Reflections on What is Software Engineering

Ewan Tempero

December 19, 2008

Introduction

This essay presents various thoughts I have had regarding the nature of software engineering, based on my experience due to teaching into a software engineering university programme and research in the area.

The Question

The starting point for this essay is the following question:

> Is there a difference between computer science and software engineering and if so what is it?

This is a question I have had to come up with a good answer for as, since 2002, I have been at the University of Auckland (UoA), which allows a Major in both Computer Science (CS), in the Bachelor of Science (BSc), and a Specialisation in Software Engineering (SE), in the Bachelor of Engineering (BE). I am in the Computer Science department, but since most of my teaching was in the SE programme I have had to answer this question both from potential students and potential employers. More recently, as I was preparing to go away on Research and Study Leave (our equivalent of "sabbatical"), I came across a discussion on this question in Software Engineering Notes (SEN) giving opposing views on the subject. What was interesting about this discussion is, had I seen it when it when it was published in 2003 I would have agreed with one side, but I am now firmly on the other side. Since reading this discussion, and having the luxury of time on leave to think about such things, I've been thinking about why my views have changed, which has lead to what you see before you.

Historical Context and Motivation

At the beginning of 2002, the SE degree at UoA was quite new — the first cohort had just begun their third (of four) year. Being so new, there was not a lot known about it. Potential new students (and their parents) wanted to know if there were jobs for SE graduates, and why they should take SE instead of CS, which had been around for many years. Industry wanted to know what the difference was, in order to decide whether they wanted to hire SE graduates instead of (or even as well as) CS. "There is no difference" was not only not a politically useful answer (there must be a difference otherwise why is UoA offering both!), it also clearly wasn't entirely correct. While a number of the courses in the SE programme began life as clones of their CS counterparts, most of them diverged to be different courses. There were also other SE courses that had no equivalent in CS, plus there were all the other engineering courses that made up the BE. So the programmes are different, but that could be considered a somewhat artificial distinction. Much more interesting is whether there is a true difference between the two.

The question is not new. There have been many discussions in various venues, including many staff tea rooms, as to what the difference is. The SEN discussion I mentioned above appeared in the November 2003 issue and consisted of a letter by Bill Griswold and a response by Peter Henderson [3]. I came across this issue as I was cleaning up my office in preparation for going on leave. I had somehow neglected to read it at the time it came out and so I took a quick look at it (SEN being one of the few periodicals I receive that I make some effort to read).

In his letter (which was a response to a proposal for an SE curriculum), Griswold expressed the view that CS is Engineering, based on the observation that much of what is associated with CS involves the construction of artefacts. He felt that SE was a sub-discipline of CS, and questioned whether making any distinction was of benefit for the development of the respective fields, in research and in teaching. Henderson (who at the time wrote a column for SEN on software engineering education) felt that there was a difference between CS and SE just as there is a difference between science and engineering. He felt that many graduates of CS programs don't have an "engineering mindset".

At the end of 2003 (that is, after my second year of teaching in the SE programme at UoA), I probably would have agreed with Griswold. I almost certainly would have agreed even only months earlier. My previous appointment of 11 years had been in the Computer Science department at Victoria University of Wellington (VUW), which offered a fairly traditional CS programme. My own background featured fairly traditional computer science courses, however my undergraduate major was Mathematics and my PhD was in theoretical computer science. By the time I left VUW

my research and teaching were firmly in "software engineering" but I considered it a sub-discipline of CS rather than something separate.

At UoA, while I was in a department in the Faculty Of Science (FoS), as most of my teaching was in the SE programme, I had more contact with the Faculty of Engineering (FoE), which administered the SE programme. The structure of the SE specialisation was strongly influenced by the structure of the BE, administered by the FoE, which had existed for nearly (at that time) 100 years. There were "professional development" courses at every level for all specialities, and "mathematical modelling" (featuring lots of calculus) for most specialities. The designers of the SE speciality were almost entirely in the Computer Science department (FoS) and had argued successfully that SE students didn't need all of the continuous mathematics, and that more appropriate courses would replace those courses the other engineering students took (discrete mathematics and so on). However there was no avoiding the professional development courses, as they were part of the engineering accreditation, and there was no avoiding the joint first year that all students took. The first year featured almost nothing relating to software (unless the half-course of programming in MATLAB counted). Fortunately for the SE programme, there was one "elective" slot at first year, and SE students were required to take the Computer Science introduction to programming course in that slot.[1]

I found the restrictions to the SE programme imposed by the BE structure just a little bit unhelpful. I did not see the need for an SE student to take a chemistry course in the first year. I'm not saying that the chemistry (and I pick on chemistry only because I don't want to list all courses) is no use, or that SE students would benefit from seeing it, but four years is little enough as it is and that chemistry slot could be used for a course containing much more directly relevant material for SE students. The line by the engineering staff as that they wanted all students to have the same foundation so that the students made their choices of speciality on an informed basis, and that as these courses were all taught by engineers it would help the students to acquire the "engineering mindset". I didn't buy it. At the time I didn't see that whatever mindset mechanical or chemical engineers had developed was going to be of any value to SE students.

At the end[2] of 2003, the first cohort completed the programme. This included a "capstone" project course where students worked in teams of two on (mostly) staff-sponsored projects. This course was "capped off" with a 2-day conference where all project teams presented the projects and then the following week there was an exhibition day open to the public that students could show off their projects.

This is when I got to see, for the first time, a "finished engineering student" (I am carefully not saying said student was a "finished engineer"!) I definitely saw something in this cohort that I had not seen in computer science graduates. It's hard to define. There was still the aspect of "we've done some cool stuff and we use cool technology" but there was also a clear indication of "we've paid attention to what the client wants" and "we've paid attention to what industry practise is." I believe this was a sign of this "engineering mindset" that everyone had been talking about.

This was the point at which I started to became less inclined to dismiss the "engineering" way of doing things, and began to pay more attention to how they did things. I still would prefer that the SE programme was a little different, but am now engineer enough to understand that's the reality that I have to deal with. As I don't know exactly what is leading to the engineering mindset, I'm less inclined to throw out courses just because I don't see they are useful, as it may be that those courses that provide the necessary foundation (that is, maybe the "traditional" engineers are right afterall!) I now also understand that there may be some value to making a distinction between computer science and software engineering.

Distinguishing on Principle

Of course attaching labels to things has its dangers. People tend to evaluate on the basis of the label rather than the actual thing, and labels are usually far too simple for that to be the right thing to do. People use labels as flags to rally around, to create "us" and "them", when other means of solving problems may be more helpful.

But we use labels to solve a problem, and its a type of solution we use to solve many problems — divide and conquer. We find we can better reason about things if we can just isolate the things we care about. If the labels provide useful distinctions, then we may be able to make more informed decisions.

For example, with a clear distinction between SE and CS, departments can, rather than debate whether or not they are doing SE or CS, can decide whether they want to be a "computer science department with a strong SE focus" or a "software engineering department" or accept that they really do do both, or whatever. Rather than argue that certain topics are "clearly important" and so "must be taught", we can examine whether the topics are more relevant to CS or SE and include them (or not) on the basis that the course is intended for a CS or SE programme, rather than trying to defend this nebulous "clearly important" criteria.

So, if we were to make a distinction between Computer Science and Software Engineering, what should it be? A reasonable starting point is to consider the difference between Science and Engineering. There are

[1] The first-year programme is now somewhat different, with the changes significantly affecting the SE programme.

[2] In the southern hemisphere academic years tend to align with calendar years

many learned discussions on this but rather than rehash them here I will stick to something simple:

> Scientists try to understand reality. Engineers try to deal with it. [3]

Like all slogans it presents a simplistic view of what is actually the case, but like all slogans it has elements of truth about it. It captures a principle that can be used to decide when something could be considered "science" and when it could be considered "engineering." The distinction is that a scientist's primary intent is to understand how the world works, whereas the engineer's intent is to build things that have to work in the world. There are many things that scientists "know" (in the sense that scientists can be said to know anything) to be true, but knowing these things doesn't impact everyday life. But then some engineer will realise that that thing can be exploited to provide some service or product that otherwise could not be done, or to provide existing services and products more efficiently.

In many discussions I've seen on what Engineering is, there has been an emphasis on building something. However I don't see that as necessarily a defining feature of Engineering, especially in the sense that building something means it is not Science. Lots of building goes on in science. Two examples are the Hubble telescope and the Large Hadron Collider. There is no question that these involved some serious engineering, but their entire purpose is to support fundamental science. Just because something is being constructed, doesn't mean no science is occurring; the question is what purpose the construction serves.

One possible purpose of construction is to determine what level of understanding science has reached. One way to test a theory is to look at what artefacts it predicts can be built, and try to build them. Scientists often call this kind of activity an "experiment", but that doesn't alter the fact that something is being built. An example here is Dolly the sheep, which represented a demonstration of the level of understanding of cloning (among other things!) but clearly involved the construction of an artefact (and no small amount of biological engineering).

I accept that other interpretations of these examples exist, but I would argue that my interpretation is not inconsistent with what happens, and so it supports my thesis that it is not whether or not construction takes place that defines engineering, but what the purpose of that construction is. If it is to help understand reality, then it is for science; if it results in an artefact that is intended to provide value to someone, then it is engineering.

Of course we take take these arguments to all kinds of extremes. For example, if I'm building something that a scientist will use for an experiment, that is, it is an artefact that will provide value to the scientist, isn't that engineering? Yes! No question. The Large Hadron Collider and my other examples are marvels of engineering. My point is that the fact that such construction took place does not mean no science was taking place — a point I will return to later.

Another point is that, by my definitions, those who do research in engineering are really scientists, and that is true. Engineering researchers operate by the same rules as scientists with regards to gathering and provision of evidence to support their claimed understanding of reality. The kinds of evidence they gather are much more likely to involve construction ("I demonstrate the validity of my ideas about inductive power transfer by building a device that is powered by it"), and the problems they consider will be much more likely to come from problems faced by practising engineers ("Science tells us that we should be able to transmit electricity through the air. Let's see if we can make that practical for everyday use.") but nevertheless they are scientists (and, mostly, engineers too).

This distinction between understanding reality versus dealing with it is useful for making decisions. A potential student who likes to explore ideas and understand how things work may be better off taking a science degree, whereas one who likes to build things and produce final products may be better off doing engineering. In deciding what courses to offer (or rather, which courses to not offer as there is never enough time to teach everything that might be useful in any degree programme), and engineering department will be more inclined to keep courses that focus on the practical realities of making things work at the expense of more "theoretical" courses (reluctantly and with great wailing and gnashing of teeth on the part of some staff).

Separating Computer Science and Software Engineering

The principle I described above is all very good for distinguishing Science and Engineering, but it is not much help distinguishing Computer Science and Software Engineering (or Chemistry and Chemical Engineering for that matter). I need to identify the part of reality that's relevant; I use "computation". So, refining my earlier statement gives:

> Computer Scientists try to understand computation, Software Engineers try to provide computation in a useful form.

For me, CS is about understanding the nature of computation at all levels, whether it's understanding what it means to be computable, determining how computation can be used to at least mimic, if not duplicate, human intelligence, finding the best algorithms to support ad-hoc networks for ubiquitous computing, or how to organise data to support fast solutions to complex queries.

SE, on the other hand, is about applying what computer scientists have learned to construct products that

[3] And artists try to interpret it?

do computation that are of value to people. There are aspects to such construction that really are not much to do with computation but nevertheless cannot be avoided when constructing software. For example, version control is something that those constructing software products need to know about but which could not really be considered a problem relating to the nature of computation (although I'm sure there are some potentially interesting mathematical problems that apply to version control).

It is possible that the lines are more blurred for CS and SE than for other fields (e.g., Chemistry versus Chemical Engineering, although it would surprise me if people in those fields argued differently). It is not uncommon for someone to start with the computation puzzle (e.g., what algorithms to use to support searching a very large distributed unorganised dataset) and that same someone to engineer a solution that makes them and lots of other people are large amount of money. The bit that makes the money requires a great deal of sophisticated engineering, and I doubt that Sergey and Larry would claim otherwise, and I also suspect that they would not argue with an interpretation that says that Google represents a successful (and ongoing) experiment demonstrating the level of their understanding about algorithms for doing search.

So how is making this distinction useful? It can help determine the contents of a course. For example, a Data Structures course for a CS degree would more reasonably contain details about how to perform asymptotic analysis of algorithms and include, for example, detailed discussions about the different kinds of algorithms for sorting. A SE version would more reasonably concentrate on practical issues surrounding choice of algorithms or data structures rather than details of the algorithms themselves, so how the mergesort algorithm works is less important than under what circumstances it is a better choice than quicksort. A reasonable assignment for the CS version of an Algorithms course would be to prove the correctness of Dijkstra's algorithm, whereas for the SE version might require an implementation that works for graphs with more than 10,000 vertices. And I must emphasise, this is not to say that a CS student wouldn't benefit from implementing Dijstra's algorithm and an SE student wouldn't benefit from doing the proof, but with a limited time-budget that doesn't allow doing both, the distinction I make provides criteria on which to base a decision.

A similar kind of argument can be used when deciding what courses to teach. It is less likely that a course on software architecture would appear in a CS programme than an SE one, because software architecture is not so much about how computation works as it is about how to organise provision of computation. That said, a software architecture course that mainly covered the formal aspects of architecture descriptions languages, or the modelling aspects of model-driven architecture would not be out of place in (my vision of) a CS programme. If someone really wanted to teach a software architecture course that focused on development of architectures (e.g., in the vein of Bass et al. [1] in a CS programme, then, in my view, that is a SE course in a CS programme. This is not bad, it's just what it is, and it's better to admit that than to make definitions of CS that allow it to be considered a CS course, or make arguments as to why such a course is really about understanding computation (I can imagine what such arguments would look like but don't see why making them is helpful.)

To give another example, there is nothing inherent in the contents of a standard networks course (e.g., as taught from Tanenbaum [2]) that directly supports the development of software. Learning the sliding window protocol is not going to be much help in figuring out how to build a system providing control software for a sleep apnea treatment device, and nor is understanding the principles of wave propagation as it applies to wireless networks. Of course knowing such a protocol exists, or that there are issues regarding wave propagation that must be considered when installing a system that uses wireless communication may help a software engineering produce a better system, but given the limited time-budget, if something has to go then those topics would be high on my list. And I would hope that any Software Engineering is also a good Engineer, and so knows to bring in the relevant expertise to cover gaps in her knowledge.

On the other hand, given the degree to which software these days uses the Internet in some way, a "networks" course that concentrates more on the top of the network protocol stack, and discusses network routing from a network security point of view, would make good sense for an SE programme. Of course there are other pressures that affect course content; for example if the only staff available to teach a networks course are more comfortable at the bottom end of the protocol stack then it may be better to have them stick to their expertise.

An Engineering Mindset?

Henderson claimed that many CS graduates didn't have this engineering mindset. Having been convinced there may be something to this, and having a better idea of what to look for, I am now inclined to agree with him. I have seen solutions produced by CS graduates that only they can use, as it requires convoluted actions on the part of the user to get the solution to behave. While it is less true now than it has been in the past, it is still the case that CS graduates do not seem to understand that just because they can make their solution work, doesn't mean it is useful to their customer, and they will provide some interesting justifications for why their solution works as it does.

This is even more true looking at the organisation of

the software. CS graduates seem to think that following a set of rules results in good software ("But I make all the fields private so my design is good!") but don't think of the consequences of their decisions in terms of the effect on future "users" (e.g. testers or maintainers). SE may not necessarily produce better designs, but they seem to have a more realistic understanding of how good they are ("Yes we made that field public because we were in a hurry to make the deadline." accompanied by shuffling of feet and shamefaced looks).

Now my observations are based on a fairly small sample so must be viewed with some suspicion! However as they are consistent with this engineering mindset and at least one other person (Henderson) seems to think the same, I am much more a believer. I am certainly less inclined to restructure our BE degree without considerable evidence that doing so won't destroy whatever it is that is creating the engineering mindset.

Of course it's not cut and dried that one programme does and the other does not produce the mindset, but at least at UoA one programme tries to whereas the other does not (and nor is it appropriate that it do so). Now that I am alerted to the mindset thing, I see it more

And I must repeat for emphasis, I am not saying that a "computer science" programme couldn't produce graduates with this engineering mindset, what I am saying is that this is really just an SE programme under a different name. I would go further and say that calling such a programme "CS" is doing noone any favours.

How did we get here?

It is worth a moment to try to understand why a discussion like this one is even necessary. I wasn't around when (for example) Chemical Engineering developed as a separate field, and it wouldn't surprise me if similar kinds of discussions took place, but the speed of the development of both CS and SE has probably played a part. Whereas the study and teaching of chemistry has been around a very long time, the large-scale use of chemistry in an industrial or everyday setting is comparatively recent. While the study of computation has been around since before useful computers existed, its organisation as a field, and particularly its teaching really only developed along with (or even trailing) the need for teaching how to construct software. This has meant that understanding computation and how to make it useful have never really been considered separately. As a consequence, what I am calling SE has often been part of the same departments that provide what I'm calling CS, and so it is unsurprising that many people see little to separate the two, or, that SE is a part of CS.

I should also address another related field, one that often goes under the name of Information Systems (IS). It was pointed out to me some time ago (by someone who was in a CS department but did software engineering) that a significant amount of SE was in fact being done in IS, perhaps (at least at the time) more than in CS departments.

CS departments tend to have one of three flavours — mathematical (because those that founded the CS department came from a mathematical background), engineering (came from electrical engineering), or business (came from a business school). In some cases, the separation has not actually occurred (or been undone), so that we have departments of Mathematics and Computer Science, Electrical Engineering and Computer Science, Computer and Information Systems, and the like. My guess is that in the cases where separation has occurred in the business flavour, what has stayed in the business school has become IS.

Of the three flavours of CS, it is likely that it is the business flavour that has been more interested in what can be done with the result, than what is possible, so it is perhaps unsurprising that there has been more interest in SE in IS departments. However SE as a field has developed to more than just business applications and so it seems to be becoming more a feature of CS than it has in the past.

As an aside, I know of at least one university in the world with separate computer science and software engineering departments (both in the same College, and with the division roughly along the lines I am using, as far as I can tell), and one university with separate CS (Faculty of Science) and Information *Science* (in the Faculty of Commerce) departments, that have a similar division.

If it ain't broke, don't fix it

As I noted at the beginning, I have had a particular reason to think about what the difference is between CS and SE, so in that sense there is a problem. However, perhaps what I am proposing will create more problems than it solves.

I've already noted the problems with labels — anyone who uses what I've written here to support some agenda based solely on the labels "computer science" and "software engineer" without providing my underlying philosophy is misrepresenting my argument. There is nothing I can do about such people so I won't try.

If following my suggestion leads to bad decisions, then that is a problem. However I do not see how it could lead to worse decisions than the current situation (I may be biased in thinking this!). The current situation is that there are arguments over what the contents of courses should be, what courses should be in programmes, and what skillsets graduates should have.

For example, I have seen any number of programmes that claim to be appropriate for anyone considering a career in software engineering. Given that they all provide quite different skillsets it seems hard to believe that they can all be right, unless "software engineering" is interpreted as "writes code," because the only common

feature of these programmes is that their graduates have had to write code during their study.

As another example, I have seen any number of discussions as to what topics should be taught in a course, with well-reasoned (and very passionate) arguments presented on both sides of the debate. It wasn't that the participants were always unaware that they were arguing over CS vs SE interpretations of the course (although that was sometimes the case), but that the different interpretations were declared to be

One specific example is accurately portraying the skillset of graduates. Potential employers often complain that they don't know (or at least aren't happy) with the skillsets of those that they have hired in the past. In least in some programmes in the world, CS graduates can have lots of algorithm development, logic, discrete mathematics, network design, relationship databases (theory, not practise), but very little writing of code, and no consideration of what is needed to determine what the systems is supposed to do or what customers care about (or even that customers exist). Employers expecting such graduates to be prototypical software engineers are going to be disappointed.

Even graduates who have taken those courses that traditionally have lots of code development, such as graphics, operating systems, artificial intelligence, and so on, have not had to think about the practical issues of producing software that someone else will use. The courses they have doing the coding for have not at all been concerned with principles relating to how to efficiently create effective software. So, while such graduates may be good at producing code that does what they want, they don't know (or even care) what's involved in producing code that does what someone else wants. It may be this attitude that most upsets employers!

This issue applies to other programmes too. I have heard claims by some people that their programme (not CS, SE, or IS) produces graduates suitable for SE positions. These claims can be evaluated by determining to what degree the graduates have been taught to provide useful computing (as opposed to just writing code as part of their degree).

According to the principle I am espousing, any programme that claims to provide some qualification for software engineering must contain courses where the main principles being taught are about how to efficiently create good software.

What makes an Engineer

I have been careful to not claim that graduates of software engineering programmes are engineers. This is not true for other engineering specialisations. Anyone who wants to be a certified professional engineer (or whatever the phrase is in your legal jurisdiction) must not only complete a university programme certified by the appropriate engineering body, but must also have some amount of actual engineering experience. Engineering firms know they are not hiring "engineers", but merely "graduates", and it is their job to turn the graduates into engineers, and there are generally various programmes within companies or supplied by the particular specialisation to support this.

The software industry almost entirely fails to take this view. The industry expects to get graduates that are "useful" on day one, and complain about how useless programmes are because this is not the case. Well that's something of an exaggeration. There are plenty of companies that have some sort of training programme for new hires. However I get the impression that these programmes are more intended to train the hires in how the company does things, rather than turning prototypical engineers into real engineers. For example, in some cases even those with some experience have to do the programmes, because they have to learn the Company Way.

Certification

This brings me to the question of certification of software engineers. Other engineering specialisations have various forms of certification. As noted above, their university programmes must be certified and the graduates must complete other requirements and go through a formal certification with the appropriate engineering body.

Arguments have been presented that such certification is not appropriate for software development because we don't know that people with such certification can be guaranteed to produce working software. Such arguments miss the point as to what certification means. The only guarantee that certification might provide is that, the person being certified has, according to the evidence presented, been exposed to the current best practise of some specialisation of engineering. Thus, when that person makes an engineering decision, he or she is likely to make the same decision as other similarly-certified people. If the decision turns out to be wrong, and it can be demonstrated that it is *not* what other similarly-certified people would have made, then that person is liable.

I know I'm am drastically simplifying a very complex concept but I believe that what I have described above is the essence of certification. If I am correct then certification of software engineers is a reasonable idea (and in fact the UoA programme is certified by the New Zealand signatory of the Washington Accord).

Are we there yet?

It's possible that there is synergy created in keeping what I have been calling CS and SE together that would be lost (and be missed) if the separation I have argued for takes place, I really don't know. But, putting on

my Engineer's hat, there are problems that need to be solved in the real world — how to decide what courses to teach, how to decide what the content of the courses are, how to present what graduates have learned to potential employers, and, if your reality includes separate qualifications in SE and CS, the need to explain the difference. These problems need to be solved now, we can't wait for the perfect solution (which any engineer knows doesn't exist). What I have presented may not be the best answer, but it is one that I think is good in an engineering sense — it provides a good enough solution that is of value to people right now. I hope you think so too.

Acknowledgements

This essay was written during 2008 while I was on Research and Study leave. It actually started while on a train going through southern Sweden, while I was a visiting researcher to the BESQ project at Blekinge Institute of Technology, Ronneby, Sweden, whose support I gratefully acknowledge.

I take full blame for everything written here but don't claim full credit. Many of the ideas I've presented are not new, and certainly not due to me. They come out of many discussions I've read or participated in, especially since being involved in the UoA programme. I can't possibly list (or even remember) everyone who has said something in my presence that I have incorporated into my thinking, but I thank you all. One person I will mention is David Parnas, whose response to a question at ASWEC 2005 crystalised my thinking on this subject, and was really the beginning to this essay. He said it better (and shorter!) than me, but those who weren't there will just have to make do with this.

References

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 2 edition, April 2003.

[2] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 4 edition, 2003.

[3] Will Tracz, editor. *SIGSOFT Softw. Eng. Notes, 28(6)*. ACM, New York, NY, USA, November 2003. William Griswold's letter to the editor, Peter Henderson's reply.