

Some Myths of Software Engineering Education

Hans van Vliet
hans@cs.vu.nl

Department of Computer Science
Vrije Universiteit Amsterdam
The Netherlands

ABSTRACT

Based on many years of teaching software engineering, I present a number of lessons I have learned over the years. I do so in the form of a series of myths, the reverse of which can be considered challenges to educators. The overall message I wish to convey is that there's more to software engineering than engineering. The engineering metaphor gives us a lot of useful guidance in shaping our profession. But there's also a downside, in that this goes at the expense of the human, social dimension that is an essential element of our profession.

Categories and Subject Descriptors

K.3.2 [Computing Milieux]: Computer and Information Science Education

General Terms

Human Factors

Keywords

software engineering, education

1. INTRODUCTION

Software engineering is vital, and so is software engineering education. The amount of software is only growing, in terms of both size and complexity. The degree to which society depends upon software puts ever greater demands on its quality. The degree to which our e-society demands solutions ever quicker requires better ways to produce systems. These developments put considerable pressure on software engineering education.

Over the past years, the software engineering community has paid much attention to organizing the knowledge we have, and ways to transform this into a curriculum. The first has resulted in SWEBOK, the Guide to the Software Engineering Body of Knowledge. It reflects a widely agreed-upon view on what a software engineer who has a Bachelor's degree and four years of experience should know. The latter has resulted in SEEK, giving curriculum guidelines for undergraduate degree programs in software

engineering. It can be seen as the education counterpart of SWE-BOK. Both are important milestones, the results of many years of experience, and many discussions within their respective working groups. The effect of these milestones on software engineering education is already visible, and is likely to increase in the years to come. This is both good and bad. The good by far outweighs the bad. I will concentrate on the bad though, my primary aim being to provoke discussion.

Others have emphasized the 'engineering' in software engineering [3], [4]. The overall message I wish to convey is that there's more to software engineering than engineering. More specifically, there's an important social dimension in software engineering, that easily gets into a tight place because of the omnipresent engineering attitude.

2. MYTHS OF SOFTWARE EDUCATION

Myth 1: A software engineering course needs a real-life practical part

The idea behind this is that we should prepare students for "the real world", which is full of inconsistencies, complex, and changes all the time. The real world also involves participants from different domains, and has political and cultural aspects. To meet this challenge, we may do a project based on real examples from industry, or introduce obstacles and dirty tricks in student exercises. The question is how helpful this is.

Students entering a software engineering course typically are quite immature with respect to software development. They may have spent less than two years at the university, and done courses on programming, data structures, and so on. In these courses, the work is usually well-organized. The problems they are given are unambiguous. And often too, there is only one right answer. In the software engineering course, they are overwhelmed with a large number of topics that are very new to them. One may of course claim that it is also possible to gently introduce some software engineering principles in other introductory courses. In practice, this is not easy to accomplish in a CS environment.

I consider my students as toddlers in the software engineering playground. So I concentrate on one or a few issues, in an orchestrated environment. Of course, I treat all of the topics in the course, and tell my favorite anecdotes. But the accompanying project highlights only a few. In later years, the students then get confronted with more real-life aspects in other courses. I have noticed quite often that appreciation for the software engineering course only comes years after students suffered it.

Myth 2: Software Engineering is like other branches of engineering

All software engineering texts discuss the relationship between soft-

ware engineering and other branches of engineering. And of course they also tell us that there are differences between software engineering and these other branches as well. But the overall message is that the similarities prevail, and that the metaphor is a useful one.

But there also is a downside to the widespread use of this metaphor. Our field has a large number of "engineering" words: building software, requirements, specification, process, maintenance, and so on. Altogether, this induces a model of how we view the software development practice. The metaphor plays an *active* role in our thought processes. See [1] for an enlightening discussion hereof.

At a larger scale, we see a similar tension between the heavy-weight, document and planning driven life cycle models from the engineering realm of software engineering, and the various lightweight approaches that emphasize the human aspects in software development. It is a major challenge to combine the virtues of both. This not only holds for the state-of-the-practice, but the more so for the educational environment, where students are entrenched in the engineering view of the world of software development.

Myth 3: Planning in Software Engineering is worse than in other fields

Many papers on software engineering or software engineering education have quotes like "... approximately 75 percent of all software projects are either late or cancelled". In his wonderful book *Death March*, Edward Yourdon quotes gurus like Capers Jones and Howard Rubin and states that "the *average* project is likely to be 6 to 12 months behind schedule and 50 to 100 percent over budget." The sometimes explicit, sometimes implicit message is that a better software education will help. And eventually maybe even does away with most runaway projects. I question this relation between the level of software engineering education and planning accuracy.

Here we may ask ourselves whether other fields fare better in this respect. The Danish economist Bent Flyvbjerg studied over 250 infrastructure projects all over the world, and found that the costs are underestimated in nine out of ten projects. At the same time, the revenues are almost always overestimated. This way projects look good, and decision makers approve of them [2].

I think many of the arguments that hold for cost and schedule overruns of infrastructure projects, are also valid for software development projects. Better function point counting mechanisms, requirements engineering methods, and the many other ingredients of our field, and us educating software engineers in them, will by themselves not do away with these issues. To quote Tom DeMarco: "One chief villain is the policy that estimates shall be used to create incentives". This is as true today, as it was in 1982.

Myth 4: The user interface is part of low-level design

The user interface of a system is important. About half of the code of an interactive system is devoted to the user interface. A recent study found that 60% of software defects arise from usability errors, while only 15% of software defects are related to functionality [5].

SWEBOK and SEEK both consider user interface design as a "related discipline". My position is that this totally ignores the fact that many software development projects currently and in the future will aim to develop systems where human use and human factors in the context of use are decisive factors for product quality. As a consequence, I take a different stance, one in which the design of the interface and the design of the functionality go hand-in-hand. We might even say: "The user interface *is* the system".

Myth 5: SWEBOK represents the state-of-the-practice

In my opinion, SWEBOK (and also SEEK) lags behind the state-of-the-practice in some areas, and it runs in front of the herd in others. In a recent European research project that investigated software en-

gineering methodologies for embedded systems, it was found that actual industrial experiences at participating companies are quite a bit removed from the average software engineering textbook or SWEBOK. And fresh graduates from our schools enter an environment that might well be characterized by these practices. Such might give a clash, and further increase the perceived distance between universities and industry.

As for the lagging behind of SWEBOK: the field of software engineering changes rapidly. New approaches such as model-driven development (MDA), service-oriented architecture (SOA), seem to make quite an impact on both research and practice, and yet have not made it to either SWEBOK or SEEK.

One major theme underlying recent developments is that software engineering becomes more and more heterogeneous: (1) distributed development involving teams from different cultures impacts work processes, (2) the combination of in-house developed software with COTS, Open Source and other externally acquired software more and more becomes a policy, if not necessity, rather than an unfortunate event, and (3) traditional, document-driven development approaches are combined with the more recent, people-driven agile development approaches to get the best of both worlds.

Both SWEBOK and SEEK still follow, in their basic structure, quite a traditional view. Though the importance of evolution is stressed at quite a few places, the surface structure of both documents still suggests a greenfield situation. Even though not intended, this is likely to influence student attitude. The emergent heterogeneous situation sketched above further complicates industrial practice, and should get some counterpart in education.

3. REFERENCES

- [1] A. Bryant. Metaphor, myth and mimicry: The bases of software engineering. *Annals of Software Engineering*, 10:273–292, 2000.
- [2] B. Flyvbjerg, N. Bruzelius, and W. Rothengatter. *Megaprojects and Risk: An Anatomy of Ambition*. Cambridge University Press, 2003.
- [3] P. Kruchten. Putting the "engineering" into "software engineering". In P. Strooper, editor, *Australian Software Engineering Conference (ASWEC 2004)*, pages 2–8, Melbourne, Australia, 2004. IEEE Computer Society.
- [4] D. Parnas. Software Engineering Programs Are Not Computer Science Programs. *IEEE Software*, 16(6):19–30, 1999.
- [5] O. Vinter, P. Poulsen, and S. Lauesen. Experience Driven Software Process Improvement. In *Software Process Improvement*, Brighton, 1996.