Software Engineering Lab – an Essential Component of a Software Engineering Curriculum

Mira Balaban
Computer Science
Ben Gurion University of the Negev, Israel
mira@cs.bgu.ac.il

Arnon Sturm
Software and Information Systems Engineering
Ben Gurion University of the Negev, Israel
sturm@bgu.ac.il

ABSTRACT

The goal of Software Engineering (SE) education is to teach theory and practice of software sciences, with an ultimate practical goal. Quite surprisingly, although standard SE programs include many hands-on courses, they do not include practical lab courses in software development and maintenance, as common in other engineering disciplines. A capstone project course, that is standard in most SE programs, cannot function as an instructive SE-lab course since it does not enable effective teaching and cannot replace a planned SE-development experiment.
This paper describes an SE-lab course that creates lab conditions, where students are faced with a deliberately challenging, planned SE development and management tasks, and the course staff provides team-based guidance. The paper analyzes the lab ideals, principles and goals, explains how they are realized in the lab content, and presents its evaluation from the students and the instructors' viewpoints.

CCS CONCEPTS

K.3.2 Computer and Information Science Education

KEYWORDS

Software Engineering Education, Lab, Questionnaire

1 INTRODUCTION

Software engineering (SE) is a practical paradigm, where practitioners apply software engineering techniques, approaches, methods, ideas, tools, criteria and values, to create and manage software applications. The goal of software developers is to create software, which is sustainable, manageable, extendable, useful, and reusable (movable) [1].

ICSE-SEET'18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5660-2/18/05…$15.00
https://doi.org/10.1145/3183377.3183395

The goal of software engineering education is to teach theory and practice of software sciences, but the ultimate goal must be practical: Graduates of software engineering should study and practice production and management of good quality software [3] [4]. Observing the curriculum in other engineering fields, e.g., Chemical Engineering, we see that it includes besides theory courses, lab courses where students get practical experience and focused instruction in using engineering techniques. The importance of such labs in engineering education and their fundamental objectives are discussed in [5]. In addition to these lab courses, there is a capstone project, where students are expected to independently initiate and deliver an engineering product. Lab courses should be distinguished from practice labs that are associated with courses and are task oriented, highly structured, and usually do not encourage broad abstract decision making.
Quite surprisingly, although standard software engineering study programs include many hands-on, studio, and project based courses, they do not include practical lab courses in software development and management, as common in other engineering disciplines. Somehow, students are expected to internalize, embed, apply and integrate all methods and turn them into a practical body of knowledge, without practical laboratory instruction. The standard software engineering curriculum includes general background and foundation courses in Mathematics, Physics and Liberal Arts, Computer-Science foundation courses, core Software Engineering courses and a final capstone project [3]. However, there are no dedicated lab courses for practicing SE methods and techniques. Indeed, Project Based Learning (PBL) [6] is based on learning via practical experience, but it relies on the students taking responsibility for their learning. Furthermore, when multiple tasks and deliverables are required (as in most software projects), close feedback is missing.
One explanation for this unusual shortage lies in the self-embedded oxymoron of the mere concept of "Software Engineering instruction". While SE targets complex, large-scale systems that require stability, interoperability and reliability, the teaching framework enforces restriction to small, short-term, relatively simple systems, where it is hard to demonstrate the essential role and value of using SE techniques. Moreover, SE techniques usually impose a "non-constructive" and delaying burden on the activity of software creation, while the "direct programming" approach is straightforward, constructive and fast rewarding. While students, after mastering some programming skills, are enthusiastic on producing software with observable performance, associated

software engineering activities like design (via modeling), testing and requirement analysis are met with a sigh. In addition, SE methods require some programming experience and maturity that most students lack. These "built-in paradoxes" are summarized in Table 1 below. The difficulties raised by the built-in paradoxes partly explain the lack of an SE-lab in the standard SE curriculum.

Table 1. SE Practice and Education Built-in Paradoxes

	SE Practice	SE Education
System Characteristics	• Complex • Large–scale • Stable • Interoperable • Reliable	• Relatively simple • Small • Relaxed stability • Usually isolated • Relaxed reliability
Techniques and Methods	Systematic design	Feasibility of direct programming
Maturity	Experienced developers	Novice programmers

Another possible explanation is that the graduation capstone project is often pointed at as the context where students are expected to exploit and integrate the overall plethora of tools taught during previous courses. Indeed, capstone projects form an integral part of most SE curricula. The goal of such projects is to let students face the challenges of applying the theoretical and technical knowledge, they gained in their studies, to solve realistic problems. In addition, such projects also enable practicing other important capabilities such as teamwork, project management, and communication and presentation skills.

The SE curriculum at Ben-Gurion University of the Negev also includes a yearlong (two semesters) capstone project. The projects are conducted by groups of 3-5 students that construct software solutions to problems of various kinds. We put special emphasis on having external stakeholders in order to stress the work with realistic problems. Such projects have been originated from various domains, including education, machinery, mobile, and medical. The projects specification imposes an iterative development, with emphasis on careful requirements analysis, modeling, prototyping, testing and validation. In general, the resulting projects are impressive and demonstrate high software development capabilities.

Nevertheless, our experience in leading this successful capstone project course for quite a few years shows that leaving all practical experience to the capstone project does not achieve the expected results and it is evident that much training is required. The major limitation is the lack of practical experience in applying SE processes and methods. Although the SE curriculum includes a well-constructed core of courses that teach SE theory, methods and techniques, there is no opportunity for ***instructive*** practical integration of methods into a working system. Consequently, quite frequently, the effective SE-education is limited. We also see that development and usage of associated SE artifacts are neglected in favor of direct programming tasks.

We believe that the capstone project course cannot replace the need for an instructive SE-lab course. This framework does not enable effective teaching, and besides, a capstone project as demanding as it might be, cannot replace planned lab instruction that is intended to trigger application and integration of methods, sometimes in deliberately extreme conditions. For that purpose, there is a need for an SE-lab, for intentional investigation and usage of SE problems, processes and methods. Such a lab would provide opportunity for an instructed integration of SE skills, methods, approaches, and technologies and would further allow discussion on software engineering related decisions.

In order to address this need, we devised an SE-lab course, where the students cope with a natural SE task in a guided manner.

The lab mimics a natural SE-development task, where usage of SE procedures is valuable and productive. It embodies the following features:

- Adopt an agile development approach.
- Maintenance challenges: Create a natural environment of successive changes due to complex evolution.
- Enable a thoughtful non-naïve customer (course staff), that plan intentionally complex requirements.
- Insist on product validation.
- Enforce SE-systematic development, including models, validation, testing, requirements analysis and reliability.
- Cope with technology challenges.
- Structured teamwork.

The goal of the lab is to acquire experience in using SE methodologies, methods and technologies. The expected learning outcomes of this course are experiencing the following aspects of software development:

- Separate essential requirements from technical organization.
- Analyze modes of usage, usability and needs.
- Analyze requirements and select appropriate architecture.
- Analyze alternative design solutions.
- Separate logical architecture from technical architecture.
- Cope with technological requirements and select technology.
- Plan evaluation and maintenance processes.
- Practice teamwork.

The contribution of this paper is in establishing an SE-lab course that adopts an instructive software engineering practice. The paper further evaluates the course from both the students and the instructors' viewpoints.

We describe the evolution of the SE–lab course until it reached its current form. Next, we discuss the course content and structure, and finally present an evaluation and discussion of the course from various perspectives.

2 HISTORY OF THE SE-LAB COURSE

The SE lab course has emerged from an initiative to introduce a technology-tools lab. This initiative failed since many students felt that at the end of their 3rd year they do not need an instructed lab for studying tools, and also since the staff of teaching assistants was not more knowledgeable on these matters than the students (sometimes it was the other way around). An observation of this lab revealed that the students missed a challenging context, i.e., an SE-challenging task, which demands using technological tools. Just coping with tools was not satisfactory.

The next stage in the evolution of this course was a common-project team-based course, i.e., a course where all students develop the same project in individually instructed teams of 5 students. The project framework was almost completely free, apart of technological requirements that included multiple user interface requirements and a database. The project subject, almost from the beginning of this stage, involved software for constructing a social forum application. This task enables combination of design, maintenance, validation and technological challenges. On the technology aspect, it requires usage of persistent storage, communication networks, multiple user interfaces and interactive behaviors. On the design and maintenance level, it requires thoughtful and flexible architecture. On the validation aspect, it requires an overall evaluation framework, including complex integration and regression testing.

This stage was a major advancement over the former tool-oriented lab. Yet, it posed several problems:

1. Floundering: The unstructured, free development framework caused some teams to flounder, not knowing how to start (see a similar observation in [7]).
2. Bad solutions: Most teams started with building a database, on top of which a thin application was developed. This led to a poorly structured application that could not cope with major requirement changes. For example, lousy handling of communication logic (mixing technology with logic), made addition of a new web-based interface more difficult. The expectations of the staff were that bad design would show its problematics when facing hard requirement changes. Practically, what happened, was that since the developed systems are still relatively small instructional size (recall the paradoxes from the Introduction section), most teams were able to cope with hard requirement change, in spite of the bad design.
3. Unbalanced teamwork: Many teams suffered from uneven experience of members. In many teams, there were one or two students with some technology prior knowledge (due to previous industry experience). These students simply controlled the project development of the team, which practically left the other team members paralyzed. In many cases, these "technology-masters" dictated the whole development. For example, students with web and database experience built a database system, used database tools to creating an application with a web interface. The dichotomy in such teams was between the members that actually built the project, to those that were unable to contribute, due to lack of technological experience. Therefore, it was difficult to rightly evaluate the contribution of individual members.
4. Grading difficulties: Due to lack of process structure, uneven student experience, excessive usage of (sometimes unnecessary) technology, badly structured systems that achieved all listed functional and non-functional requirements, and lack of prescribed student roles, it was difficult to faithfully grade the different teams and their members.

The major question that we asked ourselves involved the contribution of this course, in view of having a final capstone project. We realized that we need a better structured, directed course, where we can plan the SE content and create extreme "lab-conditions" that call for thoughtful usage of SE methods.

The next, current stage, takes a structured-lab approach, with changing prescribed team roles, and a declared software quality moto: *Abstraction before performance*. The structured approach helps students in starting the development, the team roles clarify individual contributions, and the project moto clarifies that the emphasis is on a good design, rather than showing-off mastering of technologies. In particular, the "database first" tendency, with automatically created application is not an option any more. The current project topic is construction of a *forum-management system*, i.e., a system for construction and maintenance of multiple, possibly inter-connected subject-specific human forums. The functionality of each forum is characterized by its *policy rules* that describe and constrain its structure and behavior.

In the following, we provide a comprehensive description of the SE lab and its management.

3 DESCRIPTION OF THE SE-LAB COURSE

The course starts with a couple of plenary (whole class) lectures, intended for an overall scanning of the course *ideals, principles, goals,* and *organization,* as well as for a condensed review and discussion of relevant SE methods. The overall project plan is dictated by the presented approach. Team-based work on the project starts in parallel, and is managed by bi-weekly team-individual instructed sessions.

The software **ideals** that we emphasize center around abstraction in the software artifacts. We insist on including a short, abstract, platform and technology independent description of the project goal. For example, for the Forums management application we have the specification: "A Forum enables open, real-time discussions among its users, on a topic of public interest. The Forums-maintenance system can construct and maintain forums along time, and supports different kinds of inter-relationships among forums, e.g., subject-related or user connections". Usually, this specification triggers a class discussion, as students expect mentioning web-management, or explanation of terms like "subject-related". The class discussion helps clarifying the role of platform-independence and concept-abstraction.

Another ideal that we emphasize is the essential distinction between *what* the software should achieve to *how* it is achieved. This is a difficult issue for inexperienced students, that usually tend to jump directly to the *how* challenge. This ideal implies the "no direct jump to coding principle".

The **principles** that we try to convey involve the primary status of software properties, over software writing. A natural characteristic of novices (well, as humans) is to just jump ahead and code. We emphasize the principles of testable and traceable requirements, and modifiable, and survivable code. These principles imply emphasis on well-defined requirements; first-class status for tests – organization and integration; inter-connected requirements, tests and code; and code abstraction, traceability and alternatives.

The **goals** of this course are: (1) to enable students to experience and appreciate disciplined software development; (2) to demonstrate the importance of software design and abstraction; (3) to let students select, practice, cope with and master new technologies; (4) to let students internalize, absorb and adopt the self-discipline necessary for a planned thoughtful software development and management; (5) to convey the necessity to balance between efforts invested in software design to those invested in technology mastering; (6) to acknowledge the challenges in team work.

Lab practices: The course organization is dictated by the goals. In order to create demanding SE conditions, the staff functions as a knowledgeable customer, that: (1) provides detailed well written requirement documents; (2) plans the development process; (3) specifies the expected artifacts; (4) enforces a variety of extreme situations like technology failures. In accordance with the course principles, there are no vague requirements, i.e., all requirements must be automatically testable. For example, "a user-friendly GUI" is not automatically testable, while concrete requirements about some GUI widgets are.

In order to facilitate a supportive development environment, we encourage the students to team up to groups of five students each. That way we reduce personal conflicts that may negatively affect the development effort. Each team has a coach (mentor) and there are bi-weekly 2-hours team meetings. A team meeting consists of a short (10 minutes) oral presentation of a technical subject, by a member of the team (serves the goal of coping with new technologies); a summary of the achievements by the team member that functions as the version leader (serves the goal of teamwork and disciplined development process), and a detailed planned discussion of the version content (serves the goal of planning, design, validation and traceability). The intention is to enable a constructive discussion, in a pleasant and relaxed atmosphere, rather than a test. Therefore, although each version and meeting are graded by the coach, errors or bad decisions or malfunctioning can be corrected without penalty. Moreover, efforts to locate the individual contributions of team members are deliberately restricted.

The project is developed using an agile process of four versions. During the development, there are few plenary sessions for discussing version-related general issues. The work on each version starts with a version-requirements document written by the course staff. The version document is extracted from the original general overall project requirements document. Version requirement documents usually leave some points open, to be further augmented by the teams. For example, specification of possible forum policies, password constraints, or user rules. This is done in order to enable students to practice requirement engineering. Yet, this element in the course must be restricted, in order to preserve common control and tracking for all teams.

In each version, team members change *roles*: There is a version *leader* who is responsible for the overall version results; version *client-tester* who is responsible for the version acceptance (black-box) tests, and does not participate in the version implementation; a *modeler* role and version *developers* (the modeler and the leader are also developers).

First version: This version is the most important in the project construction since it lays an initial basis for the core logic component and its associated artifacts. It is deliberately technology independent, so to enable concentration on analysis, design, management and validation. This decision was taken following several years where the first version included some heavy technology of client-server communication and persistent storage. This led most students to concentrate on mastering standard layered architectures of web-based applications, while neglecting analysis of the project under development. An even worse situation was using dedicated web-application technologies that dictated a data-base-oriented application structure. When this happens, we remind the project moto: "abstraction before performance", as such a technology-dictated structure leaves little room for abstraction. The version is split into two parts: *Modeling* and *implementation*. For each part, the teams receive a requirements document, which specifies the subset of the requirements and the associated artifacts to be constructed. In addition, this document describes the administrative individual roles of team members, and specifies the subject for the student presentation in the next meeting. For each part, the teams have roughly a two-week period in which they can independently interact with their coach, and ends in a team meeting, as described above.

The requirements document for the first part specifies a small set of essential functional and non-functional requirements and constraints for starting the project. Teams are expected to hand-in: (1) detailed use-case analysis, using activity diagrams as a means for structured process specification, and with analysis of risks, interconnections and nuances; (2) an architectural design of vertical and horizontal components for the necessary business logic; (3) intra and inter component analysis and design of structure and processes; (4) planning of validation instrumentation, in the form of a testing machine, the tests for this version and their architecture; (5) planning of a traceability means which includes requirements, tests, and code. The tester and the leader of the first version are responsible for the design of the testing machine. The whole group is responsible for the modeling artifacts.

The team meeting that follows this part mainly focuses on discussion of modeling decisions, their advantages and risks. For example, use-case design for functional requirements is associated with a concrete specification of possibilities of expected failures and implied consequences. These failures and consequences must be reflected in the use-case associated tests. The demand to use activity diagrams for describing the process of activities in a use-case is intended to enforce independent study of a modeling language that is not taught in the Software Modeling course.

Inner component structure analysis is expected to include detailed class models with associated constraints and functionalities. Concrete architectural and structure decisions are discussed in order to make sure that they are thoughtful decisions rather than casual unplanned. For example, the implication of making a Security or a Forum-policy independent vertical components are discussed using concrete examples of interactions with other project elements like retrieval and verification of user passwords or forum policies.

Architecture discussion covers also the need for a *service* layer for use-case management in every component, so to enable flexibility. Team coaches initiate discussions about changes in functionality, like modification and removal of use-cases, and the benefit in having a modular service layer. Traceability functioning is reviewed by following various trace scenarios, e.g., from a requirement to a use-case and a test, or from a test to a use-case and a requirement. The plan for a regression test, using the testing machine is observed, as well as the plan of test architecture.

The second part of the first version is mainly devoted to implementation of the first part artifacts, with addition of a few functionalities. The final result of the first version is a standalone software that can be operated via an associated testing component, management tools like logging and error-recovery tracking, and a version document that includes its requirements, models and a requirement traceability tool. The central role of a testing component right from the first version eliminates reliance on technologies that do not lend themselves to heavy testing, as we have seen in the previous stages of this course. Moreover, it conveys the importance of testing right from the beginning of a project (as writing tests is not a popular activity, and usually requires an enforced development procedure). The team meeting is devoted to demonstrating the implemented functionality and the management and traceability capabilities. If time permits, the coach tries applying an ad hoc small requirement change. This is also tried in later versions.

Second version: This version provides the first opportunity for coping with technology, and inserts the need for balancing technology with design. On the technology aspect, the version introduces two technological requirements: A client-server communication between a GUI user interface, and persistent storage. The main new logic functionality in this version includes interactive connection between users and the application. The interplay between the client-server communication to the interactive connection raises important issues of abstraction and design. The insertion of technology raises issues of stability in case of technology failure.

Teamwork on this version is preceded by a plenary session, where we discuss lessons from the first version, the technological functioning required for the second version, and the interactive communication requirement. The new technology requires teams to independently select persistent storage technology and client server communication. Team roles from the first version are switched. The new leader appoints a new tester, team members for studying and suggesting the necessary technology tools, a modeler and developers for the new functional requirements (developers can have multiple tasks, and the leader and modeler are also developers). The time period for this version is three weeks, due to the relatively heavy requirements.

Interactive communication requires careful design of the associated activities. The required interactive behavior includes fine nuances. The requirement is for different kinds of notifications to users, following a variety of triggers. The students are directed to consult the *Observer* design pattern and its different versions. Coaches workout with their teams the need to separate the interactivity logic-management from the communication technology. This is particularly important for enabling multiple user interfaces. The requirements for back notification (system pushes notifications to the user) are planned to include a demand for notification that is not database driven, like notification for a user that opens too many simultaneous sessions, or delayed notifications.

The introduction of a GUI interface calls for introduction of an appropriate modeling. We ask for *statecharts* modeling, with emphasis on appropriate level of hierarchy. The challenge here is first, to recall a modeling language that has been taught but not much practiced, but mainly, to use the hierarchy feature of statecharts, so to produce human understandable models that are directly implemented in the software. This point is explicitly checked in the team meeting.

Third version: This version introduces the requirement to add a web-based user interface. The interactive notification requirement is unidirectional for the web-based interface, i.e., there is no requirement for pushing notifications to the web interface, due to the heavy technological load, but GUI notifications can be triggered by web-interface activities. The risk with a new web-based interface is that students create a "duplicated" application, i.e., one for each interface that are inter-connected via the common database. This really happened in previous years, sometimes, with students not being fully aware of the bad architecture. In order to clarify and prevent this harmful duplication, we further introduce requirements for application-based functionalities that do not involve database. In particular, we have requirements for GUI notifications that are triggered by user actions in the web interface, without involving database storage, e.g., proper operation while database connection mal functions, or management of non-persistent information, like temporary, session-based IDs or even passwords. This version is meant to add a few more requirements, to add the new technology, and to check the quality of the software constructed so far, its stability, flexibility and robustness. The new technology requires careful design of its integration.

Fourth version: This version wraps up the development. There is no new technology, and the requirements are meant to check the features of the project. To check flexibility, there is a change of some previous requirements with the request to analyze and document the steps of the change application. To check robustness there are requirements for system load stability, including stability under contradictory user operations.

Overall project summary: The project is a framework where student experience instructed development of a demanding software. It introduces balance between design and technology, abstraction and flexibility of requirements, and enforces construction of a rich application (not a thin database client).

The instruction environment is based on team-individual meetings, where conclusions of group discussions are implemented towards the next version, and enables backwards corrections.

The SE-lab framework enables individual student experience through individual presentation, changing member roles, team administration, and independent study of models and technologies.

Grading: Grading student projects is always challenging, due to the variety of directions that a project can take. In the SE-lab, since the planned content is identical for all teams, we have developed a very detailed *final-evaluation*-scale that considers all aspects of the project, as they are realized in the final version: Functionality, robustness, associated artifacts, validation, and traceability. The final course grade is determined by the *final-evaluation-scale*, the version grades (including the backwards correction options), and the student presentations in the meetings.

In order to reduce variability of team meetings, we prepare for every round of version-meetings, a meeting-plan sheet (beyond the student technical presentation and the version-leader summary) and an evaluation sheet. During the meetings, coaches are expected to follow the meeting-plan as much as possible (deviations are, sometimes, unavoidable). The evaluation-sheet and grades are given (right) after the meetings, so not to spoil the pleasant atmosphere. Therefore, the version-evaluation sheet cannot be as detailed as the final one. Every round of team-meetings is preceded and followed by staff meetings. In the pre-round meeting, we discuss the planed requests, and in the post-round meeting, we discuss, compare and try to balance the staff experience and evaluations.

Comparison with capstone projects: Both courses deal with development of an SE system. However, this is where the comparison ends. While the SE-lab offers continuous coached meetings with an expert in the project subject, capstone projects usually receive spurious coaching by a non-expert mentor. The SE-lab introduces extreme artificially created conditions while capstone projects might not have any demanding conditions of development. The SE-lab introduces a thoughtful software design that might be unnecessary for many capstone projects, e.g., projects with simple, well-defined unchanged architecture, where difficulties lie in complex algorithmic requirements.

4 EVALUATION OF THE SE-LAB COURSE

To evaluate the effectiveness of the software engineering lab and to check whether its intentions and objectives were successfully achieved, we adopt the survey technique and devise two questionnaires. One refers to the pre-lab experience and knowledge of the students before the lab and the other refers to post-lab experience. Both questionnaires address the following aspects: Experiencing *Software Development Processes*, Experiencing *Development in Teams*, Experiencing *Requirements Management*, Experiencing *Software Design*, Experiencing *Software Implementation*, Experiencing *Software Quality*, and Experiencing *Training and Coaching*.

Pre-lab questionnaire: The questionnaire consists of questions related to the above aspects and is aimed at checking the experience and knowledge of the students in order to verify that there is a real need for the lab. We electronically distributed the questionnaire to 97 students of the last year cohort, out of which we received 22 answers. For each statement the students were required to refer to the extent to which each statement holds (from 1- limited extent to 5 – large extent). In the Pre SE-Lab column, Table 2 presents the averages of the students score with respect to the mentioned aspects. The "Tot" column reflects the overall score in that aspect, the "Lrn" column reflects the average scores of the learning in that aspect (i.e., the students have learned new techniques) and the "Prc" reflects the score of the practical experience the students get (i.e., they used already known techniques). It seems that actual experience or application of the taught theory and principles is quite limited (in all cases, beside teamwork, the scores are below 3 out of 5).

In addition to ranking statements, we also asked the students to provide us with additional comments and insights regarding their knowledge and experience before the SE-lab. With respect to Experiencing Software Development Processes, the students mentioned that although being taught in various courses, the actual execution and usage of software processes were minor. As for tools, the students mentioned that training in tools like GIT and Trello was hardly given or practiced. They mainly learned these by themselves. With respect to Experiencing Development in Teams, many students mentioned that in academic settings it is hard to coordinate among the students and that there are free riders. Nevertheless, they claim that when working together the development time is reduced and more ideas and feedbacks are emerging. With respect to Experiencing Requirements Management, the students mentioned that the subject did not get much of attention both in theory and in practice. With respect to Experiencing Software Design, the students acknowledge the subject of design patterns, yet they think that there is a little emphasize on the software design itself. Non-functional properties were also put aside.

With respect to Experiencing Software Implementation, the students claim that not much of focus or feedback were given to coding. Yet, they mentioned the background they got on code smell. With respect to Experiencing Software Quality, the students claim that much focus was given on unit testing but even those were not carefully checked. In their view, testing did not get enough emphasis throughout their studies and that their capability is limited. Nevertheless, they mentioned that they have been exposed to TDD and to integration and acceptance testing. With respect to Experiencing Training and Coaching, in general the students feel that the guidance or feedback regarding the design, coding, and testing is not sufficient.

In summary, as of the status before the SE-lab course, both the quantitative results and the students' comments indicate that further clarification, practicing, and guidance on software engineering is required.

Post-lab questionnaire: We were further interested in verifying that the SE-lab course indeed achieved its goals. Thus, we approached the two last cohorts of the program: The one that took the course during 2016 and just graduate its studies (including the completion of the capstone project), and the other that just completed the SE-lab course (2017). While talking with the 2016 cohort students, it seems that they appreciate the SE-lab course to a large extent. They claim that: "it is a very good preparation for the capstone project", "it is a critical course [in terms of importance]", "it is an important course in which all engineering aspects are integrated", "it provides a good and right pilot to learn from mistakes before the capstone project takes place".

Having the generally good feedbacks, we were interested in looking into the details and electronically distributed a post-lab questionnaire to two cohorts (the one that respond to the pre-lab questionnaire and the cohort before, from which we got the above feedback). The questionnaire includes a detailed set of questions related to the actual practices taken throughout the SE-lab (see the Appendix). With the 2017 cohort, we distributed the questionnaire to 97 students and received 17 responses, whereas with the 2016 cohort we distributed the questionnaire to 61 students are received 13 responses.

Table 2. Survey Results (means and standard deviations in brackets)

Software Engineering Aspects	Pre SE Lab (2017) – in 5 scale	Post SE Lab (2017) – in 7 scale	Post SE Lab (2016) – in 7 scale

	Tot	Lrn	Prc	Tot	Lrn	Prc	Tot	Lrn	Prc
Experiencing Software Development Processes	3.05 (0.69)	4.05 (0.79)	2.71 (0.34)	4.29 (0.88)	5.38 (0.71)	4.14 (0.81)	4.56 (0.88)	5.84 (0.75)	4.38 (0.75)
Experiencing Development in Teams	3.52 (0.47)		3.52 (0.47)	4.76 (0.68)	5.33 (0.57)	4.57 (0.62)	4.80 (0.73)	5.39 (0.39)	4.61 (0.73)
Experiencing Requirements Management	2.76 (0.35)	3.05 (0.39)	2.61 (0.27)	5.27 (0.79)	5.75 (0.25)	5.12 (0.85)	5.13 (1.09)	5.72 (0.44)	4.96 (1.18)
Experiencing Software Design	3.02 (0.56)	3.32 (0.6)	2.71 (0.39)	4.58 (0.74)	4.75 (1.01)	4.53 (0.70)	4.31 (1.23)	4.59 (1.75)	4.23 (1.15)
Experiencing Software Implementation	3.07 (0.36)	3.73 (0.94)	2.94 (0.18)	4.83 (0.56)	4.38 (0.87)	4.91 (0.50)	4.52 (1.34)	4.66 (0.70)	4.49 (1.45)
Experiencing Software Quality	2.87 (0.46)		2.87 (0.46)	4.59 (1.15)	5.06 (0.42)	4.54 (1.21)	4.52 (1.34)	4.74 (0.61)	4.49 (1.54)
Experiencing Training and Coaching	3.09 (0.59)	3.77 (1.27)	2.75 (0.03)	4.83 (0.96)		4.83 (0.96)	4.08 (1.09)		4.08 (1.09)

The two right columns in Table 2 summarize the average score in each of the aspects described before. Here the columns of "Lrn" indicate either importance of aspect, lesson learnt, and understanding of different viewpoints (i.e., of students and instructors). In all aspects, the results of the students of both cohorts indicate an improvement in both learning and practicing (above the mean (4) in the post questionnaire). In general, the students were satisfied with the SE lab course. However, as we asked them to provide ways to improve the course, they provide us with issues that require further attention. When examining in detail the various results for each of the questions as appear in the Appendix, we had the following observations.

With respect to development processes, the overall average score of the two cohorts was 4.21, yet the discussions on the pros and cons of various processes and tools (average score = 3.48) can be improved. With respect to development in teams the overall average score was 4.59, yet, the students' results indicated that the team work was not optimize (average score = 3.2). With respect to design the average score was 4.38, the results indicate that not much of design reviews was performed (average score = 2.7). With respect to implementation, the average score was 4.44, however, the results of the 2016 cohort indicate that not much of code reviews was performed (average score = 2.5). With respect to software quality, the average score was 4.52, yet, it seems that the unit testing was not planned in advance (average score = 3.24).

We next moved to read the student comments in order to get further insights. It should be noted that these are anecdotal as most students did not provide detailed comments. In the following, we refer to these comments divided into the relevant aspects, as discussed before. In brackets we indicate the cohort from which the comments were issued.

With respect to Experiencing Development in Teams, some students indicated that while teamwork is generally positive, it allows free riders, since no tight monitoring of the actual work was applied. Monitoring was mainly via team commit logs, and it might be better to determine the various roles within the team more rigorously (2016).

With respect to Experiencing Software Design, some students indicated that issues of planning and design received lesser emphasis, as the focus was on the code (2017). They commented that it would be beneficial to get more insight into actual usage of design patterns (2017).

With respect to Experiencing Software Quality, students indicated that they felt that too much attention was paid to unit-test coverage (2017).

With respect to Experiencing Training and Coaching, students in both cohorts indicated that group meetups are generally good, although guidance is sometimes limited to checking whether version requirements are met (2016, 2017). Several students suggested reducing the load (2017) in order to provide more in-depth guidance (2016, 2017) and to increase synchronization among course coaches.

5 DISCUSSION AND LESSONS LEARNED

We believe that the SE-lab course achieves its goals largely. In all desired aspects, student responses indicate improvement, in both learning and clarifying SE methods and in practicing these methods. Moreover, the instructors of the yearlong capstone project course that follows the SE-lab note that they observe meaningful improvement of student knowledge and skills.

Some aspects that constantly worry us, as course instructors, refer to *teaching* and *evaluation*. Although team meetings are pre-planned and are associated with clear evaluation criteria, individual personality, character, training, experience and knowledge of coaches still have their own impact. Naturally, coaches tend to emphasize elements of their expertise. Some coaches spend more time on software models, some prefer checking coding or validation aspects, and others have difficulties in conducting a discussion and lean towards an undesirable test atmosphere, following the written meeting-protocol. Clearly, it is impossible to expect that all coaches will be ideal and identical: Knowledgeable, mature, self-confident, open, and friendly. In the post-version staff meetings, we discuss relevant issues and try to reduce variability in version grading. In the future, we aim at having a more detailed meeting plan that precisely defines the content and procedure of team meetings. Yet, we must leave room for creativity, individual differences, and personal variety.

The annoying phenomenon of free riders is still not solved. In order to reduce the number of students with minimal or close to zero contribution, we have conducted the procedure of self-organizing teams, assuming that at their third year of studies, most students already belong to naturally created teams. This procedure was positively accepted by the students, but there

are always teams that are administratively created, or students that are added to existing teams by the SE office. Besides, students differ in their capabilities, devotion, and available time. Efforts to pinpoint the individual contribution of team members sometimes create unpleasant, dense atmosphere in meetings. Some coaches have a softer character, and feel uneasy to embarrass students in public. Such activities spoil our intention to create constructive friendly discussions in team meetings. Consequently, although we encourage coaches to try to locate free riders, we avoid explicit individual member testing.
Considering the student comments, although the general reaction is highly positive, there are several comments that deserve attention. We discuss these following the aspects we followed throughout the paper.
Experiencing Software Development Processes: Some students perceived that their actual engagement into a specific development process was limited.
Experiencing Development in Teams: The importance of teamwork is appreciated by all. Yet, in practice, there were cases in which the work was divided or performed by only few members of the team. This has caused frustration among the students as well as extra workload. We have already discussed the problematics of free riders.
Experiencing Requirements Management: Some students, in particular from the last cohort, indicated that expected version content was not well defined. Our conjecture is that in the last cohort, the course staff and the project theme have changed, and thus version contents were not clearly defined.
Experiencing Software Design: Some students expressed the need for more in-depth focus on discussing design alternatives.
Experiencing Software Implementation: It seems that the notion of code review was not sufficiently practiced, and there is a need to look for ways to integrate that experience into the course.
Experiencing Software Quality: In general, it seems that the issue of software quality turns to be addressed very well and that the students absorb the importance of a quality software.

6 RELATED WORK

Many attempts have been done in order to increase the professionalism and practice of SE graduate students. These attempts usually refer to project based learning of various types including the capstone projects.
For example, Chatley and Field [8] acknowledge the gap between academia and the required skills by the industry. They indicated the there is a need for much more hands-on experience and for balancing between teaching, learning, and assessment. To address those needs they suggest adopting a lean learning, which means learn by doing, small and frequent assessments, automation, and support by software engineering practitioners. Their main claim is that small and frequent feedback cycles increase the learning outcome. Although, we acknowledge and adopt the short feedback cycles, small assignments as we discussed in the introduction, might not be sufficient to capture SE challenges.
Recently, the notion of studios has been introduced [7] [9]. Studios mainly refer to physical location in which students can work and collaborate. Although studios further motivate students to focus on their projects. It seems that only limited guidance can be provided to the students, as they are aimed at self-directed and self-motivated learning [10]. In [9] for example, the studios are organized similarly to a capstone projects. Furthermore, the SE principles according to which the studios were designed are not explicated.
Barzilay at al [11], indicate that an advanced summary course in software that synthesizes all aspect of SE is still missing. In their work they propose a course with attempts to address the fundamentals of SE, practices and tools, productization, and technology evolution. Nevertheless, it seems that the course refers only to a sub-set of the development lifecycle and less emphasizes the abstractions required is SE.

7 CONCLUSION

We described an SE-lab course that fulfills the standard role of an engineering lab course: Offer lab experiments for practicing methods and techniques, in a planned environment that enables integration and investigation, with team-based guidance. We analyzed the ideals, principles and goals of the SE-lab, and explained how they are realized in the lab content. The SE-lab evaluation shows that students consider it to be a highly relevant core course in the program.
We pointed to several aspects that deserve further attention, mainly with relation to balanced team instruction and to student grading. Another challenge involves continuing the SE-lab ideal and principles in the capstone project. This goal requires coordination with the capstone project instructors, and mainly further efforts during the SE-lab teaching, towards embedding these software development values.

ACKNOWLEDGMENTS

We thank Achiya Elyasaf and Avi Hayoun for their help in managing the SE-lab and for their valuable insights.

REFERENCES

1. P. Bourque, R. E. (Dick) Fairley, SWEBOK 3.0, 2014.
2. IEEE, Systems and software engineering — Vocabulary, INTERNATIONAL STANDARD ISO/IEC/IEEE 24765, 2017.
3. ACM, Software Engineering 2014 Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering, http://www.acm.org/binaries/content/assets/education/se2014.pdf
4. A. Mishra and D. Mishra, "Industry Oriented Advanced Software Engineering Education," Croatian Journal of Education, vol. 14, pp. 595-624, 2012.
5. L. D., Feisel, and A. J. Rosa, "The Role of the Laboratory in Undergraduate Engineering Education," Journal of Engineering Education, vol. 94, pp. 121–130. 2005.
6. J. A. Macias, "Enhancing Project-Based Learning in Software Engineering Lab Teaching Through an E-Portfolio Approach," in IEEE Transactions on Education, vol. 55, no. 4, pp. 502-507, Nov. 2012.
7. A. J. Lattanze, "Practice Based Studio," *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, Dallas, TX, 2016, pp. 1-7.

8. R. Chatley and T. Field, "Lean Learning - Applying Lean Techniques to Improve Software Engineering Education," *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, Buenos Aires, 2017, pp. 117-126.

9. J. Lee, G. Kotonya, J. Whittle and C. Bull, "Software Design Studio: A Practical Example," *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, 2015, pp. 389-397.

10. S. R. Haynes and D. R. Mudgett, "A design studio course in application development: Lessons learned," *2016 IEEE Frontiers in Education Conference (FIE)*, Erie, PA, USA, 2016, pp. 1-8.

11. O. Barzilay, O. Hazzan and A. Yehudai, "A Multidimensional Software Engineering Course," in *IEEE Transactions on Education*, vol. 52, no. 3, pp. 413-424, Aug. 2009.

APPENDIX

Pre SE Lab Questions

Experiencing Software Development Processes	Expose to principles of quality modeling (e.g., design patterns)
Learned about software development processes	Implement principles of quality modeling
Implemented software development processes	**Experiencing Software Implementation**
Integrated approaches and tools in software development processes	Consider you implantation to include non-functional properties
Analyze the pros and cons of tools in the context of software development	Decide upon the technology to implement the software
Experiencing Development in Teams	Implement the software based of the preliminary modeling
Develop software in team larger than two students	The implementation fitted the preliminary modeling
The work is team was a challenge	Expose to principles of code quality (e.g., design patterns, bad smell)
The work in team supported the software development	Implement principles of code quality
Experiencing Requirements Management	**Experiencing Software Quality**
Learn about requirements management	Focus on testing development for the software you implemented
Learn about development of a software that is partially defined	Track the requirements vs. the code and the testing
Develop a software that is partially defined	Check the software flexibility for introducing new requirements
Pay attention to the testability of the requirements	Check the models you developed
Prioritize, decompose, and estimate the requirements	Execute the testing on the code you developed
Use tool for requirements management	Execute the testing for non-functional requirements
Experiencing Software Design	Develop the testing in parallel to the code
Learn about design considerations in software development	**Experiencing Training and Coaching**
Implement design considerations in software development	Guide on the system you develop
Learn about design considerations originated from non-functional requirements	Guidance in software development is important
Implement design considerations originated from non-functional requirements	The actual guidance has helped you

Post SE Lab Questions

Experiencing Software Development Processes	Experiencing Requirement Management
We examined various development processes	We were not required for requirement analysis
The development process was defined in advance	The project requirements were clear
We fully followed the chosen development process	We analyzed the requirements
We partially followed the chosen development process	We prioritize the requirements
We did not follow the chosen development process	We took care of the requirements testability
We integrated various methods for software development	The requirements were only used to define the project
We examined the pros and cons of the various techniques	We manage the requirement throughout the entire project
We examined the pros and cons of the various tools	We used various tool to manage the requirements
We did not examine the pros and cons of the various techniques	Requirements traceability is important when these are changing
We did not examine the pros and cons of the various tools	There is no need to trace the requirements as these are changing frequently
The importance of the development process was not clear	We manage the relationship between the requirements and design
It is important to have a development process	We manage the relationship between the requirements and code
The development process was clear	Experiencing Software Design
We adequately followed the development process	We used design consideration during the software development
Experiencing Development in Teams	The design considerations consist of non-functional requirements

We did not succeed to coordinate among the team members	Modeling was the major means for specifying the design
There were team members with limited contribution	Design patterns played an important role during the design
We succeed to coordinate among the team members to a large extent	The design was performed before coding
The teamwork helped us to fast advancing with the development	Code organization is the design
The role division was effective	The code was aligned with the design throughout the entire project
The role division did not support the development	We discussed design alternatives
All team members equally contribute to the development	The project design was known at the beginning
We learn new things when working in a team	From the view point of the students, the design is the major part of the project
The teamwork improved my communication skills	From the view point of the instructors, the design is the major part of the project
The teamwork facilitated discussion on alternative solutions	We performed design reviews
There was a need for a leader	We followed software engineering principles throughout the project design
The teamwork supported the due dates deliverable	Experiencing Software Quality
Experiencing Software Implementation	Unit testing were written before coding
We were required to select a technology for implementing the project	Unit testing were written along with the coding
The code we had took into consideration the non-functional requirements	Unit testing were written after the coding
The code followed the models we had	We had a high testing coverage
The code followed the design we had	We had an exhaustive testing coverage
Design patterns were considered throughout the coding	We had a low testing coverage
Bad smell were avoided throughout the coding	We had integration testing
The code was reviewed by another team member	We had system testing
The coding was done after determining coding rules	We had automated testing
From the view point of the students, the code is the major part of the project	The testing were executed after each change automatically
From the view point of the instructors, the code is the major part of the project	The testing architecture was discuss in details
We performed code reviews	The testing checked the system functionality with respect to the requirements
We followed software engineering principles throughout the coding	The testing checked non-functional requirements
The code was exhaustively documented	The testing checked flexibility to changes
Experiencing Training and Coaching	The testing checked reusability
The joint lectures discuss the various tools for executing the project	The testing checked the user interface
The joint lectures discuss problems raise during the project	From the view point of the students, the testing is the major part of the project
The joint lectures contribute to the project	From the view point of the instructors, the testing is the major part of the project
The group meetups were used as an examination of the project	
The group meetups were used as a focused forum for consulting and guidance	
In the group meetups we focused on understanding the requirements	
In the group meetups we focused on testing	
In the group meetups we focused on coding and problem solving	
In the group meetups we focused on a discussion on alternatives for design and implementation	
In the group meetups we focused on the project management	
The group meetups contribute to the project	