Dr. Dobb's

Software Engineering ≠ Computer Science

Software engineering seems different, in a frustrating way, from other disciplines of computer science

By Chuck Connell, Dr. Dobb's Journal
Jun 04, 2009
URL:http://www.ddj.com/architect/217701907

Chuck Connell is a software consultant. He can be reached at www.chc-3.com.

A few years ago, I studied algorithms and complexity. The field is wonderfully clean, with each concept clearly defined, and each result building on earlier proofs. When you learn a fact in this area, you can take it to the bank, since mathematics would have to be inconsistent to overturn what you just learned. Even the imperfect results, such as approximation and probabilistic algorithms, have rigorous analyses about their imperfections. Other disciplines of computer science, such as network topology and cryptography also enjoy similar satisfying status.

Now I work on software engineering, and this area is maddeningly slippery. No concept is precisely defined. Results are qualified with "usually" or "in general". Today's research may, or may not, help tomorrow's work. New approaches often overturn earlier methods, with the new approaches burning brightly for a while and then falling out of fashion as their limitations emerge. We believed that structured programming was the answer. Then we put faith in fourth-generation languages, then object-oriented methods, then extreme programming, and now maybe open source.

But software engineering is where the rubber meets the road. Few people care whether P equals NP just for the beauty of the question. The computer field is about *doing things* with computers. This means writing software to solve human problems, and running that software on real machines. By the Church-Turing Thesis, all computer hardware is essentially equivalent. So while new machine architectures are cool, the real limiting challenge in computer science is the problem of creating software. We need software that can be put together in a reasonable amount of time, for a reasonable cost, that works something like its designers hoped for, and runs with few errors.

With this goal in mind, something has always bothered me (and many other researchers): Why can't software engineering have more rigorous results, like the other parts of computer science? To state the question another way, "How much of software design and construction can be made formal and provable?" The answer to that question lies in Figure 1.

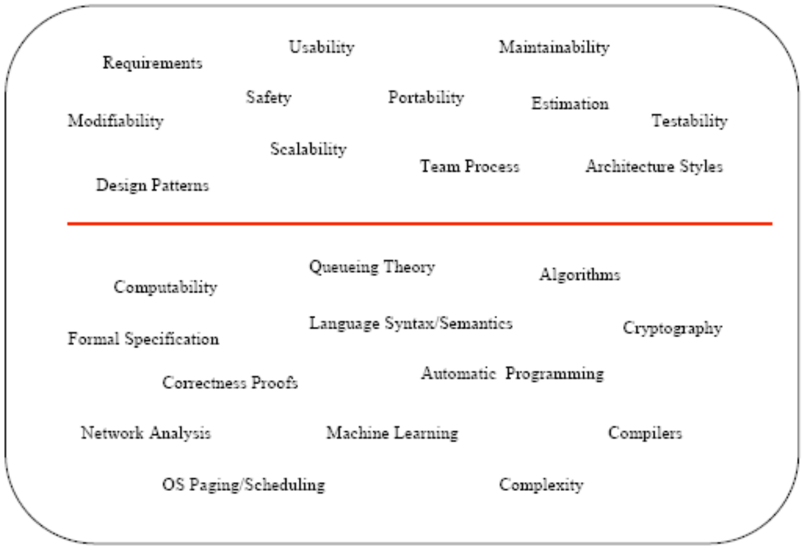

Figure 1: The bright line in computer science

The topics above the line constitute software engineering. The areas of study below the line are the core subjects of computer science. These latter topics have clear, formal results. For open questions in these fields, we expect that new results will also be formally stated. These topics build on each other -- cryptography on complexity, and compilers on algorithms, for example. Moreover, we believe that proven results in these fields will still be true 100 years from now.

So what is that bright line, and why are none of the software engineering topics below it? The line is the property "directly involves human activity". Software engineering has this property, while traditional computer science does not. The results from disciplines below the line might be *used* by people, but their results are not directly *affected* by people.

Software engineering has an essential human component. Software maintainability, for example, is the ability of people to understand, find, and repair defects in a software system. The maintainability of software may be influenced by some formal notions of computer science -- perhaps the cyclomatic complexity of the software's control graph. But maintainability crucially involves humans, and their ability to grasp the meaning and intention of source code. The question of whether a particular software system is highly maintainable cannot be answered just by mechanically examining the software.

The same is true for safety. Researchers have used some formal methods to learn about a software system's impact on people's health and property. But no discussion of software safety is complete without appeal to the human component of the system under examination. Likewise for requirements engineering. We can devise all sorts of interview techniques to elicit accurate requirements from software stakeholders, and we can create various systems of notation to write down what we learn. But no amount of research in this area will change the fact that requirement gathering often involves talking to or observing people. Sometimes these people tell us the right information, and sometimes they don't. Sometimes people lie, perhaps for good reasons. Sometimes people are honestly trying to convey correct information but are unable to do so.

This observation leads to Connell's Thesis:

> Software engineering will never be a rigorous discipline with proven results, because it involves human activity.

This is an extra-mathematical statement, about the limits of formal systems. I offer no proof for the statement, and no proof that there is no proof. But the fact remains that the central questions of software engineering are human concerns:

- What should this software do? (requirements, usability, safety)
- What should the software look like inside, so it is easy to fix and modify? (architecture, design, scalability, portability, extensibility)
- How long will it take to create? (estimation)
- How should we build it? (coding, testing, measurement, configuration)
- How should we organize the team to work efficiently? (management, process, documentation)

All of these problems revolve around people.

My thesis explains why software engineering is so hard and so slippery. Tried-and-true methods that work for one team of programmers do not work for other teams. Exhaustive analysis of past programming projects may not produce a good estimation for the next. Revolutionary software development tools each help incrementally and then fail to live up to their grand promise. The reason is that humans are squishy and frustrating and unpredictable.

Before turning to the implications of my assertion, I address three likely objections:

> The thesis is self-fulfilling. If some area of software engineering is solved rigorously, you can just redefine *software engineering* not to include that problem.

This objection is somewhat true, but of limited scope. I am asserting that the range of disciplines commonly referred to as software engineering will substantially continue to defy rigorous solution. Narrow aspects of some of the problems might succumb to a formal approach, but I claim this success will be just at the fringes of the central software engineering issues.

> Statistical results in software engineering already disprove the thesis.

These methods generally address the estimation problem and include Function Point Counting, COCOMO II, PROBE, and others. Despite their mathematical appearance, these methods are not proofs or formal results. The statistics are an attempt to quantify subjective human experience on past software projects, and then extrapolate from that data to future projects. This works sometimes. But the seemingly rigorous formulas in these schemes are, in effect, putting lipstick on a pig, to use a contemporary idiom. For example, one of the formulas in COCOMO II is $\mathbf{PersonMonths = 2.94 \times Size^{B}}$, where $\mathbf{B = 0.91 + 0.01 \times \Sigma\ SF_i}$, and **SF** is a set of five subjective *scale factors* such as "development flexibility" and "team cohesion". The formula looks rigorous, but is dominated by an exponent made up of human factors.

> Formal software engineering processes, such as cleanroom engineering, are gradually finding rigorous, provable methods for software development. They are raising the bright line to subsume previously squishy software engineering topics.

It is true that researchers of formal processes are making headway on various problems. But they are guilty of the converse of the first objection: they define software development in such a narrow way that it becomes amenable to rigorous solutions. Formal methods simply gloss over any problem centered on human beings. For example, a key to formal software development methods is the creation of a rigorous, unambiguous software specification. The specification is then used to drive (and prove) the later phases of the development process. A formal method may indeed contain an unambiguous semantic notation scheme. But no formal method contains an exact recipe for getting people to unambiguously state their vague notions of what software ought to do.

To the contrary of these objections, it is my claim that software engineering is essentially different from traditional, formal computer science. The former depends on people and the latter does not. This leads to Connell's Corollary:

> We should stop trying to prove fundamental results in software engineering and accept that the significant advances in this domain will be general guidelines.

As an example, David Parnas wrote a wonderful paper in 1972, On The Criteria To Be Used in Decomposing Systems into Modules. The paper describes a simple experiment Parnas performed about alternative software design strategies, one utilizing information hiding, and the other with global data visibility. He then drew some conclusions and made recommendations based on this small experiment. Nothing in the paper is proven, and Parnas does not claim that anyone following his recommendations is guaranteed to get similar results. But the paper contains wise counsel and has been highly influential in the popularity of object-oriented language design.

Another example is the vast body of work known as CMMI from the Software Engineering Institute at Carnegie Mellon. CMMI began as a software process model and has now grown to encompass other kinds of projects as well. CMMI is about 1000 pages long -- not counting primers, interpretations, and training materials -- and represents more than 1000 person-years of work. It is used by many large organizations and has been credited with significant improvement in their software process and products. But CMMI contains not a single iron-clad proven result. It is really just a set of (highly developed) suggestions for how to organize a software project, based on methods that have worked for other organizations on past projects. In fact, the SEI states that CMMI is not even a process, but rather a meta-process, with details to be filled in by each organization.

Other areas of research in this spirit include design patterns, architectural styles, refactoring based on bad smells, agile development, and data visualization. In these disciplines, parts of the work may include proven results, but the overall aims are systems that foundationally include a human component. To be clear: Core computer science topics (below the bright line) are vital tools to any software engineer. A background in algorithms is important when designing high-performance application software. Queuing theory helps with the design of operating system kernels. Cleanroom engineering contains some methods useful in some situations. Statistical history can be helpful when planning similar projects with a similar team of people. But formalism is just a necessary, not sufficient, condition for good software engineering. To illustrate this point, consider the fields of structural engineering and physical architecture (houses and buildings).

Imagine a brilliant structural engineer who is the world's expert on building materials, stress and strain, load distributions, wind shear, earthquake forces, etc. Architects in every country keep this person on their speed-dial for every design and construction project. Would this mythical structural engineer necessarily be good at designing the buildings he or she is analyzing? Not at all. Our structural engineer might be lousy at talking to clients, unable to design spaces that people like to inhabit, dull at imagining solutions to new problems, and boring aesthetically. Structural engineering is useful to physical architects, but is not enough for good design. Successful architecture includes creativity, vision, multi-disciplinary thinking, and humanity.

In the same way, classical computer science is helpful to software engineering, but will never be the whole story. Good software engineering also includes creativity, vision, multi-disciplinary thinking, and humanity. This observation frees software engineering researchers to spend time on what does succeed -- building up a body of collected wisdom for future practitioners. We should not try to make software engineering into an extension of mathematically-based computer science. It won't work, and can distract us from useful advances waiting to be discovered.

Acknowledgements

My thanks to Steve Homer for a discussion that sparked my interest in this question.