



The Nature of Software: What's So Special About Software Engineering?

by [Philippe Kruchten](#)

Rational Fellow

As engineering organizations across North America struggle with the concept of opening their doors to and registering -- or even licensing -- software engineers, questions naturally arise about what software engineering actually entails.

How do we qualify and evaluate software engineers? How do we validate their experiences? A first reaction may be to approach these tasks in the same way that we have done for all other engineering disciplines. However, software engineering differs from structural, mechanical, and electrical engineering in subtle ways. The differences are linked to the soft, but rather unkind, nature of software. In this article, I explore four key differentiating characteristics:



- *Absence of a fundamental theory*
- *Ease of change*
- *Rapid evolution of technologies*
- *Very low manufacturing costs*

Absence of a Fundamental Software Theory

Despite all the research done by computer scientists, there is no equivalent in software for the fundamental laws of physics. This lack of theory, or at least the lack of practically applicable theories, makes it difficult to do any reasoning about software without actually building it. During design, software can be structured and partitioned into chunks, but the real thing (once it crawls inside a computer) is actually totally unstructured, so that anything that goes wrong somewhere can corrupt something somewhere else. The absence of solid and widely applicable theory also means that the few software engineering standards we do

- ▶ [subscribe](#)
- ▶ [contact us](#)
- ▶ [submit an article](#)
- ▶ [rational.com](#)
- ▶ [issue contents](#)
- ▶ [archives](#)
- ▶ [mission statement](#)
- ▶ [editorial staff](#)

have rely on good practice alone, whereas building codes in other disciplines can trace their rules to sound physical principles.

Ease of Change

Software is, almost by definition, easy to change, so naturally, organizations want to take advantage of this characteristic. There is pressure to change software throughout its entire development and even after it's delivered. If you're building a bridge, you don't have this kind of flexibility. You cannot say, "Hmm, now that I see the pilings, I would like this bridge to be two miles upstream." But it is very, very difficult to change software in a rigorous fashion, with all ramifications of all changes fully understood and completely coordinated. Again, because of the absence of solid theory, it's hard to validate a change set and its impact without actually doing all the changes. Most of the damage that is done to software is done through changes.

Rapid Evolution of Technologies

Software development techniques, and the environment of software itself, are changing at an extremely rapid pace that does not allow for progressively consolidating a body of knowledge. This puts a lot of pressure on companies to train and re-train their software engineers, and some do not really understand why they have to spend four times more per capita on training than do people from other disciplines. This makes the initial training software engineers have received less important, except for the most general education, such as math and so on -- especially when this original training occurred twenty-five or more years ago. This rapid evolution also means that it is more and more difficult to maintain and evolve "legacy" systems -- as the recent Y2K scramble has demonstrated -- because the technologies used some ten years ago are not in use any more, and people who still have mastery over them are rather rare. The norms and standards also have to evolve rapidly to catch up with technology evolution. Finally, software engineering, unlike other disciplines, has not had the benefit of hundreds or thousands of years of experience.

Very Low Manufacturing Cost

First, I would like to note a slight shift in paradigm. Software engineers speak about design, but by this they mean only a high-level description of their intent, and then they think of program construction as akin to manufacturing. Actually it is not -- program implementation is more like preparing a cast in mechanical engineering. Also, for a software engineer, a prototype is roughly equivalent to a scale model; it's pretty incomplete. The real "manufacturing" of software entails almost no cost; a CD-ROM, for example, costs less than a dollar, and delivery over the Internet only a few cents. Often it doesn't matter if the design -- that is, the initial program -- is a bit wrong; we can just fix it and manufacture it again, as we noted above in the discussion on ease of change. We hear people refer to this as a "free bug fix release" or a "must-have upgrade." Clearly, this combination -- ease of change and low manufacturing cost -- has led the software industry into a pretty big mess. And these practices are

supported by outrageous licensing policies that allow the designer and manufacturer to assume no responsibility other than, in good cases, a promise that they will re-manufacture the product in a few days or months. You can't do that with a bridge or a car engine because the cost would be huge, and that forces engineers involved in building these things to get them right the first time.

Engineering All the Same?

For twenty years, refusing to acknowledge the four factors I described, the software industry has tried hard to pretend that software development could follow the same path as other engineering disciplines. We have failed. We also hoped that science would bring us solutions, but they have not been forthcoming. Why? To answer that question, let's look at two important ways that software engineering departs from other disciplines.

Iterative Development

The very rational and straightforward "waterfall" development lifecycle -- define and freeze requirements, create and validate the design, implement and test, then deliver -- works very well in many disciplines but has failed many times in software engineering. This project lifecycle does not accommodate changes: It does not allow you to really validate much, so you have to rely on your own warm, fuzzy feeling that "the design is OK." Nor does it allow for tactical changes in technology, or take advantage of the low manufacturing cost -- except for pushing undue costs on to consumers.

Today, software engineers take a more iterative approach to software development, which allows them to integrate changes, to refine and validate the design based on execution and not just examination, and to accommodate evolution in technology. An iterative approach would be impossible in other disciplines; you cannot build a bridge iteratively, for example.

Iterative development allows you to continuously verify the quality of a constructed prototype as opposed to demonstrating correctness *a priori*, based on fixed laws. Software has no laws that can ensure the ultimate product will perform as expected, but iterative development allows you to confront technical risks earlier in the development cycle.

Also, software engineering puts more emphasis on some techniques that have less importance in other disciplines, such as requirements management and change management, because requirements and other software artifacts may change throughout the development lifecycle and even after that.

Component-Based Development

Another dream of software engineers is to mimic with software what has happened in electronics or construction -- to develop families of standardized parts out of which you can build larger and larger sub-assemblies and ultimately complete systems. This sounds straightforward,

but actually very few can do it. Again, this is due to the lack of a strong underlying theory to rigorously define the components and their interface. The situation has not been helped by the rapid changes in technologies: No component family has had time to settle and develop as a self-sustaining industry. Only recently has a new sub-discipline emerged -- software architecture -- which tries to address, despite the lack of fundamental theory, some of the structural aspects involved in constructing large software programs and reusing software assets across product lines or product families.

So, if I agree with David Parnas¹ that software engineering should be treated on an equal footing with other engineering disciplines and not solely as computer science or some kind of enlightened craftsmanship, then I also have to acknowledge fundamental differences that make some of the more traditional approaches to engineering and engineering management inapplicable to software. Software engineering has developed its own approaches to manage its specificities over the last few years.

I believe that the registration process for professional software engineers -- as well as the construction of accredited software engineering programs -- must understand, acknowledge, and address these specificities.

Notes

¹ David Parnas, "Software Engineering Programs Are Not Computer Science Programs," *IEEE Software*, December 1999, 19-30.

References

Steve McConnell, *After the Gold Rush: Creating a True Profession of Software Engineering*. Microsoft Press, 1999.

Gilda Pour, Martin Griss, and Michael Lutz, "The Push to Make Software Engineering Respectable." *IEEE Computer*, May 2000, 35-43.

David Parnas, "Software Engineering Programs Are Not Computer Science programs," *IEEE Software*, December 1999, 19-30.

Philippe Kruchten, "Putting the Engineering into Software Engineering." *Innovations*, 4 (1), January 2000, pp. 23-24.

Koni Buhrer, "From Craft to Science: Searching for First Principles of Software Development." *The Rational Edge*, December, 2000.
http://www.therationaledge.com/content/dec_00/f_craftscience.html



For more information on the products or services discussed in this

**article, please click [here](#) and follow the instructions provided.
Thank you!**

Copyright [Rational Software](#) 2001 | [Privacy/Legal Information](#)