5-21-2012

Demarcating Computer Science

Dana Burkart

Demarcating Computer Science

Dana Burkart

May 21, 2012

Abstract

Despite its relative youth, computer science has become a well-established discipline, granting over 2% of the bachelors degrees in the United States (U.S. Department of Education, 2010). For this reason, it is important that we understand the nature of computer science and the likely direction for the development of inquiry in computer science in the future. This paper examines several perspectives on the nature of the methods of computer science inquiry. These are empiricist methods, rationalist methods, and an engineering stance. It argues that empiricist and rationalist stances play identifiable roles in the scientific nature of computer science reasoning but that the engineering stance does not. Following the trend in the maturation of other sciences, this paper recommends an overhaul in computer science curricula.

1 Identifying Science: Necessary and Sufficient Conditions

Physics, chemistry, biology, psychology: these are all unequivocally scientific fields. Labeling a field as scientific means that it aims at methodically producing formal or empirical knowledge claims. At the same time, by affixing the honorific of 'science' to some field, we are saying that its knowledge claims advance toward truth. Other fields include soil science, materials science, packaging science, computer science, information science, library

science, political science, food science, creation science, forensic science, military science, and "the dismal science." Is there a nature to scientific inquiry, such that some or most of these are *legitimate* sciences while others fail to meet the epistemic ideal which entitles them to be 'sciences'?

The demarcation problem of science is the question of how to distinguish between fields that *claim* to do science and those that legitimately do. This problem is not so easily answered: to distinguish between science and non-science, it is evident that we need a definition which will allow us to specify some set of fields which are all considered science. At the same time, it should leave out no scientific field, and similarly, should not let in any non-scientific field. To put it succinctly, our definition should keep science (as we consider it) *in* and non-science *out*. This is the requirement that any set of demarcating criteria must be both necessary and sufficient.

As it was first described by Aristotle, science is the field of inquiry that examines first causes (Aristotle, 1994). This meant that a scientific discipline examined not the 'what' of some phenomenon, but the 'how', and such a discipline almost never examined the 'why.' This description of science held until its practicality was reconsidered. Under the Aristotelian view of science, something like astronomy could not be considered science because it is merely descriptive and first causes are not considered. In other words, astronomy only categorized and catalogued the stars, took note of 'what' was out there — it did not answer the 'how.' Later, conceptions for what makes something 'scientific' began to change, and the idea that there could be a scientific approach to something (such as early astronomy) was becoming accepted (Peirce, 1957).

One more specific goal of demarcation is to identify pseudo-science. The demarcation project also seeks to distinguish natural science from other legitimate and related means of knowledge production, such as social science, engineering, and medicine. To say a field is non-science is not necessarily to make the claim that it is not producing knowledge. However, it may be a claim about *how* that discipline is going about knowledge production.

The main angle of approach when it comes to demarcating science has been to identify certain demarcation criteria. The idea here is to come up with a single set of criteria that is both necessary and sufficient. One such criterion, proposed by Karl Popper, is that of *falsifiability*: scientific theories must be able to be falsified. An objection to falsifiability is raised by Larry Laudan where he claims that with Popper's falsifiability criteria, we are forced to acknowledge as scientific "every crank claim which makes ascertainably false assertions"(Laudan, 1983, 121).[1] At first, this objection seems completely off target: Laudan seems to be making the claim that because something has been falsified, it is not science, nor was it ever. This objection is based on certain conceptions about how science is done and is really a methodological objection. Laudan is making the claim that just because a piece of knowledge *can* be falsified, that does not make it science. How we arrived at that piece of knowledge is the relevant consideration for determining whether or not it is science.

Other attempts to demarcate science, such as that by Thagard (1978),

[1]It is interesting to note the subtle distinction between claims that *can* be scientific if we are willing to reject them and the same reformulated claims that cannot because we do not have the same willingness. Does this reflect a certain attitude which makes some endeavors scientific? Or at least an attitude which makes some people scientific?

seek to define science as always progressing. Most pseudoscience, such as astrology, have given up on proposing and testing *new* claims. The criterion is insufficient, though, as it admits ufology into the fold (Laudan, 1983). Furthermore, it is not clear whether the progressive nature of an area of inquiry is always easily identifiable. Whether or not a field is 'progressing' must be determined within the field itself and may not always be clear to an outside observer.

A strong candidate for an acceptable demarcating principle comes from David B. Resnik, who argues that it is unrealistic to expect a solution to the demarcation problem to provide necessary and sufficient conditions (Resnik, 2000). Instead, he suggests that the best we can do "is to provide a list of criteria that we associate with disciplines, theories, methods, concepts, or people that we call scientific" (Resnik, 2000, 257). He brings to light practical reasons for considering the demarcation problem, such as public education, medicine, engineering, research funding, and others. In order to answer the demarcation problem, claims Resnik, we must know who is trying to distinguish between science and non-science, as well as why that distinction is wished for. In this way, a consequentialist approach is taken: i.e., the consequences of making a wrong distinction are taken into account when setting up the stringency for evaluating a given field.

This view opens itself up to criticism because of its contextualist approach — in a way, calling something a science under Resnik's pragmatic scheme does not tell us much about the nature of any particular discipline. To understand what it means to call something science in this pragmatic sense means we must ask a follow-up question: given the stakes, does the

discipline satisfy enough conditions for it to sit well? This is problematic because the term science itself has now lost its meaning. We have given up on asking whether or not the discipline is well-founded and is accurate and ambitious enough in its knowledge claims. Instead, we are asking about history and social membership.

The important point to draw from Resnik is that it may very well be a fool's errand to try and find some set of conditions which is both necessary and sufficient. The best we may hope for could be to offer some kind of template against which to measure candidates. Drawing on the work of others, I will propose one template for the empirical sciences here:

1. (Theories) The candidate discipline must propose new theories.

2. (Falsifiability) Theories within the candidate discipline must be able to make predictions which can be falsified.

3. (Progress) The candidate discipline must be able to show some sort of progress.

4. (Disambiguation) The candidate discipline should display a history of disambiguation. In other words, the scope of the field should *decrease* over time as it becomes more specialized and as related areas of inquiry become autonomous.

5. (Lineage) Being a candidate for an empirical science, the discipline should have roots in some other established (empirical or formal) science(s).

These five conditions together represent a fairly comprehensive set of criteria which can be used as a template.

The first condition seems obvious; however, it keeps out some disciplines which, while being science-adjacent, are nonetheless not rightly science. A couple of examples of this are many forms of engineering and clinical medicine. These disciplines, while certainly involving science, do not necessarily exhibit theory-creation.

The second condition makes sure that the discipline's theories allow the right kinds of questions to be asked. Theories within the discipline must be testable.

The requirement of progress keeps out fields which may have once been science but are no longer — this makes sense of statements such as "geometric optics used to be a science, but is now more of an engineering discipline" (Kuhn, 1996, 79).

Disambiguation, our fourth criterion, makes a somewhat bold normative claim about science: scientists (like philosophers) should give up problems they have figured out and those which pose enough new questions to justify delegating them to autonomous disciplines. Furthermore, when faced with a seemingly insurmountably complex problem, sub-disciplines should be created to deal with the intricacies, thus "disambiguating" the field.

Lastly, the requirement of lineage means that for something to rightly be called science, it must originate from some well-established science, be that a formal or empirical science.[2]

[2]As this is a template for empirical sciences only, we are not faced with a problem of regress, and can leave the question of where formal sciences are ultimately rooted for others to answer.

Of these five criteria, the one which requires the most defense is disambiguation. The property of disambiguation is central to the construction of a scientific field. Disambiguation serves a very practical purpose: it provides division of labor to the scientific enterprise. Without this property, scientific fields become unsustainable in a couple of ways. Becoming a practitioner of an 'ambiguous' field could be prohibitive in some cases, as the amount of knowledge encapsulated by the field is greater, and greater tenacity would be required if the practitioner is to have more than a shallow understanding. Furthermore, without disambiguation, factions may develop and practitioners might become combative. In a field with such factions, peer review lends itself to *ad hominem* attacks and over-critical refereeing.

It is important that within a field of inquiry (at least a field with well-founded norms), its practitioners agree about the nature of what they are doing. There could possibly be different schools of thought on how to carry out the science, but there should be very little dispute about what the research program undertakes to investigate.

With this template in hand, we can continue into a brief history of computer science.

2 A Short History of Computer Science

The question of whether computer science is a science is a misleading one — it presupposes that a set definition for computer science can be had. Empirical fields are easier to demarcate because the details of these fields have already been worked out. For example, it is only because we know

what it entails to do biology that we can even *begin* to answer the question of whether biology is science.

Computer science as a field grew out of mathematics in the late 1930's with Alan Turing and Alonzo Church theorizing what is now known as the Church-Turing thesis.[3] This thesis was a mathematical theory about what is computable by a special class of machines, 'Turing machines,' and it plays a central role in computer science itself. Turing machines describe with mathematical accuracy a kind of abstract machine which can compute every function which is computable.

Around the same time, a turing-complete class of machines were being built independently and with no knowledge about these theoretical advancements by a man named Konrad Zuse (O'Connor and Robertson, 2012). Thus, computer science in its formative years was as much the study of the computing machines themselves as it was a formal science. As such, early views about the nature of computer science held in one hand the computers themselves, and in the other what the computers were doing. Quickly, this situation became untenable as increasing advancements in computing hardware called for some practitioners of computer science to become highly specialized in computer hardware design. A clean break was obviously needed between those building the machines and those theorizing about them. The separate discipline of computer engineering was created to acknowledge this. Computer scientists were content to tighten their belts and focus mostly on what was done with computers, requiring perhaps a single computer organization course of new computer science students.

[3]For a quick treatment of Turing machines, see (Hagar, 2007, §2).

Since then, three prevailing views on the nature of computer science have predominated: that of the computer scientist as rationalist, as empiricist, and as engineer. This tripartite approach to defining computer science is recent, although perhaps not in computer science years.[4] The approach is well-documented, and is divisive among many computer scientists (Newell and Simon, 1976; Eden, 2007). On the whole, these three views make a couple of claims about computer science: what, specifically, computer scientists ought to be doing, and the generalization which follows — what the field as a whole should be accomplishing. While the purposes of these three 'factions' are different, the lines dividing them are not so distinct, and methodological distinctions some try to claim exist seem to be confusing at best and misleading at worst.

So to understand (or begin to understand) what it means to "do" computer science, we should look at what it is modern computer scientists do. In the next section, I will put forward some ideas about the nature of computer science. This will help us understand how the three paradigms of computer science differ.

3 Computer Science As Program Theory

Fundamentally, computer science has been the area of study involving the machines called computers and the programs that are run on them. Early conceptions of what the field was about reflect this, calling computer science "the study of the phenomena surrounding computers" (Newell and Simon,

[4]These are sort of like dog years; the short history of computer science means that just a few decades span a quarter of the field's age.

1976). This view was empirical in the sense that the machines were being experimented upon — before a new tradesman learns her craft, she must learn to use her tools. So early computer scientists were, in a sense, studying the computers themselves — hence the unfortunate naming of computer science.[5]

It is not that we have stopped doing this sort of experimentation in computer science: we have simply started separating out areas of our field which are becoming fields in their own right. This sort of individuation happens in many disciplines. The inception of computer science is owed to mathematics, and already the organization and design of computers has broken off into the distinct field of computer engineering.

Today, when we say that computer science is an empirical science, we have a different meaning. We want to say that when a computer scientist is doing an experiment, the role of the computer is that of a tool, much like a microscope. We would not say that a biologist does experiments *on* her microscope. Comparisons between computers and microscopes may seem ludicrous; however, this is only because the machinery and purpose of such an instrument seems so tangible and is well understood. To make this point clearer, we need only turn to electron microscopes, as they possess inner workings that are more abstract and not as well understood by the layperson. The disparity between the 'facts' (details on the order of atoms) and their representation is now more apparent.[6] In much the same way, we would like to claim there are 'facts' that the computer is able to represent. We describe

[5]Unfortunate in the sense that computer scientists may have *started out* studying the computers themselves, but that is no longer truly the case.

[6]For a thorough treatment of this topic see Hacking (1981).

a computer as the tool that allows us to access these representations.

So if we consider computers to be the tools, what is it that computer scientists are studying? In short the answer is that computer scientists study algorithmic programs and processes. Denning et al. (1989) identify computer science as "the systematic study of algorithmic processes — their theory, analysis, design, efficiency, implementation, and application — that describe and transform information." The distinction here between 'program' and 'process' is important. A program is the *a priori* knowledge about the process — the blueprint — while the process is that program being run on some computer.[7] With this distinction in hand, we can identify the rationalist computer scientist:

> The *rationalist paradigm* [...] defines computer science as a branch of mathematics, treats programs on a par with mathematical objects, and seeks certain, a priori knowledge about their 'correctness' by means of deductive reasoning (Eden, 2007, 135).

This satisfies a definition of computer science as being a type of mathematical endeavor. However, most computer scientists would probably agree that this does not characterize everything which falls under the umbrella of computer science.

For the (modern) empiricist and what concerns her, the rationalist approach to computer science is not sufficient. Having *a priori* knowledge

[7] In much the same way as DNA is the blueprint for biological life. Or, to put it differently, the program is the equivalent of a *fact* which the process is a *representation* of.

about a program will not do, for the simple reason that for the types of programs she studies, full understanding may not exist. These programs are more akin to theories, and as such, complete knowledge about them cannot be claimed without experimentation. The programs that empiricists experiment with are generally descriptive in nature: they might describe some natural process or simulate some system. So, in this way, the empiricist is interested in processes, not only the programs that describe them. Eden provides this definition for the empiricist:

> The *scientific paradigm* [...] defines computer science as a natural (empirical) science, takes programs to be entities on a par with mental processes, and seeks a priori and a posteriori knowledge about them by combining formal deduction and scientific experimentation (Eden, 2007, 135).

In this definition, the scientific paradigm seems to be a sort of superset of the rationalist paradigm. However, I would amend Eden's definition to be more precise about what it is empiricist computer scientists study. In addition to being interested in programs, computer scientists in the scientific paradigm study processes. Eden acknowledges this by asserting that they "take programs to be entities on a par with mental processes"; however, it is not the program itself by means of which empiricists do experiments, rather it is the process that is created by running the program.

Of course, the separation between empiricist and rationalist is not great — it is expected that each is well acquainted with the other. At the same time, sub-fields usually exhibit proclivity for one paradigm over another.

For example, computer scientists doing work in artificial intelligence, while certainly employing rationalist methods, operate empirically. The systems being built are usually complex and hinge on theories of intelligence for which there is little or no *a priori* knowledge. Theoretical computer science, on the other hand, is one of the few sub-fields within computer science which fully satisfies the rationalist paradigm and shows few signs of the other two. Almost all of the work done in theoretical computer science is done by way of formal proofs, and it easily may be the case that some theoretical computer scientists need not write a single line of code.

4 Verification - A Sticking Point

This brings us to program verification, which is a common thread throughout computer science: no matter which paradigm we look at, we can find some idea or other about how verification should be carried out. Verification, as we will use the term, is the attempt to verify a program's correctness. This, in turn, brings to light the question: what does it mean for a program to be correct?

Initially, we can define a program to be correct iff all inputs to the program result in correct outputs. However, this definition is problematic: its very statement is circular. *All* output of some program is sure to be correct in relation to its input for the program *as it is written.* In other words, assuming the program is valid, any errors made when writing it may change the output; however, the output will still be correct in relation to

the program itself.[8]

Surely, this is not what we mean when we say a program is correct, so what really makes some output correct given its associated input? We answer this by saying that the output of some program is correct in relation to the given input iff the program *as specified* would generate that output.[9] For this reason, program correctness is a nebulous concept at best — the correctness of a program lies in how well the specification of that program corresponds to the implementation of it. To put it another way:

> (Correct) A program is correct iff for each input X we get expected output E.

As the prevailing view of program verification among philosophers of computer science goes, a demarcating feature of rationalists and empiricists is how they go about verifying programs. The rationalist seeks formal proofs to verify the correctness of programs. They try to show mathematically that as it is written a program serves its intended purpose. Empiricists, on the other hand, verify programs through experimentation — a program accepting some input is written, and then run and checked against expected output. Its correctness is a measure of how well its actual output corresponds to its expected output.

The main argument against formal verification is the monumental difficulty of proving a non-trivial program formally correct. Another objection

[8] I take a valid program to be one with no syntactical errors. In other words, it compiles (if written in a compiled language) or runs (if the language is interpreted).

[9] When we talk about a program as it is specified, or the specifications for some program, we are talking about the 'plans' or purpose of that program. In other words, program specifications are a description of what the program is supposed to do. If the program *as written* corresponds to its specifications, we call that program correct.

comes from Colburn (1991, 2003) where he argues that there is an inherent flaw in the rationalist method of verification. The adequacy of formally proving a program correct rests on the assumption of the formal correctness of all subsystems affecting the program in question. A formally verified program is only proven to be correct on an ideal machine. In reality, the program relies on expected behavior in subsystems, which, unless exhaustively verified by separate formal verification proofs, means that incorrect assumptions were used in the higher-level proof. Bugs in underlying software systems and glitches in hardware might result in unforeseen behavior in a formally proven program.

These objections, in my mind, are pragmatic in nature. They say nothing against the validity of formal verification. No method of verification guards entirely against unforeseen material flaws in hardware. Furthermore, given enough resources, we could theoretically have an entirely formally verified system on which to run formally verified programs. This, of course, would not be practical, but it serves to illustrate the validity of formal proofs. Because such a system does not exist, this objection simply points out the infeasible but not impossible nature of them.

A bigger problem is that of 'specification verification.' Given some specification for a program which we are verifying against, what assurance do we have that the specification itself is correct? In other words, how do we verify a specification? Is it even possible to verify program specifications without falling into some sort of regress? Any error in the specification of some program will result in a formally-verified program which exhibits behavior we probably did not want. In this way, it seems futile to attempt to

prove the correctness of any program.

An empiricist, on the other hand, seeks to verify her programs using experiments. This can be attributed to the nature of the programs being written — generally, some sort of external system or process is being modeled. Usually these systems are natural. For example, a computer scientist looking to examine theories about the human visual system would write a program to simulate the particulars of the different theories under examination. The rationalist approach to program verification will not work here, since in this case the computer scientist does not know if her specification is correct as far as the system is concerned. The only means of verification in this situation is that of designing an experiment on the program and some input, and gauging the results against what we expect.

Of course, computer scientists need not be empiricists in just this naturalistic way; there are other systems which are empirically verified by computer scientists. Examples of this are database systems: although they are grounded in solid theory, the properties of them are empirically verified.

Here is where computer science as an engineering discipline becomes relevant. A parallel between the empiricist's method of verification and the engineer's *reliability test* can be drawn here. In fact, reliability tests are analogous to verification in software engineering. Opinions differ on this, however, and the high road is often given to the empiricists. Eden argues that there is a clear distinction between *reliability tests* and experiments. He describes the distinction:

> The purpose of a reliability test is to establish the extent to

> which a program meets the needs of its users, whereas a scientific experiment is designed to corroborate (or refute) a particular hypothesis. (Eden, 2007, 155)

I would argue that Eden is drawing a line in the sand where there should be none: just because the expected output comes from users instead of some other source does not mean the engineer and scientist are doing something different. They are still verifying program correctness, just based on different criteria. In fact the methods by which the empiricists and engineers of computer science verify program correctness are more similar than dissimilar. Reliability tests are different sorts of experiments, but experiments nonetheless. In the same way that the empiricist does not know if her program is correct in relation to the system it models, the engineer does not truly know if her program "meets the needs of its users," but the program's output might still be unexpected.

This has traditionally been how philosophers characterize the argument over verification, but the actual state of affairs has been misrepresented. Very few computer scientists buy into formal verification full-throttle. Instead, they seek to verify certain aspects of programs. For example, certain languages have what is called *typing* — this means that the type of data held in variables is specified when the program is written. Programs with this quality (often called strongly-typed languages) have the nice property that certain kinds of errors can be caught before runtime. Another property we may want to verify is that all paths of execution are reachable in the program — any unreachable path can be optimized out, thus allowing the program to

be more space-efficient. Such properties are almost never verified by hand. Whenever computer scientists talk about formal verification they are almost certainly referencing some form of *automated* formal verifier or other. An example of such a verifier is ACL2, which is "A Computational Logic for Applicative Common Lisp." This system allows the computer scientist to write a LISP program, write properties (as LISP programs themselves), and ACL2 will prove that theorems hold.[10] In general, when computer scientists talk of formal verification, they mean that they can formally verify certain properties about a program. This is an important distinction because formal verification as understood by computer scientists is not incompatible with the type of verification proposed by empiricists. In this way, more of what computer scientists do can be accounted for without needing competing views about verification: empiricists can make use of these automated tools in addition to their empirical methods of verification.

5 Applying Demarcation to Computer Science

Recall our template from the first section:

1. Theories
2. Falsifiable
3. Progress

[10] Some other formal verifiers include:
SPIN – verification of concurrent systems / programs
Coq – theorem proving system with extraction
Boogie, Spec# from Microsoft

4. Disambiguation

5. Lineage

Until this point, I have argued for an empirical view of computer science, one which incorporates elements from the formal paradigm. My next task is to apply the template for an empirical science I constructed in the first section, and see how well it fits.

Do computer scientists come up with **theories**? The answer to this question is somewhat hard to see. Experiments are conducted and hypotheses tested, so there must be some sort of underlying theories. The Church-Turing thesis is *the* central theory of computer science, and there are many theories which lie tangential to that one. These theories allow computer scientists to make predictions and design experiments which can be used to test theories and possibly falisify them — and so computer science easily satisfies the requirement for **falsifiability**. **Progress** is perhaps the easiest to see — the rapid pace of advancements over the span of the field's history points to progress.

Disambiguation is the first of our principles which poses a prima facie problem. The problem is easy to see upon examination: the simple fact that there is more than one paradigm (each with staunch supporters) that computer scientists align themselves with means that computer science as a field is not adequately disambiguated. We have, so far, listed three paradigms: that of the rationalist, empiricist, and engineer. Further, we have argued for the empiricist view over the formal one, and shown that as far as is necessary we can incorporate many, and perhaps all, of the formal concepts

within the empiricist paradigm. Talk of the engineer, however, has been spare. This is because it is here that the differences cannot be reconciled — it seems we must simply choose one over the other. The goal of the engineer is too different from that of the empiricist or rationalist — for him, computer science is a much different endeavor. The engineering of software solutions is his primary goal, not the advancement of knowledge in the field. While no normative claim is being made, these two paradigms being interwoven is harmful to computer science as a whole.

This is not a new problem, and computer science is not the first field to have faced such a crisis. Fields such as biology and chemistry have faced a similar situation, and both have resolved it by differentiating separate sub-fields, such as bioinformatics, genetics, biochemistry, and flavor chemistry. Computer science itself has already done this once with computer engineering, physics with many engineering disciplines, math with computer science. For computer science to truly disambiguate itself, it must separate out software engineering as a distinct discipline. Eden presents a normative argument for separating software engineering, claiming that many 'hard' computer science classes are being dropped from educational programs in favor of software engineering-oriented classes (Eden, 2007). His concern is that keeping the engineering paradigm within computer science will cause the field to shift away from making the sort of scientific progress it can.

Our last principle, that of **lineage**, is easily satisfied. The conception of computer science took place within mathematics in the 1930's with the question of what was and was not a computable function. Much of the opposition to thinking of computer science as a science stems from this

lineage, as it is harder to see how a discipline with formal roots can be an empirical science. But as the formal aspects of computer science become more a part of the toolset and less the main focus of research, the practice of computer science comes to resemble an empirical science more and more.

6 Importance of Curriculum

Scientists are not born — they are shaped, at least in part, by their education. This is something many scientists perhaps lose sight of — that although science is not *only* method, forming good habits helps determine the quality of the scientist. This is not a new idea — Thomas Kuhn calls this part of the *disciplinary matrix*, and attributes much of a scientist's success to her education (Kuhn, 1996). This disciplinary matrix is a specific aspect of Kuhn's more general *paradigm*, and it indicates the entire scope of a field: the education, practice, and basis for that discipline.[11]

It is important, then, that the curriculum of undergraduate "future scientists" reflect what the end product is meant to be — a well-trained scientist who is able to operate within the disciplinary matrix of her field. Computer science, with software engineering included (sometimes even being the focus) in the curriculum of many undergraduate programs, undermines this goal. Just as with computer engineering, only a single software engineering course should be required of students. Some schools have already separated

[11] Kuhn's idea of a paradigm denotes much more than what I have been using the word 'paradigm' to indicate. It indicates not only a way of looking at some aspect of the world (as it relates to a discipline), but a way that is incommensurable with its predecessors. Kuhn's paradigm is too broad for discussion here, so my use of the word will be roughly analogous to Kuhn's disciplinary matrix.

out software engineering as a separate area of study, allowing those not interested so much in the 'science' to focus on aspects of software design.

7 Conclusion

Before we can identify computer science as being a science or not, we must first demarcate computer science from computer engineering and software engineering. Computer science must disambiguate; software engineering should be completely separated out as a distinct yet related discipline. Not only are there philosophical and scientific reasons for doing this, there are strong normative reasons as well. Software engineering is an integral part of society and is the most practical application of the many discoveries made in computer science. Keeping it intertwined with computer science does a disservice to both disciplines.

The damage done to computer science by watered-down curriculum is much greater than to software engineering, however. Software engineering, being so readily applicable, lucrative, and in such demand means that computer science departments which haven't yet made this distinction have begun supplanting the bread-and-butter courses of computer science with software engineering courses which rightly belong in another major.

References

Aristotle (1994). *Posterior Analytics*. Clarendon Press.

Colburn, T. R. (1991). Program verification, defeasible reasoning, and two views of computer science. *Minds and Machines 1*, 97–116.

Colburn, T. R. (2003). Methodology of computer science. In *The Blackwell Guide to the Philosophy of Computing and Information*. Blackwell Publishing.

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

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

Hacking, I. (1981). Do we see through a microscope? *Pacific Philosophical Quarterly 62*, 305–322.

Hagar, A. (2007). Quantum algorithms: Philosophical lessons. *Minds and Machines 17*(2).

Kuhn, T. S. (1996). *The Structure of Scientific Revolutions*. The University of Chicago Press.

Laudan, L. (1983). The demise of the demarcation problem. *Boston Studies in the Philosophy of Science 76*, 111–127.

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

O'Connor, J. J. and E. F. Robertson (2012, May). Zuse biography. `http://www.gap-system.org/~history/Biographies/Zuse.html`.

Peirce, C. S. (1957). *Essays in the Philosophy of Science*. Liberal Arts Press.

Resnik, D. B. (2000). A pragmatic approach to the demarcation problem. *Studies in History and Philosophy of Science Part A 31*(2), 249–267.

Thagard, P. R. (1978). Why astrology is a pseudoscience. *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association 1*, 223–234.

U.S. Department of Education, National Center for Education Statistics, H. E. G. I. S. (2010). Degrees and other formal awards conferred surveys, 1970-71 through 1985-86; and 1990-91 through 2008-09 integrated postsecondary education data system. http://nces.ed.gov/programs/digest/d10/tables/dt10_282.asp?referrer=list.