Applying Lean Learning to Software Engineering Education

Robert Chatley

Abstract In this chapter, we describe the ways that we have applied lean and agile techniques to teaching software engineering at Imperial College London. We give details of the structure and evolution of our programme, which is centred on the tools, techniques and issues that feature in the everyday life of a professional software developer working in a modern team. We also show how aligning our teaching methods with the principles of lean software delivery has enabled us to provide sustained high-quality learning experiences. We examine two different types of course in detail: first, a 'traditional' lecture course, where we transformed the way that course is taught and assessed, aiming to create tighter feedback loops, and second a project-based course where we ask students to put agile methods into practice themselves, working in teams to build a substantial software system over a number of months. We describe concretely how we run and structure these courses to set up effective learning experiences.

Keywords Software engineering · University · Automation · Feedback
Project-based learning · Peer-instruction

1 Introduction

Lean and agile methods are prevalent in industrial software engineering today (Papatheocharous & Andreou, 2014). Scrum, Kanban and eXtreme Programming (XP) are all common in software development organisations, helping teams to develop software iteratively, in reliable and predictable ways, whilst responding to changing requirements in a fast moving world. In university Computer Science departments, we are training the next generation of software engineers, and it is therefore important that we teach these methods to prepare students for their future working lives.

R. Chatley (✉)
Department of Computing, Imperial College London, London, UK
e-mail: rbc@imperial.ac.uk

D. Parsons and K. MacCallum (eds.), *Agile and Lean Concepts for Teaching and Learning*, https://doi.org/10.1007/978-981-13-2751-3_14

Many universities and other higher education institutions are striving to bring modern industrial software development techniques into the classroom, and like any such institution, Imperial College London has been faced with the challenge of updating and evolving its software engineering education to prepare its students for modern industrial careers. Keeping pace with rapid changes in industrial practice has required changes in the way software engineering is taught. This includes teaching modern development methods and giving students hands-on experience of putting those methods into action through practical work (Anslow & Maurer, 2015; Kropp & Meier, 2014). This evolution has not been easy but, through continuous experimentation and iterative improvement, we believe that we have evolved a software engineering programme that strikes a good balance between teaching, learning, and assessment.

Given that we are teaching lean and agile methods, and believe that they have positive effects on software engineering practice, it seems natural that we use them to inform our teaching practice too. If we are looking to reduce waste, and to improve quality and feedback in our educational systems, can we apply the principles and practices that we teach to the teaching itself?

Although our own courses are focussed on software engineering, we believe that many of the lessons we have learned are transferable to other disciplines.

1.1 Perspectives on Teaching

In order to discuss the approaches that we have tried, we will borrow some vocabulary from Mark Guzdial's 2015 book 'Learner-Centered Design of Computing Education' (Guzdial, 2015). Guzdial gives us three useful terms to describe different types of learning experience. The first is *transmission*, which describes the classic lecture situation. An expert holds a body of knowledge and tries to transmit it to a—hopefully—attentive audience. This is typically a one-way interaction between one teacher and many learners.

The second perspective is *apprenticeship*, which we use to describe a learning experience focussed on the development of skills rather than theoretical knowledge, most likely through kinaesthetic learning and practical exercises. You can imagine this in a setting like a cookery class, where each student can practice a recipe repeatedly until they have mastered a dish.

The third perspective is *developmental*, which describes a personalised learning experience without a set curriculum. It focuses on taking the learner from where they are to somewhere more advanced, in a particular direction depending on their strengths and weaknesses. This sort of individual tuition works well in a situation like a piano lesson, but it is hard to replicate it with a lecture class of 150 students.

Unfortunately, we do not have the resources in our university to offer individual tuition and personally tailored programmes for every student taking Computer Science, perhaps as a student at a music conservatoire might experience. However, we will discuss how we have tried to blend these three approaches in order to improve on a style of teaching purely based on weeks of transmission followed by final exams.

1.2 The Rest of This Chapter

In the remainder of this chapter, we will illustrate how we have transformed two different types of courses to increase the value of the learning experience, and to incorporate more frequent, high-quality feedback. We move from courses primarily based on *transmission* to courses that focus on the development of skills through an *apprenticeship* model, and also incorporate individual and small group tuition from instructors and peers, moving towards more *developmental* education. In Sect. 2, we look at the evolution of a traditional lecture course, and in Sect. 3 we describe how we support different types of project-based learning. Section 4 discusses possible challenges for future adoption of similar techniques in more courses and at larger scales. We also give some qualitative feedback taken from our student survey, which is conducted across all students, anonymously, at the end of each term of study.

2 Lecture Courses

Within Imperial's Computing curriculum, we have a second-year undergraduate module called *Software Engineering Design*. The content of this module concerns methods, tools, and techniques for the development and deployment of large-scale software systems that are robust, well-engineered and easy to maintain by design. In an earlier incarnation of the course, the material concentrated on notation, formal specification languages and catalogues of *design patterns* (Gamma, Helm, Johnson, & Vlissides, 1995). This meant that students would learn a range of ways to document and communicate software designs, but these were not tied to a particular implementation language. Much of the material was thus taught 'in the abstract' and the students did not get much opportunity to put their theoretical knowledge into practice. The following comment in our student survey typified concerns that this was not the best approach:

> Would have preferred design patterns to be practiced more in lab exercises, … the patterns I understood best were the ones for which I wrote and tested actual code…

We wanted to find a way to move the focus from learning theoretical knowledge to applying and demonstrating practical skills. We hoped that this would not only improve the students' experience of the class, but also provide them with a more valuable learning experience.

Historically, teaching in this class was based largely on *transmission*. Students attended lectures twice per week throughout the autumn, took other modules during

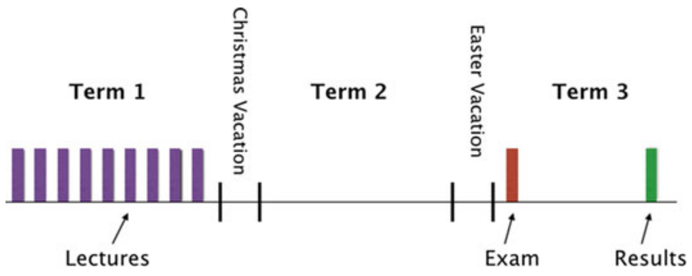


Fig. 1 The structure of a traditional lecture course at Imperial, with lectures over the autumn term, and examinations in the summer. Figure © 2017 IEEE. Reprinted, with permission, from proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)

the spring, and then had their examinations after Easter (see Fig. 1). There were tutorial classes alongside the lectures, usually with paper-based exercises, but typically only the most diligent students kept up with the exercises week by week and most left them to use as revision aids come exam time.

This approach is completely at odds with the typical delivery cycle of a modern industrial software project. The feedback cycle is very long, and a large amount of work is in process before we get to the 'quality assurance' stage. Only when we get the exam results do we really know whether we have taught the students effectively. We can think of the course as starting out with a long list of requirements for things that students should learn—a syllabus—and that we then go into a phase of transmission, after which we check the results. There is no iteration or incremental delivery—it is one big batch.

In modern software development projects, we typically strive to reduce batch size, with the aim of decreasing cycle time, decreasing risk, and increasing quality. One mechanism by which we might do this would be to employ Kanban, a lean method that focusses on flow through a system, and by using it we can aim to maximise throughput and minimise cycle time (Anderson, 2010). In software development, we want to minimise the time between someone having an idea for a feature and prioritising it, and that feature being working software in the hands of the users. In learning and teaching, we instead want to minimise the time from introducing an idea, to having a student internalise it, to verifying that their understanding is correct.

One of the tools of a Kanban practitioner is to visualise the workflow. In a software project, this is typically done with a physical or virtual 'card wall', divided into columns for the different phases that each piece of work needs to go through. The different columns map the *value stream* (Rother & Shook, 2003). Typically the board is divided into columns representing the 'backlog' of upcoming tasks, those that are in analysis, those in development, those being tested, and those ready for release, or released. Cards representing separate tasks are moved from column to column as work on them progresses. Similar boards are also often used in other agile methods such as Scrum and XP, but where in those methods the board is an information radiator to help to display the current state of the team's work, within a regular

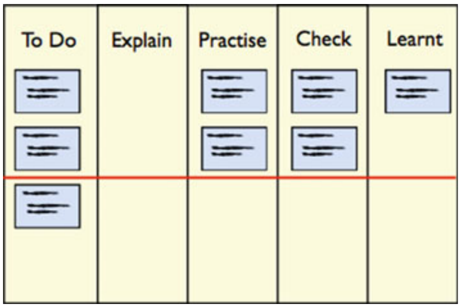


Fig. 2 The value stream of learning, which maps the different phases of learning. Figure © 2017 IEEE. Reprinted, with permission, from proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)

delivery cadence (e.g. a 2-week iteration), in Kanban the board is used as a tool to define and optimise the flow of work through the system. The key idea is to use the current state of the work to decide what to do next, and always to 'pull from the right', so that we concentrate on getting individual pieces of work finished before starting new ones (Ottinger, 2015). This way we focus on completion and keep the work in process low. A limit can be placed on the number of pieces of work that may appear in any column at once in order to enforce this focus on finishing.

We take this idea and redraw the columns on the board to form a value stream of learning. Here we list the items on the syllabus as our backlog—'to do'—and then have columns for 'explain' (transmission), 'practice' (apprenticeship), 'check' and 'learnt' (see Fig. 2). If we follow the 'pull from the right' mantra, then we want to get each item over to the right-hand side as quickly as possible. That means that we aim to do a minimal amount of transmission on each topic before the students get to practice in a hands-on exercise, and then verify the quality of their learning, obtaining feedback before we move further on in the syllabus.

Putting this into practice, we first tried the common approach of adding a small project as coursework part way through the term. However, as it took a couple of weeks to complete the project, and about the same again to get all the assignments marked up and graded, it was pretty much the end of the course before the students got their feedback. There was a wide variation in how students chose to approach the design project we gave them. Those who were more dedicated and had understood well tried out a lot of different ideas and added many features. Those who had not understood well did much less or did the wrong thing. If anything, rather than making sure that everyone had learnt the material, it seemed that we had widened the gap between the stronger students and the weaker ones. We needed something better.

2.1 *Reducing Cycle Time*

In order to give more guidance, and earlier feedback, we changed from asking students to design a whole system to asking them to consider individual design choices in different situations, and examining how implementing something one way or another

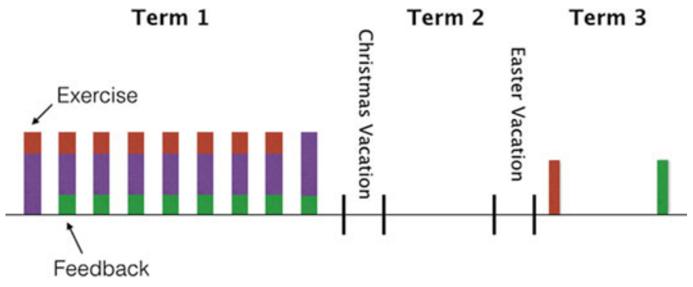


Fig. 3 Revised course structure, with a weekly cycle of assessment and feedback. Figure © 2017 IEEE. Reprinted, with permission, from proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)

would affect the future maintenance of the system. In terms of the assignments that were set, we moved from 'design a system with the following requirements, discuss the design choices you made', to a set of weekly smaller coding exercises of the form 'Add feature X to this system by using design pattern Y. Now try design pattern Z. What are the trade-offs?'. By carefully constructing a number of small scenarios to work through one-by-one, we ensured that each student had the same design issues to think about, and by making them into coding examples students got a much more hands-on, kinaesthetic learning experience.

Fitting with the weekly nature of the university timetable, this led to a weekly cycle of assessment (see Fig. 3). A new topic is addressed each week with an associated assignment, and students submit their solution later the same week. Grades and feedback are then returned within three or four working days, i.e. before they submit their next assignment.

The obvious problem with weekly assignments is the volume of grading and feedback required. Because of the limited teaching resources that institutions generally have to work with, the temptation is to reduce the frequency of assignments, e.g. to once every 2 weeks, in order to be able to deliver feedback 'at scale'. However, this is at odds with what we are trying to achieve. Relating this again to the conditions that apply in a software development project, often we strive to release software more frequently, but integrating and testing new code requires a lot of time and effort. XP promotes adopting a process of *continuous integration* (Duvall, Matyas & Glover, 2011), through which we tend to find that doing these things more often causes us to streamline processes, remove waste, and often apply automation. As Martin Fowler often says 'if it hurts, do it more often' (Fowler, 2011).

2.2 *Peer Coaching*

One change that made a big difference to both student learning and marking load was to encourage students to pair-program, which has been shown to be highly effective in a classroom environment (Williams & Upchurch, 2001). We have found that students enjoy the experience of working with a colleague—a class survey showed that from 148 students, 119 declared that they found learning to through pair-programming to be a good experience, 18 were neutral, and just 9 stated that they preferred to work individually. Students get to practice pair-programming, which is an industry-relevant skill, but not something that necessarily comes naturally to everyone; becoming good at it is difficult and requires work. The students get to coach each other and help each other to learn and understand. By engaging them in pair-programming, we had effectively set up a network of peer coaches—a *developmental* learning style personalised to each individual. Although we are aware of studies that show that constructing pairs by matching weak and strong students perhaps produces more learning, in this case, we allowed them to work with whomever they liked as we wanted to smooth the path to adoption. We may experiment with pre-selected pairs in future. Last, a major benefit in terms of giving weekly feedback on assignments was that pair-programming reduced the number of submissions from 150 to 75!

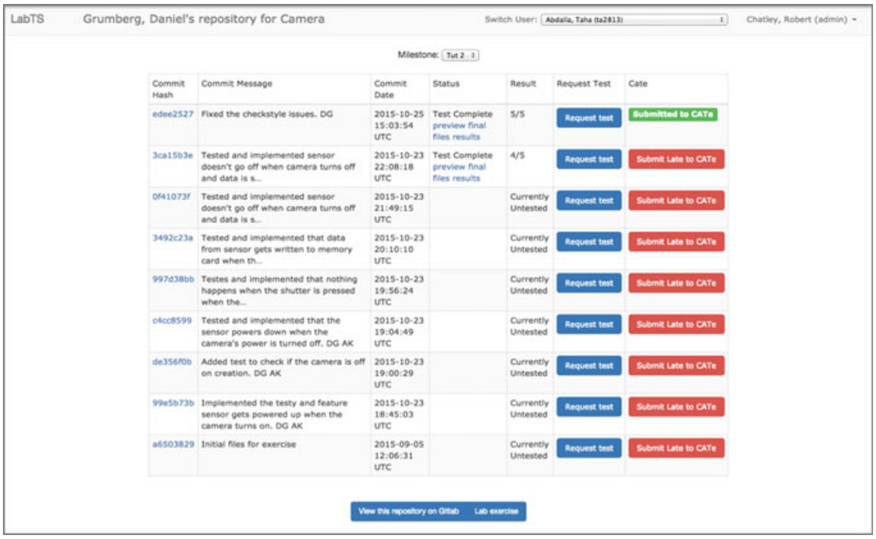
2.3 Automation

A key to reducing the burden of assessment and feedback has been to add automation where possible. This ties back to the engineering practices of XP where automation is used to allow testing of software to be done quickly, repeatably and reliably. Our approach here has been to provide tools that enable students to test their exercise solutions as they work, to detect basic problems early and allow students to fix them before they submit their work. From the first week of their first year, students learn to use version control through Git and GitLab.[1] When they start a coding exercise they clone a repository to obtain skeleton files that form a starting point and are encouraged to work in small steps, committing each change as they go. When they submit their work for assessment, what they actually submit is a Git commit hash corresponding to the version to be marked. We have also implemented a software testing tool the 'Lab Test System' (LabTS), which allows students to view and test each version of their code themselves (see Fig. 4).

For first-year courses, we provide a (partial) test suite that students can run against their code, to check the correctness of their solutions. However, later, when learning about software design, we do not want students to follow the same approach. Providing a set test suite has a consequence of defining an API that the students need to implement. In our design exercises, we want them to design their own API as part of the exercise, and to write their own automated tests against that API. Writing automated tests and utilising test-driven development is a key skill that we want to instil at this stage of the students' education.

As a mechanism to encourage students to write their own tests, we use LabTS to check a test-coverage metric, with a coverage threshold that we deem appropriate

[1]https://about.gitlab.com/.



LabTS Grumberg, Daniel's repository for Camera

Switch User: Abdalla, Taha (ta2813) Chatley, Robert (admin)

Milestone: Tut 2

Commit Hash	Commit Message	Commit Date	Status	Result	Request Test	Cate
edee2527	Fixed the checkstyle issues. DG	2015-10-25 15:03:54 UTC	Test Complete preview final files results	5/5	Request test	Submitted to CATe
3ca15b3e	Tested and implemented sensor doesn't go off when camera turns off and data is s...	2015-10-23 22:08:18 UTC	Test Complete preview final files results	4/5	Request test	Submit Late to CATe
0f41073f	Tested and implemented sensor doesn't go off when camera turns off and data is s...	2015-10-23 21:49:15 UTC		Currently Untested	Request test	Submit Late to CATe
3492c23a	Tested and implemented that data from sensor gets written to memory card when th...	2015-10-23 20:10:10 UTC		Currently Untested	Request test	Submit Late to CATe
997d38bb	Testes and implemented that nothing happens when the shutter is pressed when the...	2015-10-23 19:56:24 UTC		Currently Untested	Request test	Submit Late to CATe
c4cc8599	Tested and implemented that the sensor powers down when the camera's power is turned off. DG AK	2015-10-23 19:04:49 UTC		Currently Untested	Request test	Submit Late to CATe
de356f0b	Added test to check if the camera is off on creation. DG AK	2015-10-23 19:00:29 UTC		Currently Untested	Request test	Submit Late to CATe
99e5b73b	Implemented the testy and feature sensor gets powered up when the camera turns on. DG AK	2015-10-23 18:45:03 UTC		Currently Untested	Request test	Submit Late to CATe
a6503829	Initial files for exercise	2015-09-05 12:06:31 UTC		Currently Untested	Request test	Submit Late to CATe

Fig. 4 Screenshot of our LabTS system—a web-based tool that allows students to run automated tests on each iteration of their exercise solutions

for that week's exercise. LabTS gives each submission a score out of 3: 1 point if the code compiles, 1 point if all the tests the students have written pass, and 1 point if these tests meet the code coverage threshold and the code passes some basic layout and formatting checks. The exercises for our second-year design course are in Java, so we use a Gradle[2] build to choreograph the compilation, testing and other checks. We configure Gradle plugins to check code formatting against a given style guide, and to measure test-coverage. LabTS is then set up to run this Gradle build against each submission and report the results. Usually, if a LabTS test run does not score 3/3, it is relatively easy for the student to see what they need to do to make up the remainder of the marks. We put a policy in place for the class that if a solution does not score 3/3 on LabTS then a human marker does not need to look at it.

With this system in place, we cannot yet dispense with the human markers, but they can give more nuanced feedback on issues of design, and should not have to pick up on basic points about compilation, style, or test-coverage. Even the simple application of checks on code layout and style mean that by the time a person looks at the code, it is laid out in a way that is easy to read. This makes the most of the marker's time by allowing them to focus on more subtle design issues, and not to waste time commenting on things that can be detected automatically. For our cohort of 150 students, working in pairs, with a team of 5 or 6 markers, we can mark and give feedback with about 2 hours of effort per marker per week, which allows us to sustain weekly feedback throughout the course.

[2]https://gradle.org.

2.4 *Summary*

Changing the delivery format of this course from knowledge acquisition through transmission in lectures, to a focus on skills development through apprenticeship and practical exercises, together with the developmental support of peer coaching through pair-programming seems to have been a great success. To enable consistent progress and feedback through weekly exercises we had to solve the problem of scaling our feedback mechanisms, and have used automation techniques, as well as paring back the exercises to really focus on the core message, to make this manageable. The concrete nature of the exercises results in students feeling that their coding skills as well as their design skills are improved by completing them. They also appreciate getting weekly feedback on their work. The following comments from recent student surveys are quite typical:

> A well-structured and engaging course, which I could immediately benefit from as it helped improve the quality of my code and Java knowledge.
>
> I liked that I had to submit the tutorials every week, otherwise I would not have done them.

The weekly cycle of assessment and feedback now works really well. The small batch size and short turnaround time means that students are motivated to do the weekly assignments and this gets them to practice and to improve. Although we have not been able to automate marking completely—this seems like a grand challenge—we have found that a team of five people can now complete the feedback for the entire class in around 2 hours each week.

3 Project-Based Courses

Another key feature of Imperial's Computing curriculum is team-based practical projects. Team working is an essential component of any software engineering programme and is a key skill that many employers look for when hiring graduates. Modern software development methods focus on teamwork and collaboration for the development of software, and we feel that the best way for students to learn these is to experience them practically in project-based courses. Again, we aim to focus on *apprenticeship* and *developmental* learning, allowing students to learn for themselves and from each other through solving practical problems.

In our programme, students get experience of working in small groups from as early as the first year, but we increase the structure and process around the management of project work, along with the scale of the projects they tackle, as they progress through their education. First-year projects are left fairly freeform, with small groups and fairly short timelines, for the students to manage, however, they see fit. After that, we begin to introduce more structure as their projects grow, to allow them to experience something closer to an industrial development environment. Rather than concentrating on transmitting them the relevant theory, we focus on creating a learning experience where the structure in which projects are assessed naturally promotes an agile way of working.

3.1 Second Year—Web Application Development Projects

By the end of their second year of study, computing students should have learned the skills and knowledge needed to build a complete application. At the end of the summer term (the end of the academic year), we give them the opportunity to exercise these skills in a group project, and through this introduce some elements of agile development methods (such as Scrum or XP). The aims of these second-year group projects are to…

- explore user focussed design and development
- experience and practice an agile method in a small project
- apply software development tools and techniques
- develop team work

These projects are done in groups of four students, and run full-time over a period of 4 weeks, which we structure as four 1-week iterations. Each project team develops a web or mobile application of their own design to solve a problem that they have identified. They make the product decisions; they do not have an external customer. To emphasise an iterative approach we run the projects with the following structure:

- 1-week iterations, with groups required to demo their software every Friday.
- Demos are assessed in a lightweight way focussing on: product increment (have they delivered anything this week?) and user research (have they gathered feedback from their target users?).

Rather than marking the project entirely at the end, a proportion of the marks is available each week, so that sustained, iterative progress is rewarded—a big bang release at the end is not. We also want to steer the students towards quantitative evaluation of their own work through meaningful experiments with users. The Friday demos allow groups to demonstrate both their newly developed features and the results of that week's user trials in a short informal meeting. Each group gets 5–10 min to meet with tutors in the computer lab and showcase their latest work. We do not require a big final project report, just a few lightweight deliverables to document the purpose of the application, the overall technical architecture, and the use of appropriate development techniques.

The emphasis is on creativity, user experience, rapid iteration and vertical slicing of development. We do not go into the details of any particular project management practices or enforce that students must follow them. The weekly demos mean that the teams must integrate their features to have a working system each week, so they naturally follow a process of continuous integration and frequent release, without us requiring those practices explicitly. Our experience shows that if we require a team to demonstrate, for example, specific Scrum practices, then they tend to show them lip service, and write a report telling us what they think we want to hear, without really feeling the benefit of the practices in their projects (especially when projects are relatively small scale, as university projects tend to be). Hence, we focus on regular delivery of working software above all else.

These projects are a fun way to finish the year, allowing students to apply their knowledge and build a product. They also serve as a warm-up for the larger projects that they will undertake when they return in the third year.

3.2 Third Year—Software Engineering Group Projects

By their third year of study, computing students should be in the position where they can use their skills to engineer a substantial software system. We want to give them the opportunity to exercise and develop these skills in a relatively large group project over the course of a few months. Where the second-year projects have a small team, a short timescale, and creative freedom to develop whatever they want, third-year projects have bigger teams, a longer timescale, and a customer relationship to manage. All of this naturally requires more conscious management, and so we can support this by encouraging teams to follow agile methods more explicitly. The aims of these projects are to…

- apply software engineering tools and techniques
- apply management techniques for software projects
- develop a complex system for and with a customer, with a particular user in mind
- improve team work

Projects are done in groups of 5–6, between October and January (see Fig. 5), to a brief suggested by an academic supervisor (or in some cases an external company) acting as a customer to set requirements and guide the product direction. Students do not work on the project full-time, but alongside their lecture courses, including a Software Engineering course designed to support the project work. Each group has a different brief, but all are aiming to build a piece of software that solves a particular problem or provides a certain service for their users. Recent examples include an open-source implementation of Microsoft's RoomAlive (Jones et al., 2014), systems for estimating heart rate based on video or speech recordings, and verifying product provenance using BlockChain technology.

The aims from an educational point of view are to build the students' skills in teamwork and collaboration and to put into practice software engineering techniques that support this kind of development work.

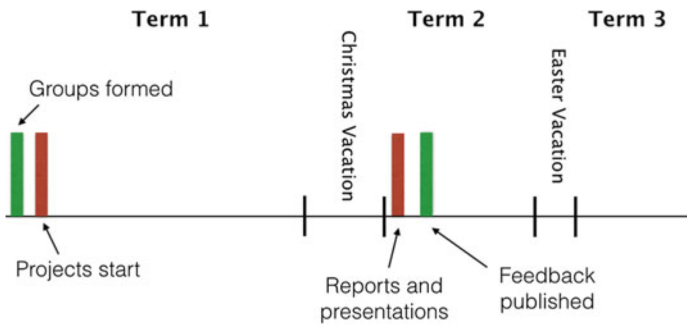


Fig. 5 Schedule for third-year Group Projects—project duration is 3 months. Figure © 2017 IEEE. Reprinted, with permission, from proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)

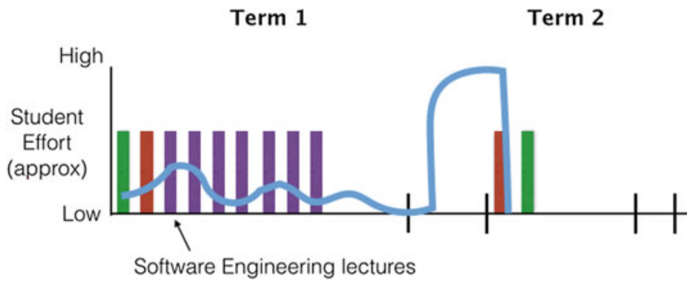


Fig. 6 Perceived effort curve for students during Group Projects (in blue). Figure © 2017 IEEE. Reprinted, with permission, from proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)

3.3 Sustainable Pace

We run these projects during the first term of the academic year when the students have just returned from their summer vacation. Something we observed in previous years was that students tended to leave the bulk of the work on their project to later in the term, with a big effort spike as the deadline neared—not working at the sustainable pace that we would hope to see in an agile project (see Fig. 6).

To encourage a more sustained pace of work, we introduced structured, time-boxed iterations. In the second-year projects described previously, students completed four 1-week iterations, and now for third-year projects we expect them to run their project as four 2-week iterations, through the autumn term, with a checkpoint at the end of each iteration where they demo their progress to their supervisors. More advanced teams could complete eight 1-week iterations if they prefer. In the 9th week, they should take a break for exams, and this then gives them the Christmas vacation to polish any final features, write up their reports and prepare their presentations which are given in January. The aims of the following structure and deliverables are…

- to encourage students to do more work on the projects earlier in the term
- to encourage sustained, iterative progress on projects
- to encourage projects to 'build a system that lets person X achieve Y', rather than research projects
- to remove any deliverables (such as reports, etc.) that do not directly add to the project

We wanted to find ways to get the students to start earlier. To this end, we have now phased out the lectures, and instead give them introductory talks introducing the structure and goals of the project on the first Monday of the new term, have them form groups on Tuesday, select projects on Wednesday and complete allocations of projects to groups by the end of Thursday. Given the way that our timetable works, this then gives them a couple of clear days to make plans and get started on the project before their lecture courses start the following week. We then run the four 2-week iterations from weeks 2 to 9 of the term (Fig. 7).

In forming project groups, so far we have let students select their own teammates. Although, as with pair-programming, there are suggestions that learning can be improved by carefully selecting students of varying academic strength and mixing them together in each group, we felt that overall, the experience of restructuring and improving our projects would be smoother if we did not have students complaining that they were not able to work with their friends. This is something that we might revisit in future, but for the moment we plan to leave the groups as self-selecting—we only constrain the team size. Before our focus on agile methods, we specified that each team should appoint a team leader, but mandating this did not seem to fit well with the collaborative nature of agile methods, so we have left it to teams to do as they think best. If they choose to do Scrum, they should appoint a Scrum Master, but that is someone who is responsible for the execution of the Scrum process and as such very different from a team leader who makes the decisions and allocates work.

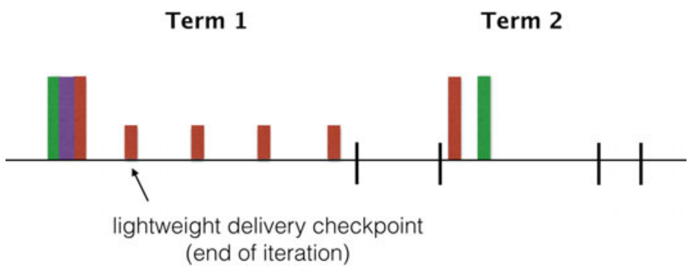


Fig. 7 Lightweight end-of-iteration checkpoints every 2 weeks. Figure © 2017 IEEE. Reprinted, with permission, from proceedings of 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)

3.4 Customer Relationships

The aim of these projects is to deliver a particular piece of software, rather than to conduct an investigation in a research area. Particularly in the case of some academic supervisors, we have had to steer them quite strongly to propose suitable projects. We found that projects that were very investigative, or required a lot of up-front reading of research papers, were generally not very compatible with our aims of delivery working software early and often.

To help to gather appropriate project briefs, we provide a template for the potential supervisors and customers to complete. The aim of using the template is to provide some consistency across project proposals, to make sure that the students have all the information they need in making their selection, and to make sure that project proposals fit well with the learning objectives in agile software development. Our project proposal template is as follows:

Please make sure that you have a particular piece of software in mind for the students to build. Your proposal should specify the following:

The project title is…

General background… a brief description of the context and purpose—a couple of paragraphs

The target user is… identify a particular user (or class of users)—be specific

The system should allow the user to achieve… be specific about the capability that the application should provide—what can you do when it exists that I can't do now.

Any technical (or other) constraints… e.g. this must be an iPhone app; this must run on a video wall; you must integrate this C++ library; you must collaborate with this external company, etc.

3.5 Lecturing Versus Coaching

Having completed their second-year projects, students should already have experience of working in an iterative way for a small project. Now, having reduced the *transmission* of Software Engineering materials through lectures, we just provide a recap lecture during the first week of term, reminding them of the agile techniques they used in their previous projects. Additional material giving more detail about specific methods (Scrum, XP and Kanban) is provided online. We also provide online material on continuous integration and delivery techniques, as well as case studies of commercial implementations of agile methods at companies like Spotify who are quite open (Kniberg, 2014) about the methods that they use.

Rather than giving further lectures, we now provide individual consulting time to all groups by holding office hours. Groups can book slots to get specific advice on any problems they are having, particularly around the areas of project management, evaluation, or other software engineering matters relevant to their project. Making these office hours optional meant that only the keen groups (often those not requiring much help) took up the opportunity. In light of that, we now mandate that each group must arrange at least two consultations over the course of the project, one during the first four weeks, and one during the second four weeks. The consultations are 30 min each, and we have around 30 groups, so this is 30 hours worth of work for the tutor over an 8 week period. We ensure that the tutor running these consultations is an experienced industry professional with experience of applying various agile methods in many different software engineering contexts. This way we can give high value, context-specific advice to each group, rather than advising them only on the textbook principles of agile. This approach, following a more *developmental* learning style, increases the relevance of the lessons taken away by each group and—hopefully—allows them to apply the specific advice directly to their project. Students can pull relevant theory as they encounter problems in their project work.

We do not mandate a set development process for the students to follow, but we encourage teams to adopt practices that might be used by an industrial team of a similar size carrying out a similar type of project. We suggest that they follow either Extreme Programming, Scrum, or Kanban (not a mixture)—and back this up with engineering practices such as continuous integration, automated testing and staged deployments. While these practices may not manifest themselves in exactly the same way between different teams, depending on the exact nature of their project, each team should be able to adopt and benefit from most of these in some guise. Again, individual coaching can help teams to adopt appropriate tools and techniques for their specific context.

3.6 Checkpoints

As previously explained, we structure these projects as four 2-week iterations. At the end of each iteration project, teams must meet with their supervisor/customer and give them a demonstration of the current state of their software. They should be able to show that they have made progress, and that their software has more (or better) features than it did at the last demo.

To structure these meetings, and to provide some consistency across groups and supervisors, we provide a simple checklist. We want to keep the assessment process lightweight for all parties, so we want to avoid writing and reading detailed reports. The checkpoint forms are quick to complete and quick to check. We provide each team with a 1-page PDF form, which they take with them when they demonstrate their product, and they ask the customer to complete and sign it. The team then scans the signed sheet and submits it as a piece of coursework. We also ask them to submit a set of three screenshots showing the current state of the digital tools they are using to manage three aspects of their project—their version control repository, their continuous integration build, and their project plan. Again, the idea is to have a deliverable that is quick to produce, and quick to check. The checkpoint form has the following questions for the customer to answer:

I certify that in this iteration I feel the group has… (check one):

- *not been able to demonstrate any new working software*
- *shown me something working, but a bit less than I had hoped for, or not what we agreed*
- *adequately delivered the features that we agreed on*
- *made better progress than I expected*
- *made amazing progress with wonderful results*

Has the list of risks to project success changed since you last met the group? What are the two main features agreed to be delivered for the next checkpoint?

Customer Signature/Date

Previously we just had binary checkpoints—either the customer was satisfied or they were not—but we have found that giving a way for customers to express their satisfaction on a scale has led to a more meaningful interaction, as well as more motivation on behalf of the teams to try to please their customers. This system of end-of-iteration checkpoints seems to be working well as a way to produce a more sustained pace of development across the term, rather than a big bang before the deadline, and also provides the benefits of agile development to the customer as they have more opportunities to see the product running, and steer future feature development so that they end up with something that meets their needs.

3.7 Summary

Our main learning in restructuring our project-based courses in Software Engineering was to focus on the regular delivery of working software. Whatever types of technical practice or project management technique we might encourage the students to adopt, they need to feel the benefits of those in helping them to deliver reliable software that meets their customer's needs, without working to a crazy schedule. As we want to foster creativity, and believe that students are more motivated when they get to choose from a wide range of projects, we need to provide support and tutoring that is specific to each team. Following a coaching model allows us to provide relevant, specific, help and advice to each team, at the point that they need it. This seems to be much more effective than more generalised transmission through lectures, especially given the varying needs of the different teams.

By coming up with an overall iteration structure, and a lightweight way of assessment through end-of-iteration checklists, it is possible for us to have a degree of consistency across the class, while still allowing different teams to work in quite different ways. We also found that this outline structure helps students to plan their work across the term, rather than leaving everything until just before the final deadline.

4 Future Directions

The methods described in this chapter are working well for us in these Software Engineering modules. But our Computing curriculum comprises many more modules besides these—modules on basic programming, compilers, operating systems, databases, logic, mathematics, etc. It is tempting to try to spread our agile and lean approaches to more modules, but we suspect there will be friction. Just as when introducing agile methods to software development organisations, change is hard. Lecturers running traditional lecture courses may be reluctant to increase the frequency of practical exercises, especially if that means more frequent assessment. The path of least resistance may be to stay with the status quo, but we believe that in transforming the modules described here we have increased their educational value, and hope that we can gradually spread this across our curriculum. Perhaps it is natural that instructors with experience of agile methods are the most keen to introduce them to their teaching, but we hope some enthusiastic colleagues will try to apply similar techniques in their classes too.

Another question is whether we can scale to larger class sizes. Lean methods allow us to improve efficiency and aim to make contact time between students and teachers more valuable. However, the close collaboration and frequent feedback we have implemented in our new structures do not relieve staff time. If we increased class sizes then we would still need more markers, more tutors, more coaches and more customers. Automation helps to remove some trivial tasks and to streamline some of the others, but so far we cannot see a way to remove the human element, and neither are we sure that we would want to.

In fact, the role of the instructor becomes critical. As agile implies high-contact collaboration, working closely together exposes problems and uncovers any lack of experience on behalf of the teacher. When coaching a project team, textbook knowledge is not enough, we need specialists with real experience of running agile projects in the wild. Removing waste from the learning experience has made the need for expert tuition the bottleneck in our system, which rather than a problem, is perhaps to be seen as a sign of success.

Acknowledgements We would like to acknowledge colleagues and students at Imperial College London for their contributions to the evolution of our curriculum.

References

Anderson, D. (2010). *Kanban: Successful evolutionary change for your technology business*. Blue Hole Press.

Anslow, C., & Maurer, F. (2015). An experience report at teaching a group based agile software development project course. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 500–505).

Duvall, P., Matyas, S. M., & Glover, A. (2007). *Continuous integration: Improving software quality and reducing risk*. Addison Wesley.

Fowler, M. (2011, July). Frequency reduces difficulty [online]. Retrieved from http://martinfowler.com/bliki/FrequencyReducesDifficulty.html.

Gamma, E., Helm, R., Johnson, R., & Glissades, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Guzdial, M. (2015). *Learner-centered design of computing education: Research on computing for everyone*. Morgan & Claypool Publishers.

Jones, B., Sodhi, R., Murdock, M., Mehra, R., Benko, H., Wilson, A., & Shapira, L. (2014). Roomalive: Magical experiences enabled by scalable, adaptive projector-camera units. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology* (pp. 637–644). New York, NY, USA: ACM. https://doi.org/10.1145/2642918.2647383.

Kniberg, H. (2014, March). Spotify engineering culture [online]. Retrieved from https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1/.

Kropp, M., & Meier, A. (2014). New sustainable teaching approaches in software engineering education. In *2014 IEEE Global Engineering Education Conference (EDUCON)* (pp. 1019–1022).

Ottinger, T. (2015). Over-starting and under-finishing [online]. Retrieved October 4, 2016, from https://www.industriallogic.com/blog/over-starting-and-under-finishing/.

Papatheocharous, E., & Andreou, A. S. (2014). Empirical evidence and state of practice of software agile teams. *Journal of Software: Evolution and Process, 26*(9), 855–866. https://doi.org/10.1002/smr.1664.

Rother, M., Shook, J., & Institute, L. E. (2003). *Learning to see: Value stream mapping to add value and eliminate muda*. Productivity Press.

Williams, L., & Upchurch, R. L. (2001). In support of student pair-programming. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education* (pp. 327–331). New York, NY, USA: ACM. https://doi.org/10.1145/364447.364614.