Reflections on Software Engineering Education

Hans van Vliet, *Vrije Universiteit Amsterdam*

The "engineering" focus in software engineering education leaves instructors vulnerable to several traps. It also misleads students as to SE's essential human and social dimensions.

In recent years, the SE community has focused on organizing our existing knowledge and finding ways to transform it into a curriculum. These efforts produced SWEBOK (the Guide to the Software Engineering Body of Knowledge; www.swebok.org) and Software Engineering 2004 (http://sites.computer.org/ccse). SWEBOK reflects a widely agreed-upon view of what a software engineer who has a bachelor's degree and four years' experience should know. SE 2004 offers curriculum guidelines for undergraduate SE degree programs. We can view SE 2004 as SWEBOK's education counterpart.

Both SE 2004 and SWEBOK are important milestones resulting from participants' extensive real-world experience and working-group discussions. Both heavily emphasize the "engineering" in software engineering.[1–3] This focus influences the contents of a typical SE course as well as the students' understanding of what SE entails. However, SE has an important social dimension that's easily squeezed out by the omnipresent engineering attitude. Here, I discuss how this limited conception of SE contributes to five assumptions that can trap SE educators:

- An SE course needs an industrial project.
- SE is like other branches of engineering.
- Planning in SE is poorly done relative to other fields.
- The user interface is part of low-level design.
- SWEBOK represents the state of the practice.

The traps idea isn't highly original. Several authors have published similar articles on the myths of formal methods, requirements engineering, and SE programs.[4] In the latter case, the authors discuss whether the new SE degree programs are a silver bullet. The traps I discuss focus on a typical SE course's content and how it represents SE to beginning students. My aim is both to provoke discussion and to highlight the challenges these traps present to SE educators.

Context

My teaching situation partly determines and bounds the traps I discuss. Typically, Dutch universities don't offer separate computer science (CS) and SE degrees. They have a three-year bachelor's program and a two-year master's

0740-7459/06/$20.00 © 2006 IEEE

In the SE course, students are suddenly overwhelmed with many new topics.

program in CS. Most students enroll in the bachelor's program right after high school. The program doesn't have much specialization and usually has one general SE course. Typically, this course includes theory and project work. The master's program generally contains a series of more specialized SE courses.

The Vrije Universiteit rates its SE course's theoretical and practical parts at 4 and 8 ECTS credits, respectively. (In the European Credit Transfer System, 1 ECTS amounts to approximately 28 study hours; a full year is 60 ECTS.) The course lasts 12 weeks. Students are scheduled to take it in the second year of their bachelor's program, which means they have little maturity in CS or SE when they enroll. The course is compulsory for students in CS, AI, and information science. Typically, 150 to 200 students enroll each year.

In terms of SE 2004, we follow a CS-first approach: students aren't introduced to SE in a serious way until the second year. Our course's content strongly resembles that of SE 2004's proposed SE201 course, presenting SE's basic principles and concepts.

Software education traps

At one time or another, I've fallen into most of the traps discussed here, as have many colleagues with whom I've discussed SE education over the years.

Trap 1: An SE course needs an industrial project

The idea behind this assumption is that we should prepare students for "the real world," which is complex, full of inconsistencies, and ever changing. The real world also involves participants from different domains and has political and cultural aspects. To meet this challenge, we might base projects on real industry examples[5] or introduce obstacles and dirty tricks into student exercises.[6] The question is, how helpful is this?

Student overload. Prior to their second year, students usually have taken courses on programming, data structures, computer organization, and so on. In such courses, instructors typically structure the work clearly and give students unambiguous problems. And too often, the problems have only one right answer. In the SE course, students are suddenly overwhelmed with many new topics. Of course, it might be possible to gently introduce some SE principles in other introductory courses. In practice, this isn't easy in a CS environment.

So, at the start of our SE course, students aren't familiar with requirements engineering (RE) and don't know how to

- write unambiguous requirements or elicit them from stakeholders from other domains,
- prioritize requirements,
- relate requirements to effort (to them, all requirements are equal, regardless of their content), or
- document requirements.

Last but not least, students don't (yet) appreciate RE's value. For example, only a few years ago, I asked students to write a requirements document as their first task. In response, one student complained, "How can I possibly write down what the system does when I haven't programmed the damned thing yet?"

The problem isn't limited to RE. Design, testing, configuration management, quality assurance, and so on all face the same issues. Combining an introduction to all these topics with a real-life case simply asks for too much. Additionally, the students aren't mature enough to appreciate the importance of many SE topics. On one hand, many issues sound obvious: pay attention to documentation, apply configuration control, test thoroughly, and so on. On the other hand, our students have difficulty appreciating issues—such as team organization and cost estimation—that software professionals know from the trenches.

Simplify (when possible). In my SE textbook,[7] I use a swimming-lessons analogy. Around 1900, Amsterdam schoolchildren typically learned to swim on the school playground, practicing proper movements while lying on wooden benches. In contrast, my father grew up in the countryside and learned to swim the hard way. His father simply tied a rope around his middle, threw him into the river, and shouted: "Swim." Nowadays, swimming lessons start off gently, in a toddler pool with Mama and plenty of flotation devices. Gradually, the amount of floating material is reduced and the pool gets deeper.

I favor a similar approach. In my SE course, I view my students as toddlers on the SE playground. I concentrate on (at most) a few issues in an orchestrated environment.

While I cover all the requisite course topics—and tell my favorite anecdotes—the class project highlights only a few targeted issues. In later years and other courses, students will confront additional real-life aspects. I've often noticed that students' appreciation for my initial SE course comes only years after they've suffered through it.

Design is one key SE issue that instructors can address in an orchestrated way—and that's also a major hurdle for most students. Design is "wicked"[8] because of the following:

- *It has no definite formulation*. We can't neatly separate the design process from the preceding or subsequent phases because they all overlap and influence each other.
- *There is no stopping rule*. No criterion exists to tell us when we've reached *the* solution.
- *Solutions aren't true or false*. Design involves trade-offs between potentially conflicting concerns. Stakeholders in the design process might define different acceptable solutions rather than one best solution.

The latter point, in particular, opens up interesting project possibilities. An instructor might ask different student groups to design the same system but with different priorities (with respect to quality requirements or requirements priorities, for example). The groups might later collectively study and discuss the different designs. (My colleague and I have reported on experiences with this approach at the software architecture level.[9])

An interesting and often-applied option is to have a dual program, in which students spend, say, half a year in industry and half a year at school. This reduces the pressure on the university to include "real-life" course elements while also increasing the likelihood that students will appreciate typical SE topics. Unfortunately, that's not an option for all instructors (yours truly included), mainly because of university systems that target full-time students who enter the university right after high school.

Trap 2: SE is like other branches of engineering

All SE texts discuss the similarities between SE and other engineering branches—as well, of course, as the differences (interesting examples compare SE with bridge design[10] and high-pressure steam engines[11]). The overall message, however, is that the similarities prevail.

Engineering's limits. Although the engineering metaphor is useful, there's a downside to it. Our field uses numerous 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 engineering metaphor plays an active role in our thought processes.[12] For example, we generally characterize the RE process as follows:

- Information (the requirements) flows from A (the user) to B (the software engineer).
- Good communication doesn't involve any frictions or blockages.
- Good reception of the information involves only extraction.

The underlying model is that requirements exist somewhere; we just have to capture them. Thus, it's a *documentation* issue. If we run into problems, there must be a blockage or breakdown in the communication channel: "Why can't the users express their real demands?"

But there are other options, such as viewing RE as an *understanding* issue. It then becomes a dialog between parties, with the requirements engineer acting as a kind of midwife. The requirements aren't something immutable "out there," but rather, they're constructs of the human mind.[13]

Social dimensions. Numerous approaches—such as participatory design, rapid application design, joint application design, facilitated workshops, early user involvement, and so on—try to overcome the traditional, functionalist view's disadvantages with respect to RE. Given the clear assignments students are accustomed to from earlier courses, they often perceive the more open attitude toward RE as confusing. One student spoke for many others in labeling it "a badly organized educational exercise."

At a larger scale, a similar tension exists between the heavyweight, document- and planning-driven life-cycle models from SE's engineering realm and the various lightweight approaches that emphasize software development's human aspects. Combining the virtues of both is a major challenge. This is true for the state of the practice and even more so for the educational environment, where students are entrenched in the engineering view of the

Although the engineering metaphor is useful, there's a downside to it.

Today, engineers from all disciplines need social competences.

software development world and are not mature enough to perceive the limits of that view.

The latter became apparent recently when several students majoring in multimedia and culture took our SE course. These students clashed with the regular CS students, who held a rather one-sided, traditional view and failed to see and appreciate the nontechnical issues involved.

Today, engineers from all disciplines need social competences, including communication, organization, and conflict-resolution skills. Also, technological possibility is no longer the only driving force behind success. Increasingly, engineers must weigh competing values such as those related to economics, quality of life, and the social and economic impact of job eliminations.[14] We must prepare our students for this future.

Trap 3: Planning in SE is poorly done relative to other fields

Many papers on SE and SE education have quotes like "Approximately 75 percent of all software projects are either late or cancelled."[15] In his wonderful book, *Death March*, Edward Yourdon quotes the Standish Group and gurus such as Capers Jones and Howard Rubin, stating that, "The *average* project is likely to be 6 to 12 months behind schedule and 50 to 100 percent over budget." And "the grim reality is that you should *expect* that your project will operate under conditions that will almost certainly lead to some degree of death march behavior on the part of the project manager and his or her technical staff."[16] The sometimes explicit, sometimes implicit message is this: A better software education will help, and might eventually even do away with most runaway projects. I question this connection between SE education level and planning accuracy.

Findings on other fields' infrastructure projects. Looking to other fields can be instructive here. Engineers are currently building an expensive high-speed railway connection to carry freight from Rotterdam's harbor to Germany (and beyond). In 1992, officials estimated total costs at 2.3 billion euro; by 2000, they raised the estimate to 4.7 billion euro. Over the same period, the connection's freight estimates continuously dropped. Many people think the connection will never make money.

In 2005, the Dutch parliament launched an inquiry into the project. It first interviewed Danish economist Bent Flyvbjerg and his coauthors Nils Bruzelius and Werner Rothengatter, who studied over 250 international infrastructure projects.[17] They found that nine out of 10 projects underestimate costs, and almost all projects overestimate revenues. The combination makes projects look good and helps ensure decision makers' approval. Now, people naturally overestimate the good and underestimate the bad, particularly in cases of uncertainty. If you ask people whether they think more people die of cancer or diabetes, they'll most likely say cancer. In fact, it's diabetes, which most people consider to be the less dangerous disease. But there are other explanations as well.

Flyvbjerg, Bruzelius, and Rothengatter cite several well-known projects with spectacular overruns:

- Suez Canal (1869): 1,900 percent over budget
- Sydney Opera House (1973): 1,400 percent over budget
- Concorde (first flight in 1969): 1,100 percent over budget
- Panama Canal (1913): 200 percent over budget
- Brooklyn Bridge (1883): 100 percent over budget

On railway projects, they found that the average project overrun is 45 percent. Next come bridges and tunnels, which have an average overrun of 34 percent.

The authors dismiss technical explanations for such project overruns. If it were simply a matter of technology, then statistically, they would have also found projects with cost underruns. They didn't. Likewise, they dismiss psychological explanations related to estimators' natural optimism. If that were true, we could assume that estimators don't learn from past mistakes. The conclusion? Estimators intentionally underestimate project costs for political reasons: the pressure is high, the parties involved have already made a deal, the project "must be done," and so on.

Software analogue. Many of the arguments that hold for infrastructure project cost and schedule overruns are also valid for software

development projects. Educating future software engineers to better count function points, engineer requirements, and approach other key tasks won't on its own resolve overrun issues. As Tom DeMarco put it in 1982, "One chief villain is the policy that estimates shall be used to create incentives."[18] This is as true today as it was then.

In one interesting software cost-estimation experiment,[19] the authors studied the "winner's curse," which has the following characteristics:

- Software providers differ in their estimate optimism: some are overly optimistic, some are realistic, and some are pessimistic.
- Software providers with overly optimistic estimates tend to have the lowest bids.
- Software clients require a fixed-price contract.
- Software clients tend to select a provider with a low bid.

The resulting contract often delivers low or negative profits to the bidder; it can also be risky for the client. In one experiment,[19] for example, Magne Jørgensen and Stein Gromstad asked 35 companies for bids on a certain requirements specification. They then asked four companies to implement the system. They found that the companies with the lowest bids incurred the greatest risks.

Both Flyvbjerg and Jørgensen emphasize the need for careful risk management. As one experienced project manager told me, "Risk management is project management for adults." Risk management definitely deserves a front seat in a full-fledged SE curriculum.

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

> *We can't worry about these user interface issues now. We haven't even gotten this thing to work yet! —R. Mulligan et al.*[20]

A system's user interface is important: In an interactive system, about half the code is devoted to the user interface. In a recent study, researchers found that 60 percent of software defects arose from usability errors, while only 15 percent related to functionality.[21] In addition, adequate attention to user interface quality can increase sales of e-commerce sites by 100 percent.[22] For Web-based systems, usability goals are business goals. To improve the state of the practice, we should integrate appropriate user interface design techniques into our software development process. The place to start this practice is SE education.

Ignoring human factors. Is the SE community integrating user interface design techniques into the development process? SWEBOK and SE 2004 offer the most relevant answers here. SWEBOK lists human–computer interface (HCI) as a "related discipline" of SE, concerned with understanding the interactions among humans and other system elements. SE 2004 takes a similar position, describing an HCI course in which user interface design concerns topics such as "use of modes" and "response time and feedback."

Both organizations reflect Mulligan and colleagues' limited view of the user interface. This view totally ignores the fact that many current and future software development projects will aim to develop systems in which human use and related human factors are decisive elements of product quality.

A broader view. Interface design and functionality design go hand in hand. We might even say that the user interface *is* the system. There are two main reasons to take this broader view of the user interface. First, the system—and hence its interface—should help users perform certain tasks. The user interface should therefore reflect the task domain's structure. The design of tasks and their corresponding user interfaces influence each other and should be part of the same iterative refinement process. Like quality, the user interface isn't a supplement. Second, dialog and representation alone don't provide users with sufficient information. To work with a system, users sometimes need to know "what's going on behind the screen."

Various studies corroborate the need to better attend to HCI in SE and CS curricula. Timothy Lethbridge,[23] for example, addresses the question of what software professionals need to know. He found that HCI is one of the topics with the widest educational knowledge gap. As Lethbridge reports, practitioners called HCI an important topic but one they'd learned little about in school. Nigel Bevan[24] shows that we must expand the traditional quality assurance approach—which emphasizes software's static and dynamic proper-

Risk management definitely deserves a front seat in a full-fledged SE curriculum.

The state of the practice is still quite a bit removed from SWEBOK and the average SE textbook.

ties—to incorporate quality-in-use aspects that address broader ergonomic issues.

Proponents of traditional SE see user interface design as a separate activity and don't include it in the mainstream software development process model. We need a more eclectic approach in which we attend to user interface issues from the start. If, as I believe, the user interface is the system, then software developers must have a basic understanding of HCI issues. SE or CS curricula must therefore offer at least an introductory HCI course.[25]

Trap 5: SWEBOK represents the state of the practice

In my opinion, SWEBOK and SE 2004 lag behind the state of the practice in some areas and run ahead of the herd in others.

Outpacing reality. As an example of SWEBOK's running ahead of practice, consider data from MOOSE (Software Engineering Methodologies for Embedded Systems; www.mooseproject.org), a recent European research project. MOOSE researchers created a Web repository with 100 entries describing participating companies' industrial experiences. Of the companies' products, researchers found that

- 50 percent were developed without an RE method,
- 75 percent were developed without an RE tool,
- 25 percent were designed without a design method,
- 50 percent were designed using a generic drawing tool or no tool at all,
- 33 percent were tested manually, and
- 90 percent were developed using some configuration management tool.

Clearly, the state-of-the-practice is still quite a bit removed from SWEBOK and the average SE textbook. Fresh graduates are likely to enter environments characterized by the practices this data reflects, which will probably further increase the perceived distance between universities and industry. Although there's some room for improvement, industry prefers evolution over revolution.

Lagging behind. SWEBOK lags behind the state of the practice because the SE field changes rapidly. New approaches—such as model-driven development and service-oriented architecture—have made a considerable impact on both research and practice, but SWEBOK and SE 2004 have yet to mention them. The same holds for my favorite topic: software architecture.

Admittedly, SWEBOK and SE 2004 discuss software architecture, but in a rather shallow way. By and large, these documents view architecture as global design. The now-prevalent view of software architecture is that it involves balancing quality and functional requirements.[26] So, architecture design doesn't follow RE, it's intertwined with it.

Many architectural decisions involve trade-offs because they affect multiple quality attributes. For example, the choice of communication protocol can have implications for both performance and security. Through architecture design, developers make these trade-offs, communicate them and their consequences to stakeholders, and document them in architectural views. Software architecture is therefore about documenting and sharing important design decisions rather than the components and connectors that result.[27]

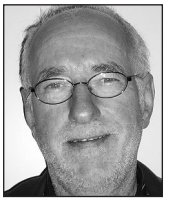
Heterogeneity. SE is becoming increasingly heterogeneous, as recent developments show:

- Distributed development, involving teams from different cultures, affects work processes.[28]
- For many organizations, combining in-house software developed with COTS with open source and other externally acquired software is now policy—if not a necessity—rather than an unfortunate event.
- To get the best of both worlds, organizations combine traditional, document-driven development approaches with the more recent, people-driven agile development approaches.

In their basic structure, SWEBOK and SE 2004 largely follow a traditional view. Although both periodically stress evolution's importance, the documents' surface structure suggests a greenfield situation, in which software is developed from scratch. Although unintentional, this structure is nonetheless likely to influence student attitudes. Because SE's emergent heterogeneity further complicates industrial practice, it should have some counterpart in education.

There's more to SE than engineering. A major challenge is to reconcile the engineering dimension with the human and social dimension in SE. Practitioners know from the trenches that both are important. Students are easily misguided if we only stress the engineering part.

About the Author



Hans van Vliet is a professor of software engineering at Vrije Universiteit Amsterdam. His research interests include software architecture and software measurement. He is the author of *Software Engineering: Principles and Practice* (Wiley, 2nd ed., 2000). He chairs Jacquard, the Dutch national research program in software engineering. van Vliet received his PhD in sciences from the University of Amsterdam. He is a member of the IEEE Computer Society and the ACM. Contact him at Dept. of Computer Science, Faculty of Sciences, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands; hans@cs.vu.nl; www.cs.vu.nl/~hans.

Acknowledgments

I'm grateful for discussions about the SE traps with Philippe Kruchten, Patricia Lago, and Chris Verhoef (of course, they don't agree with all of them). An extended abstract of this article was published in *Proceedings of the 26th International Conference on Software Engineering* (IEEE CS Press, 2005, pp. 621–622).

References

1. P. Kruchten, "Putting the 'Engineering' into Software Engineering," *Australian Software Eng. Conf.* (ASWEC 2004), P. Strooper, ed., IEEE CS Press, 2004, pp. 2–8.
2. D. Parnas, "Software Engineering Programs Are Not Computer Science Programs," *IEEE Software*, vol. 16, no. 6, 1999, pp. 19–30.
3. M. Shaw, "Prospects for an Engineering Discipline of Software," *IEEE Software*, vol. 7, no. 6, 1990, pp. 15–24.
4. H. Saiedian, D. Bagert, and N. Mead, "Software Engineering Programs: Dispelling the Myths and Misconceptions," *IEEE Software*, vol. 19, no. 5, 2002, pp. 35–41.
5. R. Dawson and R. Newsham, "Introducing Software Engineers to the Real World," *IEEE Software*, vol. 14, no. 6, 1997, pp. 37–43.
6. R. Dawson, "Twenty Dirty Tricks to Train Software Engineers," *Proc. 22nd Int'l Conf. Software Eng.* (ICSE 00), IEEE CS Press, 2000, pp. 209–218.
7. H. van Vliet, *Software Engineering: Principles and Practice*, 2nd ed., John Wiley & Sons, 2000.
8. D. Budgen, *Software Design*, 2nd ed., Addison-Wesley, 2003.
9. P. Lago and H. van Vliet, "Teaching a Course on Software Architecture," *Proc. 18th Conf. Software Eng. Education and Training*, IEEE CS Press, 2005, pp. 35–42.
10. A. Spector and D. Gifford, "A Computer Science Perspective of Bridge Design," *Comm. ACM*, vol. 29, no. 4, 1986, pp. 267–283.
11. N. Leveson, "High-Pressure Steam Engines and Computer Software," *Proc. 14th Int'l Conf. Software Eng.* (ICSE 92), IEEE CS Press, 1992, pp. 2–14.
12. A. Bryant, "Metaphor, Myth, and Mimicry: The Bases of Software Engineering," *Ann. Software Eng.*, vol. 10, 2000, pp. 273–292.
13. R. Hirschheim and H. Klein, "Four Paradigms of Information Systems Development," *Comm. ACM*, vol. 32, no. 10, 1989, pp. 1199–1216.
14. C. Lewerentz and H. Rust, "Are Software Engineers True Engineers?" *Ann. Software Eng.*, vol. 10, 2000, pp. 311–328.
15. T. Hilburn and W. Humphrey, "The Impending Changes in Software Education," *IEEE Software*, vol. 19, no. 5, 2002, pp. 22–24.
16. E. Yourdon, *Death March, The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*, Prentice Hall, 1997.
17. B. Flyvbjerg, N. Bruzelius, and W. Rothengatter, *Megaprojects and Risk: An Anatomy of Ambition*, 2003, Cambridge Univ. Press, 2003.
18. T. DeMarco, *Controlling Software Projects*, Yourdon Press, 1982.
19. M. Jørgensen and S. Grimstad, "Over-Optimism in Software Development Projects: 'The Winner's Curse,' *Proc. 15th Int'l Conf. Electronics, Communications, and Computers* (CONIELECOMP 2005), IEEE CS Press, 2005, pp. 280–285.
20. R. Mulligan, M. Altom, and D. Simkin, "User Interface Design in the Trenches: Some Tips on Shooting from the Hip," *Proc. Conf. Human Factors in Computing Systems* (CHI 91), ACM Press, 1991, pp. 232–236.
21. O. Vinter, P Poulsen, and S Lauesen, "Experience Driven Software Process Improvement," *Software Process Improvement*, Brighton, 1996; www.ece.utexas.edu/~perry/prof/ispa/icsp4/papers/o.vinter.html.
22. J. Nielsen, "Web Research: Believe the Data," Alertbox, 11 July 1999, www.useit.com/alertbox/990711.html, 2000.
23. T. Lethbridge, "What Knowledge Is Important to a Software Professional?" *Computer*, vol. 33, no. 5, 2000, pp. 44–50.
24. N. Bevan, "Quality in Use: Meeting User Needs for Quality," *J. Systems and Software*, vol. 49, no. 1, 1999, pp. 89–96.
25. G. van der Veer and H. van Vliet, "A Plea for a Poor Man's HCI Component in Software Engineering and Computer Science Curricula," *Comp. Science Education*, vol. 13, no. 3, special issue on human-computer interaction, 2003, pp. 207–226.
26. L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed., Addison-Wesley, 2003.
27. J. Bosch, "Software Architecture: The Next Step," *Proc. European Workshop Software Architecture*, LNCS 3047, Springer, 2004, pp. 194–199.
28. G. Borchers, "The Software Engineering Impacts of Cultural Factors on Multi-cultural Software Development Teams," *Proc. 25th Int'l Conf. Software Eng.* (ICSE 03), IEEE CS Press, 2003, pp. 540–545.