings since 1994, and has worked professionally in research and development, industry, military, government, business, and private and public academia - moving between computer science, software engineering, and business fields at both the undergraduate and graduate level, and even running academic computing for a few years. This paper/session will present/discuss definitions, implications, and relationships of and between the areas of computer science, software engineering, information technology, and business information systems. Included will be perspectives of the history of the past, specific needs of the present, and general directions and predictions of the future, and the implications to academia, business, and other areas.

INTRODUCTION

In addition to information in the abstract, students, faculty, staff, administrators, etc., will find occasion to need to work with people in areas not their own, and it can help to be aware of the similarities and, more importantly, the differences between various related areas.

Here are some related areas of study that, to some degree, involve information, technology, computers, etc.

- mathematics
- statistics
- computer science
- engineering (computer, software, etc.)
- information systems
- informatics, bioinformatics, etc.

What are some of the similarities and differences between these areas of study? This can be important in understanding where we have been, where we are, and where we are going - in terms of science, society, education, etc.

Realize that each area of human study is oriented to those people who tend to think in the same way, the way that field thinks and approaches problems. The Myers-Briggs personality types, as found in, for example [8], can help in this understanding. And it helps to explain ideas in terms of how that person thinks and approaches problems. At one university a long time ago, PC Write for DOS was being used as the word processor. It had issues such as converting all tabs to spaces so that indentation infor-

mation was lost when one would really like it to be retained. At a meeting of the math and computer science department, one math type could not understand the problem - it made no sense to him. In my frustration, I blurted out, "**It is a homomorphism of tabs to spaces**". His eyes lit up and he suddenly understood the problem. For non-math types, a homomorphism, in software engineering terms, (for this issue) is an association of a many to one mapping whereby the one side of the mapping loses some information that was in the many side. That is one reason why professionals in the field should teach courses in that field - they have intimate, and perhaps instinctive, knowledge of how people in that field think about, approach, and solve problems in that field.

Dijkstra provides a quote on how knowledge is partitioned.

> *Scientific disciplines have a certain "size" that is determined by human constants: the amount of knowledge needed must fit into a human head, the number of skills needed may not be more than a person can learn and maintain.* [4, p. 210]

Relatively newer fields of study, such as computer science, take a while to arrive at a consensus as to how that field thinks and approaches problems, which is one reason computer science people like all terms to be defined at the start of a discussion - there may not be consensus on what those terms actually mean.

Let us start with a thought question. The train problem (source unknown) goes as follows.

> There are two trains. One train leaves New York at 120 miles per hour bound nonstop for Los Angeles by the most direct route. The other leaves Los Angeles at 80 miles per hour bound nonstop for New York by the most direct route. When they meet, which train will be closer to Chicago?

Arrive at some answer (and some basic reasoning for your answer) before continuing (and before the trains collide!).

CHANGE

Change happens whether we like it or not. When is the software "**done**"? The software is "**done**" when it is no longer needed. Why does software need to change. In addition to the somewhat nebulous observation that the "**future is uncertain**", there are two primary reasons why software needs to change.

- Technology changes - and changes fast
- User's expectations change

Some change is externally motivated, such as laws requiring changes, security attacks requiring change, etc.

The fast and persistent change of "**information systems and computer technology**" results in a shortened form of this concept, the field of "**information technology**"

The author has usually included the following in any new course descriptions: "**The course will emphasize general and enduring principles for future needs while including specific and practical necessities for present needs.**". There is a need for trade-offs between the following.

- education (academic oriented)
- training (practice oriented)

How would you explain the difference? Think about it.

The history of the Liberal Arts [2, p. 47], [10, p. 3-8] goes back to the middle ages when the first universities were formed, starting in Bologna, to teach the seven liberal arts: grammar, rhetoric, logic, arithmetic, geometry, music, and astronomy. The origin of the word "**art**" in "**liberal art**" is from the Latin word "*artis*", meaning "**skill**". The corresponding Greek word is «**τέχνη**», the root of the word "**technology**". The distinction was that a liberal art was something created by human intellect rather than by, say, manual labor (e.g., the art of masonry) as part of the manual arts. Disciplines that might not be fully accepted by the liberal arts crowd may need some reminder of how a particular field is a primarily a liberal art and not a manual art.

The more difficult type of change involves people. In general, it can take many lifetimes for ideas to change, since people tend to retain ideas far past the time when those ideas have been replaced by newer ideas. This is a central idea in Kuhn's book "**The Structure of Scientific Revolutions**" [12].

As a case in point, the programming language FORTRAN was developed in the 1950's (and marketed by IBM as a way to make debugging unnecessary, along the lines of OS/2 being marketed by IBM as crashproof). After 20 years of ubiquitous FORTRAN use in the industry, in the 1970's, computer scientist John Backus (Extended Backus-Naur form grammars are named for him) wrote an important article that laid out the reasons why FORTRAN was not a good way to develop programs and that functional programming, using his somewhat cryptic language **FP** (Functional Programming) as an example, was a much better way to develop correct, modular, and compositional software programs. Well, most programmers ignored or laughed at him and went happily along continuing to develop and write FORTRAN programs. Now, more than 40 years later, those functional programming language concepts have crept into every popular programming system - JavaScript, Python, Lua, C#, and even Java. According to Kuhn, this is about the time for one generation to leave and the next take over with the needed changes. And who was this John Backus who had the audacity to propose replacing FORTRAN with a better way to program? Why John Backus was the co-inventer of the programming language FORTRAN.

The point of all this is that even when you can see the future (Alan Kay would say it is better to invent the future, perhaps along the lines that Steve Jobs accomplished) do not expect that needed change to happen very fast or even be recognized within your lifetime.

MATHEMATICS AND ENGINEERING

Let us start with mathematics and engineering as computer science requires both mathematics and engineering.

> *Programming is as much a mathematical discipline as an engineering discipline; correctness is as much our concern as, say, efficiency.* [4, p. 54-55]

So although mathematics is very important to the computer scientist, there is much more than mathematics to computer science.

What is mathematics? Hofstadter [9, p. 559] makes the claim that mathematics is what mathematicians do. Citing the example of Ramanujan, he goes on to assert that all mathematicians are isomorphic in the sense that they think in the same way. What exactly is that way? In part, mathematicians are able to abstract away details to such an extent that they become the butt of jokes indicating a loss of touch with reality. By the beginning of the 20th century, mathematics as a field had pretty much decided to divorce itself from reality (including philosophical questions) by making mathematics a formal system of symbols and symbol manipulation.

What is engineering? Engineering is the application of known knowledge and principles, including technology, to develop something that makes efficient trade-offs in terms of usefulness, effectiveness, time, money, cost and values, etc.

What is the difference?

The difference between engineers and mathematicians can, perhaps, be understood by way of the following story, modified from [13, p. 81] (his first important book [14] also has many interesting aspects of how mathematicians think).

> A psychologist is questioning a mathematician and an engineer in the same room. To the engineer, the psychologist asks, "There is a fire on the stove and a glass of water on the table. What do you do?". To which the engineer, without hesitation, replies, "I would take the glass of water on the table and use it to dowse the fire on the stove". The psychologist then asked the mathematician, "The glass of water is now on the window sill. What would you do?". To which the mathematician, without hesitation, replies, "I would take the glass of water on the window sill and move it to the table and in that manner reduce the problem to the previously solved problem".

The following example was used in a math class at the United States Military Academy, West Point, during the early 1970's, to illustrate the theoretical and practical concept of limits.

> At a cadet dance, a mathematician and an engineer are told the following about an attractive female on the other side of the dance floor. In the first minute, you can easily get half way to her. From that point, in the next minute you can half as far again. And so on. The mathematician figures out the limit of the infinite series and concludes that he will never actually get there, and so he does not even try. The engineer figures out the first few terms of the series and concludes that, after a few minutes, he will be close enough for all practical purposes.

What is the common trend in these stories? Let us first briefly look at problem solving in general.

Explain the primary difference between mathematicians and engineers.

PROBLEM SOLVING

At the heart of the matter, mathematics, computer science, and engineering (and many other disciplines) are concerned with problem solving. Consider three aspects of problem solving.

- First, does a solution to the problem exist?
- Second, can an effective solution to the problem be found?
- Lastly, is the solution to the problem efficient? Or, among the possible solutions, which is the most efficient, given some criteria for efficiency.

Practical example: I am in Washington, D.C. and I ask a person on the street, "Excuse me, but do you know the way to Baltimore?". To which the person replies "Yes" and walks away. A solution exists, but that is of little use if I really do need to get to Baltimore. To the next person I ask, "Please tell me how to get to Baltimore from here?". The person replies, "Well, go south on Interstate 95 to Interstate 66 then west on Interstate 66 to Interstate 81, then north on Interstate 81 to Interstate 70, then east on Interstate 70 and that will take you right to Baltimore. "Thanks." Not satisfied that I have an efficient solution, I ask a third person, "Please tell me a quick way to get to Baltimore from here." To which the person replies, "Go north on Interstate 95. Baltimore is about 30 miles from here." In a practical setting, existence and effectiveness of solutions is often not enough. Efficient solutions are required. (Note: In an academic setting, it is wise to use a landmark, such as the campus library, with which everyone is familiar.)

As a historical mathematical example with relevance today, consider prime and composite numbers.

Euclid in about 300 BC proved that there are an infinite number of prime numbers, but not how to arrive at them. They exist. Eratosthenes in about 240 BC showed an effective way to determine prime numbers as far as one was willing or able to do so - using the Sieve of Eratosthenes. If one has two large primes and multiplies them together to get a composite number, one can write a simple program to determine the two primes from the composite number. But the program may take time longer than the age of the known universe to find the primes. Not very efficient or useful. Today, no one knows of an efficient way to find those primes. And public key cryptography (today), along with digital signatures, etc., is based on the difficulty of solving this problem.

Now a mathematician is primarily concerned with existence of solutions, sometimes with effectiveness of solutions, but rarely with efficiency of solutions.

> *Mathematically, however, it seems quite unsatisfying that some quadratic equations have solutions while others do not. Historically, this problem did not worry mathematicians: solutions of quadratic equations were always thought about geometrically (not algebraically) and an equation* $x^2 + 2bx + c = 0$, *with* $b^2 < c$ *was simply regarded as an equation without solutions or geometric interpretation.* [5, p. 128]

When a mathematician writes the integral of the formula "**f(x) dx**" the mathematician does not worry about whether there is a solution, and, if there is a solution, how it is to be computed efficiently. But

when implemented as a computer program by a computer scientist or software engineer, decisions must be made. Such as:

- How is function f(x) represented?
- How are the values of x to be subdivided?
- In what order are the computations to be made?

Each of these decisions has an impact on effectiveness in that the computer program must eventually obtain the correct answer. But efficiency is an important practical concern. Efficiency is a difficult concept because of the tremendous number of tradeoffs that must be addressed (see, for example, [1].

A last example, which the author has experienced first-hand, is determining the amount everyone should pay when getting a combined check at a restaurant. With a group of math types, it could take 5 minutes or more until each bill is determined to the last penny - including the apportioned tip for each. With a group of business types, it can take about 5 seconds to determine the amount - to within about a dollar or so. Which is better? Why?

At one university, the author covered the concept of people thinking differently in an MBA course, including the math and engineering differences. One student had a husband who was a mathematician. The next week I asked her if she had told her husband about what she had learned. She said "**no**", but then with a slight grin she said, "**but I told everyone else**". She had learned something from the class.

COMPUTER SCIENCE

Returning to computer science, there are various definitions of computer science.

Donald Knuth defines computer science as the study of algorithms. [10]

Niklaus Wirth (inventer of the Pascal programming language) defines programs as consisting of data structures and algorithms. [16]

Kowalski defines an algorithm as consisting of a logic and a control component. [11]

At a series of talks given at Penn State University in the late 1980's, Tarjan used an analogy of sorting. In particular, out of all possible permutations by which a list could be sorted, we would like to use any information gained from comparisons to reduce the search needed to complete the sort. In essence, what we are looking for is a way to reduce a potentially infinite search space to a more manageable (and more finite) search space.

As such, the author has since that time defined computer science as the search for finite approximations of (potentially) infinite objects, in line with algorithmic information theory [3].

The computer scientist however, has a difficult task in that the computer scientist must, as needed, think like either a mathematician or an engineer, and be able to context switch between modes of thinking.

Computer science is sometimes called "**informatics**" or "**information science**" although the term "**information science**" can refer to a specialized part of computer science. The field of "**bioinformatics**" is concerned with the application of information and computer science to biology.

STATISTICS

The field of statistics has a deep connection to computer science, though it is may not be immediately obvious. Flip a fair coin. What is the probability that it is heads? If you say it is one half, then that is the probability to you. I can see the coin and to me it is either zero or one - I know what it is. Built into the entire field of statistics is the concept of known and unknown information by an observer and determining a "**best guess**" at what is the state of the actual information of interest.

Michael Jordan, (not the basketball player) is a leader in the field of machine learning , and who does both Computer (i.e., Information) Science and Statistics at USC Berkeley. Jordan sees computer science and statistics merging in the next 50 years. Many algorithms of interest are now probabilistic algorithms. And once data becomes too large to look at all the data, and one needs results based on many factors, query results will (and sometimes now have) error bars associated with them. In computer science, a linear algorithm is needed to at least look at all of the data once. At some point, as databases become bigger and bigger, the only way to get sub-linear algorithms is to not look at all of the data, which requires probabilistic models.

So if one has a choice of a pure mathematician and an applied statistics person to teach computer science, who might do a better job - one who does not deal well with efficiency trade-offs or one who is comfortable making decisions under conditions of uncertainty? Note: This is not an all or one decision rule - it does not fit all cases. Pure math works well with theoretical computer science while statistics works well with real-world software engineering.

RELATING THE AREAS

The field of computers is goal-oriented, with interest in problems being driven by demand for solutions to real and practical problems. The difference in each of the areas, from the point of view of a student entering the field, is in the amount of mathematical expertise required for each field. To understand the similarities and differences, let us see how an individual in each respective field of study might view and react to a specific problem.

Management information systems: We can save money on postage if we presort our mailing lists by zip code. I know that there is some way that the computer can do that. But I have so many other things to do in my management position that I will have to get our computer information systems staff to follow through with the idea.

Computer information systems: Our mailing lists can be presorted by doing an analysis of our existing (database) software system, finding the appropriate module, writing code to call the system sort routine, and modifying it to get the proper information. If no sort routine exists, the system routine is too slow, or extensive modification is necessary, we will need to contact the software engineering team who developed our applications software.

Software engineer: Tell me the size of your typical list, how fast the sort routine needs to work, and any other information you think useful. I will find a routine, based on those developed by computer scientists, that is guaranteed to perform well for your application. Our team of software engineers will update the software to meet these new requirements, install it in your system, and update your current documentation.

Computer engineer: A computer engineer is similar to the software engineer, but is primarily concerned with designing and building computer hardware.

Computer scientist: An insertion or selection sort will correctly sort in O(n*n) time where n is the number of items to be sorted. Heapsort will work in O(n*ln(n)) time. Quicksort will beat heapsort, on average, but not in the worst case. I can prove all of these properties using an axiomatic semantics and algorithm analysis techniques. I can also adapt my solutions to unusual requirements using the same methods. I understand that people actually find these sort routines useful in practical applications.

Mathematician: Here is a list. We can define a partial ordering, so it is possible to sort the list. We are done. I can now go work on more interesting problems (which may have useful applications decades from now). Oh, you say you actually need to sort list. Well then, look at all possible permutations of the list (there may be trillions of them), and at least one of them will be in sorted order. Pick one of them. (This method is called slowsort, for obvious reasons).

List the five primary areas of computer-related study.

BUSINESS

What is a business? Business was more or less considered a manual art by many. Until the 1980's, there was not even a good definition of a business. But then Hammer and Champy studied and wrote books on business process engineering, such as [7], that attempted to fill this gap - allowing business to be considered more in terms of a liberal art than a manual art. The term "**business process reengineering**" is defined by Hammer and Champy as follows.

> *Business process reengineering is the fundamental rethinking and radical redesign of business processes to achieve dramatic improvements in critical, contemporary measures of performance, such as cost, quality, service, and speed.* [7, p. 32]

So continuous quality improvement improves incrementally what one is already doing. BPR makes major changes in how things are being done.

In simple terms, Hammer has a Ph.D. in computer science and he applied object-oriented design and implementation principles to business. By their definition, a business provides value to the customer - whoever the customer (some like the term stakeholder) may be. A suitable (though not always simple) objective function is to be maximized (or minimized) to provide that value.

Issues arise in any "**business**" where the objective function is not clear. Some of these include the following. Ask yourself, what is the objective function of the following as a business, as some are run, and what issues arise?

- medical field (you get less business if everyone stays healthy)
- higher-education institutions (you get less business if students leave or do not like you)
- religious institutions (who is accountable for promises made for the afterlife)

So given a business perspective, what is an information system? A useful definition of an information system is a combination of hardware, software, data, people, and policies that provide value in relation to cost. What is the most important part of an information system? Think about it. What part cannot be replaced if it disappears?

Note that people are in integral part of an information system.

One comment of Hammer that is relevant

> *It is becoming increasingly clear that the best strategy is not one that tries to divine the future but one that responds rapidly to the present.* [6, p. 203]

TRAIN OF THOUGHT

Returning to the train problem, there are many possible solutions. If you came up with no solution, then you are probably neither a scientist, an engineer, or a mathematician.

If you collapsed each train to a single point and concluded that both trains are equally distant from Chicago, then you tend to think more like a mathematician than an engineer.

If you reasoned that trains have finite size, the back of the train leaving New York would be closer to Chicago, and therefore, unless there was a circuitous route that switched their directions, the train leaving New York would be closer to Chicago, then you tend to think more like an engineer than a mathematician.

Note that an engineer would also be concerned that the track might have been reoriented due to obstacles (e.g., hills, rivers, etc.) such that the answer might be changed. Mathematicians would tend to use a straight-line approximation between the cities.

Which way of thinking is better? Neither way of thinking is better. But they are different. What seems to be important is that different people tend to think differently.

- Mathematicians tend to think like other mathematicians.
- Engineers tend to think like other engineers.

And woe to you if you do not think like a mathematician but try to be one. Or, if you do not think like an engineer but try to be one.

Note that if you tried to find tricks around the problem, you might do well in the field of security since, as security expert Schnier points out in [15], to do well in security, you need to be able to think like the

people trying to attack you, and those people are always thinking about ways to game the system, get an unfair advantage (whatever that means), etc.

SUMMARY

This paper has looked at various aspects of fields of computer-related study, outlining similarities and differences and how these might impact the future of the respective fields.

REFERENCES

[1] Bentley, J. (1982). Writing Efficient Programs. Prentice Hall.

[2] Burke, J. (2009). Day the Universe Changed. Little, Brown.

[3] Chaitin, G. (2006). Meta Math!. Vintage.

[4] Dijkstra, E. (1976). A Discipline of Programming. Prentice Hall.

[5] Field, M., & Field, M. (1995). Symmetry in Chaos. Oxford University Press, USA.

[6] Hammer, M. (1997). Beyond Reengineering. HarperBusiness.

[7] Hammer, M., & Champy, J. (1993). Reengineering the Corporation. HarperCollins Publishers.

[8] Keirsey, D., & Bates, M. (1984). Please Understand Me. Prometheus Nemesis Book Company.

[9] Hofstadter, D. (1980). Gödel, Escher, Bach. Penguin Books Ltd.

[10] Knuth, D. (1992). Literate Programming. Stanford Univ Center for the Study.

[11] Kowalski, R. (1979). Logic for Problem Solving. Prentice Hall.

[12] Kuhn, T. (1970). The Structure of Scientific Revolutions. University of Chicago Press, Chicago, IL.

[13] Paulos, J. (1991). Beyond numeracy. Alfred a Knopf Incorporated.

[14] Paulos, J. (2011). Innumeracy. Macmillan.

[15] Schneier, B. (2004). Secrets and Lies. Wiley.

[16] Wirth, N. (1976). Algorithms + Data Structures = Programs. Prentice Hall.