Realism in Project-Based Software Engineering Courses: Rewards, Risks, and Recommendations

Pierre Flener
Computing Science Division, Dept of Information Technology
Uppsala University, Box 337, 751 05 Uppsala, Sweden
Pierre.Flener@it.uu.se

Abstract

A software engineering course is often the capstone of a general undergraduate curriculum in computer science. It is usually at least partly a project-based course, with the intention that student groups can deploy their already acquired skills on programming, verification, databases, and human-computer interaction, while applying the new material about requirements, architecture, and project management on a project. I have taught a software engineering course six times over the last few years, using a combination of ideas that I have never seen elsewhere, with a strong emphasis on realism. I here reflect on the rewards and risks of this approach, and make some recommendations for future offerings.

1 Introduction

The discipline of *software engineering* proposes a set of principles, techniques, and tools for the multi-person development of correct and efficient software within an allocated time and budget, in response to a perceived software need. The required coordination efforts set it apart from *programming*, which is typically a one-person activity and starts (normally) from a precise specification. Such programming-in-the-small is just one of the many activities within a software engineering project, also known as programming-in-the-large. The other activities include requirements elicitation and specification, architecture specification, validation and verification, and project management.

A software engineering course is often the capstone of a general undergraduate curriculum in computer science. It is often the last course taught in the final year,[1] and builds on top of courses on programming, verification, databases, human-computer interaction, and so on. The concepts of requirements, architecture, and project management are often covered for the first time in a software engineering course. It is usually at least partly a project-based course, with the intention that student groups can deploy their already acquired skills while applying the new material on a project. In that case, the pedagogical approach

[1] The 'big picture' is of course taught in some introductory first-year course.

taken fits the modern drive for *problem-based learning* [6]. There is a flourishing literature on how to teach software engineering, witness the *Conference on Software Engineering Education and Training* (CSEE&T) annual series [1] and numerous articles in software engineering journals, as well as in the general computer science education or research conferences and journals.

2 My Software Engineering Course

I have taught a software engineering course six times over the last few years, namely from 1995 to 1997 at the Computer Engineering and Information Science department of Bilkent University in Ankara, Turkey, and from 1999 to 2001 at the Information Science department of Uppsala University, Sweden.[2] Both courses were taught to fourth-year students in a project-based way. The first time I gave the course, I had thought long and hard about it. I come up with a combination of ideas that I have never seen elsewhere, namely:

- The instructor is the *coach* for all the teams.
- The instructor is the (*external*) *client*, with a software need well outside the area of expertise of probably all the students.
- The client is *insecure* about the actual software need, is largely *ignorant* about programming and computer issues, and even has *mind changes* about some objectives after receiving the first two deliverables.
- The first two deliverables are *revised at least once*, so as to accommodate feedback from the coach and the mind changes of the client.
- The software is *neither implemented nor validated and verified*, mainly for time reasons.

In the following, I will describe the course in more detail. See user.it.uu.se/~pierref/courses/SE/ for its most recent incarnation, developed from the feedback on the previous five offerings.

2.1 Goals and Overview

My main objective is to make the students familiar with the fundamental principles and techniques of software engineering. This will allow the students to acquire further knowledge later, be it through advanced courses or through self-study and practice. This also means that my lectures have a slightly theoretical flavour, as they discuss *no* particular tools. Once the fundamental principles and techniques are understood, the usage of tools is eased, whereas the study of any particular tools would not prepare the students for the usage of other tools nor of the better ones that are bound to emerge. Furthermore, the usage of tools without understanding their underlying principles and techniques is outright dangerous. The focus is on understanding why a particular principle or technique should be applied, or why it should not be applied, rather than only on how to apply it.

[2] The software engineering course at my current department was in able hands when I moved there in late 2001.

Some practice of software engineering is acquired by the students through the specification and design (but *not* the implementation, validation, and verification) of a software application, as a project for teams of about five students. For comparison purposes, the *same* project is assigned to all teams. The topic is at my discretion, but is chosen so as to be most likely *outside* the expertise of *all* the students.[3] Such a project for an *external client* has the advantage of learning what it means not to decide in-house what the requirements actually are, so that the client *must* be consulted on *every* single issue.

For pedagogical purposes, the first two deliverables are *revised* at least once, so as to learn that no such document is ever perfect, because of evolution of the requirements and because of errors in content and style. It is this essential feature of my course that makes it impossible to have the teams go on to the implementation, validation, and verification of their software, as the usually allocated time span of the course is too short. In any case, it is debatable whether a course should allow students to spend so much time applying something they have already learned in other courses, despite the joys of seeing one's product in action (if it ever runs): I have seen software engineering project-based courses degenerate into programming fests once the teams were stuck in the morass of their poor early decisions, with the hope that an actually running software product would sway the instructor.

Major methodological decisions are taken by the instructor, but the actual choice of techniques, notations, and tools is left to the teams, under the instructor's *coaching*. As a textbook, I used (the first version of) the excellent *Fundamentals of Software Engineering* [2] at Bilkent University, but switched to *Software Engineering: Theory and Practice* [4] at Uppsala University, while awaiting the now available second edition of the former. For the project management, we followed the introductory version of the *Team Software Process* (TSPi) [3] at Uppsala University. The students used or adapted its on-line supplements, such as the forms and the support tool. For the requirements process, we followed the *Volere* methodology [5] at Uppsala University. The students used or adapted its on-line Volere Requirements Specification Template.

2.2 The Project

Upon filled-out TSPi forms from each student, indicating their strengths, weaknesses, role preferences, and teammate preferences, the instructor divides the class into teams (ideally of size 5, otherwise of size 6), and assigns a TSPi management role[4] to each team member. Of course, in addition to their TSPi role, every team member is a software engineer and is expected to contribute to the entire project. The instructor is the coach for all the teams, as well as their client. The *same* project is then launched by all the teams. It is designed to be not feasible properly by less than 4 students, to avoid free-loading students.

[3] Five times, I decided on software support for the various chairs of scientific conferences. Having been a programme (co-)chair of three conferences and several workshops, with no or very little software support, made me a very convincing client. One of my Bilkent students actually went on and developed a commercial product from my requirements. In the meantime, there is a flurry of such products, enabling inspiration for the project over the internet, though the topic retains its mystery to most students.

[4] Team leader, development manager, planning manager, quality and process manager, or support manager.

Through appointments with the client, each team elaborates a **requirements specification** for a software helping to solve the problem. The client may be accidentally inconsistent or incomplete about the software need across the various requirements elicitation sessions. However, I also deliberately sneak in some inconsistency and incompleteness, in order to see whether the team eventually detects that or not. Upon constructive feedback from the client and coach, each team eventually has to produce two more versions of their requirements specification, because of errors and inadequacies in earlier versions or because the client has mind changes on some objectives. I do the latter deliberately after the second version. The teams are explicitly warned to plan for change.

Each team also has to come up with an **architecture specification** corresponding to their requirements specification, starting from its second version. Upon constructive feedback from the coach,[5] each team eventually has to produce a second version of their architecture specification, corresponding to the third version of the requirements specification, because of errors and inadequacies in the first version or because of the mind changes in the requirements.

Finally, *without* actually writing the software (according to the third requirements specification and the second architecture specification), each team has to produce a **test plan** according to which they would validate and verify the software, such that it would give them a reasonable degree of confidence.

2.3 Additional Realism

Just like in real life, it is not unusual for team members to quit (by dropping the course) or be temporarily unable to perform. Also, teams can break apart because of internal tensions. The teams are asked to have contingency plans for all this, and in any case the team leaders have weekly meetings with the coach to report on leadership issues.

I have toyed with the idea of letting the teams freely swap members, if not to operate myself such swaps. However, project re-staffing is considered bad practice because the effort of integrating a new engineer usually does not pay off, especially in struggling projects. Also, as all teams would be affected by and learn from such realism, this may be considered unfair by the designated students, unless they volunteer. Hence I have never carried out such an experiment, except when a team shrank below critical mass and I distributed its members over willing teams. It was a decision I regretted later on, as the team was very good and its ex-members lost much of their motivation while trying to integrate into their new teams.

Another way of conveying how hard it is to be integrated into an ongoing project is to rotate the documents among the teams at some checkpoint. If warned about this, the teams would take extra care to leave nothing undocumented. However, I decided this would be too drastic an experiment.

2.4 Student Evaluation

Within the exam, each team has a two-hour oral project postmortem with me, only over the final versions of their documents for the project, and, if need be, over some of the course material:

[5] Clients usually do not look at the architecture specification.

- 30% of the grade depend on the requirements specification;
- 20% of the grade depend on the architecture specification;
- 10% of the grade depend on the test plan;
- 20% of the grade depend on participation in the classroom and in meetings with the client, coach, or instructor, including the exam;
- 20% of the grade depend on participation in internal team meetings, according to an anonymous TSPi peer-evaluation conducted at the exam.

The re-exam is a written exam over the course material. It is reserved for those who failed the first exam and thus actually participated in a project team, as the project is a mandatory part of the course and can only be taken while the course takes place.

3 Course/Teacher Evaluation: Rewards & Risks

In general, my students overall loved this course, witness the following excerpts from the course and teacher evaluations, identifying the **rewards** of this course:

> "Course workload, project, content, lessons, organisation, etc were excellent. I really think I can take advantage of what I learned in this course."

> "The lectures have been *very* interesting and the course is the *best* in preparing us, the students, for real-life work at companies. [...] The attitude against students feels professional. The instructor really wants us to learn much of the course and prepare us for industry."

> "The teacher kept a good balance between academic and industry-related issues, so we can feel this course to be very useful when we start working."

I have also received various emails from former students after they spent a few years in industry as software engineers. Examples are:

> "I want to thank you so sincerely for all the knowledge you shared with us in the classes. I just wish I paid a little more attention. [...] The software engineering course you taught me is still my main reference (I still have the Software Engineering book we used as reference in the class and the class notes that some of which I took and many others were copied)." — A Bilkent'96 graduate, in May 1999

> "I had found myself at particular ease compared to many others and was nominated internal quality auditor and a vice-manager of quality assurance. I have to recognize that your course has been my guiding line through more than a year of work. And I'm indebted to you for your excellent tutorship and owe you this modest regard." — A Bilkent'97 graduate, in August 1998

However, is is impossible to have all the students on one's side, and many students who appreciated the course also came up with constructive criticism, which I always addressed in the next offering. There are indeed also **risks** in teaching the course this way, as shown in the following.

The most common complaints were about the inordinate workload, though this was usually confessed to be self-induced, witness an open letter to future classes:

> "We did not start working on the first version of the requirements specification document on the time that Dr Flener has suggested. [...] The problems cascaded to the other documents as well [...] extremely difficult to cope with. [...] Just start working on time, the rest must be enjoyable." — The Bilkent software engineering class of 1997

As hinted in this letter, I did earnestly warn the students that getting started early and well will enormously pay off. I did so every year, and no team ever heeded this advice, only to bitterly regret it later on.

Several students used the phrase 'shock therapy' for my realism-based approach, referring to the impact of the client's mind changes after the first architecture specification has been handed in. Despite my clear admonitions to expect the unexpected and to plan for change, and despite the mildness of the mind changes, if not my hoodwinking about what they might be, that impact was usually traumatic. Some students found the idea quite good, but others commented as follows:

> "I experienced more shock than learning during this course."

> "If your aim was to make our projects fail, then you have succeeded."

Learning from one's mistakes is certainly a powerful learning method, but it should not become the only or major one (unconsciously) deployed in a course. Unfortunately, a software engineering project staffed entirely by undergraduate students seems predestined to run into trouble, unless significant amounts of coaching and predecessor courses are available.

Among the most serious objections was also the following:

> "The student life/work environment is completely different from the real world and there is no way and no need for trying hard to impose any similarity."

As the more positive comments above indicate, this may be an opinion that only some students form. The only way to remedy this seems to teach the course without any project work, which is generally not done. Maybe such students, especially those who think they will not become actual software engineers, can be offered to take a theoretical exam instead of doing the project team work.

4 Instructor's Viewpoint: Rewards & Risks

It is very **rewarding** to see one's students get the best out of team work and eventually hand in quite good final versions of their deliverables. The improvement compared to their initial versions is usually amazing, which confirms the

merit of abolishing write-once documents in courses. Seeing the students rebound from poor initial choices and from the client's mind changes is also a gratifying sign that the material taught eventually pays off.

However, the amount of time (and manpower) necessary to properly conduct the course the way described is usually not allocated by study directors. This leads to severe **risks** for the instructor, as argued next.

A recurring advice in the software engineering education literature [1] is that a single instructor cannot effectively coach projects involving more than 25 to 30 students. I never needed to enforce this quota, but was once forced (for department finance reasons) to teach 35 students. It was a gruelling time, as I did not want to compromise too much on my team-wise coaching. I enforced strict, but generous, time limits for reading each submitted document. The students that year correctly complained that my annotations often just pointed to problems but did not address how to fix them. Since every team also had a strict, but generous, time budget to interview or consult me, it was hard for them to get access to all my feedback on their progress.

Even with fewer than 25 to 30 students, it is very hard to adequately conduct the course this way. Even if the instructor spends (a lot) more time than allocated to the course, drawing on spare time to do so, some students may be unsympathetic to such dedication. They may fail to weigh it in when writing course and teacher evaluations, or they may consider it unnecessary for lack of their own motivation, if not unwarranted. Despite the (much) higher weight of the positive evaluations, this kind of comments hurts.

Finally, recent industry trends such as extreme programming, the anarchic community-based development of (excellent) software such as the Linux operating system, and the so-called web-speed development of software before the burst of the IT bubble in 2001 go a long way in undermining student belief in the relevance of a classical academic software engineering course, where there is much focus on project management and non-software deliverables. As long as there are projects successfully conducted with, and because of, that more organised and slower approach, the instructor must be able to motivate the latter to the students, lest they think learning something irrelevant in practice.

5 Recommendations

If I were to teach the course again, I would still teach it the way I did so far and described here, but I would try and comply with the following recommendations. They stem from my own observations and most of them were repeatedly asked for by the students. Their implementation is often beyond the power of the instructor, but rather a policy change by the department.

The student experience is likely to be more meaningful and less painful if there were *predecessor courses on requirements specification and software architecture*. After just a few lectures on these topics, which have entire textbooks dedicated to them, within a software engineering course, the students are not ready to properly deploy and extrapolate that knowledge on even small projects within just a handful of weeks.

Similarly, a *predecessor course on technical writing* (*and typesetting*) is absolutely essential. A software engineering course is the wrong place to start actually teaching these important skills, and comes too late for that in the cur-

riculum anyway. Even students not destined for graduate studies will benefit from such a course, namely when writing course reports such as on their internship, as well as in their professional careers. Even a *predecessor course on rhetoric* would be beneficial, as many students find it hard to phrase good questions when interviewing the client. Good communication skills are essential to software engineers, despite the popular perception that they must be so-called geeks.

The instructor must enforce a limit of 25 to 30 students, especially if alone. Finding competent teaching assistants and making their contribution consistent with the one of the instructor is very hard, so ways need to be found to involve other instructors. For instance, the coaching of software engineering course projects should involve more people than its instructor. Most computer science departments have requirements, architecture, database, human-computer interaction, or validation and verification experts, who ought to coach all the teams on their topic and thereby reinforce what they already taught them in previous courses. Even in this scenario, the actual instructor can probably not handle more than 30 students.

A full-speed ten-week-period[6] is way too short to conduct the course as outlined here. Beyond the (debatable) loss of the implementation, validation, and verification of the software product, the price to pay is that not enough necessary material can be taught and exercised on small problems on time before having to be deployed on an actual project. In any case, software-project cost estimation methods, such as COCOMO, reveal five-person-teams as incompatible with ten-week-projects. The predecessor courses mentioned above would help in that direction. Otherwise, a *half-speed full-semester course* should be scheduled.

The *TSPi roles* and their detailed descriptions are much appreciated by the students to give structure and guidance to their teams and activities. If the project does not involve actual programming, validation, and verification, then the development manager and quality and process manager roles need to be given more work than prescribed in [3]. However, the students invariably perceived the TSPi management paperwork to be suffocating, so it ought to be carefully tuned down.

Learning from peers can also be added. For instance, each team can review the documents of another team, in addition to the reviews done by the client and coach.

Acknowledgements

I am deeply indebted to my former software engineering students, for making this course so enjoyable.

References

[1] *Conference on Software Engineering Education and Training* (CSEE&T) annual series. Early proceedings in Lecture Notes in Computer Science at Springer-Verlag, later ones at IEEE Computer Society Press. 1987–present.

[6] as used in Sweden

[2] Ghezzi, Carlo and Jazayeri, Mehdi and Mandrioli, Dino. *Fundamentals of Software Engineering*, second edition. Prentice-Hall, 2003.

[3] Humphrey, Watts S. *TSPi – Introduction to the Team Software Process.* Addison-Wesley, 2000.

[4] Pfleeger, Shari Lawrence. *Software Engineering: Theory and Practice*, second edition. Prentice-Hall, 2001.

[5] Robertson, Suzanne and Robertson, James. *Mastering the Requirements Process.* Addison-Wesley, 1999.

[6] Woods, Donald R. *Problem-Based Learning: How to Gain the Most from PBL*, *Helping Your Students Gain the Most from PBL*, and *Resources to Gain the Most from PBL.* Department of Chemical Engineering, McMaster University, Canada.