Reino Kurki-Suonio/dinner talk at ESEC/FSE 2003

Reflections on Theory and Practice

Having ESEC and FSE as one conference witnesses of the desire not to separate practice and theory. In the pioneering persons of computer science and software engineering these two were combined in a natural manner, since their theoretical insight developed with remarkable achievements in the emerging practice. A concrete reminder of the passing of time was the passing away of two great figures of this kind, Edsger Dijkstra and Ole-Johan Dahl, a year ago.

Myself, I also belong to the generation whose main role now is to give room and encouragement to younger people and their fresh ideas. For my own ideals I owe much to Alan J. Perlis, a legendary figure at Carnegie Mellon in the 60's. As I understood him, practice is the primary source and ultimate criterion for all theory. From him I also learnt that "one man's theory is another man's practice." Or, as we can say tonight, one person's FSE is another person's ESEC.

David Parnas, himself a student of Perlis forty years ago, likes, however, to give a different view on the word "theory," pointing out that "theoretically" is synonymous to "not really." He seems to think about the kind of theory that is developed for its own sake, without personal contact with practice, in order to achieve scientific merits and the admiration of other theoreticians.

What is theory, after all? In the broadest sense, anything that provides understanding at some level of generality can be called theory. The Oxford English Dictionary explains it as "systematic conception or statement of something; abstract knowledge, or the formulation of it." Understood in this way it is much more than what pure theoreticians would accept as theory, but even with this broad view it is clear that the development of theoretical understanding has not been the most important force in how common practice has developed in software engineering. For instance, remarkable work was done in the 60's and 70's on operating system principles and their security, but the understanding that was then achieved was shamefully ignored when PC's came around. As a teacher of programming language principles I am also rather disappointed to see that students who are familiar with current industrial practices are often quite innocent of how some important problems in programming language safety were successfully solved in the 70's and 80's.

Obviously, much more than advantages gained by improved theoretical understanding is needed to change a culture that has evolved in practice. Even if Kenneth Iverson – the father of APL and the once lively culture around it – would have been right in claiming that there are crucial advantages in associating all operators in expressions to the right, and having only one level of priorities, he would not have had any chance to change the culture of arithmetic notations. And, as far as I know, the duodecimal system with radix 12, which was preferred by Blaise Pascal and is advocated today by the dozenal societies of America and Great Britain, has been adopted only in Middle-Earth, and the hexadecimal system, which we may find even more natural, will not replace the decimal system, either. Still, Roman number notations, although still used in certain contexts, eventually had to give way for a positional system.

By the way, in 1717 this part of the world was ruled by the Swedish hero-king Charles XII, who at that time seriously considered to convert his country to radix-8 arithmetic. That was the good old days when a king could decide on such matters; today the king of Sweden has too little power even to convert his country to euros! Had Charles XII not been killed soon after getting this idea, we might have enjoyed the virtues of a power-of-two radix already long before computers.

As far as software culture is concerned, we are used to the situation that academic people advocate theoretically justified approaches that are unrealistic for practice, and that the practice is ready to accept improvements only by evolutionary patches, which do not fully bring in the desired advantages, and which make everything even more complicated. Consider, for instance, the beauty of object-orientation in its pure form, and the complexities to which its inclusion has led in languages like C, or the continual patching of operating systems that is needed for improved security.

As for theory in the sense theoreticians understand the word, and its role in software engineering, there are several misunderstandings that we have seen during the years, and still see.

Firstly, theory has been equated with such computer science fundamentals as Turing machines, computability, complexity theory, etc. Although every software engineer should have some knowledge of these, their everyday work is not really affected by them, which easily leads to the wrong conclusion that mathematical theories are irrelevant for the practice of software engineering.

Secondly, theory has been equated with laborious use of formal methods in deriving solutions to academic toy problems, and with the idea that this could easily be extended to the construction of real-life software. This is the kind of theory that Parnas identifies with “not really,” and it is associated with the misunderstanding of some theoreticians that software engineering is a purely mathematical activity.

What is the purpose of theory, after all? In physics, where one investigates the physical world, theory aims at better understanding of the fundamental particles and their interactions, which then leads to better understanding of also macroscopic phenomena. Does a similar idea work for software? Instead of the physical world we have the world of current programming languages, design formalisms, etc. The role of fundamental particles is taken by abstract machines and associated fundamental constructions in an imaginary non-physical world, whose understanding is developed by computer scientists. The hope may then be to understand the macroscopic world of software better by explaining it in terms of these theoretical constructions.

This attitude means, in fact, that theory is taken as an add-on to current practices, as sugaring that makes it possible to use formal methods and associated tools as an afterthought, and to analyze rigorously what software engineers have created independently of them. This reminds of the patching attitude to software. It allows keeping to old habits, but makes them sound theoretically better justified.

This is another misunderstanding of the role of theory, since we are not dealing with a world given to us, or with arbitrary software constructions, but with artifacts that we are to construct ourselves. Of course, there is nothing wrong in understanding the world of current languages and other formalisms better, to study their ambiguities and other unintentional properties, and to use this understanding to develop new kinds of tools. It is a problem, however, that this world is burdened with complexities and irregularities that have been inherited from the level of implementations and low-level abstractions. In particular, they are obscuring the essentials at the higher levels of design and specification. As an example, formal approaches to object-oriented specification tend to be faithful to how object-oriented concepts have been introduced in terms of programming language mechanisms, without raising the level of abstraction appropriately.

To be really usable in the practice of software engineering, a theory should not have the character of formalization, but of *simplification*, making it possible to reason as simply as possibe on those properties that are the most important to us in the artifacts that we wish to construct. Citing Dijkstra from the book published for the celebration of ACM's 50th anniversary: "Because we are dealing with artifacts, all unmastered complexity is of our own making (...), so we had better learn how not to introduce such complexity in the first place." To succeed in this, theory and practice of software engineering should not be separated. This means that mere collaboration of theoreticians and practitioners is not sufficient. Instead, the two have to be combined in individual persons, in the way they were combined in pioneers like Dijkstra and Dahl. I wish that the combined ESEC/FSE will contribute towards this end.