Connecting Good Theory to Good Practice: Software Documentation: A Case Study

David Lorge Parnas

Software Quality Research Laboratory
University of Limerick
Limerick, Ireland

1 Introduction

The motto that appeared on my slides for many years, "*connecting theory to practice*", is like motherhood; nobody dares to oppose it, at most half of us could actually do it, and many are actively trying to prevent it. Over many decades, I have been fascinated by Computer Science theory and software development practice but disappointed by both.

- The theory is often needless arcane and complex because very specialized concepts are used where more general classical concepts would do the job better.
- Software development practice generally ignores classical Engineering principles because the developers do not know how to apply those principles and do not see them applied by more experienced developers.
- I know of very few cases where theory has successfully led to improved software development practices.

2 What's wrong with the theory?

What is called "theory" in Computer Science needs to be rethought, refocussed and renamed. Much of it is not relevant to computing at all.

2.1 It is all mathematics with new names and short memories

In an effort to make their work appear more relevant, computer scientists have introduced new names for an old area. We see terms like "semantics" and "formal methods" but any well-trained mathematician would recognize the work as mathematics. The name change is actually harmful in that people who specialize in these "new" areas often do not know classical mathematics and reinvent rather bumpy versions of old

wheels. It is interesting to note that when the founding of the first Computer Science departments was debated in the 60s, mathematicians predicted that separating Computer Science from Mathematics would have this effect. Then, I thought they were wrong; today, I know that I was wrong.

2.2 Specialization has been valued over abstraction

In the history of applied mathematics, one sees a clear trend towards abstraction and generalization. For example we moved from the study of integer arithmetic to the study of real numbers and then to general algebraic structures with the same properties; abstract algebra brought great gains in simplicity, applicability, and elegance. In Computer Science "theory", one often observes "progress" in the other direction. Simpler concepts are "enhanced" by adding special features. Logic is one area where this is often done. Those who propose a new logic often build special cases into the logic rather than use a simpler, more general tool that is general enough to allow the needed structures to be defined. For example, despite some claims to the contrary, we do not need to add anything to basic logic to deal with time. In most cases, adding complexity to the rules by which we reason increases complexity and the likelihood of errors. It also leads to the replacement of well-developed general tools with prototype versions of specialized ones.

2.3 Theoreticians often look at the wrong problems.

It is far too common to see researchers working on the problems that they can solve rather than the problems that are encountered in practice. Nowhere is this clearer than with the focus on computability infinite state spaces. All of our machines are finite and the problems encountered in practice are often a result of the finite limits. We need to focus on techniques that make it easy to express the finite limits that pervade all real software systems.

3 What's wrong with current practice?

Software development today is in a very poor state. We accept bugs as normal and allow developers to charge us extra for upgrades that do what a product should have done from the start. Projects that fail to yield useful products are common. Why?

3.1 Lack of Discipline

Most Computer Science graduates do not seem to understand why the word used in English for a speciality within Engineering is "discipline". In addition to the basic mathematics and science that they will need to be sound practitioners over several decades, Engineering students learn what they have to do to be sure that they have designed products that will be fit for use. In such areas as Civil Engineering, they are taught what measurements must be made, what calculations must be done, what

tolerances are allowable, etc. Often they are given forms or checklists to make sure that they work with discipline and are less likely to overlook critical steps. Part of this discipline is that they must produce a set of highly structured, precise, and accurate documents that record key design decisions and restrict later work. Those who will review those documents have been taught calculations and other checks that should be made using them. All of this seems to be missing in most software development efforts. Programming is viewed as an art.

3.2 Failure to use mathematics as a support tool

In traditional Engineering, students are taught how to use mathematics and science to design and to check designs. In Computer Science education, the development courses and the "theory courses" hardly mention each other and developers do not know how to use what mathematics they have learned.

3.3 Poor tools and notations

Most of our programming tools are a combination of elegant but incomplete ideas and "corrections" that leave them inconsistent, incomplete and hard to use. We also have, "Undefined Muddling Languages" that are used to sketch designs rather than provide precise reviewable documentation of design decisions.

4 Why is the connection between theory and practice so poor?

Within Computer Science, I see a growing gap between two cultures: the culture of the theoretician and the culture of the programmer. Each group has become a clique, actually a set of rather closed cliques, in which there is a lot of communication within the clique and little between cliques. The practitioners propose relatively minor perturbations on the bad tools and practices that are so common today. The theoreticians focus on their own problems, trying to improve previously published theories, and ignore the (often much more difficult) problems being encountered in industry. Practitioners, on the other hand, either expect a magic tool that will do their work for them, or expect nothing at all from theory. Those who expect magic are inevitably disappointed and join those who expect nothing. Their ranks are always replenished by new, more naive developers.

In traditional Engineering, one sees researchers find their problems in industry, and solutions in academia. In software, I see the reverse. Researchers find their problems in other researcher's papers (often without any awareness of present practice) and if they find solutions, it is usually because of (industrially developed), more powerful hardware.

5 How could we do better?

I believe that the long-term solution to these problems lies in our approach to education. Education for software developers needs to be reattached to classical Engineering and Mathematics. Students need more exposure to, and understanding of, classical mathematics as well as a better understanding of disciplined Engineering practice. We can reduce the time devoted to recent CS "formal models" and current technology. We must always remember that we educate for a lifetime of professional growth, not for knowledge of currently popular, but rapidly mutating, notations.

We also need to teach researchers to look deeper. All too often, we react to the symptoms of problems, rather than the causes. If we want our research to help practitioners we must listen to them but not necessarily believe them. Often we have to solve an underlying problem, not the one that they describe.

6 Applying classical mathematics to software documentation.

The remainder of this talk will be devoted to a specific illustration of these principles. It will show how classical mathematical techniques can be applied to produce precise, but easy to use reference documentation and how that documentation can be applied to improve the effectiveness of such fundamental tasks as testing and inspection.