

The Making of a Software Engineer

Challenges for the Educator

Clemens Szyperski

Microsoft, Redmond, WA 98052, USA
<http://research.microsoft.com/~cszypers/>

Abstract. Software engineering is foremost an engineering discipline. Engineering in general and software engineering specifically has to balance many factors to achieve viable tradeoffs—an understanding of the factors as well as the viability criteria is at the heart of the educational challenge. All engineering has one ultimate goal: the delivery of artifacts (products, commercial or not) that meet the needs of those using such artifacts. All engineering lives in the intersection of people, technology, domain, and opportunity aspects. Software engineering, however, is laden with its own specific difficulties. Software as an engineering medium fills a space between the fluidity of digital content, with which software shares the representation, and the nature of machines, with which software shares the flexible and repeatable application. This brief article covers some of the authors personal observations and suggestions with a hope to inspire (and provoke) those striving to improve software-engineering education.

1 Introduction

When aiming to educate people to join some profession, few standard recipes apply. That is no different in the field of software engineering. There are fundamental issues that need to be considered and this brief article is a survey of those the author finds most compelling. Then there is the need to synthesize the whole, a curriculum, and make it greater than its parts: pointwise factoids, glimmers of wisdom, and deliberate omissions. This article does not cover such larger curricular issues, despite their clear importance.

Software engineering is foremost an engineering discipline and as such has one ultimate goal: the delivery of artifacts that meet the needs of those using them. For the sake of brevity, the term product is used throughout this Article when referring to the spectrum of artifacts, but without implying any necessity of commercial form. Products can be artifacts anywhere from: the one-off work of an individual meeting an ad-hoc need, to the work of the many over long periods of time; from meeting the needs of the few at specific times to the needs of the many over long periods; from the commercial to the societal and to the charitable framing.

Engineering as a discipline of delivering the right products has to consider a wide array of tradeoff functions, including many that are only weakly defined.

Tradeoffs include technical requirements that are in mutual conflict (perfect security versus third-party extensibility), competitive drivers (time to market or process agility), opportunity drivers (windows of technological alignment in a changing world), and many more. A deep appreciation of this fundamental situation is at the core of all engineering and needs to be conveyed to the budding software engineer as well.

All engineering lives in the intersection of people, technology, domain, and opportunity aspects. Software engineering, however, is laden with its own specific difficulties. Software as an engineering substance fills a space between the fluidity of digital content, with which software shares the representation, and the nature of machines, with which software shares the flexible and repeatable application. Software-engineering education is thus challenged to draw on broader traditional engineering disciplines while clearly extracting how it is that software is different from other engineering substances.

2 Why is software different?

One approach to understanding software as the substance of software engineering is to look at it from the perspective of crosscutting disciplines. For instance, how is software different from other engineering substances when considering the distribution of manufactured goods? Or the industrial supply chains? The opportunities of specialization in work force, industry, and society? In a recent book, David Messerschmitt and the author have analyzed these questions for a wide range of such disciplinary crosscuts [1].

In short, software is about engineering the equivalent of flexible manufacturing plants that are represented as streams of bits. That is, while traditional engineering is about the repeatable delivery of instances (a bridge), roughly following design templates that are well understood (classes of bridges), software engineering is about delivering systems (software) that then automatically deliver instances (configured, running) as needed.

It is noteworthy that traditional engineering turns to software engineering for insights when crossing the boundary into delivering automated delivery systems (such as flexible manufacturing plants [1]). The widely held myth that traditional engineering disciplines know so much better how to control their processes and the quality of their products does not generally hold. Engineering a bridge that falls outside of all known and proven templates is a costly undertaking, largely drawing on the prototyping and simulation of the specific instance to be built. Traditional engineering disciplines do tend to be more mature when it comes to established and certifiable processes that make it much more likely that defects are detected early. However, almost all such processes assume the working within an established framework and aid in the reproducible delivery of “more of the same”.

As soon as the established envelope needs to be stretched or surpassed entirely, no engineering discipline is beyond the concepts of prototyping, exploration, and scientific experimentation. Software is, ideally, always about some-

thing new—or else it should have been automated already. It is true that there is still some to be learned from other engineering disciplines, such as the scientific recording of experimental settings and experimental results. However, it is also true that non-repeatability is, in some way, the essence of software engineering. Any such argument is guaranteed to raise concern—that it could be understood as an excuse for poor software quality. This is not the intention: avoidable mistakes need to be avoided and discipline and process that help in such a cause are therefore essential. Yet, accepting those mistakes that could not have been avoided in any systematic way (and using mitigation strategies to limit damage) is an equally important aspect of software engineering—and engineering in general. There is always a remaining risk of failure.

3 The requirements myth

The definition of engineering entails the concept of synthesizing products that meet their users needs. A conceptualization of this simple observation extracts the users needs as their up-front requirements that are then to be met. In textbook examples, this seems to work out well. After all, picking any successful product after the fact, we find it compelling to extract the requirements met by that product. However, faced with the truly new, requirements capture is often doomed to under-deliver. Yet, a disproportional number of system failures can be blamed on flawed requirements capture.

There are several reasons. First, most people think incrementally and are best at phrasing their requirements as desirable improvements over an existing situation. If incremental steps are small enough or if new projects are sufficiently close to previously successful ones, then requirements can be captured with a reasonable level of completeness and precision. This is probably the single most compelling reason to use agile processes with tight feedback loops in many software projects.

At the other extreme, disruptive technology can hold the greatest promise, but almost no one succeeds at eliciting requirements in areas of disruption. Since creating disruptive technology in small increments is almost a contradiction in terms, it is useful to consider research and advanced development processes in addition to agile development processes.

Second, only products aimed at the few in specific situations—and often at precise moments in time—are likely to target coherent requirements. Products aimed at the many have to unite individual needs and perspectives. Products aimed to generalize over many situations or to bridge over longer periods have to unite domains and embrace change. Techniques like focus groups can help gather requirements and feedback even then, but only in a less conclusive fashion.

Third, software could be cast to deliver a huge number of special-case solutions that meet pointwise requirements exactly, at least in theory. However, users of such solutions themselves move through time and space, facing varying situations. What started out as a highly specialized and perhaps optimal solution

can quickly degenerate into something entirely inadequate. Aspects of training, economics, and predictability all push for solutions that are more malleable.

The reality is that engineers need to understand how to capture those requirements that can be elicited, while living with inevitable uncertainties. Agile engineering methods aim to shorten the feedback loop, but fall short on creating disruptive technology and paradigm shifts. Longer-cycle methods are better at addressing the latter, but need to be kept iterative and incremental nevertheless. The one process for all, the one tool kit for all, or the one education for all software engineers in all situations is unlikely to deliver the best overall results.

It is unfortunate that requirements and thus project goals are inextricably linked to project execution. After all, capturing requirements is known to be hard and project management and team skills are among the hardest to teach.

4 Daily essentials in the life of a software engineer

Engineering is the skillful conception and realization of tradeoffs. Almost as a law of nature, if there is the perfect solution, then it is not engineering. To be successful, engineers need to eliminate the impossible first. This is where theory contributes most forcefully—and where a lack of theoretical foundation can be most catastrophic.

Both a reasonably deep understanding and an intuitive command of the key results of central theories help. For software engineers, these include the following: computability, complexity, communication, measurement, type, and proof theory. A grasp of underlying laws of physics, such as the consequences of the inescapable speed-of-light limit or the law of the conservation of energy, help to keep the software engineers mind grounded. Since so many tasks of the engineer are non-technical in nature, it is important to have a working understanding of social, economic, and societal mechanisms.

Having eliminated the impossible, an engineer then needs to focus on the practical. Here, it is important to open up the mind for the many different fields of influence that matter. Briefly, it is all about people, technology, domains, and opportunities (markets, society).

4.1 People, Roles, Processes

In a good approximation, software engineers never create ground-up new systems, so architecture, systems theory, services and components need to take a front seat. In addition, software engineers never deal with systems or situations small enough for individualists to cope. A good understanding of team and role models that have worked in practice needs to be paired with actual team experiences. Perhaps surprisingly, rigor and semiformal to formal methods may have their main strengths in enabling precise team communications, not in the formal securing of certain properties of the deliverables (unless largely automated).

Software development itself is obviously at the center of software engineering. However, it is critically important to understand and appreciate (rather

than merely tolerate) the critical roles played by Quality Assurance (the subject formerly and improperly known as Testing), Architecture, Program and Project Management, Accessibility and Usability Engineering, User Education, Sustaining Engineering, Escalation Management, Operations, and Systems Management. Some of these roles only show prominently in large or complex projects, but—more likely than not—any software engineer will encounter them at some point.

Processes, their understanding, appropriate formalization, and desirable agility, are at the center of team effectiveness and team efficiency. Itself a rapidly evolving field, it is important that the new software engineer has a good initial understanding of what can (and cannot) be obtained by applying processes. For instance, test-driven development—an approach that effectively encourages the writing of specifications in the form of tests to pass—is an example of a technique that combines team communications with some level of automation (regression testing). Embedded as one of several key ingredients, it forms a pillar of most agile development processes.

4.2 Technology

Technology is easily the focus in software-engineering education, so there is less need to discuss this aspect here. Given the ever-changing nature of technology, educators need to take care to cover a broad range of technology. Presenting a comprehensive historic perspective is important if only to help engineers avoid the known mistakes of the past. (However, some mistakes were just inappropriate tradeoffs at the time and need to be reevaluated!) Futuristic outlooks, studies of Moores, Amdahls, and other laws, and cautionary tales of phenomena such as the memory wall—and the general trend towards latency dominance—help to keep everyone on their toes. Nothing that has been done in the past is ever good enough; nothing that is being done can ever be more than good enough.

Given the plethora of technical terms and categories that the software field produces on an ongoing basis, it is particularly important to have educators that are well-versed to project an image of currency and validity in the context of the rapidly moving blogosphere, trade press, research field, and industry developments. At the same time, the educator and indeed the entire curriculum are challenged to establish cohesion and effectiveness of mental models and tools. An example close to the author's heart [2] would be concepts such as objects, components, component frameworks, composition models, services, contracts, types, protocols, and the like. Finding plausible handles on such terms in a way that casts established technology into a framework while also helping to predict and cast coming developments is a challenge—and not one that has a single definitive answer.

4.3 Domain

Finally, yet perhaps most important of all, no engineer can function without appropriate domain knowledge. There is a huge and growing number of deeply

specialized domains, so picking one or a few of them for good is a recipe for likely unemployment. Yet, educators and practicing engineers alike have to select and embrace a few characteristic domains deeply to avoid the biggest dilemma that befalls software engineering more than any other engineering discipline: the theoretic potential to do the arbitrary paired with the practical inability to realize the specific.

No curriculum as such can be of much help here—the ancient master-and-apprentice model may well be the one approach that works. Designing educational programs to embrace deep and repeated contact with practice is therefore essential. A few longer encounters, such as year-long internships paired with reduced part-time continued studies, are likely more effective than several brief episodes. Models that encourage people to return to universities repeatedly over the course of their live-long journey would help to balance education in a world of rapid change.

4.4 Opportunity

Anyone who has worked in a startup context (or in a risky long-pole incubation effort within a larger organization) knows that the one factor that cannot be ignored is the “window of opportunity”. Itself a strangely elusive concept, such a window has to be predicted and yet, by definition, if it were perfectly predictable, it would not be an opportunity. Studying the principles of success that cause startups in some parts of the world to execute almost like clockwork (including the calculated folding of an acceptable percentage) yields many insights.

Regions like California’s Silicon Valley have established a startup ecosystem that provides anything from early-phase consulting to venture capital, from companies that focus on building companies to companies that focus on devouring the leftovers of imploded efforts. For software engineers to participate in the innovation cycles that are so fundamental to the nature of software, it helps to understand the principles that govern markets. Besides economic and fiscal mechanisms, some attention should be paid to psychological ones as well.

Attempts to develop the perfect solution outside of a framework of a perceived and carefully monitored opportunity is generally a recipe for disaster. Given the constant flux of technology, domains, and even people-level processes and expectations, it is essential to strive for a balance between the technically “perfect” in a presumed-to-be-frozen universe and the pragmatically usable in an accepted-to-be-fluid world.

5 Conclusion

Personally, I like to challenge educational programs by asking three questions:

- If I had to design the perfect questions for a day of interviews when hiring a junior engineer “out of college”, what would I hope to cover and what would I hope to find?

- If we assume radical changes in technology and our use of it, on an ongoing basis, what balance can I seek between the long-term valid and the short-term complete?
- How can we absorb today's realities to a functional and productive level, while remaining open and imaginative about what could be?

In the end, there is an area of even greater compromise and tradeoff than engineering: education of engineers.

References

1. David G. Messerschmitt and Clemens Szyperski. *Software Ecosystem—Understanding an Indispensable Technology and Industry*. MIT Press, Cambridge, Mass., 2003.
2. Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software—Beyond Object-Oriented Programming* (2nd ed). Addison-Wesley, New York, NY, 2002.