Scalable Teaching of Software Engineering Theory and Practice: An Experience Report

Solal Pirelli, *EPFL*

ABSTRACT

We report on our experience and lessons learned from teaching the theory and practice of software engineering to hundreds of undergraduate students majoring primarily in computer science. These students know how to *write code* but not *engineer software*. In particular, the teaching load appears to scale well to hundreds of students despite offering open-ended exams in an interactive theory course that focuses on solving concrete problems. We teach theory and practice sequentially, to give students time to apply what they learn, which also enables us to iterate on the theory course quickly based on results from the practice course.

1. Introduction

Software products are more than the sum of their code modules, thus students need to learn more than how to write code [10, 17]. Even for students who do not intend to become software engineers, the skill of engineering maintainable software only becomes more relevant as more fields depend on software.

While students learn some software engineering basics on their own in other courses, such as basic version control during team-based course projects, course-sized software is unlikely to trigger the problems that good software engineering practices deal with, thus students are unlikely to learn these practices on their own.

We report on our experience teaching a theory-practice pair of courses, in back-to-back semesters, to undergraduate students in computer science and related majors. We taught these courses for the past 6 years. We describe the context further in §2.

The theory course, *Software Engineering*, teaches one subject per week with a focus on solving concrete problems rather than learning facts. The practice course, *Software Development Project*, is based on the Scrum development methodology. Students take on the role of development team and share the product owner role with two staff members who act as their coaches. Projects proceed in two-week sprints and include a meeting with coaches in the middle of each sprint to ensure students do not fall behind. Students are easily motivated for their project as they own it from start to finish, but in early iterations of the theory course we found it all too easy to present facts that felt disconnected from students' "real world" of course homework and projects. We settled on interactive lectures based on concrete problems that keep students interested. We describe course contents in §3.

Because the courses' latest iterations had around 180 and 120 students respectively, numbers that grow year after year, one key concern is to scale grading while remaining valid, reliable, and fair. On one end of the spectrum, asking students to write essays does not scale. On the other end, automatically grading students' code with unit tests cannot test their understanding of software design, let alone give actionable feedback. For the theory course, our exams have short free-text questions, which can be quickly graded manually, and programming questions whose correctness can be graded automatically and whose design can be quickly graded manually. For the practice course, we use frequent grid-based reviews of students' output, ensuring the grades reflect students' performance over the entire project rather than their final output. We describe grading further in §4.

The courses changed significantly in the 6 years we cover here. The course pair structure led to a refinement of the theory course based on our observations in the practice one, such as covering asynchrony more deeply as we realized that students knew the theory behind parallelism yet did not know how to write maintainable asynchronous code. Even the split in two was an evolution as they began as one course covering both theory and practice concurrently.

While previous work found that students enjoyed active learning less even as they learned more [8], our experience does not match this: as we switched to more interactive lectures, the course evaluations improved, students explicitly brought up the interactive elements as a positive in course evaluations, and their projects improved despite more demanding project criteria. We describe how the courses evolved based on our observations and on student feedback in §5.

We learned valuable lessons for both course design and lower-level implementation details. A recurring theme is that processes should be lightweight to not burden students and staff but still have concrete constraints. Fully agile processes that focus only on results are not a good fit for students first learning how to engineer products. When and how to give feedback to students without unreasonably burdening the staff was crucial for both courses. The split into theory and practice courses is a success, though they could be two halves of a bigger course. We could not fix all problems to our satisfaction, especially around the choice of technology and its consequences in the project. We share the lessons we learned and remaining challenges in §6.

Our course material, including lecture notes and past exams, is publicly available at *github.com/sweng-epfl/public*.

ICSE-SEET '24, April 14–20, 2024, Lisbon, Portugal
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0498-7/24/04
https://doi.org/10.1145/3639474.3640053

2. Background

Learning the theory and the practice of specific subjects, such as object-oriented programming or compilers, is necessary but not sufficient to develop quality software in the real world. One way to do this is through a "capstone"-style project in which students learn by doing. However, a project alone will not teach skills that students are not even aware of. For instance, debugging code in a productive way is hard for novices [21], and thus it is unlikely that most students would figure out a good debugging method on their own within the timeframe of a project course. This is especially true in the context of a university, where students may have relied mainly on TAs for help and not learned how to look for answers on websites such as Stack Overflow, even though this is a crucial part of modern debugging [16].

The need for a theory-only course to teach the principles of software development has been recognized before [11, 19]. Such a course can establish solid foundations for a project afterwards, such as teaching the DevOps skills necessary for the technical side of project management [2]. However, a project-less course can appear disconnected from the real world, with artificially constructed theoretical examples, and thus motivate students less than a project-based course.

Teaching a theory-practice pair of courses is a natural consequence of the benefits and risks of a theory-only course. Students can first learn the core skills they need for a project that they do not already know. This involves general skills such as debugging, DevOps techniques, and testing. Students may also need to be taught concrete ways to apply subjects they have seen as individual modules in previous courses, such as parallelism. After the theory, students can then spend time in the practice course learning skills that need longer-term practice, such as how to properly divide tasks in a project [20].

Another benefit to teaching theory and practice as separate courses is that the practice enables instructors to evaluate the theory. Properly evaluating a course in software development is an open problem, since the impact of this sort of course is felt in the long term by design [6]. By teaching a project shortly after a theoretical course, instructors get faster feedback in terms of what clearly does or does not work. For instance, if students do not effectively use continuous integration in their project, then it is unlikely the DevOps theory was effective.

The theory course must teach concrete ways to develop software, which is naturally less theoretical than most theory courses, but must remain focused on principles. For instance, the original Design Patterns book [9] includes 23 design patterns. Teaching all of these in a course is not realistic. Instead, students should learn the concept of a design pattern, and some well-chosen examples they are likely to use in their project.

Scaling up to hundreds of students is a key challenge, just as it is for other courses. Smaller courses can use strategies such as hiring professional coaches [22], which helps provide a good experience to students but is expensive to scale to the level of a mandatory course in a large university, even without considering the challenge of finding a large amount of qualified coaches. Past work has noted that even ~200 students is already large [24].

Grading is a core part of the scalability challenge since the amount of course staff is typically limited. If each exam takes one hour to grade, then grading hundreds of exams is outside of the ability of a reasonably sized course staff. Options include peer grading [3] or even self-grading [26], but these raise hard questions in terms of reliability and potential to collude or cheat. One way to shorten grading time is to make exams automatically gradable, which can work to find semantic errors [18] but not to grade code design or theory. Multiple-choice questions provide a coarse-grained way to test theoretical understanding, but can both fail students who made minor mistakes, and help students who would not know where to start without a set of possible answers. Large Language Models are a promising direction [23], but are not currently reliable enough for production use.

We report on a theory-practice pair of courses we taught for six years, mainly to computer science undergraduates in the middle of their degree. The last edition of our theory course had ~180 students, while the practice course had ~120. The difference comes from the former being required for more majors. Both are worth 4 credit units out of the 30 that students take per semester. This corresponds to ~7h of work per week.

Our university has over 10,000 students, about 20% of which are in the school of computer science, which includes related fields such as data science and cybersecurity. Enrollment has been growing rapidly, especially in computer science: our class size has roughly doubled in the last 6 years. Most students attend classes in person, but after the COVID pandemic the number of students who watch recordings instead has increased to around 40% in our courses. Courses at our university do not, in general, use attendance as a grading criterion, but sometimes use quizzes and midterms that require students to be physically present at specific times.

Compared to other universities in the region, our university has a more theoretical focus, with courses designed to teach the underlying fundamentals and rarely any specific technology. This anecdotally leads to issues for both recent graduates and employers, who appreciate the theoretical depth but wish there was less of a need to learn basic practical skills on the job.

Most students have taken a set of prerequisites such as an introduction to object-oriented programming and to functional programming, CPU architecture, as well as the theory of parallelism. However, some students from other majors, as well as exchange students, have not taken these and may or may not have taken equivalent courses.

Since our university does not offer a software engineering major, our courses are the main way for students to explicitly make the jump from writing code for course projects and homework assignments to engineering real software. While students have typically worked in teams during course projects before, our courses are the first ones in which they are exposed to development methodologies, mainly Scrum.

Our teaching workload is equivalent to around 1 full-time position for the theory course and 1.5 full-time positions for the practice course. This takes the form of two co-lecturers and ten teaching assistants, mostly undergrads who did well in previous editions of the courses.

3. Contents of the courses

Our overall objective is to teach students to develop real-world software from start to finish. *Software Engineering*, the theory course, focuses on recognizing common needs; knowing which techniques can help and why simpler techniques don't work; designing, testing, and implementing programs; and being able to constructively criticize software written by others. *Software Development Project*, the practice course, teaches students how to work in a team, including dividing tasks, planning, testing, and demoing their work. Students develop Android apps from scratch in teams of 6, coached by a pair of TAs, using the Scrum methodology. Teams choose what app they want to build and can use any Android-based technology they want. Overall, our goal is to shift students' mindset from *writing code* to *engineering software products*, keeping in mind the limited time we have and the number of students we handle.

In terms of methods, we focus on modern teaching practices, in particular interactivity. In the theory course, this translates to exercises interspersed with lectures. The lecturer never speaks for more than 15 consecutive minutes, after which students do practical or theoretical exercises for 5–15 minutes, the lecturer interactively reviews the solutions with students, and the cycle begins anew. We also emphasize frequent repetition: there are three exams rather than one with increasing weights in the final grade, and lectures that do not follow an exam begin with a quiz on the previous lecture's contents.

We want students to remember high-level solutions, even if they do not remember the exact details. Each section in a lecture is a question, such as "how can we reuse concepts across software systems?" to introduce design patterns and "what makes code debuggable?" to introduce defensive programming, invariants, and logging. Students can look up details when they need them, but only if they know there is a solution in the first place. For instance, a student who has never heard of defensive programming is likely to either slowly reinvent the wheel or be led astray by solutions to the wrong problem, such as how to navigate code faster in their favorite debugger.

We present the theory schedule in Table 1, with lecture objectives. Each lecture is 120 minutes, plus breaks and some time for a quiz at the beginning, or an exam review for lectures after an exam. The lecture on Requirements is only two-thirds the length of the others, since one third is used to motivate the course, which works well because the exercises for Requirements are shorter as they are more theoretical. Infrastructure is the most applied lecture, as it uses Git exercises to illustrate version control. The Testing lecture includes dependency injection, which is key to testing Android apps. Debugging is mainly not about the act of debugging itself, but about writing readable and debuggable code to facilitate debugging. Design is a practical look at using modularity and abstraction, illustrated with design patterns to decouple an app's UI and business logic. Performance is a high-level view focused on metrics and system design, not on micro-optimization. Evolution teaches how to read existing code due to the necessity of doing so in the real world [25], as well as documentation and versioning.

Mobile Platforms introduces the differences between traditional platforms and mobile ones using Android as an illustration, such as app lifecycles and efficiency concerns due to battery use. The Asynchrony lecture was born from our observation that students could not apply in practice the parallelism concepts they had seen in theory, and thus focuses on the Future abstraction [5] and its use in design, implementation, and testing. Teamwork is about the concepts students will need in their project, especially Agile and code reviews. Security comes after the final exam due to logistics: our lecture slot is on Friday, and thus we need time to grade the final before the winter break.

We chose not to teach the specifics of UML or any modeling tool. Instead, we present the concept of discussing design in a team through a shared modeling language, with UML as an example. This is in line with our focus on high-level solutions over tools.

We present the project schedule in Table 2, which is simpler as most of the course is a set of Scrum "sprints". The first week consists of a "bootcamp" for individual students, during which they write a basic Android app while also forming teams. Once students have picked teams, the second week consists of a team "bootcamp" that includes common tasks most teams need regardless of their app idea, such as using a database. The first two weeks also serve as time to pick a project idea. We require that apps must use some online service, have a concept of users, an offline mode, and use at least one phone sensor. This ensures all teams encounter the same core challenges. We explicitly forbid teams from writing their own backend, as this is too much of a time sink given the low time budget of the course.

At the beginning of each sprint, teams meet with the coaches for a demo of their app in its current state, a retrospective on the previous sprint, and planning for the upcoming sprint. Teams are asked to pick items from their backlog and create a list of tasks assigned to team members before the meeting, but coaches can help if desired and override decisions if necessary. Teams also meet with their coaches in the middle of each sprint, to ensure all is going well, though the need for this meeting can be waived at the coaches' discretion.

In addition to writing code, which must always come with tests, students must review each other's code. This comes in the form of code reviews on pull requests, with mandatory checks that at least one person has signed off on a pull request before it can be merged. Coaches occasionally perform code reviews as well, mainly to show what a good code review looks like.

Each sprint, students must write a short summary of the past sprint focused on what went right or wrong in terms of process. For instance, students may note that they started their work too late, or underestimated the time merging multiple branches into their main branch would take. Summaries force students to self-reflect and avoid making the same mistakes again. It also gives coaches a written record so they can proactively confront teams that seem to be about to repeat their mistakes.

At the end of the course, teams submit a video demo of their app, all of which are made available to the entire class. The staff then selects which teams should present live to the entire class. There is no exam, instead students are graded individually each sprint and graded as a team at regular intervals.

#	Title	Objectives
1	Introduction Requirements	Motivate the course Formalize and use requirements. including internationalization, accessibility, and ethics
2	Infrastructure	Contrast 1st, 2nd, and 3rd generation version control systems, use version control and continuous integration, communicate effectively in commit messages
3	Testing	Understand what and when to test, evaluate tests with code coverage, adapt code to enable fine-grained testing
4	Debugging	Develop readable and debuggable code, isolate the root cause of bugs, use a debugger to better understand and debug code
5	*Exam #1*	
6	Design	Apply modularity and abstraction in practice, compare ways to handle failures, reuse concepts with design patterns, decouple business logic and user interface code
7	Performance	Compare performance metrics and scales, create appropriate benchmarks, profile code to find bottlenecks, choose adequate algorithms and designs for performance
8	Evolution	Find one's way in a legacy codebase, safely evolve such a codebase with refactorings, document and quantify changes, establish and use solid foundations with versioning
9	*Exam #2*	
10	Mobile Platforms	Contrast traditional and mobile platforms, understand new metrics such as power use, know the basics of mobile apps such as lifecycles, user permissions, and app stores
11	Asynchrony	Understand asynchrony in practice, build maintainable asynchronous code with Futures, write tests for asynchronous code, and design asynchronous software components
12	Teamwork	Contrast development methodologies including Scrum and Waterfall, apply agile principles in practice, divide tasks in a team, write useful code reviews
13	*Exam #3*	
14	Security	Design and use threat models, analyze the trusted computing base of a piece of software, design secure software components, know the root causes of common vulnerabilities

Table 1. Week-by-week schedule of the theory course, in the first semester.

#	Work	Description
1	Individual bootcamp	Create a "Hello, World!" Android app per student, including tests and CI
2	Team bootcamp	Create a standard app per team, with authentication, maps, API calls, basic UI, and caching
3-13	Sprints	2-week sprints with weekly meetings, students mostly in control of the product backlog
14	App showcase	Each team demoes their app to the rest of the class

Table 2. Week-by-week schedule of the project course, in the second semester.

4. Grading

Course grades must be valid, reliable, and fair [1, 14]. *Validity* is the degree to which evaluations are trustworthy, i.e., accurately reflect students' mastery of the material. *Reliability* is the degree to which the same answer is assessed consistently, regardless of who writes the answer an, who grades it, when it is graded, and other such environmental factors. *Fairness*, which can be viewed as a part of validity and reliability, is the absence of bias. Overall, students must know in advance the objectives they must achieve, know what graders are looking for in an answer, and not worry that factors outside of their control may harm their grade.

Scalability is another key objective in a large classroom: the course staff must be able to grade each exam in reasonable time, without having to compromise the other objectives. For instance, a multiple-choice question set with a single correct answer per question is very scalable, but compromises validity since minor mistakes or misunderstandings can lead to large losses of points, while students who would not be able to answer a question if it was free-form may be able to guess the correct choice.

We grade how students fare as software engineers, for a short period of time in the context of the theory course's exams, and for a longer period in the project course. In particular, exam statements are written in the context of a team with customers, and all grading criteria are defined in terms of the consequences of good software engineering practices. For instance, a question involving adding a cache for Web requests in an existing system does not mention design patterns, and its grading criteria do not include the use of any specific design patterns, but rather we ask and grade that the cache should be self-contained and modular.

Our exams, like our project, are open book. Exams are a sort of very short sprint in which students must answer questions from hypothetical teammates or customers and implement tasks they have been assigned. The former are theoretical questions, and the latter are practical questions. We give one example of each type as Figure 1 and Figure 2 respectively. Importantly, each question is short enough that the time spent reading is small compared to the time spent thinking about and writing answers and ends with the grading criteria and available points. Answers to theoretical questions are always required to be 1–2 sentences, ensuring students know how to concisely express their thoughts and lowering the time spent grading each answer. Answers to practical questions involve both high-level design decisions to be graded manually, and implementation code that can be graded by a tool such as automated tests or code coverage.

Project grades consist of two equally weighted components: the individual performance of each student, graded each sprint, and the overall deliverable of each team, graded each third of the course. Both grades consist of multiple criteria graded on a 5-level scale: *Excellent*, *Good*, *OK*, *Poor*, *Very Poor*. We provide the full grading grid to students, which includes expectations of each level for each criteria, and summarize them here as Table 3 and Table 4. In exceptional cases, when the individual and app grades differ significantly, the staff may adjust students' final grades. This adjustment is unfortunately subjective, but we only have to use it for a handful of clear-cut cases in practice.

Users report that your app freezes when they open its image gallery, which shows images in a scrollable grid. This is the function run to display the gallery:

```
void openImageGallery() {
    List<Image> images = getImages();
    displayImages(images);
    initializeButtons();
}
```

In one sentence, explain why the app is freezing:
...

Users complain your image gallery uses too much mobile data. In one sentence, propose the first step towards improving this:
...

For each sub-question, you will receive up to 5 points for a concise answer based on good engineering practices.

Figure 1. Example of a theoretical exam question.

Your colleague wrote a command-line client to list users on your company's platform but fell ill before they could write tests. Your task is to write a test suite for this client.

Make minimal modifications to `Client.java` to make it testable: (1) it should not have hardcoded dependencies on its environment; and (2) the new implementation should have the same behavior as before.
Write tests for the modified client in `ClientTests.java`.
You do not need to worry about running the app, since your company's API is not implemented yet.

You get 15 points if you make minimal and clean modifications to the code such that its behavior does not change. You get 20 more points if you provide useful and maintainable unit tests that fully cover the branches of the client class.

Figure 2. Example of a practical exam question.

Rubric	"Excellent" grade requirement
Planning	Large increment, fitting with the sprint backlog, with time for review and merging
Code	Maintainable, robust, and documented code at the levels of both functions and modules
Tests	Tests for all or almost all cases
Reviews	Thorough code reviews that consider design and likely future evolution of the codebase

Table 3. Individual grading criteria in the project.

Grading the individual planning component is the most difficult of the bunch, due to the subjective nature of what is a "large" increment in the context of a project. Depending on the project, the existing codebase, and the technology involved, whether a given piece of work is enough to merit an *Excellent* grade can be hard to decide. Similarly, ensuring students take on enough work at the beginning of a sprint is a difficult problem. In practice, an end-to-end scenario that provides some value to users is usually enough to get an *Excellent* grade in planning.

The theory course also includes graded quizzes at the start of most lectures, which together count for 5% of students' grades. This is intended to motivate students to attend the quizzes and thus practice their learning and find gaps in their understanding before exams.

To mitigate the impact of any single event on a student's grade, such as feeling sick on the day of an exam, we have three exams rather than one in the theory course, we drop the worst quiz grade, and we drop the worst sprint grade in the project. Furthermore, the first app grade in the project only counts for 10% of the overall app grade, and mainly serves to "wake up" teams whose code is not up to par.

To improve reliability and fairness, all grading is done in pairs. Each exercise in an exam is entirely graded by the same pair of TAs, who must agree on each grade. Project coaches are in pairs and must also agree on each grade. This also helps with knowledge transfer between junior and senior TAs.

Scalability is baked into our grading design. We must grade an entire exam in at most a week, which in practice means an afternoon parallelized across pairs of staff members. We also must grade each team's sprint in the project course quickly so that the team can know how to adjust for the next sprint.

Having a rubric is a necessity, as taking decisions on a case-by-case basis takes too much time. For exams, we start from the public criteria given in each question, and graders create a rubric per criteria as they grade. We have found that having 3, or at most 4, criteria per question is a sweet spot for fine-grained grading that scales. Tools such as Ans.app and Gradescope help make this practical. For the project, thanks to the detailed grid of criteria and levels per criteria, the main time sink is reading each students' code in the pull requests they made that sprint.

We also pre-filter answers to practical questions to remove answers that do not pass basic smoke tests we give to students along with the statements. This ensures graders do not lose time manually evaluating the design of code that does not work at all. This is similar to a real-world code review: a software engineer proposing code changes that do not even pass smoke tests will likely be met with frustration that they did not run these tests before asking others to review the code.

We found overall that factors that help scalability also help with reliability and validity in general. For instance, requiring TAs to not spend time nitpicking the exact naming of variables not only saves time but also leads to more valid grades. Just as a software engineer would not be happy if their colleagues decided to fight every variable name, a student will be disappointed to get a poor grade because of small disagreements if their code is otherwise well designed and passes all tests.

Rubric	"Excellent" grade requirement
Functionality	The app provides clear value to users and fits within the Android ecosystem
Resilience	The app is resilient to failures, user error, and malice, with corresponding tests
Maintainability	The code is modular, clean, and documented. Another team could take over productively.

Table 4. App grading criteria in the project.

In terms of time, it takes a pair of TAs 3 to 4 hours to grade 200 answers to an exam question, which includes not grading the answers that fail smoke tests. Thus 10 TAs can grade an entire exam in one afternoon. For the project, it takes around 2 hours for one coach to grade one team, and 30 minutes for the other coach to double-check this grading, thus it takes 5 hours for a pair of coaches to grade 4 teams.

Discouraging and detecting cheating is a challenge any course faces, but the open book nature of our grading makes this easier. Plagiarism and unauthorized collaboration are the only forms of cheating we must look out for. The former can be done with automated tools as well as paying attention for obvious style changes during grading. We allow students to copy code if they indicate the source and the code is under a license that permits such copying, just as a real-world project would. The latter is done during exams by TAs who look at students' laptops to ensure they are not emailing or chatting with each other, and during projects by looking out for red flags and by discussing students' code with them during the weekly meetings to ensure they understand what they are doing in general.

Tools based on Large Language Models, like ChatGPT, are a new and interesting challenge which we had to consider in the latest edition of the course. We decided to allow students to use ChatGPT as long as they indicated they had done so, in the same way they would give credit to a human helping them. Our exams often naturally lead ChatGPT astray as it behaves similarly to a very confused student, though the latest GPT-4 model is less bad. Typically, ChatGPT provides lengthy answers that try to cover every possible angle without actually taking a stand, which is what some students do even without ChatGPT. These answers get few, if any, points, since we do not grade based on keywords. Minimizing the length of statements, which we do to ensure reading is not a bottleneck for students, also helps confuse ChatGPT, possibly because it provides less context. For code-based theoretical questions, ChatGPT often provides a long code snippet as an answer even if the question is about design and should not be answered with code.

One interesting fact we noticed when testing our questions with ChatGPT and various prompts related to software engineering is that the quality of training data varies widely across subjects. Scrum in particular leads to dubious answers such as answering that "the principle of transparency in Scrum" means the daily standup really is the place for in-depth technical discussions, a question we intended to be easily answered in the negative.

5. Evolving the courses

Our courses today are drastically different from six years ago. First, they used to be a single course, worth fewer credit units than the combined amount they are currently worth. Second, the theory was focused on facts, not concrete problems. Third, the project was focused more on the process than on the outcomes. We report on these three points in detail, on changes that did not pan out and that we reverted, and on the impact of the changes we made.

The changes we made are based primarily on feedback from students, which comes from many channels. The most direct one is a survey we run after each theory exam, whose anonymous completion is worth 2% of the exam points and thus has a high response rate. We also get direct feedback in course evaluations that students fill through the university portal, though these lack incentives and thus have a response rate closer to 50%. Student questions during lectures also give us feedback on what concepts need better explanations. Indirectly, we also obtain feedback from TAs answering questions about exercises, grading exams, and coaching projects. The latter in particular gives us concrete and larger-scale feedback on whether students are able to apply in practice the concepts we taught in the theory course.

We split theory and project into separate courses, the biggest change in terms of logistics and also the most successful one. Part of the reason for this change was specific to the study plan of computer science majors at our university: splitting the course in two allowed us to increase the workload since it is now split over two semesters. But the main reason behind the split is that there are too many theoretical subjects the project depends on, and teaching these concurrently is unrealistic. Students who first learn about requirements, or applied modularity, or testing, or essentially any of our theoretical subjects realize their existing code has structural flaws that will lead to problems down the line. Students naturally want to rewrite parts of their code to remove these flaws, but this is hard to balance with the need to add features to have a useful app at the end of the course.

With two separate courses, we can teach subjects at the depth we believe to be necessary, in an order we believe makes sense, instead of constraining ourselves based on what students need in the first weeks of their project. Students can focus on the exercises in the theory course and the project in the practice course, instead of sacrificing exercise time for project time and thus implementing concepts they are not sure they understand.

We shifted from memorizing facts to solving problems. While the concepts underlying our courses have not changed, we used to teach specific facts without much motivation behind them, and to evaluate students' knowledge of these facts. This also led to spending too much time on details, such as policies for naming variables and classes since they are easy to describe and easy to grade. Furthermore, details do not lend themselves to interactivity, since exercise answers typically repeat what was said in a lecture and are thus not useful within a lecture. We now see a much deeper understanding of key concepts in the project course. For instance, students used to see continuous integration as an opaque process, but now know why it rejects their code.

The shift toward problem-solving helped with the validity of theoretical exam questions and the scalability of practical ones. Grading facts in theoretical questions is trivial to grade when using multiple-choice questions, but minor misunderstandings cause students to get zero points for an answer that was almost right in their head. Grading open-ended theoretical questions is slower, but our policy of requiring concise answers and our use of tools make it feasible, as we discussed above. While validity was not an issue in practical questions before, scalability was. Grading every potential low-level issue takes time. Grading a student's overall approach is faster. This does not mean we no longer consider low-level details, only that we group them into a general "maintainable code" rubric for which a student can get full marks even with a small number of minor mistakes, just as a code reviewer in a real project might approve a pull request even if they have a few nitpicks.

We noticed that students enforce low-level policies in the project by themselves, such as insisting on naming, formatting, and choice of data structures in code reviews. In fact, one of the main issues we notice in students' code reviews is the same one we eliminated in our course: too much focus on low-level facts at the expense of a high-level view. Coaches must sometimes point out design flaws in code that students merged after a "thorough" code review by a student pointed out every minor naming issue.

We shifted the project focus from process to outcomes, with earlier and stricter feedback on outcomes. The goals we want students to achieve have not fundamentally changed, but we now communicate them better and intervene earlier in case of problems. Due to the limited time budget of our course, we do not have the time for students to discover long-term issues on their own without sacrificing the quality of their project. Grades used to surprise students, even with regular coach feedback.

We now enforce stricter constraints throughout the project, and explicitly grade each sprint as it happens instead of handing out grades for the entire project at once. Constraints include a level of code coverage below which students can no longer merge pull requests, mandatory continuous integration, overriding students' backlog order if their app does not yet meet course requirements such as an offline mode, and reducing the definition of done for tasks that take more time than expected. Students still learn from their failures, but the impact of these failures is now contained to one sprint or two at most instead of the entire project.

Since we made expectations explicit, we can now grade based on outcomes we ask for rather than compliance with specific steps. The fact these grades are frequent and binding ensures students follow coaches' advice. For instance, it used to be common for students to not merge their code because they did not think it was ready. This was despite reminders from their coaches that iterating with a smaller feature that fully works is better than a larger feature that does not. Teams would thus come to their end-of-sprint meeting with little to demo since no new feature worked end-to-end, defeating the purpose of the Scrum method. While real-world customers would complain, students do not perceive coaches as customers, and thus took their complaints as advice. Now that these complaints are in the form of a grade, students realize they must correct course.

Since we drop the worst sprint grade when computing the final grade, coaches can reassure students the first time this happens that, as long as the students learn from their failure, it will not impact their grade. This also helps coaches assign a "poor" or "very poor" grade without feeling bad about it.
We also decreased the overhead of the Scrum process by moving to 2-week sprints instead of 1-week ones. This enables students to take on bigger tasks in a sprint, and to pivot to smaller tasks if they realize in the middle of the sprint that they will not be able to complete the entire task they originally planned. We still kept weekly meetings due to our past experience with teams that had failed "too much" in a single week already given what they can quickly recover from, but the mid-sprint meeting is now merely a short standup meeting. This enables coaches to intervene if they need to, without taking time from teams that work well since they would anyway organize standup meetings.
Not all our changes were satisfactory. We tried adding guest lectures, proposing projects with existing customers, and peer feedback in the project, but these did not pan out.
We intended guest lectures to show students how real-world software development works, but found they were not worth the time investment and typically not well attended by students. Our time budget means any guest lecture sacrifices one subject we want to teach, and thus should have high value. Students did not care much for the guest lectures, and some even perceived it as the staff not bothering to do their teaching job. This may partly be due to us not investing enough time in helping guest lecturers design their lecture.
We asked a local company and some student associations such as a music festival to propose project ideas and act as joint product owners with the coaches, but the resulting projects were hard to grade and usually did not motivate students more than if they had chosen their projects. This failure may come from our "half-attempt" at it, since most teams were still working on their own idea, and thus the sponsored projects were more of an add-on than a focus of the course. Previous work describes successful attempts at capstone-style projects [7, 15].
We asked students to give feedback on their teammates in the project, but received muted reactions. Despite our assurances that this would not impact their teammates' grades, students were not keen to discuss problems openly, except in extreme cases. Students typically report such extreme cases to their coaches anyway, so the peer feedback did not help. Since one cannot realistically argue that the coaches will both read the peer feedback and not be influenced by it at all, perhaps feedback available only to teammates and course lecturers but not to coaches would help.
The changes we made were broadly well-received, both in terms of student evaluations and project outcomes. The theory course's grade in university-mandated evaluations went up from being mostly "good" to "very good" on a 4-point scale. Students still complain that the project course takes more time than its allotted credits imply, but they overall feel that the courses are worth it and also produce better apps.

6. Lessons learned

Having described our courses, the way we grade, and how the courses evolved, we now share **the lessons we learned** in the past six years and **what could be improved next**.
Theory before practice is a good idea. In our context of a computer science major in which students do not spend time on concrete aspects of software development, teaching the theory and practice in parallel does not work well. Projects expand to fill whatever time the students have available, and thus students sacrifice the theory side. There are also too many subjects one needs to cover to establish solid foundations before starting a project. Covering these subjects in parallel with projects is too late to be useful in the short term.
A bigger course split in two, instead of separate courses, may work better even with the same overall amount of time, since students could spend more time per week on the project and better amortize fixed time costs such as standup meetings. However, repetition would be less spaced over time.
Concrete problems help motivate theory. We noticed this in both our courses: students appreciate that we start from problems and explain how to solve them, including examples of these problems. When students are in the middle of making a mistake in their project, sharing an anecdote about real-world failures or even a failure in a past edition of the course works much better than appealing to theoretical concepts.
TAs should maintain a database of concrete anecdotes, both from their own experience and from real case studies, to have good examples to give to students.
Problem-solving exams can scale as long as the grading rubric is focused and the graders trained to maintain this focus. Multiple-choice questions are not a necessity for theory, and unit tests are only one part of a scalable strategy for practice. This requires writing exams with scalable grading in mind.
The staff must be trained to focus when grading and avoid the temptation to spend time on overly detailed feedback. Such feedback is well-intentioned but leads to cutting corners once graders realize they do not have enough time to grade all submissions. Giving concrete expectations to graders in terms of time and rubric details seems to work well.
Graded quizzes are not worth the stress on both students and staff. Quiz grades have two constraints: they must be worth enough to merit the time the staff spends preparing them, but they must not be worth too much since they are small. These constraints are not satisfiable together. New quiz questions must be written every course iteration if they are graded, and the staff must spend time grading them and answering regrade requests. This is too much of a time commitment given the low ceiling on how much the quizzes can reasonably weigh in the final grade.
Quizzes are useful but should not be graded. They give short-term feedback on students' understanding to both students and lecturers. Not grading them would alleviate student stress, not require staff time every year, and allow peer instruction [12, 13] when correcting answers in class. At most, one could grade whether students attempted the quizzes, to incentivize students to participate in them.

Project coaches must be proactive, much more than they may reasonably expect. In the short time frame of a semester, letting a student make the same mistake twice in a row and telling them that this time they really must do it right is too late. Forcing a merge of a smaller feature that the team wishes to expand first, for instance, makes coaches feel overly imposing but is required to avoid teams failing too much. Similarly, coaches must grade according to the criteria and levels they are given, regardless of subjective criteria such as feeling the team put a lot of effort, to avoid teams getting different grades due to different coaches.

Project coaches should receive formal training, not only oral discussion during staff meetings or emails. Just as concrete problems help students learn software development, giving coaches concrete examples of common project problems would help them make correct and uniform calls. Coaches should also be given estimates of the time they need to spend grading, just like graders for exams.

Code reviews work well as peer instruction in a project, with the added benefit that they resemble real-world practice. The main challenge is to ensure students' reviews are thorough and uncover design issues rather than superficial syntax nitpicks. The lack of an objective way to assess a code review makes this more difficult: it is possible, though unlikely for students in a course, that a piece of code is so flawless that no review could point out real issues. One way to address this is to insist on the knowledge transfer aspect of code reviews, which is a priority for software engineers [4]. Requiring students to ask questions so that they truly understand why the author of the code chose a particular design and implementation yields inquisitive reviews, without making students feel that they are attacking the author. Asking students to explicitly leave *positive* comments praising the parts they find particularly well-made and explaining why also helps.

Teams should be able to request a *parallel* code review of some of their pull requests from coaches every sprint. Coaches reviewing pull request *instead* of students deprives students of an opportunity to practice. On the other hand, reviewing code *after* students have already done so can be seen as grading the student review rather than helping. Coaches should perform in-depth reviews at the request of teams but require the team to release a review at the same time, so that the two reviews do not influence each other. This does not require synchronization other than agreeing on a time to release a scheduled email.

Students often default to Waterfall when uncertain, for instance making long-term plans about the final state of their app even before they have tried the technologies available to them on Android. The existence of previous course projects in the curriculum that have well-defined deliverables using well-known technologies does not help. Defining only a backlog of tasks per sprint does not seem to be enough.

Teams should be asked for concrete short-term plans as a compromise, instead of requiring only a description of their app at the beginning of the semester and a backlog of tasks at every sprint. The first plan could be a "minimum viable product" description of what the app should look like after a sprint, with an adjusted plan for the next sprint after each sprint.

Grading open-ended projects is hard but worth it. Even with well-defined criteria and levels, coaches do not always find it easy. This is not surprising, but it should not distract from the fact that open-ended projects are worth it from a motivational standpoint. An even older edition of the course, prior to the work reported in this paper, instead asked all teams to develop the same application. From what we can tell, this was a worse experience for students. While students today did not experience this older version of the course, they did experience projects with well-defined criteria, as mentioned above. Thus, in course evaluations, they often explicitly mention the open-ended factor as a positive.

Project requirements should be more specific, requiring features instead of abstract concepts such as "user support". The requirements we have work reasonably well, but some teams still spend time reinventing the wheel, such as implementing their own authentication system. Forcing students to use a specific system would defeat the point of the open-ended and problem-solving nature of our courses but asking them to have a realistic set of features may help. Requiring email verification, password recovery, and two-factor authentication would dissuade all teams from writing their own authentication system while highlighting the benefits of code reuse.

The time team meetings take is bimodal, especially at the beginning of the project. Good teams quickly find their pace and finish meetings in barely more time than it takes to demo an app and review the sprint plan. Even when coaches ask for an oral retrospective, good teams have thought about it and are able to answer quickly and concisely. On the other hand, teams that need more time from the coaches require *much* more. This time investment is worth it from a teaching perspective, but hard to square with fixed time slots per team.

Projects should have office hours and shorter meetings, instead of doing everything in a single meeting. A short meeting slot, with assigned coaches, could be used for the demo and a retrospective. A much longer slot, with whichever staff members are available, would help teams that need more help.

Tool reliability shapes students' experience. The list of features in tools such as continuous integration services, device emulators, and development environments matters less than the reliability of these tools. Continuous integration is the biggest culprit in our project course, and Android is unfortunately not as good a platform as it should be in this regard. Students often run into tests that only fail in continuous integration. Even with a recorded video of the emulator during testing, a feature Android only recently added officially, finding a root cause is hard. Thus, students frequently report spending most of a sprint trying to debug continuous integration issues, which demotivates them. Students finish the course wondering if continuous integration is really worth it given the pain, defeating a course objective.

Another form of reliability is documentation and obsoletions. The recommended way to write Android apps changes almost every year, but the documentation does not always follow, and students find plenty of resources on the Web that are obsolete despite being relatively recent. Students are less likely to search for solutions on their own if they often find outdated results.

We have no concrete idea on what could be improved for this final lesson about tooling reliability. Ideally, we would select the course technology based on tooling support, so that students' first foray into real-world development is at least supported by solid tools. However, many possible candidates are unavailable for reasons beyond our control. Web-based solutions require knowledge of JavaScript that our students do not have, and that cannot be learned at the same time as a time-constrained project. iOS requires developers to use Apple machines, which is not a reasonable ask from students. Desktop applications such as using JavaFX would not be as motivating, especially since students could not develop apps based on real-time locations, which are a popular project basis.

7. Conclusion

We described our experience teaching a theory-practice pair of software engineering courses over the last 6 years.

We successfully scaled to almost 200 students in the theory course and 120 in the practice one using the equivalent of 1 and 1.5 full-time teaching loads respectively, while increasing the interactivity of the courses and the quality of the grading. The theory course in particular could scale further since its material is now complete and thus needs less attention.

While previous work suggested interactive courses may be less well-received by students than traditional ones [8], we did not observe this. Students mostly gave top evaluations to our theory course. In our internal course survey, 63% of students approved having exercises done during lectures while only 14% of students disagreed.

8. Data availability

The course contents, including lecture notes, exams, and meta-level documentation for lecturers, are publicly available at *github.com/sweng-epfl/public*. The exact feedback from students, such as the course evaluations, cannot be shared publicly.

References

[1] Allen, J. and Lambating, J. 2001. Validity and Reliability in Assessment and Grading: Perspectives of Preservice and Inservice Teachers and Teacher Education Professors. (Apr. 2001).

[2] Alves, I. and Rocha, C. 2021. Qualifying Software Engineers Undergraduates in DevOps - Challenges of Introducing Technical and Non-technical Concepts in a Project-oriented Course. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2021), 144–153.

[3] Aniche, M., Mulder, F. and Hermans, F. 2021. Grading 600+ Students: A Case Study on Peer and Self Grading. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2021), 211–220.

[4] Bacchelli, A. and Bird, C. 2013. Expectations, Outcomes, and Challenges of Modern Code Review. *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA, 2013), 712–721.

[5] Baker, H.C. and Hewitt, C. 1977. The Incremental Garbage Collection of Processes. *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages* (New York, NY, USA, 1977), 55–59.

[6] Breaux, T.D. and Moritz, J. 2023. A Metric for Measuring Software Engineering Post-Graduate Outcomes. *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2023), 283–295.

[7] Bütt, E., Person, S. and Bohn, C. 2022. Student-Sponsored Projects in a Capstone Course: Reflections and Lessons Learned. *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2022), 254–264.

[8] Deslauriers, L., McCarty, L.S., Miller, K., Callaghan, K. and Kestin, G. 2019. Measuring actual learning versus feeling of learning in response to being actively engaged in the classroom. *Proceedings of the National Academy of Sciences*. 116, 39 (2019), 19251–19257. DOI:https://doi.org/10.1073/pnas.1821936116.

[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc.

[10] Garousi, V., Giray, G., Tuzun, E., Catal, C. and Felderer, M. 2020. Closing the Gap Between Software Engineering Education and Industrial Needs. *IEEE Software*. 37, 2 (2020), 68–77. DOI:https://doi.org/10.1109/MS.2018.2880823.

[11] Gestwicki, P. 2018. Design and Evaluation of an Undergraduate Course on Software Development Practices. *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2018), 221–226.

[12] Gopal, B. and Cooper, S. 2022. Peer Instruction in Online Software Testing and Continuous Integration - A Replication Study. *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2022), 199–204.

[13] Gopal, B. and Cooper, S. 2021. Peer Instruction in Software Engineering - Findings from Fine-Grained Clicker Data. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2021), 115–121.

[14] Gordon, M.E. and Fay, C.H. 2010. The Effects of Grading and Teaching Practices on Students' Perceptions of Grading Fairness. *College Teaching*. 58, 3 (2010), 93–98.

[15] Gorka, S., Miller, J.R. and Howe, B.J. 2007. Developing Realistic Capstone Projects in Conjunction with Industry. *Proceedings of the 8th ACM SIGITE Conference on Information Technology Education* (New York, NY, USA, 2007), 27–32.

[16] Li, A., Endres, M. and Weimer, W. 2022. Debugging with Stack Overflow: Web Search Behavior in Novice and Expert Programmers. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Software Engineering Education and Training* (New York, NY, USA, 2022), 69–81.

[17] Li, Z., Arony, N., Devathasan, K. and Damian, D. 2023. "Software is the easy part of Software Engineering" - Lessons and Experiences from A Large-Scale, Multi-Team Capstone Course. *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (Los Alamitos, CA, USA, May 2023), 223–234.

[18] Liu, X., Wang, S., Wang, P. and Wu, D. 2019. Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics. *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2019), 126–137.

[19] McBurney, P.W. and Murphy, C. 2021. Experience of Teaching a Course on Software Engineering Principles Without a Project. *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education* (New York, NY, USA, 2021), 122–128.

[20] Meulen, A. van der and Aivaloglou, E. 2021. Who Does What? Work Division and Allocation Strategies of Computer Science Student Teams. *International Conference on Software Engineering.* (May 2021). DOI:https://doi.org/10.1109/icse-seet52601.2021.00037.

[21] Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L. and Zander, C. 2008. Debugging: The Good, the Bad, and the Quirky – a Qualitative Analysis of Novices' Strategies. *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (New York, NY, USA, 2008), 163–167.

[22] Paasivaara, M. 2021. Teaching the Scrum Master Role using Professional Agile Coaches and Communities of Practice. *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2021), 30–39.

[23] Pinto, G., Cardoso-Pereira, I., Monteiro, D., Lucena, D., Souza, A. and Gama, K. 2023. Large Language Models for Education: Grading Open-Ended Questions Using ChatGPT. *Proceedings of the XXXVII Brazilian Symposium on Software Engineering* (New York, NY, USA, 2023), 293–302.

[24] Porquet-Lupine, J. and Brigham, M. 2023. Evaluating Group Work in (Too) Large CS Classes with (Too) Few Resources: An Experience Report. *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1* (New York, NY, USA, 2023), 4–10.

[25] Ryan, B., Soria, A.M., Dreef, K. and van der Hoek, A. 2022. Reading to Write Code: An Experience Report of a Reverse Engineering and Modeling Course. *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)* (2022), 223–234.

[26] Walser, T.M. 2009. An Action Research Study of Student Self-Assessment in Higher Education. *Innovative Higher Education.* 34, 5 (Dec. 2009), 299–306. DOI:https://doi.org/10.1007/s10755-009-9116-1.